DESIGN OF INTERVENTIONS FOR INSTRUCTIONAL REFORM IN SOFTWARE DEVELOPMENT EDUCATION FOR COMPETENCY ENHANCEMENT

Thesis submitted in fulfillment of the requirements for the Degree of

DOCTOR OF PHILOSOPHY

By

Sanjay Goel

Department of Computer Science & Engineering and Information Technology

JAYPEE INSTITUE OF INFORMATION TECHNOLOGY
A-10, SECTOR-62, NOIDA, INDIA

April, 2010

DESIGN OF INTERVENTIONS FOR INSTRUCTIONAL REFORM IN SOFTWARE DEVELOPMENT EDUCATION FOR COMPETENCY ENHANCEMENT

Thesis submitted in fulfillment of the requirements for the Degree of

DOCTOR OF PHILOSOPHY

By

Sanjay Goel

Department of Computer Science & Engineering and Information Technology

JAYPEE INSTITUTE OF INFORMATION TECHNOLOGY
A-10, SECTOR-62, NOIDA, INDIA

April, 2010

DECLARATION BY THE SCHOLAR

I hereby declare that the work reported in the Ph.D. thesis entitled **"Design of Interventions for Instructional Reform in Software Development Education for Competency Enhancement"** submitted at **Jaypee Institute of Information Technology, Noida, India,** is an authentic record of my work carried out under the supervision of **Prof. J.P. Gupta and Dr. Mukul K. Sinha**. I have not submitted this work elsewhere for any other degree or diploma.

(Sanjay Goel)

Department of Computer Science & Engineering and Information Technology

Jaypee Institute of Information Technology, Noida, India

April 9th, 2010

SUPERVISOR'S CERTIFICATE

This is to certify that the work reported in the Ph.D. thesis entitled "**Design of Interventions for Instructional Reform in Software Development Education for Competency Enhancement**", submitted by **Sanjay Goel** at **Jaypee Institute of Information Technology, Noida**, **India** is a bonafide record of his original work carried out under our supervision. This work has not been submitted elsewhere for any other degree or diploma.

(Prof. J.P. Gupta)
Jaypee Institute of Information Technology

April 9th, 2010

(Mukul K. Sinha)
Expert Software Consultants Ltd.

April 9th, 2010

In revering memory of my grandparents,

Sh.(late) Chiranji Lal Goel, a dedicated teacher, who taught me that work is its own reward, and

Smt.(late) Shanti Devi Goel, who personified simplicity and patience.

TABLE OF CONTENT

CHAPTER-7

THE PHENOMENON OF 'LEARNING' 182

CHAPTER-8

A FRAMEWORK OF PEDAGOGIC ENGAGEMENTS IN SOFTWARE DEVELOPMENT EDUCATION 216

ACKNOWLEDGEMENTS

This work is the result of a long personal journey across a variety of professional experiences: learning, designing, teaching, and also leading teams. This work has humbled me and has made me realize the magnitude of my 'ignorance' about 'learning,' and also many intricacies of software development. During this journey, I have had the good fortune of wonderful and engaging interactions with experts and scholars of diverse disciplines.

To top the list, I am highly grateful to hundreds of software professionals from all over the world who have participated in many surveys, polls, and discussions during the course of this research. I am thankful to all my past, present, and future students, who are my main inspiration for this work. Some of the past students are my most valued professional consultants, collaborator, and critiques. I also want to express my gratitude to all my enthusiastic colleagues in the Department of Computer Science and Engineering at the Jaypee Institute of Information Technology for their confidence, support, and active collaboration in contextualizing and administering many ideas in their various courses.

I am most indebted to Dr. Mukul K. Sinha, my mentor for the last twenty years, for innumerable professional lessons and values that I have learnt from him. His systems thinking approach, ability to take risks, commitment for excellence, and coaching has been a great source of strength for sustaining this long and personally enriching inquiry. Only a few blessed people have the good fortune of receiving such selfless mentoring.

I am highly thankful to Prof. J.P. Gupta for his affection, generosity, and patience. But for his blessings and whole-hearted support, it was not possible to try out many instructional interventions that have provided very useful insights for this thesis.

Numerous discussions with Prof. M.N. Faruqui encouraged me to maintain my enthusiasm for carrying out this research. His critique and wisdom reflected the depth and breadth of his vision, as well as multifaceted and rich experience in higher education. I am also thankful to Prof. S.K. Kak for his interest and motivation. He introduced me to the SPINE project that became a very

important reference for this work. I have also learnt many lessons about curriculum design, and also critical inquiry, from Prof. S.L. Maskara. Several discussions with Prof. A.B. Bhattacharyya and Mr. H.S. Dagar were very enriching. Few Indian researchers get the benefit of such comprehensive editorial support, as was extended by Mr. Dagar. Moral support extended by Prof. S.K. Khanna, Prof. (Late) Prof. C.S. Jha, and Dr. Y. Medury has been very encouraging in this journey. I am also highly grateful to the co-authors, reviewers, and editors of all my papers.

It will not be proper if I forget to acknowledge the lessons I learnt about the value of context and holistic thinking at the Indira Gandhi National Centre for the Arts (IGNCA) during my tenure there from 1995 to 2002. My way of thinking and perceptions about excellence, diversity, scholarship, aesthetics, education, and its relationship with human life evolved significantly during the process of innumerable interactions with Dr. Kapila Vatsyayan, Prof. P.S. Filliozat, Prof. T.S. Maxwell, Prof. Saskia Kersenboon, Prof. R. Nagaswamy, Prof. Aditya Malik, Dr. V. Filliozat, Prof. B.N. Saraswati, Prof. Frits Stall, Prof. Anil K. Jain, Prof. Sutcliffe, Prof. Gary Marchionini, Prof. S.P. Mudur, Prof. Ranade, Pierre Pichard, Prof. John Emigh, and many others during the course of designing several interactive multimedia learning systems at the IGNCA.

Lastly, and equally importantly, I am highly thankful to my parents, wife, brothers, sister, and two sons for their care, love, and also tolerance for my carelessness. Not many people get as much pampering at home as has been extended to me since my childhood. Their generosity to take care of all my responsibilities at home enables me to focus more on my studies and work.

ABSTRACT

Community and culture significantly influence value orientation, perceived needs, and motivation as well as provide the ground for creating shared understanding. All disciplines have their own cultures, and all cultures evolve through cross-cultural exchanges. The computing community has created and documented a sound body of knowledge of software engineering (IEEE SWEBOK). It is one of finest examples of multi-cultural synthesis of many disciplines especially engineering, computer science, and even social sciences. With the very large scale worldwide endeavor on computing or software engineering education, it is now time to leverage education and 'learning' related research to create and document a theoretically sound body of knowledge of software developers' education. Such a body of knowledge should naturally require us to synthesis the evolving disciplines of software engineering and higher education.

In this thesis, we discuss our study and investigations about the following types of questions:

1. How has software development education evolved, specifically with reference to educational research?
2. What is meant by competent and professionally oriented computing engineers, especially with respect to software engineering? What are the essential attributes? What is the relative importance of these attributes?
3. What is the degree with which the various components of traditional processes of engineering education succeed in creating opportunities for enhancing these competencies? What students think about their educational experiences? What students think works well for them? What processes do professional engineers recommend?
4. What pedagogical practices succeed in developing competencies, and under what circumstances? What comes in the way of implementing these strategies? What kinds of lectures are effective for learning in the views of students and faculty? What factors block students from effective learning? How to overcome these difficulties?
5. What kind of instructional interventions are required? How can the existing education theories/strategies/methodologies be used to educate competent computing engineers? Do we need new theories of learning for software development education? If so, what would be main aspects of such a theory of learning?

In this study, the research processes included a wide-ranging survey of published literature in diverse areas of software development, computer science and IT education, engineering education, professional and higher education, learning theories, thinking, instruction design, and human development. The research also included a study of a large number of comments written by professional software developers about contemporary issues related to software development processes, required competencies, endorsements, etc., in various professional forums. More than three hundred professionals of more than sixty organizations from various countries have been consulted and/or surveyed on various issues. More than one thousand undergraduate computing students, and more than one hundred faculty members, have also been surveyed on selected issues.

We have proposed a three-tier taxonomy of twelve competencies and a comprehensive unified framework of pedagogic engagements in software development education. We have also discussed some instructional interventions developed by us, manifesting some aspects of this framework. All these interventions were administered in a chosen set of existing computing courses. Some new courses have also been developed in the process. The development of the framework of pedagogic engagement, and these interventions for instructional reform of software development education, has been an intertwined and highly spiral process.

We hope that our proposed framework of pedagogic engagement in software development education will help the community of software development educators and researchers to create a variety of interventions that will help in extending the 'Software Engineering Body of Knowledge' (SWEBOK) to 'Software Development Education Body of Knowledge' (SDEBOK). Designers of educational programs for other professions can also adapt this framework and methodology.

LIST OF FIGURES

LIST OF TABLES

CHAPTER-5

CHAPTER-6

CHAPTER-7

CHAPTER-8

CHAPTER-9

APPENDICES

ANNEXURES

1

CHAPTER 1: INTRODUCTION

Creativity combined with our understanding of nature, material, medium, other humans, and artifacts has always helped us in devising new processes for performing old tasks, and also devising new tasks in our personal, social, professional, and organizational lives. New processes and tasks require the use of existing artifacts in new ways, and also the creation of new artifacts. Often new processes bring advantages in terms of increased speed, reliability, scale, safety, comfort, or flexibility and/or savings in effort, energy, material as well as costs. In addition, humans have also used themselves both as the source of raw energy through physical labor, and as controllers through psychomotor skills to perform these tasks. Taming of animals, tapping of natural energy, steam engine, electricity, etc., helped to reduce our role as energy suppliers. Mankind could focus more on the other two tasks of being the controller and process designers. With the availability of control systems in the last century, our role as controllers has also reduced significantly, and more human energy is now available for the creative work of devising new processes and new tasks. Artisans, engineers, designers, and technologists play a key role in identifying the opportunities and developing new processes and tasks in diverse domains of human activities. Strength, malleability, expected life, and various other affordances of the material and medium influence and constrain our design activities. The digital computer is the most malleable artifact created so far, and it can be further used as a material and a medium to rapidly create a large variety of new artifacts in a very flexible way. This power has given an unprecedented boost to the development of new processes, as well as new tasks in all domains of human activities.

Engineers and technologists plan, design, develop, test, integrate, deploy, maintain, improve, reverse engineer, re-engineer, as well as evaluate components, products, applications, systems, services, standards, processes, and methodologies encompassing various artifacts. Their disciplines are differentiated with each other on the basis of the artifacts they build and focus on. In order to identify and create opportunities of devising new processes and ways in various domains, they need to understand the needs and nuances of those domains as well as humans' individual as well as social behavior. This is often the most critical and creative task, especially when the subsequent engineering processes are very rapid and fairly stabilized. US Accreditation

Board for Engineering and Technology (ABET) defines engineering as follows: "*Engineering is the profession in which knowledge of the mathematical and natural sciences gained by study, experience, and practice is applied with judgment to develop ways to utilize, economically, the materials and forces of nature for the benefit of mankind."*

Unlike science, *engineering and technology* are oriented towards conception, design, invention, development, application, improvement, and production with an emphasis on current and future needs of society. They require holistic thinking involving integration of many competing demands, theories, data, and ideas as well as decision making based on incomplete data and approximate models. The theorizing attempts go beyond the search of causes, and are focused on new processes and applications. Engineering is *not* just applied science*; it is as much about process as it is about technical knowledge*. An engineer's task involves conceiving and designing products, processes, and systems, and to predict their behavior using science. Scientists create models to understand natural phenomenon with known outcomes, whereas engineers create models to predict outcomes for systems. The use of heuristics distinguishes engineering methods from scientific methods. *Engineering* is further distinguished from *Technology* by its focus on more complex problems that involve use of more diverse resources, more diverse groups of stakeholders with varying needs, wider range of conflicting technical, engineering and other issues, more abstract thinking, originality, infrequently encountered issues, and work progress in spite of insufficiency of standards and codes of practice. *Technological work* needs mastery of discipline and context specific current knowledge, techniques, skills, and tools. A higher focus on quality and timeliness are its distinctions. Broadly, the educational programs of engineering and technology recognize many of these distinguishing aspects of the discipline and respond in various ways through their curriculum and educational methods.

As per the ACM-IEEE joint report [1], *Computing means any goal-oriented activity requiring, benefiting from, or creating computers.* It includes: designing and building hardware and software systems for any of a wide range of purposes, processing, structuring and managing various kinds of information, doing scientific studies using computers, making computer systems behave intelligently, creating and using communications and entertainment media, finding and gathering information relevant to any particular purpose, etc. Computing engineers

are concerned with four kinds of artifacts: (i) software, (ii) digital ICs and other hardware, (iii) embedded systems, and (iv) digital content. For the last four decades the demand of software developers has been increasing at an accelerated rate. Jalote [2] summarizes the growth of Indian software industry as follows: "*It started primarily as a subcontractor for technical manpower. … shifted to doing complete parts or phases of projects, usually the later phases of coding and testing. … matured to providing complete solutions offshore. …most leading companies are operating in the high-end software services business. … a large number of software companies matured to CMM level 4 or level 5…*"

In the last few years, there has been an exponential growth in engineering education, especially in India and China. This growth has led to an era where fresh graduates of computer science related disciplines are easily absorbed in the industry. Indeed, to satisfy the growing demand for software, very large volumes of engineers from other engineering disciplines are also absorbed as well. All engineering graduates are considered to be ready for a direct fit with the requirements of the IT industry [3]. The core competencies developed in all engineering disciplines are considered to be sufficient, and the companies rely more on their own finishing schools for specialized computer science and IT knowledge.

With the advent of the Internet, it has become possible to outsource software development tasks to remote sites, making India an attractive destination, both technically as well as financially. This has resulted in an exponential *increase in the demand of software developers* in India, especially in the last decade. It has become a challenging opportunity for Indian academic institutions to provide an adequate pool of software professionals of desired quality to the rapidly growing Indian software industry.

To meet this challenge, the Indian academic institutions have been able to expand fast and *satisfy the industry's need of software professionals quantitatively*. However, the quality of 'most of' the professionals they generate is *below the desired industry expectation*. The software industry associations, as well as the academic regulatory bodies, have repeatedly shown their concern emphatically about the sub-standard quality of 'the majority of' fresh software professionals [4]. Most of the software houses spend around six months to one year in their post-induction, in-

house software development education and training of fresh engineers. It clearly indicates that there is a significant gap in the technical education that academic institutions impart to their software graduates, and what technical expertise the industry expects in them. A *competence mismatch* exists between academic technical offering and software industry employability. We elaborate upon this in the second and third chapters.

It is not proper to consider software industry as monolithic group. Even in India, there are different kinds of companies, those involved in software services, and those involved in new product development in large or small companies. There are huge differences in the requirements of these categories. Often, India's highly dominant software service industry's immediate requirements dwarf other requirements, which are more futuristic and even more compatible with the goals of excellence in higher education. In sections 2.9 and 2.11, we especially examine the needs different kind of software industry. A NASSCOM-KPMG study [5] argues that key skills required by the industry are not met by the current educational system. It quotes the following observations from a World Bank study on science and technology manpower in India published in 2001: (i) faculty lacks industry rigor, R&D background, and exposure to tools, (ii) students lack opportunity and encouragement for creative thinking, (iii) inflexible and rigid curriculum is not exposed to innovation/industry, (iv) teaching is examination oriented without focus on communication and problem solving skills, (v) continuous evaluation is often not systematized, and (vi) examinations are often memory based, and encourage partial studying through ample choice.

Organizations and their clients have limited tolerance for inept performance. Often engineers engage directly with clients in complex interactions. Educators are expected to teach competencies that are relevant and enhance an organization's performance [6]. Stephen says, "Anyone not aware that this is a time of change in higher education is asleep at the helm" [7]. Universities around the world have become increasingly aware of the need to be able to demonstrate, in a quantifiable manner, the skills and attributes that their graduates are imbued with during their learning experience [8].

State of Indian Contribution in Computing Research

There are over a million software engineers working in India. Further, there are over two thousand colleges offering degree level educational programs in computing. The IT industry's share in India's GDP is more than 7%. Seven Indian IT companies have been listed in the top 15 technology outsourcing companies of the world. However, Indian organizations' contribution to computing research literature remains very meager. The ACM digital library gives access to almost 0.3 million papers. *Less than 0.7% papers have been contributed by authors having Indian affiliations.* Before 2005, this fraction was only 0.3%. During 2005 to Feb 2010, it increased to 1.3%, which still is a very small number, given the huge number of software engineers and colleges offering computing degrees in India.

A focused search (using affiliation option under advanced search) in March 2010 showed that some of the largest India-based IT companies, i.e., TCS, Infosys, Wipro, HCL, Satyam, Oracle India, have together so far collectively published less than 100 papers that are indexed on this digital library. This library does not include a single paper from other very large Indian IT companies like Tech Mahindra, Patni Computers, and Birlasoft. On the other hand, Microsoft India and IBM India have published approximately 300 papers, and Microsoft and Google have contributed 3,885 and 582 papers respectively. *This highlights that the mismatch is not just in terms of immediate specific needs of industry, but also long term goals of professional excellence*. This numbers highlight the gross mismatch between published contributions and the size of India's IT industry, and the number of computing professionals in the industry or academia. *In addition to meeting industry's short term needs, software development education can also stimulate the overall growth of India-based computing research contributions by arousing interest among future software developers.*

Section 1.1: Basis for the Need for Reforms in Computing Education

This thesis attempts to contribute towards bridging this competence mismatch by providing ideas for instructional reforms in computing education with special reference to software development. Unlike other disciplines of engineering, computer scientists have always remained interested in understanding the phenomenon of 'learning.' Artificial intelligence and computer based teaching were the earlier sub-disciplines within computing that required and encouraged computer

scientists to understand various issues associated with 'learning'. The International Federation for Information Processing (IFIP) established a technical committee on education in 1963. In its very early years, The ACM also founded special interest groups SIGCSE (Special Interest Group on Computer Science Education) and SIGCUE (Special Interest Group on Computer Uses in Education). More recently, the ACM has started SIGITE (Special Interest Group on Information Technology Education).

Reforms in engineering education have a long but slow history. Felder [9] remarked, "We teach primarily mechanics, and not reasoning methods; memorization and routine application, and not analysis, synthesis and evaluation. We don't encourage creativity and independence of thought, and in fact often do our best to discourage them." Sadly nothing much has changed on the ground. *The community that is responsible for transforming the lifestyle of the world has not yet transformed its own educational process.*

Many engineering faculty have never practiced engineering [10]. The curriculum's focus on content is disconnected from engineering practices [11-12]. Felder and Brent [13] reported on some recent studies that measured the intellectual growth of engineering students during their studies using Perry's model of epistemological development [14]. It was observed that the engineering education failed to elevate a significant number of students to level 5 as per Perry's nine-level model, and the average growth after four years of college was only one level, with most of the change occurring in the last year.

Our exploratory study has shown that the kind of activities that a typical engineering student is generally engaged in, does not help in enhancing creativity, critical thinking, and innovative problem solving [11-12]. However, the last decade has seen an increasing recognition of the need for transformation. A certain section of policy makers, universities, accreditation agencies, and faculty members have made tremendous contributions to bring the much needed transformation. Many accreditation agencies have even transformed their accreditation criteria in the last few years. This is expected to drive an unprecedented transformation of instructional programs in responding institutes. This challenge can only be met by undertaking large scale research in engineering education.

Recognizing the need to re-engineer the engineering education a recent report '*Educating the Engineer of 2020*' [15] suggests that "*the engineering education establishment should endorse research in engineering education as a valued and rewarded activity for engineering faculty as a means to enhance and personalize the connection to undergraduate students, to understand how they learn, and to appreciate the pedagogical approaches that excite them.*"

One of the founding fathers of modern education, Franklin Bobbitt observed that curriculum should aim to teach those subjects that are not sufficiently learnt as a result of normal socialization. In 1920s, he proposed a five step process for curriculum design: analysis of human experiences in a field, job analysis to identify specific activities, deriving objectives to identify the abilities required for specific activities, selecting objectives as the basis of students' activities, and planning in detail. Paulsen and Peseau [16] proposed a framework of Zero Based Curriculum Review process that starts with first operationalising the curriculum goals as categories of required professional competencies, and then identifying appropriate knowledge base learning objectives and also behavioral objectives in terms of professional practices, and skills with respect to required professional competencies.

Woods et al [17] proposed the following process for engineering faculty: (i) identify the skills you wish your students to develop and communicate their importance to the students, (ii) use research, not personal intuition, to identify the target skills, share some of the research with the students, (iii) make explicit the implicit behavior associated with successful application of the skills, (iv) provide extensive practice in the application of the skills, using carefully structured activities, and provide prompt constructive feedback on the students' efforts, (v) encourage monitoring, (vi) encourage reflection, (vii) grade the process, not just the product, and (viii) use a standard assessment and feedback form.

An exploratory informal discussion with large number of faculty members of engineering institutes with teaching experience ranging from a few months to several decades, and coming from different departments of engineering, sciences and management, it was found that most were not aware of any literature in educational research. Hence, by and large engineering

education methods have remained unaffected by such research. In 1982, Professor Richard Felder [9] presented a revolutionary thought that *'does engineering education have anything to do with either one.'* The curriculum and educational committees of the ACM, IEEE, AIS, AITP, LACS, IFIP, etc. have mostly ignored the rich educational literature related to curriculum design, instruction design, assessment methods, theories of learning, human development, epistemology, and sustainable development. Only a few of the available theoretical models and frameworks in these education related areas have been examined, reviewed, and/or used by the researchers of software development education.

UNESCO has labeled 2005-2015 as the decade of education for sustainable development. In this decade, bodies like National Science Foundation (NSF), USA and the National Academy of Engineers (NAE), USA have emphasized the need of systematic research in 'learning' to transform engineering education. In 2006, NAE identified the following research areas for engineering education [18]:

1. Engineering Epistemologies: Research on what constitutes engineering thinking and knowledge (technical, social, and ethical aspects) within social contexts now, and into the future.
2. Engineering Learning Mechanisms: Research on engineering learners' developing knowledge and competencies in context.
3. Engineering Learning Systems: Research on the instructional culture, institutional infrastructure, and epistemology of engineering educators.
4. Engineering Diversity and Inclusiveness: Research on how diverse human talents contribute solutions to the social and global challenges and relevance of our profession.
5. Engineering Assessment: Research on, and the development of, assessment methods, instruments, and metrics to inform engineering education practice and learning.

Section 1.2: Evolution of Software Development Education

In this section, we discuss the evolution of software engineering education. Table 1.1 gives a list of some of important reports examined in this discussion.

Table 1.1: Some important reports on computing curriculum

1. ACM curricula committee for CS (1965)	18. ACM/IEEE (UG and PG) (1991)
2. ACM Curricula for CS (UG and PG) (1968)	19. Model Indian curriculum for CSE (UG) (1993)
3. COSINE' IEEE for CS in EE (1968)	20. IFIP curriculum for CS (UG) (1994)
4. COSINE' IEEE for CS in EE (UG) (1971)	21. LACS curriculum for CS (UG) (1996)
5. ACM curriculum on IS (UG) (1972)	22. ACM curriculum on IS (UG) (1997)
6. ACM curriculum on IS (PG) (1973)	23. IFIP curriculum for Informatics (UG) (2000)
7. IEEE Model Curricula for CSE (UG) (1975)	24. AICTE curriculum for CSE (UG) (2000)
8. IEEE Model Curricula for CSE (UG) (1977)	25. AICTE curriculum for IT (UG) (2000)
9. ACM Health Computing Curriculum (UG and PG) (1978)	26. ACM IEEE curriculum on computing (2001)
10. ACM Curricula for CS (UG) (1978)	27. SEI-CMU Software Engineering Body of Knowledge Ver 1.0 (1999)
11. ACM Curricula for CS (PG) (1981)	28. ACM/AIS/AITP curriculum for IS (2002)
12. ACM curriculum on IS (UG and PG) (1982)	29. IEEE SWEBOK (2004)
13. IFIP curriculum for CS (1984)	30. ACM-IEEE curriculum for SE (2004)
14. CMU curriculum for CS (UG) (1985)	31. ACM-IEEE curriculum for CE (2004)
15. LACS Model Curriculum for CS (UG) (1986)	32. ACM- IEEE curriculum for CS (2005)
16. ACM report on Computing as a discipline (UG and PG) (1989)	33. ACM-IEEE curriculum for IT (2005)
17. SEI model curriculum for SE (UG) (1990)	34. LACS curriculum for CS (UG) (2007)

Beginning of Computing and Computing Education

Computing in the form of processing: understanding, creation, manipulation, communication, expression, and rendering of symbols has always been a very important natural activity of human mind. Though the use of the term *computing* is not limited to be used in the limited context of processing of formal mathematical symbols, computer software transcends such boundaries to support processing of diverse range of symbols. With the invention of computing machines, the field of computing has advanced beyond one's imagination. Computing has transformed many aspects of everyday lives for a vast majority of mankind. The role of computing has been evolving from enhancing efficiencies through otherwise by-passable support systems to creating real-time mission critical systems. The initial application domains driving computing till 1960s were code breaking, engineering calculations, scientific simulation, as well as repetitive data processing in defense, space, government, insurance, banking, and some other large business organizations. Some attempts of language translation and information retrieval were also made even in 1950s. Outgrowing the initial goal of doing repetitive mathematical calculations, computers have already permeated almost all spheres of human activities even including arts and

sports. The socio-cultural effect of computing and communication technology is much wider, deeper, and faster than the effect of other technologies. Computing has also been used to expand our understanding of mind and reasoning.

India's decimal number system inspired ninth century Persian mathematician Mohammed ibn Musa al-Khowarizmi to write a book on calculating using this number system. Based on his name, *Algorism* slowly started referring to arithmetic operations in this number system. These algorisms were strictly mechanical procedures to manipulate symbols. They could be carried out by an ignorant person mechanically following simple rules, with no understanding of the theory of operation, requiring no cleverness and resulting in a correct answer. The word *Algorithm* was introduced by Markov in 1954 [53]. Before the 1920s, the word *computer* was used for human clerks that performed computations. In 1936, Turing and Zuse independently proposed their models of the computing machine that could perform any calculation that can be performed by humans. In the late 1940s, the use of electronic digital computing machinery based on stored program architecture became common.

In the late 1950s saw the arrival of high level languages. The Association of computing Machinery (ACM) was founded by Berkeley in 1947. It started its first journal in 1954. Mathematical logic and electrical engineering provided the foundation for building modern computers. The personnel training responsibility was largely taken up by the manufacturers themselves. Most early programmers were math graduates, many of them were women. In the 1950s, a large numbers of private computer schools emerged to fill the burgeoning demand [19]. The word *software* was coined by John Tukey, famous statistician, in 1958. The words *computer science*, *information systems*, *information technology*, *system analysis*, and *system design* were being used even before. Dunn of Boeing [20] defined Information Technology as a body of related disciplines which lead to methods, techniques, and equipment for establishing and operating information processing systems. He also provided a simple definition of information systems as a connective link between five basic management functions of defining objectives, planning, gathering resources, execution, and control. In 1968, the computer science study group of NATO Science Committee coined the word *software engineering* to imply the need to transform software design and development into an engineering-type discipline.

Till 1970's, computing was often regarded as a subfield of one or more of a mixture of disciplines of mathematics, operation research, electrical engineering, statistics, industrial engineering, and management. Many of existing undergraduate programs of these disciplines were modified to accommodate some of the naturally fitting aspects of computer *science*. Mathematics departments taught practice and science of programming and numerical analysis. The electrical engineering department emphasized on design and construction of electronic digital computer, and management schools paid more attention of design of information systems. Initially, masters and later undergraduate degree programs and departments of *computer science* were emerging as offshoots of the mathematics departments in colleges of science and arts. Stanford established its computer science department in 1962, and by the late 1960s many universities in United States had started computer science departments. Concurrently, the management schools and others interested in business data processing applications focused on *information systems,* and started developing these programs. The engineering schools offered *computer technology* and *computer science* programs, and also computer as an option in various existing programs.

Early Curriculum Recommendations by ACM

The Association of Computing Machinery (ACM) unified the pioneering efforts of several universities and stimulated the process through its two independent curriculum committees established in the mid 1960s. The International Federation for Information Processing (IFIP) established a technical committee on education, TC-3, in 1963. Simultaneously, Various other professional agencies like the Computer society of the Institute of Electrical and Electronics Engineers (IEEE), and Data Processing Management Association (DPMA) made significant contributions in these efforts.

The first ACM Curriculum Committee on Computer Science (C^3S) was formed in 1964. In its preliminary recommendations [21], the committee posited that computer science is concerned with information in much the same way as physics is concerned with energy. It mainly identified careers in *systems programming* and *application programming* for computer science students. It distinguished computer science from mathematics by highlighting that while mathematician is

interested in discovering the syntactic relation between elements based on a set of axioms which may have no physical reality, the computer scientist is interested in discovering the pragmatic means by which information can be transformed to model and analyze the information transformation in the real world. The final report, Curriculum'68, considered development of programming skills as an important by-product rather than the main purpose of the computer science programs. It emphasized that computer science programs must provide the student with the intellectual maturity to stay abreast of their discipline, and also interact with other disciplines through liberal education. The curriculum recommendation [22] identified three major categories of computer sciences subject areas. These were *information structures and processes*, *information processing systems*, and *methodologies*. The first category of *information structures and processes* concerned with representations and transformations of information structures, and theoretical models for such representations and transformations. It included data structures, programming languages, and models of computation. The second category of *information processing systems* included computer organization, translators and interpreters, computer and operating systems, and special purpose systems. The last division of *methodologies* focused on broad areas of applications of computing which have common structures, processes, and techniques. It incorporated numerical mathematics, data processing, symbol manipulation, text processing, computer graphics, simulation, information retrieval, artificial intelligence, process control, and instructional systems. The committee recommended the inclusion of at least two courses from each of three categories for a masters program in computer science. For the undergraduate program, the essential computer science courses included introduction to computing, computers and programming, introduction to discrete structures, numerical calculus, data structures, programming languages, computer organization, and systems programming. The committee recommended the inclusion of at least two of the following computer science courses for indicated specialization: (i) compiler construction for applied systems programming and data processing application programming, (ii) switching theory for all other than scientific application programming, (iii) sequential machines for computer organization and design, (iv) numerical analysis-I for scientific application programming, and (v) numerical analysis-II for scientific application programming.

Early Engineering Perspectives

Electrical engineering departments identified computers as one of their main components. The Committee on Computer Science in Electrical Engineering (COSINE Committee), National Academy of Engineers (NAE), USA published recommendations for infusing computer science in electrical engineering curriculum. This led to the formation of *computer engineering* programs in electrical engineering departments. Developments in computers started to help in developing new methods of solving engineering problems. The COSINE committee strongly recommended [23] a total reorientation of electrical engineering curricula from analog and continuous to digital and discrete. In 1968, the computer science study group of NATO science committee coined the word *software engineering* to imply the need to transform software design and development into an engineering type discipline. This, however, was given legitimate attention as an academic discipline in the late 1970s.

In 1971, the COSINE committee recommended the start of a new undergraduate program called computer engineering within electrical engineering departments. This program was conceived as an engineering program with emphasis on the concepts of design of software, hardware, and systems. It proposed three specialization options under this program: (1) digital systems engineering, (2) software systems engineering, and (3) theoretical computer science and engineering. A juxtaposition of the COSINE subject list with the list suggested in C^3S' Curriculum'68 for computer science shows that, while on one hand, C^3S recommendations had ignored the hardware and design aspects, the COSINE recommendations ignored discrete structures and data structures. In 1975, IEEE computer society education committee identified and addressed this dichotomy in their recommendations, and proposed a new undergraduate program on *computer science and engineering* integrating courses in hardware systems, software systems, and theory of computing [24]. These courses were expected to constitute approximately 50% course requirement. The remaining 50% courses were to be in the areas of humanities and social sciences, physics, chemistry, communication, mathematics, economics, electronics, and engineering sciences as per Engineers Council for Professional Development (ECPD) guidelines. Sloan [25] and Engel [26] compared the new evolving recommendations of C^3S and model curriculum of the IEEE Computer society and concluded that the two were virtually same with

respect to their recommendations in the area of software engineering and program design. Their emphasis differed with respect to hardware and logic design on one hand and theory on another.

Early Information Systems Perspective

The Curriculum Committee on Computer Education in Management (C^3EM) of ACM published a position paper in 1971 [27]. Education for improving organizational productivity through information technology was the main motivation for this and subsequent committees in this area. This committee felt concerned about the unfavorable attitude of computer science departments towards applied problems. A few years later, this committee evolved into the ACM Curriculum Committee in Information Systems (C^2IS). In its recommendation report submitted in 1972 and 1973 [28-29], it identified requisite knowledge and abilities of information system graduates and grouped these into six categories of people, models, systems, computers, organization, and society. The ACM curriculum committee of computer science did not pay specific attention to this aspect until 1980s, and depended on general liberal education to provide the necessary breadth without specifying their specific recommendations.

These two C^3EM reports explicitly recognized two categories of information system programs at masters as well as undergraduate level: (1) technically trained systems designers, and (2) managerially oriented information analysts. The committee recommended the inclusion of five major topic areas of computer science, information systems, management, operations research, and systems design techniques. In 1973, this committee published its recommendations for undergraduate programs, and strongly argued for starting undergraduate programs in information systems in the light of very high manpower requirement at programmer and systems analyst level. It encouraged the computing centers as well as departments of computer science, business, electrical engineering, and industrial engineering to start undergraduate programs with their chosen concentration options on technology or organization. The committee also recommended one-year masters program in information systems for these students.

A few years later, this committee evolved into the ACM curriculum committee in information systems (C^2IS). It is not clear why the committee chose not to explicitly include computer

programming as a compulsory course in the technology concentration. This anomaly was corrected in the 1982 recommendations of C^2IS.

In later decades, a new trend of domain specific computing programs emerged. This trend resulted in establishment of many programs like medical or health informatics, geo-informatics, bio-informatics, chem-informatics, social informatics, and so on. In 1978, the ACM curriculum committee on health computing published its recommendations [30]. In many of these domain specific programs, up to 50% of the course content was related to domain specific foundations and domain specific aspects of informatics. The remaining courses focused on generic mathematics, statistics, information systems, computer science, and general education. ACM curriculum committee cautioned against somewhat frivolous proliferation of specialized programs [31]. However, in current era, specialized programs addressing the needs of specific domains are becoming important.

The 1981 report of C^2IS [32] emphasized that the demand of personnel with technical and organizational skills is relatively much greater than the demand for solely technical skills or organizational skills. It expressed its general concern over the ad-hoc basis of instruction of systems analysis and design. In its 1982 report [33], this committee proposed separate MS and MBA programs for the two streams of information systems.

As per the 1982 recommendations of C^2IS, considering the nature of the professional work of information system specialists, a strong emphasis (more than 20%) was placed on social sciences and humanities including economics, psychology, and English. It was argued that such a background helps in development of many essential attributes of requirement and systems analysts. The hiring of computing professionals in India has always been highest for information systems and software engineering related work. However, it is surprising that such undergraduate engineering programs have not been developed in India. The three year Master of Computer Application (MCA) programs also have a relatively heavier dominance of computer science and management related courses, and pay only little attention to these aspects related to the computing profession. The lack of strong industrial participation in curriculum design,

professional inclinations of curriculum designers, and educational politics in India may have contributed to this phenomenon.

Liberal Arts Perspective

The model curriculum recommended by Liberal Arts Computer Science Consortium (LACS) attempted to define computing program in terms of their approach towards data structures and algorithms [34]. It proposed that a computer science program is more interested in the formal properties of data structures and algorithms, a computer engineering program focuses more on their realization, and an information systems program is more orientated towards applications. Even after two decades with many changes in computing arena, in its 2007 model curriculum, LACS has only slightly modified their original definition of computing programs. The realization part has now been partitioned into two categories of linguistic and hardware realization.

The 1986 report and all subsequent reports of LACS, put more emphasis on discrete mathematics and place it along with first introductory computing course before other mathematics courses. In addition to two introductory computing courses, the 1986 report proposed four core computing courses on computer organization, algorithms, theory of computation, and principles of programming languages. These recommendations were only marginally revised by the consortium even after ten year [35]. In its 2007 recommendations, software development has been added to this category.

A typical liberal arts computer science program is more broad-based than specialized programs, and it includes more than 50% non science courses in the area of humanities, social sciences, etc., [36]. It is unfortunate, that such programs do not exist in India, and software development education is mainly linked with engineering programs. This possibly has contributed to a nearly non-existing or marginal inter-disciplinary activity between computer science and these areas. In the west, it is not uncommon to have a degree in computing and philosophy, computing and art, and so on. Perhaps, it is time to consider the option of a liberal arts oriented design degree with specialization in computing in India.

Changing Role of Mathematics in Computing Curriculum Recommendations

In the first decade, the computer science curriculum was lesser oriented towards business data processing needs. Interestingly, discrete structures and three courses in numerical methods were not considered as part of mathematics courses. Instead they were included as essential computer science courses. The committee further suggested a minimum of six mathematics courses for undergraduate programs. The committee proposed essential inclusion of courses in related areas of mathematics, statistics, electrical engineering, philosophy, linguistics, industrial engineering, and management. Overspecialization at undergraduate level was discouraged by the committee, and it also encouraged the deep involvement of computer science faculty in computer applications. Scientific simulation and engineering calculation oriented applications encouraged to put a strong emphasis on numerical methods.

The strong emphasis on numerical methods decreased gradually through subsequent recommendations, and it was eliminated from the core in nearly all subsequent recommendations of the ACM, IEEE Computer society, as well as other bodies except International Federation of Information Processing (IFIP). Computing curricula [37] does not specify any minimum required weight of numerical techniques for any of the five computing discipline – computer science, computer engineering, information systems, software engineering, or information technology. It is not recommended even as an elective course for the later three disciplines.

On the other hand, discrete mathematics was increasingly being recognized as more central and fundamental for computer science than calculus [38-40]. There were proposals to teach discrete mathematics as the first mathematics course, and the model curriculum for liberal arts degree in computer science responded favorably [41] [34]. In 2001, 76% faculty members are reported to have felt that discrete mathematics should be a prerequisite to data structures [42]. However, many universities and institutions were slow to respond to this change. A survey [43] showed that even in late 1980s, nearly 30% universities and institutions in USA did not include discrete mathematics, and nearly 27% maintained numerical algorithms in the core curriculum of computer science.

Possibly because of IFIP influence, for quite some time, numerical techniques continued to be part of the core curriculum of many computing programs in India for some time. The current model curriculum recommended by the All India Council for Technical Education, India [44-45] has not included numerical mathematics as a core course for both the commonly offered undergraduate computing programs of engineering institutes: (1) computer science and engineering, (2) information technology. Unfortunately, even discrete mathematics is excluded from the list of AICTE's information technology curriculum.

Over the decades, with the advent of faster, cheaper, smaller, reliable, networked, and mobile hardware, as well as user friendly and multi-layered software, the computer applications have rapidly expanded much beyond the scope of computational science around numerical techniques, modeling and simulation, and operation research. Lethbridge [46-48] found that in the list of the most important twenty-five subject topics of the university curriculum, professional software engineers did not include a single topic of mathematic. Though computational science is recognized as an extremely valuable closely related discipline, the recommended core body of knowledge of computing curricula with specialization in computer science, computer engineering, software engineering, information systems, or information technology, do not include these courses any more [49-50].

Further, the ACM-AIS-IEEE joint report [51] has recommended a lowered minimum requirement for mathematical foundation for programs in software engineering, information systems, and information technology. ACM-IEEE joint curriculum recommendation on software engineering [52] has included only one topic of mathematics 'discrete mathematics' as part of the essential core. Recently, differentiating computer science from mathematics, Fant [53] argues that rather than computational issues, computer science is more concerned with issues related to creation and actualization of process expressions.

Human and Social Aspects in Computing Curriculum

Till the 1970s, sociological, economic, and educational implications of developments in computer science were not considered as major responsibility of computer science. The report

recommended that computer science faculty should cooperate with concerned departments to develop courses in these areas, and computer science students should be encouraged to take these courses. However, computers were been increasing recognized as agents of social change. Professional bodies started paying more attention to understanding the social impact of computing.

In 1976, IFIP added a new technical committee, TC-9: Relationship between Computers and Society. The ACM curriculum committee also responded to this trend, and included *computers and society* as a strongly recommended elective in Curriculum'78 [54]. It was also suggested that such a course should be taught by computer science faculty. The committee recommended that meaningful computer applications should be cited and reviewed throughout the elementary material. The committee posited that structured programming along with social, philosophical, and ethical considerations are of such importance to the development of computer scientists that they must permeate the instructions at elementary levels. In all subsequent recommendations of the ACM, IEEE, IFIP, and others this proposal was further strengthened and this course was often included in the core. Most of the subsequent recommendations provided a more central position to this area. IFIP [55] recommended *computer and society* as part of the core for six variants of computing programs. Computing curricula [37] specifies '2' as the minimum weight of legal, professional, ethical, and social aspects on a scale of 0-5 for all their five forms of undergraduate computing discipline. However, some studies [46-48] showed that in spite of strong recommendations from professional bodies, this area received lesser than required attention during formal education in the opinion of responding practitioners.

The C^3S published a survey of computer science education [56]. This report was a mere catalog of various reports and papers without any observations or conclusions. It badly failed to critically review the previous literature or propose future trends. A year later the committee on computer science published their new recommendations, Curriclum'78. Mathematics requirements were mostly unchanged, and the report was criticized for being retrogressive in this aspect. The committee posited that structured programming along with social, philosophical, and ethical considerations are of such importance to the development of computer scientist that they must permeate the instructions at elementary levels. The core computer science and mathematics

courses constituted less than 50% of course requirement. Additional course requirements were proposed to be fulfilled through electives and courses in humanities, sciences, engineering, and social sciences. General liberal arts requirements were expected to give breadth to the program. The report was criticized for taking a fragmented approach [38].

In 1981, the C^3S submitted its recommendations for master's level program in computer science. It prescribed that the basic intention of master program is to develop students' critical and professional thinking and intuition to enable the graduates to take sound professional decisions with awareness of ACM code of ethics. Development of written and oral communication skills, cognizance with pertinent literature in their field of choice, teamwork, and leadership skills were also included among the prescribed goals. However, the committee did not make any specific recommendations to ensure that the curriculum meets the stated objectives. It recommended a list of thirty masters level courses, and classified into following five categories: (i) programming languages (six courses), (ii) operating systems and computer architecture (seven courses, including computer communication networks), (iii) theoretical computer science (four courses), (iv) data and file structures (four courses), and (v) other topics (nine courses). The C^3S failed to use this opportunity to make a defining and novel contribution towards curriculum design through these reports of late 1970s and early 1980s. The curriculum committee's reports of late 70s and early 80s have been later criticized for being reactive rather than proactive [57].

In the last few years, with the emergence of new specialization tracks of human computer interaction and also entertainment computing, sociology, art, philosophy, and psychology related courses have become even more important. Some of the recent programs include many courses from these areas by replacing courses of natural science, management, and electronics [58]. Currently, out of thirteen technical committees of International Federation for Information Processing (IFIP), four committees directly relate to human aspect of computing: (1) Education (working since 1963), (2) Relationship between Computer and Society (established in 1976), (3) Human-Computer Interaction (working since 1989), and the most recent (4) Entertainment Computing (founded in 2002). These committees seek to promote use of models, theories, and methods of social science, human sciences, ethics, psychology, culture, education, and aesthetics

in both design and evaluation of user orientated computer systems and humanization of system design process.

The AICTE model curricula for computing disciplines [44-45] have not taken cognizance of these developments and place the curricula only in the limited context of natural science, mathematics, physical aspects of engineering, and business. The important and pervasive context of human culture and society has not even been included in the agenda.

Beginning of Consolidation

The 1980s was the period of maturation and organized growth of computer science programs in many countries, including India. United Nations Educational, Scientific and Cultural Organization (UNESCO) commission IFIP to propose a modular curriculum especially for developing countries. IFIP submitted its first recommendations in 1984, and revised recommendations in 1994. The IEEE Computer society and ACM jointly specified criteria for the computer science curriculum [59-60]. It mandated a broad based computer science core giving even emphasis on computer theory, algorithms, data structures, programming concepts and languages, and computer elements and architecture. It insisted on inclusion of social implications of computing within the core computer science segment of the program. Mathematics and science were recognized as supporting disciplines, and the criteria sought to provide breadth through humanities, social sciences, and other disciplines. Advanced computer science topics were recommended to be addressed through electives.

The ACM task force in cooperation with Computer society of IEEE [61] started to define the computing discipline and observed that the three paradigms of theory, abstraction, and design are equally important and fundamental to computing. Computer science mainly deals with theory and abstraction, whereas computer engineering deals with abstraction and design. The task force identified two broad area of competency development: (1) discipline oriented thinking, and (2) tool usage, with the first being the primary goal of curriculum. It felt concerned about the neglect of laboratory exercises, team projects, and inter-disciplinary studies. The task force identified three purposes of laboratories in computing courses: (1) demonstrate how principles covered in lectures apply to design, implementation, and testing of software and hardware, (2) emphasize

the use of tools and processes, and (3) introduce experimental methods. Further, the task force provided a novel curriculum design framework by dividing each of these sub-areas into three parts of theory, abstraction, and design. The task force identified nine sub-areas of computing. It observed the need of diversity and well-intentioned experimentation in computing curricula.

The joint ACM/IEEE-CS curriculum task force published its report in 1991. The report [62] represented a unified set of recommendations from two major societies in a variety of academic contexts, including liberal arts, sciences, and engineering. This task force chose to exclude information systems from its agenda, and included all other variants like computer science, computer engineering, computer science and engineering, informatics and other similar program under the single title of computing. It emphasized the importance of breadth, laboratories, social, ethical, and professional issues, theoretical foundations, communications skills, design experience, and teamwork. It strongly advocated the integration of social and professional context of computing along with theory, abstraction, and design into the curriculum. The task force also identified twelve unifying and recurring concepts that are pervasive throughout the discipline.

In 1990s that accreditation agencies of engineering programs in some countries, mainly USA, UK, Australia, Canada, Singapore, and Japan, became explicitly concerned about desired educational outcome. USA's Accreditation Board for Engineering and Technology (ABET) played a stimulating role in this movement.

Goldweber et al [57] reviewed the previous curriculum related literature incorporating some educational literature. They classified the various pedagogical approaches into six different categories of viewing computing as (i) mathematics, (ii) engineering and design, (iii) art, (iv) science, (v) social science, and (vi) inter-disciplinary. They identified anthropology, applied psychology, computer science, cultural studies, economics, ergonomics, ethics, history, linguistics, management, mathematics, philology, philosophy, semiology, sociology, and politics as relevant disciplines. It criticized the Curriculum'91 for its coverage of social and professional context as an afterthought. This group considered the development of truly inter-disciplinary computing curriculum as the next challenge.

Software Engineering Perspective

Wassermann and Freeman [63] argued that computer science forms only a small portion of necessary education of a software engineer, and software engineering differed from other engineering that have their foundation in natural sciences. This was a novel observation that deserved more attention. This observation may have encouraged the subsequent committees to integrate more content about social and human sciences into mainstream computing courses, as was observed in some of the later recommendations. They considered a software engineer as a generalist, and drew an interesting analogy with a family physician who must have wide range of skills in addition to the core knowledge of medicines and diseases. They posited that a software engineering is an applied computer scientist, and the curriculum content must include problem solving, design, implementation, management, and communication skills. In addition to writing and speaking, the recommended communication skills included willingness to listen to others and sensitivity to the viewpoints and value systems of others. They also recommended the inclusion of accounting or economics or business administration, psychology, industrial engineering practices, and history or political science in the software engineering curriculum.

In his much debated talk called "On the cruelty of really teaching computing science," Dijkastra emphasized on formalism [64]. He declared *software engineering* as a self-contradictory doomed discipline. He called for banning the anthropomorphic metaphor in computer science, and insisted that programmer must also give formal proofs for the correctness of their programs. He advised that an introductory programming course should be taught as a formal mathematics course, and students should not be required to test their programs through implementation.

Certainly, mathematics education helps in developing some type of problem solving skills. However, by reducing computer science to formal mathematics, one of the founding fathers of computer science was under-estimating the huge growth of the software industry, and the important role software was to play in everyday life. In this debate, some supported him and others like Hamming, Parnas, Karp, Sherlis and Winograd criticized his 'extremism' and reminded that proofs are tedious and fallible, and engineering is not about optimality or perfection, it is reasonableness in terms of reliability, cost, time, and effort.

The serious shortfall of manpower and software crisis provided the necessary enabling conditions for the fast emergence of the 'doomed discipline' of software engineering as applied computer science that called for an engineering approach. The Software Engineering Institute (SEI) was founded in 1984 at the Carnegie Mellon University. This institute made significant contributions to the development of educational programs in software engineering. This was the start of some specialized programs in software engineering in USA, and also in Europe [65-66a].

In 1990, SEI presented a model curriculum for undergraduate engineering program in software engineering. As compared to ABET's accreditation criteria of engineering program, in this curriculum, the humanities and social sciences requirement was increased by reducing electives and mathematics and science components. Further, two ABET categories of engineering science and engineering design were merged into a single category of software engineering sciences and design. None of basic engineering science course was retained in this curriculum. In many ways, this curriculum was a reflection of a twelve year old proposal [63].

A new kind of engineering discipline was finally beginning to get its recognition, which claimed its foundations in the science of artificial constructs, mind, society, and engineering methods rather than material. This is a phenomenon that has been largely ignored by Indian engineering educators, even after so many decades. The curriculum recommendations categorized computing courses into four categories: (1) software analysis, (2) software architectures, (3) computer systems, and (4) software process. This indicated the signs of the beginning of integrated curriculum in computing.

In 1999, SEI-CMU published a report to define the discipline of Software Engineering [67]. The mathematics requirements included mathematical logic and proof systems, discrete mathematical structures, formal systems, combinatorics, and probability and statistics. Topics in numerical methods or calculus were not included. This report also included the computing topics of data structures and algorithms, computer architecture, operating systems, and programming languages. The software product engineering related areas were identified as software requirement, design, coding, testing, and operation and maintenance. Software management

areas encompassed management of process, risks, quality, configuration, process, and acquisition.

Based on a long industry-academia consultative process, SWEBOK [68] provided an excellent document that elaborates upon ten main knowledge areas under the categories of software requirements, software design, software construction, software testing, maintenance, software configuration management, software engineering management, software engineering process, software engineering tools and methods, and software quality. In a very sketchy manner, SWEBOK also elaborates upon the desirable topics of related disciplines of mathematics, computer science, computer engineering, management, project management, quality management, software ergonomics, and systems engineering.

For the first time in its history of nearly forty years, a computing curriculum recommendation made some reference to some education theories. SWEBOK elaborates upon technical competencies that software engineers with four years of experience should have. It identifies ten knowledge areas. Appendix D in their report suggest the desired level of competence as per Bloom's taxonomy to classify various knowledge areas with reference to ten knowledge areas of software requirements, design, construction, testing, maintenance, configuration management, engineering management, engineering process, tools and methods, and quality. This report is currently undergoing a revision exercise, and some more knowledge areas like software engineering economics are being considered for inclusion.

Deficient Educational Perspective Till the End of Last Century

The 1991 report of the ACM/IEEE-CS curriculum task force was seminal as it approached the issue with broader educational objectives and looked at the curriculum as a unified artifact. Leaving the former *fragmented* approach to curriculum design, this committee tried to create a *connected curriculum* [69]. However, this as well as all earlier mentioned curriculum recommendations related to computer science and engineering, appear to have over-sighted or ignored the simultaneously growing literature in educational research and curriculum design to theoretically ground their approach and broaden their perspective.

In the absence of such a theoretically grounded perspective of 'education,' the recommendations were highly skewed towards content and application with academic and technology orientation for curriculum design. These recommendations did not pay sufficient attention to other aspects of education that are better addressed through incorporation of complementary orientations for curriculum design. These orientations were cognitive process, society centered, and humanistic approach for curriculum design [70]. Scragg et al [71] called for developing insight based curriculum through insight-building activities. They argued that computer science is a fundamentally creative endeavor, and expressed concern at the lack of appropriate vocabulary in computer science curriculum.

Gersting and Young [72] in their paper "Content + Experience = Curriculum" proposed experiential aspect of computer science curriculum to complement the content part, and argued that providing and evaluating experiences is a major responsibility of the faculty. However, even they did not ground their proposal into educational theories. Meanwhile, Carson [73] argued that it is not its application, but effect on thinking that makes sciences relevant. He suggested that teaching within the discipline needs to be subordinated to the central task of teaching about the whole culture. He expressed concern at the substitution of liberal education's curriculum goals of humanism and citizenship with economic and political goals. Clarke and Reichgelt [74] examined the curriculum of sixty universities and colleges and found that most provided only a list of the courses, and a summary of the objectives.

Indian Approach

Recognizing the growth potential, Government of India sponsored Indian Society for Technical Education (ISTE) to propose the first model curriculum in this area. The ISTE interacted with academia, industry, and professional bodies like Computer Society of India (CSI) and Institution of Electronics and Telecommunication Engineers (IETE) and proposed a curriculum in 1987. The group over-sighted most of the important international up-to-date recommendations and manpower requirement projections with respect to computing education, especially with respect to information systems and software engineering. It nearly failed to foresee the tremendous growth of offshore and outsourcing software service industry that already existed even in the

1970s, started to take off in the mid 1980s, and was growing fast in the late 1980s and the early 1990s.

The model curriculum proposed by this committee and published by Rajaraman [75] did not make a mention of this growth or any up-to-date study related to manpower requirement. It only included an outdated report of 1980 on manpower requirement by the Indian Planning Commission. He did not mention any rational reasons or arguments for this retrogressive curriculum that did not find it suitable to put even a single computing course in the first year, and chose to put discrete mathematics in the fourth semester. The committee ignored the already well recognized developments in database management and software engineering. This paper also did not relate itself with the large body of educational research literature. Most surprisingly, none of the ACM or IEEE reports related to curriculum recommendations are included in the reference list. Instead only one UNESCO-IFIP [55] recommendation was included as a reference. However, possibly as an afterthought, for comparison purpose, Denning et al [61] was referred.

Rajaraman [75] distinguished the proposed Indian curriculum from the western model [61] as one with a bias towards electrical engineering. He did not respond well to the real demands and trends of the local or global industry. The growth of undergraduate computing education was slow till the early 1990s. Even in 1993, approximately 3000 students were completing their undergraduate engineering degree in this discipline. However, the growth of Indian education programs in this area has been phenomenal in the subsequent years, and this number has multiplied by more than fifty times in the last last fifteen years. Availability of low-cost desktop computers is the main contributing factor to this growth. It has fuelled the demand for more software, and hence trained manpower, especially in the software sector. The setting up of computational facilities in educational institutes became much cheaper. This phenomenon was largely over-sighted or under-estimated by the curriculum designers. Even today, the curriculum of many universities has not deviated much from the earlier model curriculum. Rajaraman's paper raised the issue of faculty shortage; the issue is much more serious today. Every year, more than 2,00,000 undergraduate students enter colleges to study computing courses. However, most of the required knowledge related to information systems and software engineering is picked up on the job.

The model curricula designed by AICTE, India [44-45] for undergraduate engineering programs in computer science and engineering and information technology totally ignore the integration and experiential aspects of curriculum design. Most carelessly, the curricula even failed to project basic working definitions of either of the disciplines. With reference to humanities and social studies courses, the committee seems to have totally succumbed to the short sighted economic goals. There is only one language/communication course in the first semester that can qualify as a non-management humanities course. All other humanities courses have been replaced by management courses. It seems that to the curricula have been designed without seriously examining any of the earlier recommendations of any of the educational research literature or even specific curriculum related recommendations of international professional bodies, like the ACM, IEEE, or IFIP.

Section 1.3: Research Approach

Community and culture significantly influence value orientation, perceived needs, and motivation as well as provide the ground for creating shared understanding. All disciplines have their own cultures, and all cultures evolve through cross-cultural exchanges. The computing community has created and documented a sound body of knowledge of software engineering [68]. It is one of finest examples of multi-cultural synthesis of many disciplines especially engineering, computer science, and even social sciences. In the last decade, the disciplines of design and aesthetics are also providing very interesting enrichment opportunities for this body of knowledge. With the very large scale worldwide endeavor on computing or software engineering education, it is now time to leverage education and 'learning' related research to create and document a theoretically sound body of knowledge of software developers' education. Such a body of knowledge should naturally require us to synthesis the evolving disciplines of software engineering and higher education.

The phenomenon of 'learning' has been extensively studied by psychologists, educationists, sociologists, philosophers, engineering educators, and even computer scientists working in artificial intelligence and e-learning. Computing educators take very important curricular and educational decisions without referring the rich theories of curriculum design or education. This

oversight is analogous to the misconception that "software engineering = programming" which just requires knowledge of some programming language.

In late 1980's, engineering methods had to be combined with the elements of computer science to create large scale software systems. Similarly, now with the exponential growth of education in computing disciplines, the scale of the impact of the computing faculty's decisions is far reaching. The computing student community is no more limited to highly gifted few any more. The scale of computing faculty's educational responsibilities is continuously expanding. Quality of software development education is an important issue that needs to be urgently addressed. Hence, there is an urgent need to enrich the culture of software development education with the help of educational research. For sustaining this unprecedented expanding scale of computing education, we now need theoretically sound educational frameworks. More so because of severe shortage of experienced faculty, especially in countries like India where this expansion has been exponential, resulting in quality difference between the best and worst programs to be even more than an order of magnitude.

The published research in computing education or software engineering education does not sufficiently leverage this research in education. In the various curriculum reports of 1960s to 1980s by the ACM as well as IEEE, there is no reference to educational models or theories. Even in the 1990s, we find few such attempts. In the absence of such references, it is not surprising that the curriculum committees limited their goal to cataloguing various content areas and describing and sequencing the required courses, resulting in a fragmented curriculum. They did not attempt to argue or propose curriculum models for holistic education of computing professionals.

An attempt of this type may have encouraged the curriculum designers and educators to create an integrated curriculum, as was happening in some other disciplines. Aning et al [76] have observed that in general, engineering faculty is not aware of cognitive science research that has potential to improve engineering pedagogy and mention about recent efforts by NSF to bring together engineering and education faculty. It is not surprising that the computing curriculum

designers not only ignored the pure education research, but also applied educational research such as science education.

Subsequently, the trend started changing, and some authors at annual computing education conferences like the ACM SIGCSE, ACM SIGITE, ASEE-IEEE FIE, etc., started examining, reviewing, and/or using some well established theoretical models and frameworks like Bloom's taxonomy and Kolb's experiential learning. However, the papers presented at IEEE CSEE&T show a very poor record of leveraging even such highly popular theories. Interestingly, some learning theories have also been used by the HCI, Information systems, and multimedia communities for guiding their design objectives and processes. A large number of papers in the ACM SIGCSE, ACM SIGITE, IEEE CSEE&T, or IEEE Transaction of Education are like experience reports, and do not make a good attempt to theoretically ground their work in educational research. However, many other streams of higher education, including engineering and science education, have leveraged educational research to enrich their research.

A meta-analysis [77] of computer science education research posits that the majority of the work done in the past has been done by computer scientists reflecting on their own teaching practice. These authors stress that there is a need for more dedicated researchers in computer science education. They observe that in more established educational research, like science education research, the studies carried out are not limited to researchers' own teaching practices so much as on other teachers' practices. Not many such studies have been reported in computer science and engineering education. The research method developed and used in this research is an attempt to fill this gap.

The data collection and analysis goals have gone much beyond the boundaries of the courses taught by the researcher. An attempt has been made to integrate the techniques of qualitative as well quantitative research methods to take the advantages of both. Research processes included a wide-ranging survey of published literature in diverse areas of Software development, computer science and IT education, engineering education, professional and higher education, learning theories, instruction design, and human development. Research also included study of a large number of comments written by professional software developers about contemporary issues

related to software development processes, required competencies, endorsements, etc., in various professional forums. More than three hundred professionals of more than sixty organizations from various countries have been consulted and/or surveyed on various issues. More than one thousand undergraduate computing students, and more than one hundred faculty members, have also been surveyed on selected issues.

This dissertation is concerned with understanding and suggesting ways to expand the context of software development education with the help of existing theories on 'learning', epistemology, human development, education, and instruction by applying analytical, qualitative, and quantitative methods to investigate the following types of questions:

1. How has software development education evolved, specifically with reference to educational research?
2. What is meant by competent and professionally oriented computing engineers, especially with respect to software engineering? What are the essential attributes? What is the relative importance of these attributes?
3. What is the degree with which the various components of traditional processes of engineering education succeed in creating opportunities for enhancing these competencies? What students think about their educational experiences? What students think works well for them? What processes do professional engineers recommend?
4. What pedagogical practices succeed in developing competencies, and under what circumstances? What comes in the way of implementing these strategies? What kinds of lectures are effective for 'learning' in the views of students and faculty? What factors block students from effective 'learning'? How to overcome these difficulties?
5. What kind of instructional interventions are required? How can the existing education theories/strategies/methodologies be used to educate competent computing engineers? Do we need new theories of 'learning' for software development education? If so, what would be main aspects of such a theory of 'learning'?

In this thesis, we propose a comprehensive framework of pedagogic engagement in computing courses for developing multi-dimensional competencies with respect to the requirements of software development. We have fairly comprehensively examined the published record of major developments and ideas in the history of evolution of computing curriculum since the 1950s. Further, we have identified the distinguishing characteristics of software development. We referred to published literature, and also carried out several exploratory surveys and polls among software developers to understand the profession from their perspectives. We take a position that software development is not an extension of any single discipline.

With respect to the needs of this distinguished profession, we have studied and collated the published recommendations by several accreditation boards, professional bodies, and researchers. We have also carried out several surveys among working professionals to understand their perspectives about the required competencies that must be emphasized by the educational process of software developers.

Based on these studies and surveys, we have identified *twelve core competencies* for software developers from various approaches, and organize these in the form of a three-tier taxonomy. We then elaborate upon the context and meaning of each of the twelve core competencies in the light of various multi-disciplinary theories and findings, and also our own reflections, empirical results, and interpretations.

During the course of this study, we have studied a large number of theories of education, 'learning', intelligence, human development, curriculum design, and thinking. Tables A'1.1a and A'1.1b in Annexure AN1 list some of these important theories and modes. We have selected some of these, and used them for designing our generic framework of pedagogic engagements as well as specific interventions for instructional reform in software development education.

Our proposed framework of pedagogic engagements in software development education includes (i) core activities of software development, (ii) distinguishing characteristics of software development profession, (iii) three-tier taxonomy of twelve core competencies, (iv) five-dimensional ladder of professional and human

development, (v) three-dimensional perspective of the knowledge domain of software development, (vi) two core principles for facilitating deep learning, and (vii) a four-dimensional taxonomy of pedagogic engagements over (v).

Finally, as exemplar case studies, we also elaborate upon some instructional interventions designed and administered by us in some chosen set of computing courses. These interventions are manifestations of some aspects of our proposed framework of pedagogic engagements for software development education. Some new courses have also been developed in the process.

Investigations related to curricular aspects like specific programming languages, methodologies, or formalism are not included within the scope of this work. We believe that the proposed framework is fairly comprehensive, reusable, and robust. It can be used to design many more interventions in software development education. Designers of educational programs for other professions can also use this framework and methodology.

Section 1.4: Thesis Layout

The first chapter of the thesis gives an overview of the motivation, objective, background, research method, and results of the reported work. In addition, we also discuss the evolution of computing curriculum in the last five decades.

In the second chapter, the required core competencies for software developers are explored with the help of published recommendations of accreditation agencies, professional societies, and published research. Fresh survey has also been carried out for this investigation. These competencies are then consolidated into a three-dimensional taxonomy. More literature is explored to consolidate the competency requirements of the software services and software product companies.

The third chapter analyzes the distinguishing features and multidimensional aspects of software development with a view to further analyze the required competencies. In this process, a large number of software professionals were consulted on various issues related to software development and required educational inputs. The three-dimensional taxonomy of competencies

proposed in the second chapter is distilled and revised into a three-tier taxonomy of twelve competencies.

In the fourth to sixth chapters, we discuss the meaning of the identified twelve competencies in the context of software development work. The basic competencies are discussed in fourth chapter. The competency driver-habits of mind are elaborated in fifth chapter and competency conditioning attitudes and perceptions are discussed in sixth chapter. We draw upon multi-disciplinary published literature and empirical studies in the process. Each of these chapters deals with a different category of competencies as per our taxonomy.

The seventh chapter gives an overview of various quantitative and qualitative surveys among computing students, software developers, and faculty of engineering institutes. We conducted these surveys to empirically investigate the phenomenon of 'learning' in computing/engineering disciplines. In this chapter, we essentially discuss the rationale for student-centric active learning.

The eighth chapter gives the most significant theoretical contribution in this work. We consolidate all our earlier findings discussed in the earlier chapters with the results of carefully chosen classical and contemporary 'learning' theories. These theories have been chosen with respect to their applicability for software development education. We propose a unified framework of pedagogic engagements in software development education. This framework focuses on development of required core competencies for software development as consolidated in the third chapter and discussed in the fourth, fifth, and sixth chapters.

Some aspects of this framework are manifested in some instructional interventions discussed in the ninth chapter. The tenth chapter provides a summary, and suggests future scope of research.

CHAPTER 2: IDENTIFICATION OF CORE COMPETENCIES FOR SOFTWARE ENGINEERS

Education programs seek to develop certain generic and discipline specific competencies of students. Educationists, accreditation agencies, professional societies, as well as forums of industry often engage in discourse about the essential and desired competencies as outcomes of education programs. Passow [78] has interpreted the competencies to mean the skills, abilities, knowledge, attitudes, and other characteristics that enable a person to perform skillfully (i.e., to make sound decisions and take effective action), in complex and uncertain situations such as professional work, civic engagement, and personal life. Further, she has viewed expertise as the proficient coordination of multiple competencies that leads to consistently effective performance in a variety of complex, unique, and uncertain situations.

Section 2.1: Study Report on Core Competencies for Engineers with Specific Reference to Software Engineering

We first discuss the various studies related to the core competencies required for general engineering graduates, and come up with the set of *general engineering competencies* normally accepted among the researchers [78a]. With this set of competencies as a starting point, we did an extensive survey among software engineering practitioners, to find out which *subset of engineering competencies are more important for the software engineering graduates.*

Bordogna [79] quotes an NSF report (published in 1989) which identified *integration, analysis, innovation and synthesis,* and *contextual understanding* as key capabilities for engineering students. He also posits that the essence of engineering is the process of integrating different forms of knowledge to some purpose, and an engineering student must experience the 'functional core of engineering'- the excitement of facing an open-ended challenge and creating something that has never been. He proposes that a *21st century engineer must have the capacity* to:

i. design, in order to meet safety, reliability, environmental, cost, operational, and maintenance objectives,
ii. realize products,

iii. create, operate, and sustain complex systems,
iv. understand the physical constructs and the economic, industrial, social, political, and international context within which engineering is practiced,
v. understand and participate in the process of research, and
vi. gain the intellectual skills needed for lifelong learning.

Dodridge [80] classifies the attributes of engineers into two broad categories of (i) *knowledge and understanding* and (ii) *skills*. Dodridge (2003) as well as Mason [81] refer to a 1998 survey by the EMTA (Engineering and Marine Training Authority) that identified *practical skills, multiskilling, computer literacy, communication skills, management skills, personal skills,* and *problem solving skills* as the most important skill deficiencies among engineers. Hoscette [82] and Erlendsson [83] have identified some workplace defects and leading causes of failures in engineering. As per their observation, the major concerns are *passivity*, *non-responsiveness, uncritical thinking, technical incompetence, inept or poor communication skills, poor relations with the supervisor, inflexibility, poor and lax working habits,* and *too much independence.*

Successful Practices in International Engineering Education (SPINE) is a benchmark study [78a] focusing on the analysis of successful practices in engineering education in ten leading European and U.S. universities including MIT, CMU, and ETH, Zurich. The study attempted to measure the perceived importance and assessment of fifty-one parameters on *quality of education, teaching methods, engineering competencies, general professional skills,* and *aspects of reputation of institute* through a quantitative analysis. In the SPINE project, *543 professors of these universities, 1372 engineers and 145 managers* of European and US companies were questioned. A summary of their findings is given at Annexure AN11.

We administered a survey among Indian engineers and managers working in Indian and multinational IT companies to obtain their perceptions on the importance of *forty-nine parameters of engineering education*. For the purpose of our first empirical study [84] conducted in 2004-06, we added two additional general professional competencies: (i) *awareness of environmental issues,* and *(ii) sensitivity towards socio-economic aspects for sustainable technological development.*

The abovementioned twenty-three competencies were included in this list. Other parameters on *teaching methods*, *quality of education*, and *aspects of reputation of institutes* were the same as in the SPINE survey. The results of this survey with reference to the teaching methods are discussed in Chapter seven. The other two set of surveyed parameters are not included in this thesis. Respondents were requested to assign numeric ratings to these parameters on a scale of 0 to 10, with 10 being the highest importance in terms of the parameter's criticality and potential contribution in preparing students for a successful professional career.

Fifty-four experts working in fifteen companies responded. The responding experts had industrial experience ranging from 1.5 years to 35 years with an average experience of 7.5 years, which is inferred to be slightly higher than the industry average, given the average age of employees in the Indian IT industry is only 27-30 years [5]. The Collection of these responses was spread over a period of approximately one year from 2003 to 2004. Table 2.1 provides a brief summary of the survey results about the importance of competencies. More details are provided in Appendix A1.

Table 2.1: Most important engineering and general professional competencies, as rated by Indian engineers and managers working in Indian and multi-national IT companies (2004)

No	Competency	Category
1	Problem solving	Pivotal
2	Analysis/Methodological skills	Critical
3	Basic engineering proficiency	Critical
4	Development know-how	Critical
5	Teamwork skills	Critical
6	English language skills	Critical
7	Presentation skills	Critical
8	Practical engineering experience	Critical
9	Leadership skills	Critical
10	Communication skills	Critical

Problem solving skill was also identified as the most important competency by the responding engineers in the SPINE project [78a], as well as a University of Arkansas study [85]. Problem

solving is the ability to identify and solve problems, when and where they occur. Domelen [86] quotes Steward (1982), "... all problem solving is based on two types of knowledge: knowledge of problem-solving strategies, and conceptual knowledge." Gary [87] argues that curriculum should provide opportunities for transforming a problem statement into a model, conjecturing solutions, selecting or developing the appropriate mathematics, examining the analysis, and continuing to transform the conjecture into a solution. Bruner [88] proposed that preparing students for solving real-life problems require a different paradigm of education and learning skills, including self-directed learning, active collaboration, and consideration of multiple perspectives. Problems of this nature do not have "right" answers, and the knowledge to understand and resolve them is changing rapidly, thus requiring an ongoing and evolutionary approach to 'learning'.

The findings of this study, based on the ratings assigned by Indian engineers and managers working in the Indian and multinational IT companies, as summarized in Table 2.1, are highly compatible with the findings of the SPINE project, which examined the requirements for Europe and the USA in a more general context of the engineering industry. However, importance of *development know-how*, *practical engineering experience, research know-how,* and *specialized engineering proficiency* have been rated at a *higher level by the respondents of the current study*, *as compared to the respondents of the SPINE project*. We can explain this difference by examining the nature of the Indian IT industry. This difference may perhaps be partially attributed to the fast obsolescence in the IT industry. Further, the Indian IT industry is mainly a "service industry." Many companies want to have "industry ready" engineers. Often some companies mention some specific IT skills like the ability to program in specific computer languages, and the use of development tools as recruitment criteria for fresh engineers.

Interestingly, the *importance of other language skills has been rated very low* as compared to the SPINE rating. As Indian IT companies begin to play a larger role in non-English speaking countries, this is likely to change marginally. Some companies have already started recommending potential recruits to acquire skills in languages like Japanese. There are some noticeable differences with respect to the NASSCOM-KPMG and also Indian Task Force reports

[89] that classified *spoken English, team-working, initiative/enthusiasm,* and *motivation/drive* as desirable skills rather than necessary skills.

Two competencies not examined by the SPINE project, and introduced in this study, were *awareness of environmental issues* and *sensitivity towards socio-economic aspects for sustainable technological development*. The first among these has come out as 'obligatory' while the second has been rated as a *desirable* competency.

Hence, we find conclude that the identified core competencies for general engineering graduates were also required by software engineers, but there were major difference in their order of importance.

Section 2.2: Necessary Competencies as Educational Outcomes for Software Engineers as Recommended by Accreditation Boards, Professional Societies' and Other Approaches

Curriculum content is no longer the key as the accreditation agencies in many countries have transformed their accreditation criteria and standards in terms of core competencies. A major shift has taken place from input-based criteria to outcome-based approach. NAE in their vision report for 2020 [15] recommends that engineering schools should vigorously exploit the flexibility inherent in the outcome-based accreditation approach to experiment with novel models for baccalaureate education. Subsequently, we carried out an extensive study of the recommended outcomes by accreditation boards of some countries. We examined the recommended outcomes by Accreditation Board for Engineering and Technology (ABET) (United States) [90-92], United Kingdom Standards for Professional Engineering Competence (UK-SPEC) [93], Institution of Engineers, Singapore (IES) [94], Engineers Australia Accreditation Board [95], and Japan Accreditation Board for Engineering Education (JABEE) [96]. These are summarized in Annexure AN1.

There are great similarities in the competency set identified by the accreditation agencies of the US, UK, Australia, Japan, and Singapore. Nine out of the eleven competencies identified by the ABET, US continue to reappear with some modifications in the competency list prescribed by accreditation agencies of all of these countries. However, some agencies have broadened the

scope of some of these competencies to be more comprehensive. For example, the JABEE has broadened *ability to work in multi-disciplinary teams* into *ability and intellectual foundation for considering issues from a global and multi-lateral viewpoint,* and also has put it at the first position of their list. We considered these competency lists to be ordered on importance as perceived by respective agency. While there are many similarities in the order proposed by these agencies, the JABEE has ordered their list differently. It gives highest importance to *the ability and intellectual foundation for considering issues from a global and multi-lateral viewpoint* and *understanding of the effects and impact of technology on society and nature, and of engineers' social responsibilities (engineering ethics).* Table 2.2 gives a summarized and composite view of some of the most commonly distinguished and identified competencies by the ABET, UK-SPEC, EA, JABEE, and IES. *We use these results to further expand and refine our initial set of competencies (Annexure A1) for further investigations.*

Table 2.2: Comparative analysis of some common competencies distinguished and identified by some accreditation agencies

S.No	Competency	Position in the respective list					
		ABET EC2000	UK-SPEC	IES	EA	JABEE	Average position
1	Ability to apply knowledge	1	2	1	1	3	1.6
2	Design skills	3	2	3	5	5	3.6
3	Problem solving skills	5	--	4	4	4	4.25
4	Technical competence	11	1	5	3	4	4.8
5	Ability to work in multi-disciplinary teams	4	4	9	6	1	4.8
6	Sensitivity towards ethical and professional issues	6	5	10	9	2	6.4
7	Communication skills	7	4	6	2	6	5
8	Sensitivity towards global, societal, and environmental issues	8	5	8	7	2	6
9	Readiness for life-long learning	9	5	7	10	7	7.6

Section 2.2.1: Impact on Curriculum and Future Directions

The recommendations of the various accreditation agencies in the US, UK, Singapore, Australia, and Japan have already affected educational programs, not only in their respective countries, but also in other countries. Many universities have redefined their program objectives, delivery mechanism, and assessment systems to incorporate graduate attributes in teaching programs [97-97a]. For example, as per National Academy of Engineers (NAE) report, Olin College of Engineering [15] has identified the following characteristics for their graduates:

a. Superb command of engineering fundamentals.
b. Broad perspective on the role of engineering in society.
c. Creativity to envision new solutions to problems.
d. Entrepreneurial skills to bring these visions to reality.

Macro level reforms are being realized through micro level redesigning of every course with a focus on fostering specific competencies [8]. Curriculum now gives more emphasis on design, practice, collaborative learning, humanities, social sciences, and sustainable engineering [98]. Faculty development programs have been organized to help them understand the underlying pedagogical issues [99]. Learning theories and epistemological frameworks are being used to shift the focus of teaching, learning, and assessment processes on competency development [100-100a].

Section 2.2.2: Indian Scenario

One of the nine Indian inventors included in the list of top 100 inventors under 35, Vikram Sheel Kumar, thinks that the biggest challenge an Indian student faces is *finding the space to develop an independent mind* [101]. Some of the senior industry managers in some industrial sectors feel concerned about the lack of *positive attitude, behavioral aspects, ability to cope up with challenges, sincerity, integrity, ethics, self-analysis, discipline,* and *independent thinking* among fresh engineering graduates [102]. It is very ironic that while '*availability of highly skilled manpower' has been identified as the most important factor* that is driving the increasing momentum of R&D off-shoring/outsourcing industry in India; *'quality of higher education' has been identified as one of the main inhibitors* [103].

The accreditation criteria defined by the National Board of Accreditation (NBA) of the All India Council of Technical Education (AICTE) [104], has not yet responded to the abovementioned contemporary models that emphasize carefully identified attributes and competencies based on national and global needs. One of the major objectives of NBA is to encourage the institutions to continually strive towards the attainment of excellence. The details of the parameters and their weights as prescribed by the NBA are given in Annexure A1 (Table AN1.1). This clearly shows the NBA is still silent about the core competencies, and continues to assess undergraduate and

postgraduate engineering programs with respect to several inputs rather than focusing and encouraging the institutes to develop a set of carefully identified competencies.

Section 2.3: Some other Contemporary Recommendations About Desired Competencies of Engineering Graduates

According to the Engineering Professors Council (EPC), United Kingdom, the key skills for engineering are communication skills, general IT user abilities, application of numbers, working with others, problem solving, and improving own learning and performance. It also identified the following primary competencies for engineers [105]:

a. Transform existing systems into conceptual models
b. Transform conceptual models into determinable models.
c. Use determinable models to obtain system specifications.
d. Select optimum specifications and create physical models.
e. Apply the results from physical models to create real target systems.
f. Critically review real target systems and personal performance.

The National Academy of Engineers (NAE) suggests that the essence of engineering—the iterative process of designing, predicting performance, building, and testing—should be taught from the earliest stages of the curriculum, including the first year [15]. Further, the NAE [106] has identified the following attributes for engineers of 2020:

a. Strong analytical skills.
b. Practical ingenuity: skill in planning, combining, and adapting.
c. Creativity (invention, innovation, thinking outside the box, art).
d. Communication.
e. Business and management.
f. Leadership.
g. High ethical standards and professionalism.
h. Dynamism, agility, resilience, and flexibility.
i. Lifelong learners.

Rugarcia et al [107] proposed the following categories of necessary skills for engineers:

a. Independent, interdependent and lifetime learning skills,
b. Problem solving, critical and creative thinking skills,
c. Interpersonal and teamwork skills,
d. Communication skills,
e. Self-assessment,
f. Integrative and global thinking skills, and
g. Change-management skills.

Cabrera et al [108] classified the professional competencies for engineers into three main categories of *group skills, problem solving skills and professional awareness*. The *group skills* include developing ways to resolve conflict and reach agreement, being aware of the feelings of members in group, listening to ideas of others with open mind, working on collaborative projects as member of a team. The *problem solving skills* encompass ability to do design, solve an unstructured problem, identify knowledge, resources, and people to solve problem, evaluate arguments and evidence of competing alternatives, apply an abstract concept or idea to a real problem, divide problems into manageable components, clearly describe a problem orally, clearly describe a problem in writing, develop several methods to solve unstructured problems, identify tasks needed to solve an unstructured problem, visualize what the product of a design project would look, weigh the pros and cons of possible solutions to a problem. The third category of *professional awareness* comprises of an understanding about what engineers do, the language of design, the non-technical side of engineering, and the process of design.

Passow [78] collated some of the earlier research on competencies and expertise in the context of engineering education. She cites Stark et al [110] who surveyed faculty members of nearly 400 universities to find the faculty's perception of adequate emphasis in different professions, and found that the engineering faculty viewed *conceptual competency*, as the most important competency closely followed by *integrative competency* (melding multiple competences to make informed judgments), and *communication competency*. *Professional ethics, technical competence, motivation for continued learning, career marketability, and contextual competence* (examining the context from a variety of view points) further expanded this list. This study showed that *adaptive competence* (propensity of modify, alter, or change elements of professional practice), *professional identity, and scholarly concern for improvement* were also viewed as reasonably important by responding faculty members.

She has carried out a meta-analysis of twelve empirical studies that had collectively surveyed more than ten thousand engineering graduates about the importance rating of competencies. She has classified the competencies in three groups of *top, intermediate, and bottom clusters*. The *top cluster* includes *problem solving, communication,* and *data analysis*. The *intermediate cluster* includes *ethics, life-long learning, teamwork, engineering tools, design,* and *math, science, and*

engineering knowledge. The *bottom cluster* comprises of *contemporary issues, experiments,* and *understanding the impact of one's work.*

Further, Passow's meta-analysis showed that in addition to the competencies identified by ABET, *decision-making, commitment to achieving goals, the ability to integrate theory and prac*tice effectively in work settings, leadership skills, and *project management* are also extremely important competencies. This study also concluded that respondents from computer science, computer engineering, and software engineering background rated *design* and *engineering tools* at a relatively higher level as compared to other engineering disciplines.

We use these recommendations and results to further expand and refine our initial set of competencies (Annexure A1) for further investigations.

Section 2.4: Recommendations of Some International Professional Societies Related to Computing

Recommendations for Computer Science

The Joint Task Force on computing curricula of the IEEE Computer Society and the ACM has published several reports related to computing curricula. These reports make clear recommendations on this issue with reference to specific undergraduate programs in computer science, software engineering, computer engineering, and information technology. The final draft on computing curricula, 2001, suggested the following broad level characteristics of computer science graduates [1]:

a. Systems-level perspective.
b. Appreciation of the interplay between theory and practice.
c. Familiarity with common themes.
d. Significant project experience.
e. Adaptability.

This report also suggested the following general skills for computer science graduates:

a. Communication.
b. Teamwork.
c. Numeracy.
d. Self-management.
e. Professional development.

Recommendations for Software Engineering

In 2004, the same task force made specific recommendations about undergraduate degree programs in software engineering [52]. It suggested that graduates of an undergraduate software engineering program must be able to:

a. show mastery of the software engineering knowledge and skills, and professional issues necessary to begin practice as a software engineer,
b. work as an individual and as part of a team to develop and deliver quality software artifacts,
c. reconcile conflicting project objectives, finding acceptable compromises within limitations of cost, time, knowledge, existing systems, and organizations,
d. design appropriate solutions in one or more application domains using software engineering approaches that integrate ethical, social, legal, and economic concerns,
e. demonstrate an understanding of and apply current theories, models, and techniques that provide a basis for problem identification and analysis, software design, development, implementation, verification, and documentation,
f. demonstrate an understanding and appreciation for the importance of negotiation, effective work habits, leadership, and good communication with stakeholders in a typical software development environment, and
g. learn new models, techniques, and technologies as they emerge and appreciate the necessity of such continuing professional development.

Recommendations for Computer Engineering

In their final report 'Curriculum guidelines for undergraduate degree programs in computer engineering' [111], the task force identified the following characteristics for computer engineering graduates:

a. System Level Perspective.
b. Depth and Breadth (of knowledge).
c. Design Experience.
d. Use of Tools.
e. Professional Practice.
f. Communication Skills.

Recommendations for Information Technology

In April 2005, the same task force also proposed a draft computing curricula for information technology. This report suggested [112] that pervasive themes for IT program outcome should be *user centeredness and advocacy, information assurance and security, the ability to manage complexity, a deep understanding of information and communication technologies and their associated tools, adaptability, professionalism,* and *interpersonal skills*. This report also recommends that an IT graduate must acquire the ability to:

a. use and apply current technical concepts and practices in the core information technologies,

b. analyze, identify, and define the requirements that must be satisfied to address problems or opportunities faced by organizations or individuals,
c. design effective and usable IT-based solutions and integrate them into the user environment,
d. assist in the creation of an effective project plan,
e. identify and evaluate current and emerging technologies and assess their applicability to address the users' needs,
f. analyze the impact of technology on individuals, organizations and society, including ethical, legal and policy issues,
g. demonstrate an understanding of best practices and standards and their application,
h. demonstrate independent critical thinking and problem solving skills,
i. collaborate in teams to accomplish a common goal by integrating personal initiative and group cooperation,
j. communicate effectively and efficiently with clients, users and peers both verbally and in writing, using appropriate terminology, and
k. recognize the need for continued learning throughout their career.

Recommendations for Information Systems

In 2004, the ACM, Association for Information Systems (AIS), and Association of Information Technology Professionals (AITP) published a joint report on 'Model curriculum and guidelines for undergraduate degree programs in information systems,' and characterized this discipline as 'Technology-enabled Business Development.' They have divided the representative capabilities and knowledge expected for Information System graduates into the following categories [113]:

a. Analytical and critical thinking: organizational problem solving, ethics and professionalism, and creativity.
b. Business fundamentals.
c. Interpersonal, communication, and team skills.
d. Technology.

Indian Recommendations

NASSCOM-KPMG [5] and the Government of India Task Force [114] identify *written English, logical reasoning, problem solving and numerical ability, programming skills, listening/empathy, assertiveness and confidence, integrity, values and discipline, sociability, dependability,* and *reliability* as necessary skills for IT professionals. These reports identify *spoken English, foreign language, accent understanding, comprehension/creativity, initiative/enthusiasm, team-working, multitasking and time management,* and *motivation/drive* as desirable skills.

It may be noted that the recommendation of NASSCOM as well as that of Government of India Task Force are more influenced by the over emphasized requirements of software service

industry, as shown later in Table 2.4. It sadly ignores the requirements of product development related work in small or large companies.

Hence, we conclude that, *not only Indian engineering education accreditation agency, the AICTE (ref: Section 2.2.2), but also the premier trade body and the chamber of commerce of Indian IT industry, NASSCOM, and also the task force created by the central government's ministry of communication and information technology, have also not yet shown futuristic directions in this regard. The mammoth growth of IT education in India has and continues to take place in an eco-system that is conditioned by serious absence of futuristic vision in the apex institutions.*

We use the recommendations discussed in this section to further expand and refine our initial set of competencies (Annexure A1) for further investigations.

Section 2.5: Some Contemporary Recommendations on Desired Competencies of Software Developers

The US based Professional Aptitude Council (PAC) conducts a pre-employment aptitude examination for IT professionals. This has also been recently launched in India [115]. This examination consists of questions on nine parameters of *problem solving, linear logic, mathematical ability, technical knowledge, applied technical skills, coding skills, creativity, work style,* and *personality composite*. It identifies *attention to detail, interpersonal skills, adaptability/flexibility, persistence, sense of urgency*, and *creativity* as IT related personality constructs. *Listening, adaptability to new technology, time management, visualize/conceptualize, multi-tasking, business culture, "be the customer" mentality, constructive criticism, organizational skills, stress management, idea initiation,* and *project management* are also highly valued skills in the IT industry [85]. Chang [116] and Erlendsson [117] suggest additional competencies like *knowing how to learn rapidly, ability to advocate and influence (persuasion), mentoring, decision making*, and *ability to manage complexity.*

Kelley and Caplan [118] carried out a comparative study of *star and average performers at Bell Labs*, which showed that *taking initiative* was ranked as the most important strategy by star

performers, while it was least important for average performers. On the other hand, *ability to give good presentations* was a core strategy for average performers, while it was peripheral for the top engineers.

Turley and Bieman [119] studied the *competencies of software engineers in a Fortune 500 computing company*. They found *concern for reliability/quality, focus on user needs, algorithmic and structured thinking, pride in quality/productivity, emphasis on elegant and simple solutions, mastery of skills/techniques, help other, innovative, maintenance of "big picture" view, enjoy challenges, seek help from other, lack of ego, attention to detail, pro-active nature, team orientation, reuse, desire to improve things, perseverance,* and *strength of conviction* are more common competencies of these software engineers.

They further identified that the top 30% software engineers demonstrated significantly higher levels in competencies like *help others, pro-active role with management, strength of convictions, mastery of skills/techniques,* and *maintenance of "big picture" view.*

The exceptional software engineers in this study distinguished themselves in terms of their *result orientation* and *sense of mission*, whereas non-exceptional software engineers distinguished themselves in terms of higher *perseverance* and *methodological approach*.

They also cited and highlighted the following observations made in earlier behavior oriented software engineering research:

a. The development process was not linear: *designers operated simultaneously at various levels of abstraction and details*.
b. Experienced designers took the users view before proceeding to design. … high-rated systems analysts were more likely to work for a *productive relationship with the users* and specify more requirement than the low-rated analysts. They would reject more hypotheses, try several strategies, apply heuristics, set more goals, and look for analogies to prior problems.

Armour [120] suggested that software developers need *domain specific training*, learning to learn, and structuring mechanism of the representation form.

Connor et al [121] have identified *new and specific technical skill, computer literacy and IT skills, multi-skilling and greater flexibility, the ability to deal with change, an ability to continue learning, re-skilling,* and *the greater importance of personal and generic skills* as key themes in their assessment of skill trends.

eXtreme Programming (XP) principles, rules, and practices are based on five core values: communication, simplicity, feedback, courage, and respect [122]. Shore and Warden have further elaborated upon these values [123]. *Communication is aimed at giving the right information to right people when they can use it to its maximum advantage*. Simplicity means to be able to discard unnecessary things. Feedback is to learn the appropriate lessons at every possible opportunity. Courage is required to make the right decisions, even when they are difficult, and to tell the stakeholders when they need to hear it. Respect implies treating oneself and others with dignity, and to acknowledge expertise and mutual desire for success.

Hazzan and Tomakyo [124] highlight the importance of *mental habit of abstraction and the ability to make transitions between levels of abstraction* as an important skill for software developers. Further, relating software engineering to Schön's work on reflective thinking and professions [125], they also posit that *mental habit of reflection and the ability to move across the ladders of reflections* are closely associated with software engineering processes. Agile methods like eXtreme Programming draw their strength from the possibility of continuous improvement through reflection.

Sodiya et al [126] expanded *Goldberg's Big Five personality factors* by adding a sixth factor of cognitive ability, and collected the personality traits of nearly 500 software engineers working in different stages of software engineering: requirement engineering, system design, coding, testing/implementation, and delivery/maintenance in Nigeria.

Their findings showed that *agreeableness: the tendency to be compassionate and not antagonistic towards others, was a universal personality trait among high performing software engineers.* This tendency includes being pleasant, tolerant, tactful, helpful, trustworthy, respectful, sympathetic, and modest. The high performing software engineers further showed high levels of *cognitive ability of abstract thinking, analysis, concentration, and visualization*. The other common personality trait among this group was found to be conscientiousness: *the tendency to be self-disciplined, to be dutiful, achievement and competence oriented, thorough, consultative, and orderly*. *Openness to experience: the tendency to enjoy new intellectual experiences and ideas, imaginative, curious, and broadmindedness* was also found to be a common trait of high-performing software engineers, particularly involved in systems testing and integration, management of software process, and deliver/maintenance. *Extraversion: the tendency to seek stimulation and enjoy the company of others* was not found to be a common personality trait of high performing software engineers. Neurotocism: the tendency to experience unpleasant emotions relatively easily was found to be universally low among this high performing group.

Recommendations for Software Architects

Bass et al [127] have identified that in addition to the knowledge of architectural concepts, software engineering, design, programming, technologies and platforms, the following general competencies are important for software architects:

a. Communication skills: Oral and written communication skills, presentation and convincing skills, see and address multiple viewpoints, consulting skills, negotiations skills, understand and express complex topics, listening skills, approachable, and interviewing skills.
b. Interpersonal skills: Team player, diverse team environment, creative collaboration, consensus building, balanced participation, diplomatic, mentoring, conflict resolution, respects for people, committed to others success.
c. Leadership skills: decision making, initiative, innovative, self-motivated and directed, committed, dedicated, passionate, independent judgment, influential, ambitious, mentoring, coaching, training.
d. Workload management: work under pressure, time management, priority assessment, result oriented, estimation, ability to concurrently work well on multiple complex projects and systems.
e. Skills to excel in corporate environment: passion for quality, art of strategy, work under supervision and constraints, organizational and work flow skills, process oriented, entrepreneurial, assertive without being aggressive, open to constructive criticism.
f. Information handling: detail oriented while maintaining overall vision and focus, see the larger picture, good at working at an abstract level.
g. Personal qualities: credible, accountable, responsible, insightful, visionary, creative, perseverant, practical, confident, patient, empathetic, work ethics.

h. Skills for handling unknown and unexpected: tolerant to ambiguity, risk taking/management, problem solving, reasoning, analytical skills, adaptable, flexible, open mindedness, resilient, and compromising.
i. Learning: good grasping power, investigative, observation power, adept at using tools.
j. Domain knowledge.
k. Knowledge of industry's best practices and standards.
l. Knowledge of business practices.

We use the recommendations discussed in this section to further expand and refine our initial set of competencies (Annexure A1) for further investigations.

Section 2.6: A Perspective from the Professional Codes of Conduct, Ethics, and/or Practice

Many professions have established professional societies that continuously help and guide their members to understand their professions not only in terms of technical advancements, but also evolving understanding of their profession's context. Professional codes are often designed to motivate members of an association to behave in certain ways. Codes of ethics are 'aspirational,' because they often serve as mission statements for the profession, and thus can provide vision and objectives. Codes of conduct are oriented more toward the professional, and the professional's attitude and behavior. Codes of practice relate to operational activities within a profession. These codes also help them to face and handle professional dilemmas. Primarily, these codes are designed and used to inspire, guide, educate, and discipline the members. Codes 'sensitize' members of a profession to ethical issues and alert them to ethical aspects they otherwise might overlook. Codes inform the public about the nature and roles of the profession. Codes also enhance the profession in the eyes of the public. These codes of conduct, practice, and ethics are not static, and keep on evolving to respond to new challenges and understanding.

All professional societies related to engineering and computing have defined a code of ethics and/or professional practice. Professional societies like the ACM and IEEE also insist that the professional education programs must also educate students with these prescribed codes. The IEEE-ACM joint computing curricula task force on software engineering [52] takes the position, *"to help insure ethical and professional behavior, software engineering educators have an obligation to not only make their students familiar with the Code, but to also find ways for students to engage in discussion and activities that illustrate and illuminate the Code's eight*

principles, including common dilemmas facing professional engineers in typical employment situations." SWEBOK [68] includes the software ethics under the knowledge area of software quality.

We have examined the codes of conduct, ethics, and/or practice of following societies:

1. American Council of Engineering Companies, 1980
2. National Society of Professional Engineers (NSPE), 1993
3. The Institution of Engineers, Australia
4. American Association of Engineering Societies, 2000
5. American Society of Civil Engineers, 1996
6. American Society of Mechanical Engineers
7. American Institute of Chemical Engineers, 2003
8. IEEE (Institute of Electrical and Electronics Engineers), 1990
9. ACM (Association of Computing Machinery), 1993
10. Information Processing Society of Japan, 1996
11. ACM-IEEE Code for Software Engineers Ver 5.2, 2002

The ACM-IEEE Code for Software Engineers Ver 5.2 has eight clauses that address issues related to public, client and employer, product, judgment, management, profession, colleagues, and self. The codes of all the above mentioned societies including ACM-IEEE Code for Software Engineers Ver 5.2, have following common features:

1. The first and the most important recommendation in all these codes is that concerned professional shall fulfill their professional duties by holding paramount the *safety, health and welfare of the public*. Several clauses of ACM-IEEE Code for Software Engineers Ver 5.2 reflect this concern and objective. These are clause number 1 (1.01 to 1.08), 2 (2.07), and 4 (4.01).
2. The second very important commonly address issue in all these codes is the directive advising their members to *undertake technological tasks for others only if qualified* by training or experience, or after full disclosure of pertinent limitations. Several clauses of the ACM-IEEE Code for Software Engineers Ver 5.2 reflect this concerned and objective. These are clause number 2(2.01), 3(3.04), 4(4.02), 5(5.04), and 7(7.08).

3. The third uniformly occurring instruction to their members is to *act for each employer or client as faithful agents or trustees*. Clause no 2 (2.01 to 2.09) of ACM-IEEE Code for Software Engineers Ver 5.2 expresses this concern in several ways.
4. The fourth identical facet in all these codes is an advice to their members to *issue public statements only in an objective and truthful manner.* Clause no 1 (1.06) and 6 (6.07) of ACM-IEEE Code for Software Engineers Ver 5.2 are expressions of this desired virtue.
5. The fifth regular feature of all these codes is a guidance to *avoid improper solicitation of professional assignments.*
6. The sixth common element of these codes is the suggestion that the members shall continue to *develop relevant skill, knowledge, and expertise throughout their careers and shall actively assist and encourage those under their direction to do likewise.* Clause no 8 (8.01 to 8.06) of ACM-IEEE Code for Software Engineers Ver 5.2 are expressions of this desired trait of software professionals.
7. The seventh common aspect of these codes is about *promoting an ethical approach among colleagues*. Clause no. 5 (5.01 to 5.12) of ACM-IEEE Code for Software Engineers Ver 5.2 are expressions of this desired trait of software professionals.
8. The eighth regular tenet of these codes is guiding the members to *continuously strive for quality, excellence, and adherence to highest professional standards.*

We use the spirit of these recommendations to further expand and refine our initial set of competencies (Annexure A1) for further investigations.

Section 2.7: Classical and Contemporary Recommendations on Desired Competencies of Graduates

In the above sections, we notice a high emphasis on human and social related competencies that go much beyond the scope of technical competencies. Hence, in order to get a better insight into these aspects from the perspective of university education, in this section, we look at the classical as well as contemporary recommendations about university graduates in general. In the 1850s, a pioneer philosopher of modern higher education, John Henry Newman, wrote a seminal work 'The Idea of a University Defined and Illustrated' [128]. As part of this work, he included a

discourse on 'Knowledge Viewed In Relation To Professional Skill.' In this discourse, he insisted that

> ***University training aims at raising the intellectual tone of society**, at cultivating the public mind, at purifying the national taste, at supplying true principles to popular enthusiasm and fixed aims to popular aspiration, at giving enlargement and sobriety to the ideas of the age, at facilitating the exercise of political power, and refining the intercourse of private life. education should give the ability **to see things as they are**, to go right to the point, to disentangle a skein of thought, to detect what is sophistical, and to discard what is irrelevant … to fill any post with credit and to master any subject with facility, **to accommodate himself to others** … to throw himself into their state of mind, how to bring before them his own, how to influence them, how to come to an understanding with them, how to bear with them, … to be at home in any society … [to have] common ground with every class … [to know] when to speak and when to be silent … to ask a question pertinently … [to] be able to converse and gain a lesson seasonably ,,, [and to enjoy] the repose of a mind that lives in itself, while it lives in the world.*

Franklin Bobbitt posited that because of unpredictability of future roles, the curriculum should insist on general education and **developing individuals' intellect** rather than just aiming to train them for specific work. He also insisted that education must aim at developing a **respect for many of the classic authors of "great books**." These thoughts were also resonated in Robbins Report (1963) [129] that suggested that the purpose of higher education is not simply the "instruction of skills suitable to play a part in the general division of labour" and "the advancement of learning," but also, "**to promote the general powers of the mind … and transmit … a common culture and common standards of citizenship**." Martha Nussbaum [130] posited that the purpose of liberal education is to cultivate humanity (**world citizenship**), and she suggested that to achieve this goal, three capacities need to be cultivated. The first among these is capacity for **critical self-examination and critical thinking** about one's own culture and traditions through logical reasoning: consistency of reasoning, correctness of facts, and accuracy of judgment. The second capacity is to **see oneself as a human being who is bound to all humans** with ties of recognition and concern. The third capacity is for **narrative imagination**: the ability to empathize with others and to put oneself in another's place through imagination.

The American Association of College and University [131] has declared the following learning outcomes as essential for all college graduates:

a. **Knowledge of human cultures and the physical and natural world** by engagement with big questions, both contemporary and enduring

b. **Intellectual and practical skills:** Inquiry and analysis, critical and creative thinking, written and oral communication, quantitative literacy, information literacy, teamwork and problem solving
c. **Personal and social responsibility** through active involvement with diverse communities and real-world challenges: civic knowledge and engagement—local and global, Intercultural knowledge and competence, ethical reasoning and action, foundations and skills for lifelong learning
d. **Integrative learning** through the application of knowledge, skills, and responsibilities to new settings and complex problems

García-Aracil and Van der Velden [132] have proposed their competency classification based on six categories of organizational, methodological, participative, specialized, generic, and socio-emotional competencies. The organizational competencies incorporate working under pressure, accuracy, attention to detail, time management, working independently, and the power of concentration. The methodological competencies comprise of foreign language proficiency, computers skills, understanding social, organizational/technical systems, documenting ideas and information, problem-solving ability, analytical competencies, and learning abilities. The participative competencies encompass planning, coordinating and organizing, economic reasoning, negotiating, assertiveness, decisiveness, persistence leadership, as well as taking responsibilities and decisions. The fourth category of specialized competencies essentially means knowledge of field specific theories and methods. The fifth category of generic competencies include broad general knowledge, cross-disciplinary thinking/knowledge, critical thinking, documenting ideas and information, problem-solving ability, and written as well as oral communications skills. The final category of socio-emotional competencies incorporate reflective thinking, assessing one's own work, economic reasoning, working in a team, negotiating initiative, assertiveness, decisiveness, persistence, adaptability, leadership, getting personally involved, taking responsibilities, decisions, loyalty, integrity, tolerance, appreciating of different point of view.

Their study showed that the best paid jobs required high levels of participative and methodological competencies, the worst paid jobs emphasized on organizational competencies, and high specialized knowledge contribute to higher wages in some professions like medical science, mathematics (including computing), and engineering. They finally concluded that new emerging work situations require individuals with enhanced levels of participative, methodological, and socio-emotional competencies.

We use the recommendations discussed in this section to further expand and refine our initial set of competencies (Annexure A1) for further investigations.

Section 2.8: A Comprehensive Distilled View on Desired Competencies

We have consolidated the abovementioned competencies recommended by engineering accreditation boards, engineering and computing professional agencies (including the code of ethics), and various thinkers of higher education, engineering education, and computing education. These recommendations were have been made with reference to graduates, engineering graduates, and computing graduates. *Appendix A2 gives a comprehensive summary* of these competencies in an alphabetical order of competencies. The importance of so many competencies with reference to software developers education has not been empirically examined in the earlier ranking studies, e.g., SPINE [78a], Bailey and Stefaniak [85], and our own [84]

Section 2.9: Further Empirical Investigations on Required Core Competencies for Engineering Graduates with Reference to the Indian IT Industry

Our earlier SPINE based empirical study (Appendix A1) discussed above had its own limitations. It mainly suffered from two deficiencies: (i) The examined competencies were generic in nature that were applicable to all fields of engineering, and these were not grounded in the specific competency literature related to software development. (ii) The software industry was considered a homogeneous entity and did not distinguish between the product based small or large companies and/or large companies mainly involved in offering software services to their clients.

Hence, in 2007, we took another survey. Based on the findings of our first study and various published recommendations about the desired recommendations as proposed by accreditation boards, professional bodies, as well as researchers, *we significantly revised and expanded the list of surveyed competencies from twenty-three to thirty-five*. Table A3.1 in Appendix A3 maps the competencies of the old (Appendix A1, Table A1.1) and the new list.

Some important competencies, listed in Appendix A2, were still not distinguished in our empirical study, conducted in 2007 (Appendix A3). Some of the important competencies of Appendix A2 that were not examined in 2007 included - curiosity, domain competence, abstraction, algorithmic thinking, knowledge of physical and natural world and intercultural knowledge, reflection, self acceptance and self regulation, and workload management.

Seventy-one experts working in thirteen companies with additions like Accenture, Borland Software, SUN, and TCS responded. The responding experts had industrial experience ranging from 1 year to 22 years, with an average experience of 5.6 years. The data was analyzed in a similar manner to our earlier SPINE-based study. For classification of competencies we added another category at the top to distinguish the topmost recommendation and termed it as 'Existential.' Table 2.3 provides the summary of the 2007 results.

Table 2.3: Most important competencies as rated by Indian engineers and managers working in Indian and multi-national software companies (Revised Study 2007) (More details in Table A3.2, Appendix A3)

Category	S.No.	Competency (SNo as per Appendix A2)
Existential	1	Perseverance, commitment, and hard work (13)
	2	Ability to work in teams (1)
Pivotal	3	Ability to apply knowledge (2)
	4	Integrity and authenticity (25)
	5	Analytical skills (6)
	6	Accountability and responsibility (25)
	7	Technical competence (31)
	8	Problem solving skills (22 and 23)
Critical	9	Listening skills (1)
	10	Attention to detail (15)
	11	Project planning and management (24)
	12	Quality consciousness and pursuit of excellence (25)
	13	Critical thinking (26)
	14	Readiness for lifelong learning (9)
	15	Design skills (11)

Table 2.4 enumerates the important competencies of Table A3.2 (Appendix A3) that were rated with higher importance, differently for three different segments of software industry: (i) software services related work at large companies, (ii) product development related work at large or mid-size companies, and (iii) product development related work at small companies. In Section 2.11, we interpret the implications of these findings.

Table 2.4: The most important competencies for software development work related to software services and product development

Category	**Software services related work in large companies** (SNo as per Table A3.2, Appendix A3)	**Product development work in large/mid-size companies** (SNo as per Table A3.2, Appendix A3)	**Product development work in small companies** (SNo as per Table A3.2, Appendix A3)
Existential	Ability to work in teams (2)	Ability to work in teams (2)	Perseverance, commitment, and work (1)
		Ability to apply knowledge (3)	
Pivotal		Perseverance, commitment, hard work (1)	Accountability and responsibility (6)
			Ability to apply knowledge (3)
			Problem solving skills (8)
			Research skills (17)
Critical	Perseverance, commitment, and work (1)	Accountability and responsibility (6)	Attention to detail (10)
		Analytical skills (5)	Analytical skills (5)
		Problem solving skills (8)	Integrity and authenticity (4)
		Research skills (17)	Readiness for lifelong learning (14)
			Technical competence (7)
Obligatory	Listening skills (9)	Integrity and authenticity (4)	Quality consciousness and pursuit of excellence (12)
		Critical thinking (13)	Critical thinking (13)
		Design skills (15)	Design skills (15)
		Technical competence (7)	

Section 2.10: Classifying the Core Competencies for Software Developers

Using Marzano's Dimensions of Learning for Classifying the Competencies

Dimensions of Learning [138], is a comprehensive model of learning and learning process. It structures the various aspects of learning along the following dimensions: (1) attitudes and perceptions, (2) acquire and integrate knowledge, (3) extend and refine knowledge, (4) use knowledge meaningfully, and (5) productive habits of mind. As per this model, all learning takes place against the backdrop of learners' attitudes and perceptions and their use of productive habits of mind. Dimension 4 subsumes dimension 3, which in turn subsumes dimension 2. This means that when learners extend and refine knowledge, they continue to acquire knowledge, and when they use knowledge meaningfully, they are still acquiring and extending knowledge.

In 2006, we adapted this model to design a three-dimensional taxonomy of desired competencies. Dimensions 2, 3, and 4 represent different aspects of learning in three hierarchical levels. As there are no orthogonal relations among them, in this discourse, they are merged into one. The new merged dimension can be viewed as having *three internal hierarchical sub-levels*. We suggested that, in essence, there are only *three dimensions of learning*:

a. Dimension 1: Attitudes and Perceptions,
b. Dimension 2: Productive Habits of Mind, and
c. Dimension 3: Acquisition, Integration, Extension, and Meaningful Usage of Knowledge.

Learners' *attitudes and perceptions* about the purpose of learning, as well as roles of teacher, self, and peers determine their motivation, and very significantly influence depth and performance of their learning.

Productive habits of mind: critical thinking, creative thinking, and self-regulation facilitate their learning process.

Acquisition, Integration, Extension, and Meaningful Usage of Knowledge is directly manifested in the software developers' work. It includes competencies like technical competency, problem solving, and communication skills.

The core competencies (for software engineers) studied and identified by us till that time (starting with the set of competencies for general engineers) were mapped in these three dimensions of learning. We posited that attitudes and perceptions affect a professional's ability to practice. The most important element of education should be to develop required attitudes and perceptions. Under the conditions of the right attitudes and perceptions, professionals use their productive habits of mind to acquire and integrate knowledge. Attitudes, perceptions, and productive habits help them to extend, refine and use knowledge for meaningful tasks. The first version of our taxonomy was published in 2006 [139]. It is summarized in Table 2.5.

Table 2.5: Taxonomy of core competencies for software developers - ver. 1, 2006

Dimension 1 **Attitudes and perceptions** (S. No. as per Table A3.2, Appendix A3)	**Dimension 2** **Productive habits of mind** (S. No. as per Table A3.2, Appendix A3)	**Dimension 3** **Meaningful usage, extension, and acquisition of knowledge** (S. No. as per Table A3.2, Appendix A3)
1. Perseverance (1) 2. Sense of urgency and stress management (29) 3. Adaptability and ability to multi-task (18) 4. Ability to work in homogeneous, multi-disciplinary, multi-locational, and multicultural teams (2) 5. "Be the customer" mentality (19) 6. Listening (9) 7. Sensitivity towards global, societal, environmental, moral, ethical and professional issues and sustainability (34) 8. Systems-level perspective (including knowledge integration, consideration for multilateral viewpoint, and user-centeredness) (20) 9. Ability to assist others through mentoring and philanthropic donations (30) 10. Entrepreneurship (35) 11. Readiness for lifelong learning. (14)	12. Attention to detail (10) 13. Numerical ability (26) 14. Critical thinking (13) 15. Creativity and idea initiation (22)	16. Technical competence (7) 17. Ability to apply knowledge (3) 18. Analytical skills (5) 19. Design skills (15) 20. Decision making skills (21) 21. Problem solving skills (8) 22. Communication skills (16) 23. Organizational skills (23) 24. Project planning and management (11) 25. Persuasion skills (28) 26. Experimentation skills (25) 27. Constructive criticism (27) 28. Knowledge of contemporary issues (32) 29. Research skills (17) 30. Mentoring skills (24) 31. Wealth creation skills (31)

Additional Competencies

Four important competencies of Table A3.2 (Appendix A3), later identified by us, were not distinguished in this taxonomy. These were – *Integrity and authenticity* (No 4 in Table A3.2), *Accountability and responsibility* (No 6 in Table A3.2), *Quality consciousness and pursuit of excellence* (No 12 in Table A3.2), and *Cost consciousness* (No 33 in Table A3.2).

Some other very important competencies, listed in Appendix A2, were also not distinguished in this taxonomy. Some of main competencies of Appendix A2 that were not classified in 2006 included - curiosity, domain competence, abstraction, algorithmic thinking, knowledge of physical and natural world and intercultural knowledge, reflection, self acceptance and self regulation, and workload management.

In **Annexure AN3**, we briefly discuss the details of some *others models about classification of competencies*. These include Bloom's taxonomy of educational objectives [133], Anderson and Krathwohl modification of Bloom's taxonomy [134], Costa's *model of intellectual functioning* [135], Kennedy's four perspectives on professional expertise [136], The classification of college

graduate's competencies as proposed by Stark et al [137], Marzano's revised taxonomy [140], earlier *classifications* cited by García-Aracil and Van der Velden [132], and Kelly Coate [141] schema for curriculum design

In Section 3.11, we will discuss a revised version of our taxonomy of competencies.

Section 2.11: Chapter Conclusion

The overall findings of the revised study, as summarized in Table 2.3 and Appendix A3, gave new insights into the importance of desired competencies in software industry. The respondents gave highest importance rating to many newly added competencies that related to attitude and values rather than skill or knowledge. These include perseverance, commitment, and hard work, integrity and authenticity, accountability and responsibility, quality consciousness and pursuit of excellence, "be the customer" mentality, and systems-level perspective. Similarly newly added generic cognitive skills of attention to detail, critical thinking, decision making skills, and creativity and idea initiation were also rated very high by our respondents. *Very interestingly, contrary to the popular interpretation of communication ability, listening skill was rated much higher than the communication, presentation, or persuasion skills.*

Further, the findings of the revised study, as summarized in Table 2.4, are especially useful for curriculum designers and computing faculty.

> For the *Software Services Industry*, the ranked list of top competencies recommended were: *(i) ability to work in team, (ii) abilities related to perseverance, commitment and hard work,* and *(iii) listening skills.* Interestingly, all these competencies require development of attitude and perspectives, usually not the focused goal of the commonly prevailing academic process.
>
> For *large and mid-size IT Product Development Industry*, the required pivotal/critical competencies were: *(i) ability to work in teams, (ii) ability to apply knowledge, (iii) abilities related to perseverance, commitment and hard work, (iv) accountability and responsibility, (v) analytical skills, (vi) problem solving skills,*

and *(vii) research skills*. Here again, all these competencies also relate to attitude, perspectives, and thinking habits, that are usually not focused upon the commonly prevailing academic process.

For *small IT Product Development Industry*, the required pivotal/critical competencies comprise of all that are required for a large product company (with some minor change in their ranks), along with a few *additional* critical competencies: *(i) attention to detail, (ii) readiness for lifelong learning, (iii) quality consciousness and pursuit of excellence*.

It clearly *indicates the nature of the gap which needs to be filled.* These finding create a strong case for overhauling the software development education system in every aspect. The educational programs have to be conceptualized very differently from the training programs. Education has two goals of nurturing as well as training. Webster defines 'educate' as "to develop mentally, morally, or aesthetically especially by instruction," "to provide with information," and also "to condition to feel, believe, or act in a desired way." Hence, the education, especially higher education, is expected to help in growth of human beings to advanced levels. Training is concerned with development of 'skills.' Education on the other hand, has a wider goal of cultivating 'valuable competencies' to develop wise and competent professionals and citizens.

It is not sufficient to only aim to train technically skillful software engineers. The education system has to aim to develop competent software development professionals. Consequently, while development of skill and technical knowledge is certainly important, the development of attitude, perspective, and thinking ability is even more important. It is also imperative to understand that these learning outcomes can be achieved mostly through *changes in academic process*, and also *inclusion of a few additional courses.*

Further, the findings of the revised study, as summarized in Table 2.4, give even more interesting inputs, especially for educators in India, where the software service industry is currently dominating the software industry. Table 2.4 also shows that the competency needs for the usual

work in very large companies, who are currently the largest recruiters from engineering campus, are very limited. Because of very high visibility and recruitment potential, these companies are currently in a position of influencing the management of educational institutes. The finding of Table 2.4 show that if Indian software educators try to orient the goals of their educational programs for this sector, their students will not be suitable for the other two sectors that are growing silently. *Based on our finding, we take a position that in order to inculcate excellence, the educational community should create more partnerships and communication channels with the companies that are involved in product development in large, mid-size or even small sector.*

We also need to educate our students that the software industry is not monolithic, and the most dominant voice is not the most futuristic voice. With increasing pressures on profit margins in the post-recession period, and fast growing software service industry in many other countries, we cannot hope to run our software industry solely as a service industry with the current nature of less challenging low cost work. The most natural allies for educational institutes, that will help us bring excellence by being more demanding users of our product, i.e., students, are small sector product development companies. There is an increasing trend of start-up companies. The educational institutes should create partnership and even facilitate their growth. The students also need to be motivated to aspire to work for such companies, and prepare themselves accordingly. How to forge such partnerships and communication channels is beyond the scope of this dissertation.

Further, in the light of several other identified competencies and deeper reflections about learning, we have recently revised our thinking about this classification scheme as well as the core competencies. We believe that these dimensions have interdependence and are not orthogonal. In our new taxonomy, we do not consider our three categories as independent dimensions. As there is an inter-dependence of these categories, we model these competencies as a three-tier taxonomy of twelve core competencies. The details of this revised and comprehensively distilled taxonomy are discussed in the third chapter after discussing the distinguishing features of software development.

CHAPTER 3: DISTINGUISHING FEATURES OF SOFTWARE DEVELOPMENT AND REQUISITE TAXONOMY OF CORE COMPETENCIES

As our study and investigations showed, most of competencies required for general engineering are also required for *software engineering*, but the latter *does require additional competencies* that are critical for their profession. This prompted us to *investigate the distinguishing features of software development*, so that we would be able to propose what *types of instructional reforms would result in addressing the competency mismatch.* It would also help formulate what *changes in curricula* (deletion/modification/addition of courses) would be necessary to facilitate the changes.

Technology professionals are expected to serve human needs through designing, building, evaluating, maintaining, testing, and modifying systems, processes, and components. In the process, they engage in creation, operation, maintenance, application, and destruction of forms of matter or energy, and/or information. Software developers are concerned with information aspects of such endeavors. Software provides support to acquire, store, search, filter, transfer, transform, and/or destroy information. The transformation may involve content transformation, form transformation, composition, and/or decomposition. Software is not just enabling people to do their activities in newer ways.

It is also empowering them to do new activities to satisfy their needs at multiple levels of Maslow's need hierarchy [141a], and also seek happiness through expanded levels of positive relatedness, and also enhanced levels of autonomy and competence. Software-enabled newer ways of self-expression are helping people to experience higher levels of self acceptance. Hence, software-based artifacts are also reshaping the cultural landscape of society.

It is not proper to call software engineering an extension to any single discipline. Like languages and mathematics, the problem domains and solution possibilities are unlimited, and hence, software engineers can find opportunities to integrate their disciplinary knowledge with any

other discipline, and also grow in that domain. The *discipline of computing is inherently highly inter-disciplinary and is continuously expanding through collaborations with other disciplines of human understanding*. Surely, concepts are shared back and forth, but software engineering has its own distinguishing features. It, without doubt, shares many common practices like project management, and making design tradeoffs with other engineering disciplines. However, some of the following distinctive aspects distinguish it from many other engineering disciplines.

Section 3.1: Programming as an Art to Software Engineering

In the initial days, programs were mostly written by individuals, and the programming tasks were handled more like an art. A set of mathematical and algorithmic courses formed the major portion of computing education.

As computer programs started handling more complex tasks of various applications, their size increased. Large programs need modifications and enhancements throughout the period of their active use. The development and maintenance of large programs was no more an individual effort, but a *teamwork which requires disciplined engineering processes with systematic documentation of activities of analysis, design, coding, testing, deployment*, and maintenance. This was a qualitative jump in the software development process, and by the mid 1980s various courses covering different aspects of Software Engineering became an integral part of computing education. Software development evolved out of a mathematical art to an engineering activity for handling complex tasks. Similar to the other engineering activities, *problem solving and technical competence would be the key core competencies required for software engineering*.

Engineering with a Difference

Software engineers do not manage high volume manufacturing, or mandatory repeated implementations. Unlike other artifacts, replication of software artifacts is easy, and often without any costs. A much larger number of skilled developers and testers have to collaborate to create and evolve many software artifacts. Therefore, only skilled workers are required for software engineering related activities, and group work is much more important. Software engineering is comparatively much younger to other engineering disciplines and the theories, best practices, and essential development tools are still evolving. Further, the obsolescence and

technological changes are very rapid. Consequently, *disciplined lifelong learning is a must, for software engineers.*

While all other engineering disciplines aim to create physical products by combining material and physical processes, software is created from ideas and existing software, and is largely independent of the material and physical processes. Hence, it has *much lesser dependence on physical sciences and constraints*. This gives them the freedom to *create imaginative virtual spaces and services limited only by human thinking*. Hence, there is considerable room for variant approaches to defining and solving problems and creative thinking.

Designs often involve a large number of layers of discrete abstractions and complex interactions among a very large number of components. The methods of dealing with this complexity are not mature and effective enough. Consequently, projects face higher uncertainty factor, and a design cycle often requires several iterations. According to estimates, *80% of software projects fail to meet their original objectives, schedules, and budgets* due to a failure to manage this complexity [142].

Because of the underlying digital phenomenon, noise, fluctuations, uncertainties, or errors can result in unpredictable outcomes and software crashes. Further, the inherent invisibility makes is vulnerable to failures and unpredictable behavior. This often causes the absence of indicators of failure before total catastrophic failure. Hence, the designer's task to create and manage "soft failure modes" becomes more complex.

A lot of available software is highly vulnerable to failures. With exponentially expanding size and complexity of software, its reliability is becoming an even more challenging issue. Fortunately, most software artifacts are not potentially life threatening or a source of human injury. This, however, has unfortunately has contributed to insufficient sensitivity towards risks and reliability among developers. In the last two decades, more attention is being given to developing and following good engineering practices to minimize software risks. This issue is further elaborate in Section 6.2.

In this section, we discussed how the software development process evolved from an individual's art to a large team's engineering effort. However, some characteristics like knowledge intensive nature of the work, much lesser dependence on physical sciences and constraints, inherent invisibility, discrete abstractions, and complex interactions, etc., make it a significantly different from all other forms of engineering. Hence, we posit that while many elements of traditional model of engineering education are not in alignment with the requirements of software development.

Section 3.2: Debugging as a Core Activity in Software Development

A software bug is an anomaly in behavior of running program [142a]. Detection and fixing of bugs is an ongoing process in software development and evolution. Every phase of testing in Software Development Life Cycle (SDLC) i.e. unit testing, integration testing, system testing and customer testing is typically followed by bug fixing. According to Humphrey [371], more than half a typical software organization's effort is devoted to finding and fixing defects. Indeed, such is the emphasis on debugging that many software companies test their prospective employee with buggy code. Proficiency in debugging is integral to deep understanding of software development. Though the possibility of bug introduction lies in every phase of Software Development Life Cycle, but a majority of them can be traced to either design or implementation issues. A significant amount of research has been and continues to carried out on debugging. In March 2010, a word search on "debugging" in ACM Guide showed approximately 23,000 papers. Out of these, approximately 11,000 papers have been published since 2005. In no other developmental activity including engineering, an exactly analogous activity does not play such a central role.

IEEE Standard Classification for Software Anomalies [142a] provides comprehensive methodology for classifying bugs in each phase of bug life cycle. It presents customizable framework for software organizations which serves two objectives. First, it facilitates effective bug tracking by enforcing bug classification in each phase of bug life cycle i.e. Recognition, Investigation, Action and Disposition. Second, it provides with data of bug classification which can be analyzed to identify problematic areas responsible for common bugs. Additionally, a

reflective analysis of bug related data of project and/or release, can help in measuring impact of a process change followed for a particular release or project.

Section 3.3: Process Centric System Development and Maintenance in Software Engineering

The term 'engineering' in the context of software development refers to having a ***systems approach*** to problem solving, and also following a disciplined ***process-centric/oriented approach*** for assuring the quality of the deliverable software system, and their ***maintenance and evolution throughout their life cycle***. The traditional approach of engineering education does not pay much attention to user interaction and evolution, and hence, is not completely suitable for software.

The various aspects of the system development process mainly deals with human processes and engineering management processes. Due to the absence of physical material, the software maintenance activity is not about managing wear and tear. Instead, it is focused on *learning about the misunderstood and changing requirement, removing development errors, and continued development*. Consequently, it is *imperative for software developers to develop the process-centric system development approach as well as the competencies for software maintenance and continuous evolution.*

Section 3.4: Software as an Integral Part of Business, and Need for Comprehension for Software Maintenance

The engineering approach to software development, coupled with continuous exponential advancement in computer hardware technology, brought a higher level confidence among its users, and they started to look at this integrated field as 'information technology.' Software has now become an integral part of business processes of a large number of organizations. Today, any development of business/application software must presume that the *developed software would go through repeated changes*. To satisfy their clients by the quality of their service, *software professionals must be capable of comprehending the existing client software*, and then performing the required modifications/enhancements as per their evolving requirements.

Often a good amount of very old program and/or open source code is re-used, and is blended with new code in the enhanced and newer versions of software. A very large number of developers are engaged in maintaining and evolving the work of other developers. Reuse-based development methodologies are becoming more popular. Hence, a good familiarity with existing components, open source, and the ability to comprehend programs are very important. This is presently not at all addressed by computing educational curricula. Therefore, *software developers need to develop the ability to comprehend the software developed by others, and also write software that can be easily comprehended by other developers.* Increasing dependence on large amounts of Free and Open Source Software (FOSS) makes it even more crucial.

Section 3.5: Role of Empathy and Social Sensitivity in Software Development

Software development in many ways is all about people: users, customers, developers, and managers [143]. All other engineering disciplines attempt to boost human performance by building tools, programs, and systems for supporting physical processes, but on the contrast the software engineers do so by supporting their cognitive processes.

User Empathy

Many artifacts and services created by software engineers offer a much higher level of cognitive, social, and emotional engagement opportunities to users. Unlike all other engineers, software developers do not spend much time building their systems, instead they spend more time trying to figure out what the systems should do. The activity of software development is like 'writing,' where the real task is actually knowledge acquisition, construction, structuring, and representation.

Software design solutions and approaches are often manifestations of the designer's thinking process, rather than the expressions of a physical phenomenon. Hence, an understanding of people's cognitive processes like how they think, how they plan, how they assess situations, how they represent and structure information, how they decide, and other related mental processes is not only helpful but often an essential requirement of their work. This makes their task of

requirement elicitation much more complex. The analysts and designers often have to understand the difference between what clients and users ask, want, and actually need. Designs that are suitable for a context are not necessarily as suited for other contexts. Software developers need to understand users' requirements from multiple perspectives. Software projects usually require a significantly higher level of 'communication' for customer interaction and support. Therefore the analysis, design and architectural part of the software calls for an integrative balance of structured as well as unstructured thinking. The implementation part is easier and mostly depends upon structured thinking.

The main concern of software developers has been gradually shifting from making 'inexpensive' software to 'quality' software to 'appropriate' software. In order to create such software, developers have to acquire the mental adaptability to clearly understand the nuances of processes, and identify the automation possibilities in various application domains. Consequently, they need to have not only an interest in the work of other human beings, but also an ability to understand their experiences as well as beliefs.

Because of the people-centric nature of the activity, the software developers have to deal with professional challenges related *to intellectual property, security, privacy, anonymity, offensive content, and cyber regulation*, etc. It is crucial for software engineers to *respect cultural diversity, and appreciate the conflicts and complexities of the human mind*. *User empathy*: seeing things from users' perspective, and aptitude for 'narrative reasoning' are essential for those software engineers who analyze and design the software requirements specifications.

Group work

As compared to the other kinds of engineering industries, the software industry places a much deeper level emphasis on group work. Large multi-locational, multi-cultural global teams concurrently work in different parts of the world to meet the requirements of clients of varied cultural backgrounds. The majority of an engineer's time in the software industry is spent working with other programmers. The nature of group work among software engineers is not limited to process-centered coordination. Whitehead [144] observes that software engineering projects require many software engineers to collaboratively create a large number of artefacts

incorporating code, requirement specifications, architecture descriptions, design models, test plans, etc. In the software industry, many tools have become popular for facilitating process-centered coordination, ensuring mutual awareness, traceability and consistency, and also collaborative creation of software development related artifacts [145].

Shared development environment, engagement with the team, constant feedback, reviews, and continuous testing and integration are some of the hallmarks of software development methods [146]. All this require a significant amount of group work. eXtreme Programming strongly relies upon practices like daily meetings and pair programming. It uses the practice of pairing not just for code development, but also for design, refactoring, as well as testing [147].

Thus, *empathy and social sensitivity* and are the core competencies required for development of appropriate and quality software. Development of social sensitivity is not to be taken at the periphery, but at the core of the software developers' education program. Good exposure to 'human and social sciences' as well as arts, particularly literature, can help in this regard. Engagement (comprehension, analysis, and so on) with well constructed literary narratives, requires the person to imagine characters' position and experiences. Hence, it can be a great help in nurturing these abilities. *These aspects are discussed further in Sections 4.4 and 6.3.*

Section 3.6: Project Scoping and Estimation for Software Contract

Usually, the software projects are based on **contracts** between the clients and vendors of software services. Earlier, in most of the on-shore projects, the clients were charged on the basis of manpower engagements. The software industry is now ready to take up software contract on a fixed-cost basis. *Project scoping and estimation are the two most challenging tasks of the software development process, which are not adequately covered by current computing educational curricula.*

Section 3.7: Learning New Domain and Knowledge Structuring in Software Development

The software development processes essentially try to map the application domain requirements to programming constructs. Application domain training and even certifications

have become a common part of continuous training programs of software developers. Software projects require a higher emphasis on abstraction, reflection, modeling, and information organization of various distinct application domains. Therefore, the opportunity of inter-disciplinary work is higher, and is further increasing. A deep understanding of various functions in specific application domains is highly valued in the software engineering industry. The software development education program needs to expose the students to *diverse types of domains, and also nurture the ability to learn nuances of newer domains.*

Software as a Medium to Store Knowledge

Armour [148] argued that software is not a product, but the fifth medium to store knowledge after DNA, brain, hardware (artifacts), and books. For storage, it represents the knowledge in space and expresses the stored knowledge in space and time. He posited that all kind of human knowledge is now being transcribed into software, because software has a wider range of valuable *storage and structuring characteristics* as compared to all the previous medium: quite persistent like books, update frequency is only slower than that of brain, intentionality (our ability to change it deliberately) is higher than the hardware and books, ability to self-modify is higher than DNA, hardware, and, books, and activeness (ability to affect the outside world) is relatively unlimited.

He took the position that the difficult and time consuming part in software development is not transcribing the already acquired knowledge into an active form, but acquiring and structuring the knowledge with concern of completeness, consistency, and usability. He viewed *software development as a learning activity*, rather than a production activity, and advocated that *software developers need more training in learning, and knowledge structuring mechanisms* rather than in software itself. *Hence, exposure to disciplines like cognitive psychology that try to understand human understanding and learning becomes much more relevant for software developers.*

Section 3.8: Software Development Process for Ill-defined Problems

A very large number of software developers face new problems every day. As application domain requirements are embedded in real experiences, it is usually very difficult to map and concisely describe it as a software problem. Therefore, real-life software problems are ***mostly ill-***

defined problems. Often multiple iterations and representations are required to define the problems. Projects usually require at least an incremental innovation, and significant amount of development. Further, usually there are no unique best solutions, only multiple acceptable solutions to partially solve the real-life problems. Software developers need to explore new opportunities, identify hidden requirements, generate new concepts, incorporate novel elements, innovate new user interfaces, functional and architectural designs, reuse components, uncover hidden faults, derive new use, and do many other such tasks. Accordingly, there is higher focus and challenge on integration, continued evolution, reuse, creativity, and flexibility. *In Section 4.5, we shall further elaborate upon ill-defined problem solving.*

This characteristic of software problems, and the development process, also contributes to make it a multi-dimensional activity. Agile methods are increasingly being accepted to develop software primarily to address this aspect of software problems. *Reflective thinking, multi-perspective thinking, critical thinking, creative thinking, and innovative problem solving significantly contribute in transforming complex ill-defined problems into simpler well-defined problems.*

This requires them to be not just skilled, but also reflective and creative. Familiarity with a variety of creative works can help in enhancing individuals' creativity. Hence, *reflective engagements with the discipline of aesthetics, arts, literature, and design can play a significant role in nurturing reflective thinking and creativity of software engineers. Reflective thinking is discussed later in Section 5.2*

Problem-centric learning methods have been found to be more effective for developing the competence to solve ill-defined problems [156]. *Further, software problems are not only ill defined, they are also socio-technical soft problems (discussed in Section 6.3) that require a soft systems approach. Consequently a systems-level perspective becomes very important for software developers.*

Section 3.9: Empirical and Qualitative approaches in Software Development Research

Software engineering in many ways is all about people: users, customers, developers, and managers [143]. Unlike many other disciplines of engineering, for their regular day-to-day practice, often software engineers have to *collect and analyze qualitative data.* They collect such data through brainstorming, interviews, conceptual modeling, and observation for activities like requirement engineering, project planning, use case and task analysis, etc. Usage logs, documentation, static and dynamic analysis, bug tracks, etc. are used for activities of program comprehension, testing, reverse engineering, etc., [149].

Further, researchers in software engineering often investigate extremely complex processes in software developments involving a large number of professionals who use highly complex skills. The questions related to cognitive, behavioral, and social aspects of developers and other stakeholders, are also of immense importance and interest to software engineering researchers. The analytical and quantitative research paradigm, which is otherwise well accepted in other disciplines of engineering, is not sufficient for investigating real-life issues involving humans as well as their interactions within themselves and also with technology [150].

Many a times, experimentation is not even possible, and qualitative data is the main, and sometimes, the only source of information. Hence, the researchers in many computing areas, e.g., information systems, software engineering, human computer interaction, and entertainment computing are increasingly relying upon the empirical and qualitative research methods like case studies, action research, survey, etc. These research methods are already very popular in fields like business, social work, psychology, sociology, political sciences, education, information systems, urban planning, architecture, and so on. Like information systems, urban planning, and architecture researchers, the software engineering research community mostly uses these methods with a pragmatic and result-oriented view, rather than from a philosophical stand. Consequently, an understanding of these methods is becoming increasingly important for software engineering researchers, as well as practitioners.

Acceptable data sources and research methods characterize different disciplines. The differences in the two cultures of natural science and humanistic research were highlighted by Snow in 1959 [151]. Qualitative data and methods are not used in science and engineering disciplines. They confine themselves to analytical and quantitative approaches. Qualitative approaches are used in social sciences. However, software engineering practitioners as well as researchers use *analytical, quantitative, and also qualitative approaches*, and give the opportunity to integrate these two cultures. *A heavy dependence on qualitative data and increasing currency of qualitative methods among software engineering practitioners as well as researchers is a distinction that further distinguishes the field from other disciplines of engineering.* This brings it relatively closer to disciplines like architecture and information systems. *We include the quantitative as well as qualitative data analysis techniques in our proposed framework of pedagogical engagements in software development education (Table 8.6).*

Section 3.10: Software Development: Whole-Brain Activity

Diverse activities of software development are not confined to any single type of thinking style. *Diverse types of left- as well as right-brain thinking skills are integrated to create good software.* Abstraction, logic, reduction, critical thinking, etc., are considered as left-brain activities whereas concretization, intuition, creativity, holistic thinking, etc., constitute right-brain thinking. Only interesting persons can develop interesting software. Usually software programs are complex systems. They are executed on computing environments that are examples of complex systems. Software is usually a critical subsystem of a larger technical and/or organizational/social system. Further, the development life cycle of software is another example of a very complex social system. Hence, like all long-term system development activities, *software development also requires several cycles of left- and right-brain activities: abstraction and concretization, logic and intuition, critical thinking and creativity, reflection and experimentation, micro-scoping and macro-scoping, as well as reduction and holistic thinking respectively. Software development is a whole-brain activity.*

The whole process requires an integration of the ability of abstract conceptualization, and an active experimentation with concrete experiences and reflective observation [152]. Software development leverages developers' strengths in varied types of intelligences as identified in

Sternberg's *Theory of Triarchic Intelligence* [153], Herrmann's *Four Quadrant Model of the Brain* [154] and also Gardner's *Multiple Intelligence Theory* [155].

In addition, to the ability to solve ill-defined problems, it is imperative *to develop diverse types of thinking skills, comprising whole-brain activity, among software professionals. Our proposed framework of pedagogical engagements in software development education, discussed in Section 8.3, aims to offer such whole brain engagement.*

Section 3.11: Revised Taxonomy of Core Competencies for Software Developers

Integrating all our earlier theoretical and empirical studies about core competencies, recommendations of various organizations as well as researchers, the first version of our earlier taxonomy, and our latest understanding of the distinguishing features of the software development activity, we have further revised our list of desired core competencies for software developers. With reference to the specific context of software development, *we created a comprehensive set of thirty-three competencies (Appendix A2)*. It *subsumed and expanded the thirty-five competencies of Table A3.2 (Appendix A3)*. Some additional very important competencies were also included in this comprehensive set of Appendix A2. These included - *curiosity, domain competence, abstraction, algorithmic thinking, knowledge of physical and natural world and intercultural knowledge, reflection, self acceptance and self regulation, and workload management*.

Through further grouping based on logical closeness, hierarchy, and/or dependency, we have further reduced the set from thirty-three to twelve core competencies (Table 3.1) each with extended meanings. The competencies in the comprehensive set are *subsumed within one or more of the revised set of twelve competencies*. We have dropped 'wealth creation skills' from our revised core set, as this was categorized in the lowest ranked category of competencies based on the rank given by our respondents, and it also does not integrate well within our final twelve competencies.

Table 3.1: Core competencies for software developers

Twelve core competencies
1. Technical competence
2. Computational thinking competence
3. Domain competence
4. Communication competence
5. Complex problem solving competence
6. Attention to details
7. Critical and reflective thinking
8. Creativity and innovation
9. Curiosity
10. Decision making perspective
11. Systems-level perspective
12. Intrinsic motivation to create/improve artifacts

Annexure A4 gives the mapping of thirty-five competencies of Table A3.2 (Appendix A3) with this reduced inclusive set of Table 3.1. The popular beliefs about the competencies, and also curriculum, place maximum importance on technical, communication competencies, and/or problem solving competencies. This table very strongly brings out the utmost importance of the development of a *systems-level perspective* among software developers. *Seventeen out of the thirty-five of our* earlier identified competencies relate to it. *Ten out of thirty-five competencies relate to reflective thinking* and *nine relate to critical thinking*. Eight out of thirty-five competencies relate to *decision making perspective*. Seven competencies relate to communication competence and also *intrinsic motivation to create/improve artifacts*. Six competencies relate to *computational thinking, curiosity, and domain competence*. *Attention to detail* and *Creativity and innovation* relate with five competencies. All these are emerging as important goals for education programs for future software developers.

These results are significantly different from all earlier results, including our own. Surely, like other engineers, problem solving skills are important for software developers (ref: Table 2.1). However the nature of the problems they are required to solve are significantly different from other engineering disciplines. Hence, their *problem solving skills needs to be driven and conditioned by a systems-level perspective, critical and reflective thinking, decision making perspective, curiosity, intrinsic motivation to create/improve artifacts, curiosity, attention to detail, and creativity and innovation.*

Based on several models of organizing the competencies and models related of learning and human development, we have also revised the structure of our taxonomy of core competencies. We posit that the three categories of our earlier taxonomy, given in Table 2.5, have interdependence and are not orthogonal. Consequently, in our revised taxonomy, we do not consider these three categories of competencies as orthogonal dimensions. As there is an inter-dependence of these categories, we model these competencies as a three-tier taxonomy of core competencies. We have also classified the revised set of twelve core competencies into three-tier taxonomy, as given in Table 3.2. It includes five basic competencies, three competency driver-habits of mind, and four competency conditioning attitudes and perspectives. This is also reproduced as Table 8.1. The arrows in this table indicate the direction of influence.

Table 3.2: Three-tier taxonomy of core competencies for software developers

Basic Competencies	**Competency Driver-Habits of Mind**	**Competency Conditioning Attitudes and Perspectives**
1. Technical competence 2. Computational thinking competence 3. Domain competence 4. Communication competence 5. Complex problem solving competence	6. Attention to details 7. Critical and reflective thinking 8. Creativity and innovation	9. Curiosity 10. Decision making perspective 11. Systems-level perspective 12. Intrinsic motivation to create/improve artifacts

With reference to our three-tier taxonomy given in Table 3.2, the basic competencies are necessary for software developers to contribute to the development of useful and quality software. The highly pervasive core competency driver-habits of mind are necessary to continuously develop, refine, and enhance the basic five competencies. These in turn also help in developing even more useful and high quality software. The most pervasive and enduring competency conditioning attitudes and perspectives create necessary conditions for creating meaningful software and wiser professional software developers. Thus, these competency conditioning attitudes and perspectives guide and regulate the application of competency driver-habits of mind. This in turn helps to create meaningful, appropriate, ethical, and very high quality software. The competency driver-habits of mind are also necessary to continuously develop, refine, and evolve the highest level competency conditioning attitudes and perspectives.

Basic Competencies

The basic competence for software developers includes skill, rules, and knowledge related to various technical activities of software development, computational thinking, application domains, communication, and general purpose complex ill-defined problem solving. These contribute to the development of useful and quality software.

Technical competence is manifested in the practical as well as intuitive understanding required for the executing various technical tasks related to software development. Computational thinking as an approach to problem solving, creating services, interfaces, and behaviors, and also understanding human behavior. Understanding layers of data and process abstraction forms the core for computational thinking.

Since, application domains of software include all kinds of domains, it is imperative for software developers to understand the concerns, focus, aim, knowledge structures, and thinking approaches of application domains. The communication competence for software developers is significantly different from the communication competence for sales professionals. It is essential for understanding the needs of the consumers, the difficulties of their clients and co-developers. It is required for knowledge acquisition as well as knowledge sharing.

Software development is not about finding answers to well defined problems but solving complex ill-defined problems. Performance on well-defined problems is not correlated with performance on ill-defined problems. Hence, education processes need to give special emphasis on this.

Competency Driver-Habits of Mind

Software development is more of a cognitive activity rather than a construction activity. With respect to the multifaceted activities of software development, three mental habits: attention to details, critical and reflective thinking, and creativity and innovation, have been identified as the most important for software developers. These habits contribute to continuously develop, refine, and enhance the basic competencies and create more useful and high quality software.

Inconspicuous nature of software and the necessity of thoroughness, long attention spans, consistency, etc., make 'attention to detail,' the most essential mental habit for soft developers. Critical thinking is necessary for controlling errors in logical and analytical reasoning at various stages of software development. Software development is essential an evolutionary activity that requires continuous reflection about the product as well as processes to uncover and alter the limitations of both. Much of software is increasingly becoming concerned about user's experience.

Creative people are needed to design of new innovative software products for users and new procedures and tools for software developers, as well as management of software development. Development of these habits has to be put as a core learning outcome of all courses.

Competency Conditioning Attitudes and Perspectives

Attitudes and perspectives affect a professional's motivation, expectation, and also ability to practice. Curiosity, decision making perspective, systems-level perspective, and intrinsic motivation to create/improve artifacts are especially important with reference to the requirements of the profession of software development. These attitudes and perspectives guide and regulate the application of competency driver-habits of mind to create meaningful, appropriate, ethical, and very high quality software.

Curiosity is recognized as a source of critical thinking and also creativity. Software developers are required to have a high level of curiosity to learn 'how things work,' 'how to create things that work,' and also find out 'what may be consequences and risks.' Today's software developers need to be deeply interested in learning not only about the power of information and software technology, but also needs and even possibilities of human beings.

Decision making is about choosing intelligently among less than perfect possibilities. The decision making process requires software teams to blend short term as well as long term perspectives. Long term perspective focuses on sustainability that includes concerns for stability, efficiency, and scalability.

Systems thinking is seeing wholeness, seeing interrelationships rather than individual things. Software developers use systems to develop systems for supporting systems. Many software systems are socio-technical systems and the software development systems are essentially social systems.

Intrinsically motivated state has been found to more conducive to creativity. Hence, computing students' intrinsic motivation for creativity needs to be enhanced for creating conditions for self actualization through creation.

In the next three chapters we discuss the meaning and relevance of these twelve competencies in the context of software development and the education of software developers.

CHAPTER 4: SOFTWARE DEVELOPERS' EDUCATION FOR DEVELOPMENT OF BASIC COMPETENCIES

BusinessDictionary.com defines competence as "*knowledge of, and skill in the exercise of, practices required for successful accomplishment of a business, job, or task*." Competence is manifested as performance. In 1980's Rasmussen, Reason, and Norman elaborated upon a *three level hierarchy of human performance*: *skill based, rule based, and knowledge based*. As per this model, the lowest level of human performance is *skill-based at which the behavior is controlled by a stored sequence of action in space and time.* Expert programmers can create low level programming constructs *without conscious engagement*, whereas novices have to think about such compositions [157]. The expert programmers can make skill-based errors in routine actions because of intrinsic factors of inattention or over-attention. The middle level of human performance is *rule-based at which the behavior is controlled by stored if-then rules. The highest level of human performance is knowledge-based at which the behavior is controlled by deliberate logical and analytical reasoning.* This behavior is invoked by beginners who start performing a task or by experienced persons who face a novel situation. The errors at this level occur either because of resource limitation of conscious mind or incomplete/incorrect knowledge.

For developing software, the developers have to engage themselves at all these three levels with respect to following five basic competency domains.

1. Technical Competence
2. Computational thinking competence
3. Domain Competence
4. Communication Competence
5. Complex Problem Solving Competence

Software developers' performance is result of integration of skills-based aspects with rule-based and knowledge-based reasoning. Practice sharpens skills-based aspects of their professional tasks. Rich experiences with varied cases enrich their rule-based reasoning. Knowledge-based reasoning requires critical and reflective thinking. Under the enabling conditions created by

'competency conditioning attitudes and perceptions,' software developers use their 'competency driver - habits of mind' to acquire, integrate, apply, refine, and extend their competence, i.e., skill, rules, and knowledge in all these five competency domains.

Section 4.1: Software Developers' Education for Development of Technical Competence

Technical competence of professionals is manifested in the practical as well as intuitive understanding required for the executing various technical tasks of a profession. Mosby's dental dictionary defines technical competence for dental care professional as "*the ability of the practitioner, during the treatment phase of dental care and with respect to those procedures combining psychomotor and cognitive skills, consistently to provide services at a professionally acceptable level.*" A professional's technical competence requires a coherent and integrated understanding of factual, conceptual, and procedural knowledge in the subject area. It needs the ability to use tools, techniques, procedures, best practices, and standards to solve problems. Further, it requires an intuitive understanding of what is technically feasible, scalable, and reusable.

Engineering and design professionals need to understand the current state of the art, and emerging technologies. Further, they need to be able to use this understanding to assess tractability of the problems [158]. They need to have the patience and wisdom to consider a restricted subset of the problems, till the technology advances to a level where a solution of the original unconstrained problem can be attempted. As experience and technology matures, the focus shift shifts from short term goals to higher long term goals which expand to encompass an entire class of problems. It is imperative for them to have a good understanding of the limitations and risks associated with each piece of work.

In our 2009 survey on required competencies for software developers, twenty software professionals assigned 'technical/domain competency' and 'analytical/design skills' an average rating of 2.95 and 3.0 respectively on a scale of 0-4. A majority of these responses, 60% and 63% respectively recommended these to be critical or very important competencies.

With reference to Appendices A2 and A3, Technical competence of software developers also includes the following:

1. Ability to apply knowledge
2. Technical competence to solve the software solvable problems using tools and techniques
3. Use of open source software
4. Knowledge of industry's best practices and standards
5. Appreciation of what is technically feasible
6. Ability to identify the risk level of each piece of work
7. Design skills
8. Numerical ability

Curriculum designers continuously face and address the challenge of identifying the required technical competencies suitable for their respective industries. However, often this process gets disengaged from the real continuously evolving industrial requirements. With reference to professional courses like engineering, it is critical to continuously collect required inputs from relevant industry and update the curriculum. The computing industry is evolving faster than the academic discipline of computing. There is a continuous complaint from the industry about severe shortage of well prepared graduates. The continuously evolving work profile of computing engineers is not appropriately reflected in the educational programs. Most of large software companies have their own education wings to train and retrain their developers. Typically large companies have mandatory technical training for their staff every year. The training programs are focused on several aspects like core technologies, development methodologies, project management, etc.

Based on a long industry-academia consultative process, SWEBOK [68] provided an excellent documentation of required technical competencies that software engineers with four years of experience should have. The SWEBOK report gives details about the ten knowledge areas related to software engineering: software requirements, design, construction, testing, maintenance, configuration management, engineering management, engineering process, tools and methods, and quality. With reference to different topics in these areas, Appendix D of the

SWEBOK report specifies the desired level of competence out of the six levels as Bloom's taxonomy: Knowledge, Comprehension, Application, Analysis, Synthesis, and Evaluation.

Important Technical Activities in Software Development

We have attempted to further understand the full spectrum of required technical competencies from the perspective of real technical work profiles in the software industry. We first catalogued the various technical and technically oriented activities through a consultative process. *Sixteen professional engineers in the software industry with high quality and rich industry experience as well as strong academic background were consulted.* Collectively, these experts have a rich work experience of over 330 man-years in various activities of software development. We have grouped various technical activities under seven major categories. Appendix A5 catalogues all these technical and technically oriented activities related to software development activities. These categories relate to planning, design, realization, evaluation, and client interface activities. Our classification also includes two categories of ubiquitous and over-arching activities.

Planning activities relate to project and risk planning. The design related activities encompass diverse design activities at various stages and multiple levels. The design activities are followed by realization activities. Realization refers to the class of activities that relate to implementation and maintenance. Evaluation activities relate to selection and evaluation of tools, technology, products, and process. Many activities require intense interfacing with the client. These client interface activities relate to requirements and support.

Some activities are embedded within almost every function. These ubiquitous activities include process support activities that apply across all phases of a project. The activities in abovementioned six categories need the support of certain overarching activities that are apples companywide across the projects.

Based on this catalogue of activities, *we administered a survey among another group of software developers. Fifty-seven software professionals responded to our survey.* About 14% of these experts have more than fifteen years of experience in software development, 11% have worked for more than 10 years, 42% have more than five years experience, and 21% have more than two

years of work experience. They work in companies like IBM, Oracle, Cadence, EBSCO, TCS, HCL, Wipro, Mahindra Satyam, Bloomberg, LGSoft, Samsung, Deloitte Consulting, and CRIS, etc. The views of our respondents on the most important activities that must be included in the main goals for new curriculum for the future generation of software developers are given in Table 4.1.

Table 4.1: Most important activities that must be included in the main goals for a new software curriculum

1. Algorithm/Computational Procedure/Component and Interface Design (79% respondents)
2. Application/Product/System Design/Prototyping (75% respondents)
3. Product/Requirement Definition and Specification/Requirement Engineering/Visualization/Consulting (75% respondents)
4. Code Analysis, Program Comprehension, Re-documentation (68% respondents)
5. Innovation and research (66% respondents)
6. Application, Component Development/System Integration (65% respondents)
7. Group work, people management, and leadership (65% respondents)
8. Estimation and Costing, Project Scheduling (63% respondents)
9. Product/Process Quality Assurance and Control (60% respondents)
10. Validation and Verification (Testing) (58% respondents chose)
11. Technical Documentation, Presenting Ideas and Insights (54% respondents)
12. Test Design (52% respondents)
13. User Interface Design (47% respondents)
14. User Acceptance, End-user Documentation, Deployment and Roll-out, Customer support (45% respondents)
15. Security Architecture Design, Architecting, Component Selection (42% respondents)
16. Project Monitoring and Control (40% respondents)
17. Tools and Technology Selection and Evaluation (40% respondents)
18. Usability/Value/Impact Analysis (39% respondents)
19. Resource Planning and Management, Staffing and Team Development (36% respondents)
20. Risk Planning and Mitigation (36% respondents)
21. Build and Release, Configuration Management (36% respondents)

Many activities of Appendix A5 received the support of less than 35% professionals. We have not included such activities in Table 4.1. This list is part of our proposed framework of pedagogical engagements in software development education (Table 8.3, first column).

Pedagogical Perspective

In order to perform these activities, software developers need to have an *integrated understanding* that hardware and software are two extreme ends of the possible solution space, with the possibility of varying levels of interaction and exploitation between two. The correct identification of the system/environment boundary to define the system/environment interface is crucial to the ability to successfully define, design, and develop a functional system. This often requires a *tradeoff in decomposition and allocation of functionality to hardware and software*

sub-systems. The need to define interfaces early is critical in order to support modularity, multi-team development, and testability.

There is a distinction between a hierarchical decomposition for project management vis-à-vis decomposition for driving development. The managers and the developers need to understand this distinction, and also the corresponding mapping between the two views. They also need clarity about the system-environment boundary, interface, system metrics, constraints, and acceptance criteria.

Given the drastic reduction in the time to market a product, it is essential that the product is brought to the market the earliest. This forces a designer/developer to exploit all means possible, including third-party tools, libraries, and sub-systems. Today, software development does not only require logic building ability, but also hugely depends upon system/platform knowledge. Efficient development requires inclusion of available in-house source code, commercial off the shelf components (COTS), and also open source software. Hence, good awareness and ability to select, include, and modify, the available software components and subsystems into new systems are now imperative for software developers.

They also need to have theoretical, practical, and intuitive understanding of the *entire programming stack* that includes hardware (CPU, memory, cache, interrupts, microcode), operating system APIs, binary code, assembly, static and dynamic linking, libraries, compilation, interpretation, garbage collection, heap, stack, memory addressing, processes and threads, *understanding of space-time tradeoff,* and *data structures and algorithms.*

They must have *hands-on experience* with at least *two different instances of* each of the following: *architectures, operating systems, programming languages, programming paradigms, compilation systems, DBMSs, glue/scripting languages, IDEs and productivity tools* like profiling, testing, CASE tools, version control systems, etc.

They need to learn *code optimization, performance tuning, defensive programming, assertions, and mixed paradigm programming*. The curriculum should address *code organization* within and

across files, source code tree organization, source code version control, build automation, deployment, and roll out. Creating awareness and appreciation of *upcoming technologies and standards* is strongly recommended as an agenda for curriculum designers for software development education.

Pervasive Knowledge Areas

Today, *web and multimedia* (including graphics) nearly have omnipresence in computing systems. Hence, the students must learn multimedia and graphics programming, including use of special APIs for the purpose. All computing students must be given some practice with web-database architecture and programming.

Embedded systems place special requirements on interfacing peripheral and communication protocols. Exposure to *peripheral interfacing and communication protocols* is highly recommended. *Security* has emerged as a big concern for users and a challenge for computing professionals. The education program must give good experience with secure programming and security APIs. Use of *mobile phones* as a computing platform is growing exponentially. Students must be exposed to developing software for at least one such platform.

Use of *open source* for developing software has become very popular in recent years. Therefore students must be comfortable with identifying, evaluating, modifying, and integrating open source for their work.

In Section 9.2.1, we discuss our experience in infusing some of these elements in regular computing courses.

Need for higher focus on debugging

Many characteristics like significant work in new development in every project, discrete abstractions, complex interactions among a very large of components, inherent invisibility, large groups of developers, continuous evolution, etc., make *software highly vulnerable to errors.* Software errors (bugs) result because of *lack of attention and also because of misconceptions* related to programming, operating systems, compiler, and tools, libraries, etc. Software errors

can be reduced by developing proper technical competence. The students need to *learn to avoid, anticipate, identify, track, and remove bugs* that often arise due to their misconceptions in their as well as others' source code.

Debugging activity is by and large *ignored by curriculum*. SWEBOK [68] refers to debugging in a casual manner, and does not include it at all in its appendix D of specific topics. Debugging has been more seriously attention in interim revision of CS2001 [159]. We take a position that computing curriculum need to address this issue more seriously. Students need to be well versed in the use of tools and techniques for identifying and rectifying errors. The students also need to be exposed to common bugs, their consequences, and remedies. *The computing curriculum and education programs need to give much more emphasis on debugging. In our proposed framework of pedagogical engagements for software development education, we include this aspect in Section 8.1.* They need to learn to use debugging tools for interactive debugging, static analysis, and dynamic analysis.

Debugging activity requires lot of analytical effort. Metzger draws an analogy between programmer and safety analysts who seek to prevent future problems by doing a root-cause analysis of significant events, e.g., accidents, near misses and potential problems. For effective debugging, he suggests the usage of *root-cause analysis techniques* like 'cause and event charting' and 'faulty tree analysis.' *We include these as part of proposed framework in Table 8.6.*

Metzger observes that *design errors* may occur because of *errors in data-structure, algorithm,* or *interface specifications related to user-interface, software-interface, or hardware-interface* [157]. Annexure AN4 gives a summary of his observations.
We have created a taxonomy of software bugs based on misconceptions related to programming, operating systems, compiler, and software architecture [159a]. Our taxonomy of software bugs is given in Appendix A6. *We propose to enrich the courses with sufficient exposure to some of these bugs from each category.*

Education program needs to give them opportunities to acquire, apply, extend, refine, and integrate their technical competence. Technical competence includes skill, rules, knowledge related to various technical activities as discussed in this section. Much of the routine behavior of experienced programmers is rule-based. These rules are often implicit and unarticulated by them. They use their rules to organize things into patterns [160]. Experienced software developers encode their rules in such a way that enables them to apply their rules with much lesser effort than a novice for solving the same problem. However, their rules depend upon their expertise, and may not cover all cases.

Metzger [157] catalogues the software errors because of rule-based reasoning into two broad categories: (i) misapplication of good rules occur when a time-tested rule is applied by overlooking the additional conditions that warrant another rule, (ii) application of a bad rule occurs when conditions are wrongly represented, or ineffective/inefficient action is chosen. More details of these are discussed in Annexure AN4. *Hence, it is necessary for them to understand the scope and limitations of their rules.*

Traditional methods of teaching fail to take such a comprehensive perspective of technical competence. In **our recently concluded survey** "Software developers - (How) Did your college help you in your development?" (Table A10.2 (i) part-I, Appendix A10), huge proportion of the respondents felt that as compared to all other academic engagements, their *projects did much better to develop their design skills (92% respondents felt so), implementation skills (90%), debugging skills (84%), technical competence (76%), and analytical skills (75%).* Laboratory work (70%), knowledge transmission oriented lectures (54%), and homework and tutorials (48%) were considered as effective for developing *technical competence*. Laboratory work and industrial training (84% and 49% respectively) were found to effective for *implementation skills.* Laboratory work, industrial training, and mentoring of juniors (86%, 35%, and 31% respectively) were found to effective for *debugging skills*. Laboratory work (63%), research literature survey (58%), thinking oriented lectures (54%), homework and tutorial assignments (42%), discussion with other students (38%), and industrial training (33%) nurtured the *analytical skills*. Laboratory work (61%), industrial training (49%), and thinking oriented lectures (47%) were found to be the main contributor for development of *design skills*. *Traditional knowledge*

transmission oriented lectures and discussions with others were found to be least effective for development of analytical skills whereas written examinations were found to be least effective for development of design skills.

Section 4.2: Software Developers' Education for Development of Computational Thinking

Traditionally, software was regarded as belonging to the domain of 'applied mathematics.' Many experts view software development as a special type of mathematical problem solving activity which requires the developers to use various mathematical thinking processes like step-by-step approach to decomposition, abstraction, pattern recognition, spatial and temporal modeling, deduction and induction, and synthesis.

In his much debated talk called 'On the cruelty of really teaching computing science,' in 1989, Dijkastra emphasized on formalism [64]. He further identified the following two *radical novelties* of programming: (i) conceptual hierarchies deeper than a single mind ever needed to face before, and (ii) in a discrete world small changes do not imply small effect. In 1991, the joint ACM/IEEE-CS curriculum task force [62] identified *twelve unifying and pervasive concepts of computing - binding, complexity of large programs, conceptual and formal models, consistency and completeness, efficiency, evolution, levels of abstraction, ordering in space, ordering in time, reuse, security, and trade-off and consequences.*

In our 2009 survey on required competencies for software developers, *twenty software professionals assigned abstraction thinking and algorithmic thinking an average rating of 2.9 and 2.8 respectively on a scale of 0-4.* An overwhelming majority of these respondents *(70%) recommended these to be critical or very important competencies* with respect to the requirements of software developers' multi-faceted professional activities.

With reference to Appendices A2 and A3, computational thinking of software developers also relates to the following:

1 Abstraction and transition between levels of abstraction, representation skills spatial and temporal modeling skills, structuring skills, and theorizing

2 Algorithmic and structured thinking. Logic, pattern matching, logical what-if analysis, problem decomposition and synthesis, etc.
3 Analytical skills
4 Attention to detail: breadth, depth, clarity, accuracy, preciseness, specificity, relevance, significance, completeness, consistency
5 Problem solving skills
6 Critical thinking
7 Design skills
8 Creativity and idea initiation

Algorithmic problem solving activities

Expert programmers think and develop algorithms rather than think in specific language syntax [179]. In 1979, Kowalski postulated that an algorithm consists of logical and control components [180]. The logic components define the knowledge that is needed to solve the problem. The control component determines how to use and sequence such knowledge to do so. Muller and Haberman [181] have enumerated algorithmic problem solving activities.

Problem comprehension is the first activity that involves reformulation of the problem statement in terms of data items, initial state, goal, assumptions, constraints, and scale. This is the most critical thinking stage for designing algorithms. For five consecutive years (2002-07), in data structure and algorithm courses, we emphasized on this aspect by engaging students to generate examples of increasing complexity in terms of scale, diversity, assumptions, goals, initial state, constraints, tolerance, and exceptions. The students were required to first develop the algorithms for the simplest possible case of each problem. With each additional case of increasing complexity of the problem, they were required to identify the limitations of the existing solution, and then modify the same to meet more complex demands. A comparison of problem solving strategies of best performing students of one such class with the best performers of a later class where the faculty used a more traditional textbook oriented approach, showed that the students of the first group showed a much higher level of sophistication in their approach to solve algorithmic problems.

The second activity of decomposition is the identification, naming, and listing of subtasks and data items with attributes, objectives, and roles. Analogical reasoning, generalization and abstraction are used for identifying similarities between problems, and extracting prototypes of problems from analogical problems in different contexts. This helps in identifying a problem's prototype for its categorization.

This is followed by the problem's structure identification, i.e., composition, identifying the relation between subtasks, data items, state transitions, data flow, and distinguishing between logic and control. Schematizing a problem's structure using diagrams helps a great deal in this process. Flow chart has great limitations in terms of its inability to show data or states.

A new diagramming technique called 'concept mapping' (Appendix A19) has been developed and used in various classes as mentioned above. The students who were exposed to concept mapping in their introductory data structures course continued to use it, or a self-modified notation, even after graduating. Based on this analysis, this notation has been re-introduced in an introductory data structure course, and the concerned faculties as well as students are finding it useful.

Finally, algorithm thinking requires evaluation and appreciation of efficiency and elegancy, reflecting on problem-solving processes and strategies to draw conclusions for the future, and verbalization of ideas and differentiating between an idea and its implementation.

Lethbridge's Study on Most Important and Influential Topics

Lethbridge et al [46-48] surveyed approximately 200 practicing software engineers and managers. Their report shows that *five out of the thirteen subject categories did not contribute even a single topic to the list of twenty-five most important and influential topics,* while these categories were felt by the respondents to be over emphasized in the curriculum. *These subject categories are theoretical computer science, mathematical topics in computer science, other hardware topics, general mathematics, and basic science.*

Computational Thinking: Beyond Traditional College-level Mathematics and Algorithmic Thinking

In 2009, we initiated an online discussion among the online community of software professionals on LinkedIn. Nearly 30% respondents felt that proficiency in mathematics indicates a high capability to handle abstractions, the ability to go into detail, ability to plan and approach a problem in a methodical/structured fashion. On the contrary, the other *majority suggested that this relationship between mathematics and software has been exaggerated, and gave reasons like mathematics education does not necessarily enhance lateral thinking for problem solving. However, many respondents grounded software development competency into puzzle-solving ability.*

Wing [183] viewed *computational thinking as an approach to problem solving, system designing, and also understanding human behavior*, by drawing on the concepts fundamental to computer science. Isbell et al [182] shift the emphasis from algorithm to interaction and suggest that computing problem solving is not so much about finding answers but more about creating services, interfaces, and behaviors. Fant [53] argues that, unlike mathematics, computer science is more concerned with issues related to creation and actualization of process expressions.

In our experience, students without good background in school level mathematics, especially in topics like algebra, geometry, trigonometry, functions, etc., have been found to perform poorly in software development oriented courses. However, performance in college level mathematics courses like higher calculus, differential equations, and linear algebra, etc., seem to have no correlation with the performance in software development skills of college level engineering students. There are many exceptional programmers whose performance in college level mathematics has been poor, and there are many poor programmers with very good performance in college level mathematics.

According to Wing [183], computational thinking is about *producing executable descriptions*, i.e. automable or automatically manipulatable models. *She strongly recommended that computing faculty teach courses on computational thinking* which includes thinking in terms of: constraints, abstraction, decomposition, heuristics, algorithms, recursion, concurrency,

synchronization, efficiency, elegance, tradeoffs between processing and storage, caching, interpreting code as data and data as code, and prevention, detection and recovery from worst-case scenarios. Two relevant intuitions for computing are the concepts of *having something and being in a state* [184]. In 1975, the chief designer of many programming languages, e.g., Pascal and Modula, Nicklaus Wirth, wrote a book titled, 'Algorithms + Data Structures = Programs.'

There is a need to review the college level mathematics content from this perspective. Whenever mathematics courses succeed in engaging students in representing real-life problems into mathematical or computable problems, and then solving those problems using mathematical tools, they provide direct help in enhancing the analytical thinking skills required for software development. *Courses on puzzle-solving and mathematical modeling have a higher potential to make such direct contribution. We include this aspect in our propose framework of pedagogical engagements (Table 8.6).* In 1999, SEI- CMU published a report to define the discipline of software Engineering [67]. The mathematics requirements included 'mathematical logic and proof systems,' 'discrete mathematical structures,' 'formal systems,' 'combinatorics,' and 'probability and statistics.'

Isbell et al [182] also take a position that though *computing overlaps with various disciplines like mathematics, science, engineering, arts, humanities, and social sciences, it is neither of these* and is a discipline in itself that requires a distinguished kind of mindset which they term *computationalist thinking*. They posit that the *equivalence of model, language and machine* is the key idea of computing. According to them, *computing marries the representations of some dynamic domain and dynamic machine to provide theoretic, empirical, or practical understanding of domain or machine.*

Computational thinking requires thinking in terms of *data attributes, data flow, relationships, and state transitions*. It also involves thinking about *system-environment boundary, interface, system metrics, scale, sequence flows, transactions, composition, exception handling, testability, evolution, and documentation*. Today, *user interaction has become equally important*. Isbell et al [182] posit that computationalist thinking focuses on *model, abstraction, interpretation, scales*

and limits, simulation, and automation. They insist that computationalists must understand how to *create, analyze, and critique models*.

Abstraction as an Integral Part of Computational Thinking

Hazzan and Tomakyo [124] highlight the importance of mental habit of abstraction and the ability to make transitions between levels of abstraction as an important skill for software developers. Computational thinking involves stepwise refinement with different notations at different levels. It involves thinking about reality at different levels of abstractions and to model the same through executable formalisms. The fundamental feature of computational thinking is abstraction of a situation/system/problem in such a way that the selected details in the model make it executable by a machine. The choice of the selected executable abstractions of the problem is driven by its purpose [185]. The purpose may be: (i) automation, or (ii) simulation either to get deeper insights or to create virtual worlds.

Abstraction is informally described as the process of mapping a representation of a problem onto a new representation. Philosophers like Aristotle, Hume, and Locke have taken a reductive perspective of the abstraction process and see it in terms of the filtering-away of irrelevant components and specifics, with the aim of extracting content or meaning. Constructivist perspective of abstraction emphasizes selection and combination of relevant constituents. Each new abstraction identifies a new phenomenon and becomes a potential constituent for further abstraction [186]. Abstraction concepts include association, aggregation, composition, classification, or generalization.

The computing worlds consist of *things (objects), events, and actions (activities, processes, and operations)*. Kramer viewed computational abstraction as generalization to identify the common core or essence, manipulating symbolic and numerical formalisms, and also moving from an informal and complicated real world to a simplified abstract model [187]. Wing [183] sees it first as a process of deciding what details we need to highlight/ignore, and then choosing an appropriate representation to model the relevant aspects of a problem. It takes several iterations to fine tune computational abstractions. *The maximum challenge is to gather a 'complete' overview of the given problem.*

Computational abstractions are to be discovered by balancing creation against reuse, with a strong preference for reuse of things that are already tried and tested.

Abstraction of Real World

Nicholson et al [188] caution that since software developers solve problems that exist in the real world, their *solutions must ultimately succeed in the real world*, not just on the abstract level used to define the solution. They also suggest critical evaluation of computational abstractions because abstractions may become too generic/specific. The details removed in an abstraction may reemerge in a way that requires that they be considered. Any representation can have consequences for how the subject of the abstraction is understood. The existing computational abstractions may cross into new contexts by accident or default, and the same subject may recur at multiple layers of abstractions with different aspects and context. They insist on identification of the context of use and then defining the computational abstraction accordingly. For identification of the context of use, their recommendation is to understand the abstractions that are already used within the relevant context, and the socio-political context thereof. Software developers also need to identify the reusable ideas/components in the application and technology domain. Finally, regarding *simultaneously working with multiple layers of abstractions*, it is important to understand how the different layers of abstraction relate to each other, and always clearly indicate the layers being currently dealt with.

A Key Principal for Designing Hierarchy of Abstractions

In his classic paper, Miller had suggested that humans have an upper limit of the number of items that they can simultaneously hold in their temporary memory for further cognitive processing. This is in the range of seven plus/minus two [189]. Software developers should keep this in mind as they develop their hierarchy of abstractions.

Pedagogic Perspective

In **our recently concluded survey,** "Software developers - (How) Did your college help you in your development?" (Table A10.2 (i) part-II, Appendix A10), a majority felt that as compared to all other kind of academic engagements, their student *projects did much better to develop their*

algorithmic (72%) and abstraction (57%) thinking respectively. Algorithmic thinking was felt to improve through engagements of thinking oriented lectures (60%), laboratory work (58%), research literature survey (40%), and knowledge transmission oriented lectures (36%). Abstraction competence was felt to improve through engagements of research literature survey (40%), thinking oriented lectures (38%), and laboratory work (32%). *Discussion with others, knowledge transmission oriented lectures and written examinations were felt to be least effective with respect to development of abstraction competence. Discussions with others, faculty as well as other students were found to be ineffective with respect to development of algorithmic thinking.*

Student assignments need to be designed keeping the objectives of strengthening various aspects of computational thinking as discussed in this section. These assignments can be designed as per our proposed framework of pedagogic engagements in Chapter 8.

Section 4.3: Software Developers' Education for Development of Domain Competence

All professional societies, including those that are associated with the professions of engineering or software development, strongly advocate that their profession's main aim is to work for the welfare of society. Welfare requires a balanced fulfillment of the human needs at multiple levels of need-hierarchy as per Maslow's need-hierarchy [141a], in compliance with concerns of sustainable development. Many technologies have mainly been supporting human activities that facilitate fulfillment of lower level human needs as per Maslow's model of need hierarchy. Potentially, *software can even support some human activities that facilitate fulfillment of people's needs in various domains even at upper levels of Maslow's hierarchy*. Outgrowing the initial goal of doing repetitive mathematical calculations, computers have already permeated almost all spheres of human activities. *Software not only supports, but also facilitates the reorganization of business and/or production process itself. Similarly, the social networking is now helping to transform and create new form of human-social interactions.*

After decades of experience in creating and using software solutions, few domains, especially those that are related to science, engineering, governance and business, are more mature than

many others in terms of understanding of domain specific software possibilities both by software developers as well as concerned domain experts. Many new domains are fast emerging as big users of software. Both domain as well as IT experts are exploring new possibilities of creating IT enabled operations and services in these domains. A large number of *new users and applications* are emerging in the domains of *business analytics, mass communication, customer relationship management, social marketing, security, energy management, environment management, compliance governance and risk management, healthcare, life sciences, and collaborative work. A significant number of novel applications are being developed for arts and sports as well.*

In our 2009 survey on required competencies for software developers, twenty software professionals assigned 'technical/domain competency' an average rating of 2.95 on a scale of 0-4. A majority of 60% of these respondents recommended it to be a critical or very important competency with respect to the requirements of software developers' multi-faceted professional activities. In order to acquire required domain knowledge in varied application domains, a broad understanding of various processes and diverse human tasks is very helpful for software designers.

With reference to Appendices A2 and A3, domain competence of software developers also relates to the following:

1 "Be the customer" mentality
2 Analytical skills
3 Design skills
4 Imagination: storyboarding, extrapolation, visualization, cognitive flexibly: ability to transfer and models of solutions of one situation/field to another, multi-perspective thinking, lateral thinking, creativity and idea initiation, and innovation
5 Problem solving skills
6 Project planning

Armour [120] [148] viewed *software development as a learning activity*, rather than a production activity, and advocated that *software developers need more training in domains, learning, and knowledge structuring mechanisms rather than in software itself.* Effort to acquire required domain knowledge varies from few days to several decades depending upon the complexity of the problems. Domain training, and even certifications, have become common part of continuous training programs of software developers. Since the application domains are now virtually encompassing all kinds of human activities, it should be presumed that a fresh graduate is required to work in a new domain, to start with.

It is very common for software teams to do lot of rework because of insufficient understanding of the application domain. Lack of domain knowledge is a very significant problem in software projects and because of his deficiency, the *requirements appear to fluctuate* [168]. Domain specific knowledge enables developers to identify problems in logic [157]. Most software developers generally have a tendency to blame the users for fluctuating requirements. Hence, domain knowledge is well recognized as the key contributor to enhancing productivity of software development processes.

The software development processes essentially try to map the application domain requirements to programming constructs. Shirley identified four levels of skill in student programmers' work: expedient, constructional, operational, and structural. Using the structure of the problem to devise the solution is considered the most sophisticated approach to programming [169]. A student at the structural level of programming skill first carries out an interpretation of the problem within its domain, then structures the problem before coding.

Many domain specific languages (DSLs) have been developed and continue to evolve for various domains. Domain driven design approach is becoming increasing popular among software designers. It is based on the premise that primary focus of software designers should be on the domain and domain logic, rather than on the particular technology used to implement the system. In general, students need to learn to capture the processes involved in a domain, identify the actors, events, schedules, compliances, etc., map the information flow, decision making based on information, identify the gaps and redundancies, optimize the processes through business and

process re-engineering, and most importantly, substantiate the value delivery of IT in the enterprise.

Domain specific conceptual knowledge and technical skills are important aspects of the domains. However, *domains are characterized by prominent thinking processes*. Different kinds of thinking processes are prominent in different domains. Increasingly, software is being developed to *support the cognitive processes of domain specialists both for analytical as well as creative tasks.*

Software is no more limited to only providing rapid and reliable data storage/transfer/access. It is increasingly transforming computers as *cognition support systems* through various devices for data transformation, analysis, and synthesis. In order to develop *domain specific cognition support systems, an understanding of domain specific cognitive tasks is essential for the software developers*. A sound understanding of a specific domain enables software developers to look for problem cases, failure modes, and benefits to the actual users.

Computing does not just open new ways of doing domain specific activities; it also requires new conceptualizations of the domain that require automated processing. The opportunities of automated processing in turn further open new ways of re-conceptualization in application domains [176]. Sometimes, software developers can also help in inventive problems solving in specific domains by *infusing different thinking patterns developed through their experience in other domains*.

Breadth and Diversity

Diversity of Disciplines

In 1973, *Biglan classified academic disciplines along three dimensions*. Each of these dimensions were broadly classifies into two categories - *hard vs soft, pure vs applied, and life vs non-life* [170-173a]. Hence, for our purpose, we will treat these dimensions as bi-level. As per this classification, hard disciplines follow a single common paradigm, whereas the experts of soft disciplines differ in their methodologies and concepts. Table 4.2 summarizes this classification.

Table 4.2: Biglan's classification of disciplines

	Hard		Soft	
	Life	**Non-life**	**Life**	**Non-life**
Pure	Biology, Biochemistry, Genetics, Physiology, etc.	Mathematics, Physics, Chemistry, Geology, Astronomy, Oceanography, etc.	Psychology, Sociology, Anthropology, Political Science, Area Study, etc.	Linguistics, Literature, Communications, Creative Writing, Economics, Philosophy, Archaeology, History, Geography, etc.
Applied	Agriculture, Psychiatry, Medicine, Pharmacy, Dentistry, Horticulture, etc.,	Civil Engineering, Telecommunication Engineering, Mechanical Engineering, Chemical Engineering, Electrical Engineering, Computer Science, etc.	Recreation, Arts, Education, Nursing, Conservation, Counseling, HR Management, etc.	Finance, Accounting, Banking, Marketing, Journalism, Library And Archival Science, Law, Architecture, Interior Design, Crafts, Arts, Dance, Music, etc.

The hard-pure disciplines are concerned with universals and simplification, whereas soft-pure disciplines are concerned with particular cases. The thinking approaches significantly differ for these categories. The hard-pure disciplines have an atomistic approach and rely more on linear logic, facts, and concepts whereas soft-pure disciplines have a holistic approach, and rely more on the breadth of intellectual ideas, creativity and expression. The hard-applied disciplines focus on problem solving and application of knowledge to create products and techniques, whereas, soft-applied disciplines focus on personal growth, reflective practice, and lifelong learning to create protocols and procedures. The hard-pure disciplines are concerned with mastery of physical environment, whereas soft-applied are concerned with enhancement of professional practice.

As per this classification, computer science is classified in hard, non-life, applied category of disciplines. The algorithm design and programming part of software development surely belongs to this category. However, software development also includes many other tasks like project management, requirement analysis, user interface design, and usability analysis. These tasks relate to people, and *hence software developer like engineering or computer science cannot be classified to the single category as per this classification*.

Application domains of software include all disciplines and hence are well spread over all categories of Biglan's classification. Hence, it is imperative for software developers to

understand the concerns, focus, aim, knowledge structures, and thinking approaches of application domains that belong to all categories of Biglan classification.

Unlike traditional engineering based waterfall model, evolutionary approaches to software development view *users' requirements as tentative, evolving, and open to change*. Paulsen and Wells [174] found that as compared to the students of pure fields like science, fine arts, social science, and humanities, the students of applied areas like business and engineering hold more naïve beliefs about the structure of knowledge and speed of learning. *Engineering students were also found to have more naïve views about certainty of knowledge.* Theirs, and earlier, research showed that as compared to engineering students, the students of soft fields like social science, fine arts, humanities, education, and business are more likely to view knowledge as diverse, tentative, and open to change. Hence, a *good grounding in soft disciplines becomes even more important* with respect to current and emerging trends of software development methodologies. The teaching of computing courses also needs to be restructured in order to develop these epistemological beliefs.

Because of the nature of their curriculum, engineering students in computing disciplines are already well over-exposed to the approaches of hard-pure and hard-applied disciplines. This tends to limit their perspective and approach. *Hence, it is strongly recommended that engineering students of computing disciplines are well exposed to soft-pure and soft-applied disciplines also, especially as application domains. We have further developed and included this approach in our proposed framework of pedagogical engagements in software development education (Table 8.7 and Table 8.8).*

Good observation skills, enquiring mind, diversity of interests, empathy, and reflection skills, are the prerequisites for building the required domain understanding. The software development education program needs to expose the students to diverse types of domains and domain categories as per Biglan's classification. It also needs to nurture the ability to learn nuances of newer domains. This exposure to application domains needs to introduce the students to specific attributes: (i) context: users, functions, concerns, constraints, compliance requirements, (ii) operations: procedures, practices, methods, evolution of IT applications, (iii) domain specific

vocabulary and its semantics, (iv) domain experts' cognitive tasks, and (v) challenges: complexities, risks, uncertainties, and complications.

Diversity of Learning Styles

Kolb [152] identified four main learning styles. Kolb also discovered prominent patterns of correlation of the styles with respect to domains, and also with concerned persons' functions [175]. These four styles are given in Table 4.3. Rather than following the commonly popular perspective that subjects are linked with specific learning styles, *we take a position, that different styles are relatively more suitable for learning different aspects of a single subject. Hence, an integration of these styles enhances learners' ability to learn different aspects of any domain*. Kolb proposed 'experiential learning cycle' for facilitating deeper learning.

A liberal arts kind of broad based educational model that includes exposure to diverse disciplines, not just science, mathematics, engineering, or management is potentially suited to make the students more ready for developing software for diverse domains. Breadth of multi-disciplinary exposure also offers the opportunity to enrich and diversify students' repertoire of learning style. However, such multi-disciplinary courses will be effective, only if they succeed in *engaging the students in prominent cognitive tasks with the domain experts to some extent*. We use Kolb's style to enrich our proposed framework of pedagogic engagements in section 8.3.1.

Table 4.3: Kolb's learning styles

1. Divergent: involves reflection on concrete experience, requires abilities of concrete experience as well as reflective observation. This style is associated with valuing skills: relationship, helping others, and sense making. Such people have broad interests and tend to be imaginative and specialize in arts, literature, psychology, etc. Effective communication and relation building requires this style.
2. Convergent: involves active experimentation to test/apply abstractions, requires abilities of abstract conceptualization as well as active experimentation. This style is associated with decision skills like quantitative analysis, use of technology, and goal setting. Such people like to deal with technical rather than people related aspects, and tend to specialize in technology and medicine. Bench engineering and production requires this style.
3. Accommodative: involves active experimentation on concrete experiences, requires abilities of concrete experience as well as active experimentation. This style encompasses a set of competencies that can best be termed acting skills: leadership, initiative, and action. Such people tend to specialize in education, social service, sales, communication, nursing, etc. Decision making in uncertain situations requires this style.
4. Assimilative: involves reflection on abstractions; requires abilities of abstract conceptualization as well as reflective observation. This style is related to thinking skills: information gathering, information analysis, and theory building. Such people tend to specialize in mathematics and physical sciences. Planning and research activities require this style.

Pedagogic Perspective

In **our recently concluded survey,** "Software developers - (How) Did your college help you in your development?" (Table A10.2 (i) part-II, Appendix A10), a large fraction of 61% felt that as compared to all other kind of academic engagements, their projects did much better to develop their domain competence. This was followed by research literature survey and knowledge transmission oriented lectures (51% each), laboratory work (39%), homework (35%), written examinations and mentoring juniors (31% each).

Many universities have started domain specific computing degree programs at master's level e.g., bio-informatics, digital arts, computational finance, computational economics, health informatics, computational mathematics, computational physics, computational social science, computational psychology, archive and museum informatics, etc. In such programs the non-computing discipline is strongly shaped and heavily influenced by computing principles. Typically, the students in such programs have an undergraduate degree in the domain. In 2005, we also proposed the design of a two year master's program in *Archaeo-heritage Informatics* [177], given in Appendix A7.

Recommendations for Breadth Courses for Developing Domain Competence

Training in general systems thinking helps in quickly understanding even unfamiliar areas [178]. Weinberg considered that linguistic and mathematical competencies are essential foundations for general systems thinking. In Section 6.3, we further elaborate upon system thinking.

Broad based education in diverse disciplines is likely to enrich linguistic sensibility and competence. While computing courses need to bring a higher focus on a systems approach in their delivery, the *breadth courses in other disciplines can also very significantly contribute to develop general systems thinking by specifically bringing it as one of the prominent learning objectives*. In order to help in general systems thinking, the courses need to be selected and redesigned with this aim. In order to develop system thinking and ability to learn a new domain, the breadth courses too should try to enhance their focus on: (i) diversity and multi-perspective thinking, (ii) inter-disciplinary integration and applications, (iii) and systems approach.

Repeated exposure to *complexity, complications, nonlinearity, uncertainties, and risks,* and as highlighted and illustrated within the context of each of the specific breadth courses is likely to significantly enhance their ability to understand the nuances of unfamiliar domains, and also to orient their mindset to decision making in complex situations.

Within the context of many knowledge disciplines in sciences, mathematics, engineering, management, social sciences, and humanities, a body of knowledge has already been created around systems and systems thinking. Appendix A8 suggests some such breadth courses that can develop and reinforce systems thinking and help in developing the ability to learn new domains.

Section 4.4: Software Developers' Education for Development of Communication Competence

There are several kinds of communication in a software development project [161]. These include communications (i) between the development team and the customers, (ii) between the developers and the project manager, and (iii) among the developers. A typical project manager in IT spends about 90% of the time in communication with various stakeholders. Often communications competence is misinterpreted as making exciting presentations or impressive speaking or writing skills. *However, the communication needs of software developers are very different from the communications needs of sales or marketing professionals*. Communications skills do not make up for the deficiency in thinking ability. Good communication requires keeping track of who, what, when, and why. It mainly involves *listening with understanding and empathy*.

The communication competence of software developers encompasses the need to communicate their difficulties and vision to their clients, management, colleagues, and end-users, and also preparing technical documentation, and also end-user documentation. One needs to keep himself in the shoes of the end-user to give a useful product. *Communication encourages the exchange of ideas and project related knowledge among the people engaged in the project: clients, managers, and developers.*

Communication among project participants is formalized through various documents. The forms of documentation include requirements, specifications, architectural documents, detailed design documents, quality documents, and also low-level design information such as source code comments. Hence, effective verbal and written communication skills are also essential for software developers.

In our 2009 survey on required competencies for software developers, twenty software professionals assigned 'communication skills' an average rating of 2.75 on a scale of 0-4. A majority of 65% of these respondents recommended it to be a critical or very important competency with respect to the requirements of software developers' multi-faceted professional activities.

With reference to Appendices A2 and A3, communication competence of software developers also relates to the following:

1 Ability to work in teams
2 Listening skills
3 "Be the customer" mentality
4 Persuasion, negotiation, consensus building, and conflict resolution skills.
5 Mentoring, coaching, and training skills
6 Organizational skills

Agile manifesto emphasizes face-to-face communication over written documents. Extreme Programming (XP) relies on four values: simplicity, communication, testing and courage. Chau et al [162] posit that software engineering is a knowledge-intensive process with a very strong need for *knowledge sharing support* to enable software organizations to:

7 effectively share domain expertise between the customer and the development team,
8 identify the requirements of the software system,
9 capture non-externalized knowledge of the development team members,
10 bring together knowledge from distributed individuals to form a repository of organizational knowledge, and

11 retain knowledge that would otherwise be lost due to the loss of experienced staff; and

12 improve organizational knowledge dissemination.

They observe that while traditional software development approaches support knowledge sharing primarily by documents or repositories, agile approaches rely heavily on socialization through communication and collaboration.

Cockburn [163], one of original authors of the agile manifesto, highlighted the following principles regarding communication in setting and running of software projects:

1 Larger teams need more communication elements.

2 Interactive, face-to-face communication is the cheapest and fastest channel for exchanging information.

3 Increased communication and feedback reduces the need for intermediate work products.

Outsourcing and offshore development has added to the communication related challenges in the software development process. Documentation becomes more important with offshore development. Xiaohu et al [161] cite previous research that had shown that because of *communication and coordination issues, distributed projects take about two and half times longer to complete as similar projects where the project team is centralized*.

Today's software development situation sees two types of trends: document driven outsourcing, and offshore project and agile approaches emphasizing a lot of face-to-face communication. Attempt to blend these two are also underway.

Responding to our online polls (results summarized in Table 4.6), one of the senior-level respondents wrote, *"Software development life cycle is completely dependent on the communication effectiveness. Communication tends to break at every stage of the software development life cycle. Hence communication skills (mainly comprehension and listening) are of paramount importance in software engineering."* Another senior level respondent recalled his

customer saying "*being convinced about giving the work to you is a challenge because we are not sure if your team has accurately understood our requirements.*"

Christiansen [164] has concluded that these additional challenges arise because of factors like *different cultures, different languages and accents, thin communication channels, different platforms, and different time zones.* Christiansen has also given *some suggestions to overcome these challenges*: put stress on synchronous communication, adapt to and understand other cultures, put emphasis on spoken language skills, rotate people between shores, use artifacts properly, align IT infrastructure, use requirement specifications with care, and invest time and money in transferring implicit knowledge.

Backer et al [165] had studied aspects related to competency requirements for computing professionals with respect to various *micro-level communications skills: writing, reading, speaking, listening, presentation, and nonverbal*. They had also studied extent of engagement of technical professionals in these micro level communications. *With reference to software developers, communication involves frequent translations between the domain/business descriptions to/from technological descriptions, in both directions.*

The braintrack.com provides the summary of responses of the perceived importance of communication abilities for various professionals. Table 4.4 summarizes their finding with respect to programmers and systems analysts.

Table 4.4: Perceived importance of communication skills by programmers and systems analysts (Source: braintrack.com)

Communication skill	Respondent programmers who find it important for their work	Respondent systems analysts who find it important for their work
Written comprehension	77%	66%
Oral comprehension	66%	63%
Oral expression	60%	60%
Written expression	56%	56%

In order to understand the relative importance of micro-level communication skills for software developers, we conducted two polls among software developers.

The first poll, Poll-A, asked the respondents to choose the most important micro-level communication skill with reference to the requirements of software development work. The micro level communication skill considered included: (i) speaking with clarity, (ii) making impressive presentations, (iii) writing with clarity (iv) reading with comprehension, and (v) listening with understanding and empathy.

With respect to these micro level communication skills, the second poll, Poll-B, asked them to identify most serious weakness of typical Indian engineering graduates. We respectively received 84 and 69 responses for these two polls. Table 4.5 gives the summary of the profile of the respondents.

Table 4.5: Profiles of the respondents for the two polls about communication competence among software developers

		Poll-A **Importance** (84 responses)	Poll-B **Weakness** (69 responses)
Age	>55 years	Nil	5%
	35-54 years	42%	26%
	25-34 years	54%	58%
	18-24 years	4%	11%
Company size	Enterprise	29%	36%
	Large	29%	29%
	Medium	21%	14%
	Small	21%	21%
Job function	Consulting	21%	38%
	Engineering	37%	31%
	Product	21%	15%
	Sales	14%	Nil
	Academics	7%	8%
	IT	Nil	8%

Table 4.6 summarizes their responses.

Table 4.6: Summary of responses for these two polls about communication competence

Micro level communication skill	**Poll-A Importance** (84 responses)	**Poll-B Weakness** (69 responses)
Listening with understanding and empathy	55%	37%
Writing with clarity	16%	14%
Speaking with clarity	15%	33%
Reading with comprehension	7%	5%
Making impressive presentations	4%	8%

The responses show a very interesting aspect. All the respondents who identified "making impressive presentations" as the most serious communication skill related weakness of Indian engineering graduates, belong to the youngest age group of 18-24 years. It is further very interesting to note that 50% of the respondent in this age group thought so. In the last few years, there has been a sudden increase in the emphasis on communication skills; often this is misunderstood in terms of making impressive presentations. Our poll shows the extent of this misconception among our fresh engineers.

One of the responding enterprise IT architect from India commented, *"Indian engineering graduates tend to just skim the surface and are not that well prepared during discussions and meetings."*

Pedagogic Perspective

Many studies have showed that multi-paradigm disciplines like humanities, social sciences, and psychology had a positive influence on self-reported growth in communication skills by students. However, Li et al have found that self-perceived gains in communication skills most significantly depended upon the *degree of their integration into the social community of the university rather than their discipline of study* [166]. Further, the quality of curriculum was found to be the most significant factor for influencing their social integration.

In **our recently concluded survey,** "Software developers - (How) Did your college help you in your development?" (Table A10.2 (i) part-II, Appendix A10), a large fraction of *84% felt that, as compared to all other kind of academic engagements, their discussions with other students did much better to develop their communication skills. This was followed by mentoring juniors and discussions with faculty (71% each), discussion with others (51%), and industrial training (45%).* All other engagements were found to be inadequate in this regard by the respondents.

Etlinger [167] has suggested a *framework for teaching communication skills to computing students*. This framework has three concentric circles. The inner circle is about the critical ideas: (i) purpose - inform, instruct, or persuade, (ii) strategy- form of communication, organization of information, and tone of communication, and (iii) audience- hostile or receptive, supportive or

neutral, internal or external, interested in the entire artifact or only in part of it. The middle circle of skills includes: reading and writing, listening and speaking, reviewing and evaluating, and, thinking. The outer circle focuses on process issues, more general traits, and quality.

Rather than only depending upon the communication skill related courses offered by humanities, language or management department, many authors have experimented with the strategy of including improvement of communication skills as one of desired learning outcome of their regular computing courses and also project work. Many view project work as an effective way of improving desirable kind of communication skills. At some universities, special courses on technical communication have been offered by the computing faculty.

The communication competence for software developers is significantly different from the communication competence for sales professionals. It is essential for understanding the needs of the consumers, the difficulties of their clients and co-developers. It is required for knowledge acquisition as well as knowledge sharing. *We posit that active and collaborative engagements as included in our proposed taxonomy of pedagogical engagements in Sections 8.3.1 and 8.3.4 respectively contribute towards developing the required communication competence of computing students.*

Section 4.5: Software Developers' Education for Development of Complex Problem Solving Competence

Programming and Problem Solving

Gomes and Mendes view *programming as problem solving* [190]. Booth identified conceptions of programming and 'learning to program' generally held by students [169]. As per her model, *a student's conception about programming* grows in sophistication from the initial level of computer related activity to problem oriented activity to product oriented activity. She also identified the stages of increasing sophistication in a student's conceptions of 'learning to program': (i) learning a programming language, (ii) learning to write a program in a programming language, (iii) learning to solve problems in the form of programs, and (iv) *becoming part of the programming community.*

Further, the *conceptions related to programming languages* grow in sophistication from simplest levels of viewing a programming language as a utility program that enables programs to be written, to the second level of code as a set of instructions, commands, symbols, and constructs. The third level in this order is viewing programming language as a means of communication between programmer and computer to enable communication between computer and user. The *highest level views programming language as a medium of expression for the programmer to express solutions*.

In our 2009 survey on required competencies for software developers, twenty software professionals assigned 'problem solving ability' an average rating of 3.2 on a scale of 0-4. A large majority these respondents (80%) recommended it to be a critical or very important competency with respect to the requirements of software developers' multi-faceted professional activities.

With reference to Appendices A2 and A3, complex problem solving of software developers also relates to the following:

1 Ability to convert ill-defined problematic situations into software solvable problem
2 Problem orientation, problem definition and formulation, generations of alternatives
3 Emphasis on elegant and simple solutions
4 Ability of infusing different thinking patterns developed through their experience in other domains
5 Inclination for reuse and synthesis by integration
6 Solution implementation and verification
7 Project planning and management
8 Sense of urgency and stress management

Good Solutions

Conceptualizing programming to become part of the programming community and conceptualizing programming language as a medium of expression of solution are indicators of the possibility of multiplicity of solutions for the same problem. The solutions may suffer from

shortcomings like over-simplification, over decomposition, under-decomposition, or disordered management of complexity in a disordered manner. It is not sufficient to somehow solve complex problems. The need of elegance, i.e., ordered management of complexity, increases with increasing number and diversity of items, relations, correlations, and systems of relations. Elegant management of complexity affords overall comprehension and continuous orientation. Like good literature, architecture, or some other work of art, elegant software exhibits clarity, simplicity, precision, minimized interfaces, orderliness, coherence, and consistence, without compromising on integrity and performance. Avoiding unnecessary complications is a necessary requirement for elegant solutions. Software developers' aesthetical sense, **urge for elegance**, patience, and **systems–level perspective** are necessary driving force for imbibing elegance in their solutions.

Developing elegant software also requires good understanding of the specific problem and also its context. Multidimensional complexity of application domains has to be registered, facilitated, and expressed in software constructs. Hence, not just technical and computational thinking competence, but domain competence is also necessary for developing elegant software.

Good software also includes defense, error-handling, and recovery mechanisms for various kinds of errors: hardware-level, programming, or user induced. It also affords testability and portability.

Problem Solving

In a study [191], almost unanimously, i.e., 97.7% of 1023 experts rated 'problem solving' as an important element of human intelligence. In its most simplistic interpretation, a problem is something that cannot be solved in a single, obvious step. Gomes and Mendes also provide some of the following interesting definitions of problem and problem solving:

Pérez et al. - problem is a situation for which there isn't an evident solution.

Perales - problem is any situation that produces, on one hand, a certain degree of uncertainty and, on the other, behavior in search of a solution.

Gagné - problem solving is a process where the apprentice/learner discovers a combination of rules previously learned that he/she can apply to reach a solution for a new problematic situation.

Nickols [192] opines that what characterizes a problem is *uncertainty about action*, having a goal and not knowing how to achieve it. Problem solving depends upon *cognitive processes* of problem anticipation and identification, problem understanding, problem definition, problem formulation, problem representation, generations of alternatives, decision making and planning, implementation and integration, monitoring, evaluation, improvisation, and solution communication.

Jonassen [193] has proposed a taxonomy of problems based on variations in problem types and representations. The problem types vary in a three dimensional continuous space of three factors: *structured-ness*, *complexity,* and *degree of domain specificity*. *Software problems are domain specific, complex, and ill structured.* Based on the cognitive task analysis of various kinds of problems, Jonassen has identified eleven different kinds of problems. *Software developers typically deal with all these kind of problems. Nickols' typology of problem solving approaches* [198] comprises of (1) repair approach, (2) improvement approach, and (3) engineering approach. Software developers mainly adopt the later two approaches, because seemingly, repair problems in software systems are actually improvisation problems. Annexure AN6 gives more details of these models.

Expert programmers are found to be good at logical thinking; many of them also enjoy puzzle-solving [194]. Metzger [157] has viewed debugging as a search problem like mathematical problem solving that is solved using a variety of search strategies like binary search, greedy search, depth-first search, and breadth-first search. In his classic, 'How to solve it,' famous mathematician Polya [195] listed four phases of problem solving: (i) understand the problem, (ii) plan the solution, (iii) execute the plan, and (iv) review the results. More details are given in Annexure AN6.

Drawing analogies between debugging and mathematical problem solving, Metzger [157] explains many *heuristics for solving debugging problems*: (1) stabilize the problem, (2) create a

standalone test case, (3) categorize the problem with reverence to correctness, completion, robustness, and efficiency, (4) describe the problem according to a standard methodology, (5) explain the problem to someone else, (6) recalling a similar problem, (7) drawing diagrams like control flow graph, data flow graph, and complex data structures with pointers, and (8) choosing a hypothesis from historical data. Further, he has also suggested some strategies like *program slice strategy, deductive reasoning strategy,* and *inductive reasoning strategy* for debugging.

Cognitive psychologists have studied problem solving methods for the last few decades. Galotti collates some general domain independent techniques for puzzle-like problems [196]. These include: generate and test, means-ends analysis, working backward backtracking, reasoning by analogy. Annexure AN6 gives more details of these.

Creative Problem Solving

Stoycheva and Lubart [197] have elaborated upon a creative problem solving process. This process involves five main activities of data finding and mess finding, problem finding, idea finding, solution finding, and acceptance finding. The first activity of data and mess finding is carried out by collecting data from the senses, experiences, knowledge, feelings, opinions, emotions, memories, fantasies, future projections, interaction with others, information on the social roles, and situation. Data collection can also be purposefully unstructured, random, and divergent. Data and mess finding also involves evaluation of relevance, interconnectedness, and importance of collected data. Through the analysis of this data, mess is discovered, created, or recreated.

The second activity of problem finding (or problem defining) is most creative in the problem solving process. Creative people try out many formulations and interpretations until one is found that best fits the data and offers the best opportunities for solving the problem. In this reference, Nickols [198] insists on defining the problem and also the solution state. He recommends to clearly detailing out the boundaries, distinguishing characteristics, the nature, and meaning of solution state. He [199] also insists on defining objectives and goals through inquiring about what are we trying to achieve, preserve, avoid, or eliminate?

The third activity of creative problem solving, idea finding, aims to filter out the most promising options which are identified for further elaboration. It involves multi-perspective thinking about concepts and experience. The fourth activity of solution finding is about examining selected alternatives from multiple perspectives for their pluses, minuses, and other interesting aspects. It involves exploring and finalizing the criteria for evaluation of alternatives. Further, alternatives are evaluated using the chosen criteria, and the most appropriate is chosen for implementation. Finally, the activity of acceptance finding is about successful implementation. It also requires envisaging how different stakeholders will react to the innovation.

With reference to abovementioned creative problems solving process, *we recently conducted a LinkedIn Poll among software professionals*. Seventy-six software professionals responded to this poll. The respondent professionals were well distributed in terms of age and job functions. In all, 7% respondents were older than 55 year, 33% were in the age group of 35- 54, 47% were in the age group of 25- 34, and 13% belonged to the age group of 18-24 years. In terms of job function, their distribution was: 13% in consulting, 38% in engineering, 38% in product, and 13% in creative functions. All respondents belonged to large or enterprise organizations.

With respect to the problem solving skills for software work, the respondents identified the most serious weakness of Indian engineering graduates as follows:

a. *Idea and/or solution finding (36%),*
b. *Problem (re)formulation (22%),*
c. Implementing (18%),
d. Mess identification (12%), and
e. Stakeholders' acceptance (12%).

Some respondents also commented as follows:

> *"…engineers tend to assume the cause of the problem and jump to solutioning. This usually leads to compounding the situation. Engineers tend to overlook the importance of problem assessment, analysis and ascertation."*
> *"…our grads are enthusiastic and want to provide a quick fix to the problem, which is preventing them from thinking in terms of the "5 Why's"…"*

It shows the importance of creativity with reference to problem solving through software development. This issue is discussed in details again in Section 5.3.

Section 4.5.1: Expert problem Solvers

Research on expertise has shown that it takes approximately *ten years to turn a novice into an expert*. *Hence, the four year undergraduate education needs to prepare the student to make the rest of the progress.* In early 1970s, Gordon Institute proposed a famous four-stage conscious competence theory. As per this theory, the competence has four stages: *unconscious incompetence, conscious incompetence conscious competence, and unconscious competence*. Nonaka added a fifth stage to this and called it *reflective competence* [200].

Winslow [201] refers to the five levels of expertise as suggested by Dreyfus and Dreyfus in 1985. These are the levels of *novice, advanced beginner, competence, proficiency, and expert*. In the specific context of computing professionals, Denning [202] has refined Dreyfus levels, has added two more levels (master and legend) after expert. *We have merged Gordon Institute's and Denning's levels into a single ladder. The merged levels are shown in Table 4.7. First seven levels of this are included in our proposed framework of pedagogical engagements in software development education (ref: Table 8.2, first column).*

Table 4.7: Competency ladder (Integrating the ladders by Gordon Institute, Dreyfus and Dreyfus, and Denning)

Level	Description with respect to software professionals
1 Unconscious incompetence	Does not recognize the competency deficit, nor desires to learn.
2 Conscious incompetence	Recognize the competency deficit, without addressing it.
3 Novice (beginner)	Aims to learn objective facts, features, and rules for determining actions without being context sensitive. Focus on syntax etc. *Learn through memorization and drill*.
4 Advanced beginner	Recognizes common situations that help in recalling which rules should be exercised, starts to recognize and handle situations not covered by given facts, features and rules *Learns through problem solving* and repeated practice with common situations.
5 Entry-level Professional (competent)	Performs most standard actions without conscious application of rules after considering the whole situation. Handles new situations through appropriate application of rules, can design systems. May lead. *Learns through advanced problem solving, projects, extensive practice in common and exception situations, and participation in professional networks.*
6 Proficient professional	Effortlessly deals with complex situations, no longer has to consciously reason through all the steps to determine a plan, appropriate actions come from experience and intuition. Design and mange complex systems, ingenious solutions. *Learns through apprenticeship to experts,* coaching, putting self into wide range of situations, membership and contributions to professional networks. Teaches others.
7 Expert	Consistently inspiring and excellent performance. An expert generally knows what to do, base upon mature and practical understanding. Performance standards are well beyond those of most practitioners. Extensive experience with large systems, appreciate subtle and indirect design issues and customer concerns, leads well. High productivity. *Learns through apprenticeship to masters, advanced coaching, and development of breadth*. Years/decades of experience.
8 Master	Capacity for long range strategic thinking and action. Sees historical drifts and shifting clearings. Has developed a *distinctive style*. Has *produced innovations*, altered the course of history in the field. Teaches others to be experts and masters. Develops new methods, admired for long. *Learning by working with other masters. Creates and leads professional networks.*
9 Legend	Has attained high standing. Work has widely accepted impact. *Shapes directions of the field.*

Costa and Kallick [203] have identified sixteen characteristics of what intelligent people do when they are confronted with problems, the resolution to which is not immediately apparent. These are listed in Annexure AN6.

Problem solving requires cognitive and meta-cognitive processes and also *affective and conative elements of self-confidence, perseverance, open-mindedness, motivation, and mindful effort. Meta-cognitive* aspects have been discussed under 'critical and reflective thinking.' The affective and conative elements are elaborated in sixth chapter.

Galotti [196] describes some findings related to factors that hinder problems solving. *Mental set* is the tendency to adapt a certain framework, strategy, or procedure, or more generally, to see things in a certain way instead of another. It causes people to make certain unnecessary

assumptions even without awareness. *Incomplete or incorrect representations* make problems solving much harder.

Jonassen [193] and Galotti [196] have discussed the individual differences in problem solvers. The *prior experiences* of problem solvers enrich their *mental corpus of problem schemas*, enabling them to recognize different problem states, and move faster towards implementation. *Expert programmers* have been found to have this characteristic [204]. They are persistent, and their mental models of their program comprehension exhibit the following *five characteristics*: *hierarchical and multilayered, explicit mapping between layers, recognition of basic patterns, well-connected internally, and well-grounded in the program text*. They also choose and mix their richer mental models in an opportunistic way [201].

Experts in any domain are able to more easily *pick up more perceptual information, recognize more patterns, create more hypotheses, perform skills, and also represent the problems at more deeper and abstract levels*. *Expert programmers* have *good problem solving skills, determination, and persistence.* They gather clues, in the form of facts and information to help in problem solving, and are also efficient planners [194]. They are also more likely to reflect and check errors in their thinking. Expert programmers have the habit of breaking down the problems into minor sub-problems [205].

Problem solvers with higher cognitive flexibility [206] and cognitive complexity can consider more alternatives, and hence, are better experts. The epistemological beliefs of the problem solvers about the nature of problem solving also affect their natural ways of approaching the problems. The stages of cognitive development discussed later in Section 6.1 effect these beliefs.

Pedagogic Implications

Jonassen [193] and Linda S. Gottfredson [207] have consolidated earlier research on problem solving and highlighted the distinctions between academic and practical problems. These differences are given in Table 4.8.

Table 4.8: A Comparison of typical academic and real life problems

Academic Problems	Real life practical problems
1. Tend to be formulated by other people	1 Require (re)formulation.
2. Well-defined or well-structured	2 Ill-defined or ill-structured
3. Tend to be complete. Presented with all the parameters and constraints. Usually consist of a well-defined initial state, a known goal state, and a constrained set of logical operators.	3 Require information seeking. One or more elements of the ill-defined problem are unknown or not known with certainty. The goals of real-life practical problems are usually vaguely defined with unstated constraints.
4. Typically posses only a single answer	4 Usually possess multiple acceptable solutions.
5. Tend to encourage single method of obtaining a correct answer	5 Allow multiple paths to solution.
6. Require application of a finite number of concepts, rules, and principles	6 Present uncertainty about useful and usable concepts, rules, and principles as well. Further, in case of ill-defined problems, the relationships between concepts, rules, and principles may be inconsistent between cases.
7. Divorced from ordinary experience	7 Embedded in and require prior experience. This requires the problem solver of ill-structured problem to distinguish important from irrelevant, and construct a problem space for generating solutions.
8. Tend to be of little or no intrinsic interest	8 Require motivation and personal involvement

Real-life ill-defined problems are not constrained by the content domain, may require the integration of several content domains, their solutions are not predictable or convergent, possess multiple criteria for evaluating solutions, and no explicit means for determining appropriate action. They require the solver to express personal opinion or belief, make judgments, and also defend them. Earlier it was believed that experiences with well-defined problem solving easily transferred to solving ill-defined problems. However, research in problem solving has demonstrated that performance on well-defined problems is not correlated with performance on ill-defined problems.

In **our recently concluded survey,** "Software developers - (How) Did your college help you in your development?" (Table A10.2 (i) part-II, Appendix A10), a large fraction felt that as compared to all other kind of academic engagements, their student projects (78%) did much better to develop their problem solving skills. This was followed by laboratory work (59%), thinking oriented lectures (51%), discussions with other students (49%), homework (37%), research literature survey (36% each), industrial training (33%), and discussions with faculty (31%). Discussions with others and traditional knowledge transmission oriented lectures were found to be least effective in this regard by the respondents.

Complex Problem Solving Techniques

Literature on ill-defined problem solving offers some excellent general purpose techniques that have been used in various professions, especially management and design. These techniques essentially help in analysis of complex ill-defined problems. Some of these techniques are given in Table 4.9.

Table 4.9: Some techniques for solving complex ill-defined problems

1. Flow charts (understanding how a process works)
2. Concept mapping
3. Systems diagrams (understanding the way factors affect one-another)
4. SWOT (Strength, Weakness, Opportunity, and Threat) analysis
5. Appreciation (extracting maximum information from facts by repeatedly asking 'so what?')
6. 5 Why's (asking "Why?" five times, successively, to understand the ultimate root cause),
7. Cause and effect diagram (identifying possible causes of problems)
8. Affinity diagrams (organizing ideas into common themes)
9. Appreciative inquiry – 4D approach (solving problems by looking at what's going right in four phases of problem solving: Discovery, Dream, Design, and Deliver)

Many of these techniques, especially, flow chart, system diagram, 5 why's, and cause and effect diagram are already being used by many software developers in the industry. Flow chart and systems diagrams are already being used in some computing courses. Various kinds of conceptual modeling diagrams, especially UML diagrams are used by software designers. Metzger [157] recommends the usage of Nassi-Shneiderman diagram and Warnier-Orr diagrams for program design conception stage. Most of the other techniques in above list can also be very effectively used by all software developers for various activities of software development. Hence, computing students should be well exposed to these techniques through their curriculum. Integration of these and some other similar techniques in software development education is on our future agenda. Active engagement in our proposed framework of pedagogic engagements incorporates using these techniques in various problem solving activities (ref: Table 8.5). Further, they should also need to learn to adapt existing techniques, and if required, also develop new techniques especially diagrammatic techniques.

Further, with reference to software development, analyzing and solving complex ill-defined problems usually requires approaching problems and solution from a systems-level perspective. The details of systems-level perspective are discussed in Section 6.3. Evolutionary nature of software development also makes it necessary to continuous reflect upon the problem and iterate

over the solutions. Hence, in the context of software development, reflective thinking, discussed in Section 5.2 is also very important for complex ill-defined problem solving.

Section 4.6: Chapter Conclusion

In this chapter we discussed that the basic competence for software developers includes skill, rules, and knowledge related to various technical activities of software development, application domains, communication (mainly in terms of understanding user needs and knowledge sharing with different stakeholders), computational thinking, and general purpose complex ill-defined problem solving.

Repeated practice with similar problems enhances skill. Variety, richness, and complexity of problem cases actively examined, solved, and/or critiqued by learners expand their 'rule base' and 'actionable knowledge base,' and hence, their competence. The implicit rules, their limitations, and exceptions are learned and refined by reflective practice. Problem cases with subtle differences can result in rule failure to solve problems. Such situations create conditions for the learner to recognize the limitations and exceptions to their rules and further refine them.

Hence, during software developers' education, a large variety and number of such experiences are necessary for them to build a sophisticated, rich, and actionable mental repository of implicit rules. *No single method of teaching and learning can help the learner to build such a repository. Only a proper integration of active, integrative, reflective, and collaborative engagements with theoretical, as well as practical, problem cases can help to create a large number of such varied opportunities.*

As per our studies discussed in this chapter, *student-centric pedagogical activities, especially projects* have been found to be most effective for development of all basic competencies, discussed in this chapter. Well designed projects, if administered properly can engage the students in a variety of learning oriented tasks. We further discuss this issue in seventh, eighth, and ninth chapters.

Rule or rule-base refinement is a knowledge-based activity that requires revising the mental model of the problem, knowledge domain, and/or the mapping between the two. This exercise is driven by the mental faculties of attentions, critical analysis, reflection, and also creativity and innovation. Hence, we call these mental faculties the competency driver-habits of mind. In the next chapter, we carry out a detailed discussion about these faculties as we see them in the context of software development education.

CHAPTER 5: SOFTWARE DEVELOPERS' EDUCATION FOR DEVELOPMENT OF COMPETENCY DRIVER-HABITS OF MIND

As discussed in Section 4.5, Costa and Kallick [203] suggested sixteen habits of mind of intelligent people to solve unfamiliar problems (Annexure AN6). Good professionals develop powerful habits of mind to use their intelligent thinking behavior for solving problems within their professional settings. These thinking habits distinguish them from the novices. *Thinking is the creation of a mental representation of what is not in the immediate environment* [209]. *Thinking continuum spans from one extreme of automatic thinking to another extreme of controlled thinking. Pure association is the simplest form of automatic thinking. Automatic thinking occurs in situations where repetition encourages decision making based on previously learned responses. In controlled thought, in contrast, we deliberately hypothesize a class of objects and experiences, and then view our experiences in the light of these hypothetical possibilities.* Formal thinking, visual imagination, scenario building, creation are forms of controlled thinking. According to Piaget, in formal thinking, the reality is viewed as secondary to possibilities.

Software development is more of a cognitive activity rather than a construction activity. With specific reference to debugging, which is one of the key challenging activities of software development, Metzger puts forward a systematic approach by integrating the thinking perspectives of six approaches: detective, mathematician, safety expert, psychologist, computer scientist, and engineer [157]. With respect to the multifaceted activities of software development, following the following three mental habits have been identified as the most important for software developers:

i. Attention to details
ii. Critical and reflective thinking
iii. Creativity and innovation

Those with a history of successful thinking efforts of some kind are much more willing to make more thinking effort of that kind. They know from past that they can productively engage in such thinking. Hence, it is imperative for software development education to nurture these habits among the students.

Section 5.1: Software Developers' Education for Development of Attention to Details

Thoroughness and concern for different perspectives and aspects, including very small or routine matters, are very important for all software developers. They need to carefully examine the objects/ideas under consideration in terms of form, function, relationship, and perspective. Programming requires habit of long attention spans typically lasting several hours and often for several days on a single problem. Software designers need to work at varying levels of abstraction, and ensure consistency in terms of interpretation and implementation across these levels of abstraction. This requires a keen attention to details, and the ability to correlate them across the various levels of abstractions encountered.

Importance of attention to details for software development work

Further, given the inconspicuous nature of software, its visibility limited to the side-effects affected in the system's environment, it is imperative that the limited visibility is accurate and consistent with the desired objectives and behavior. This requires careful planning and execution, with particular attention to exacting details. Expert programmers have the habit of paying attention to minor details [194].

In our 2009 survey on required competencies for software developers, twenty software professionals assigned 'attention to detail' an average rating of 3.3 on a scale of 0-4. An overwhelming majority of 90% of these respondents recommended it to be a critical or very important competency with respect to the requirements of software developers' multi-faceted professional activities.

With reference to Appendices A2 and A3, 'attention to detail' of software developers also relates to the following:

1 Good grasping power and attention to detail: breadth, depth, clarity, accuracy, preciseness, specificity, relevance, significance, completeness, consistency
2 Listening skill
3 Quality consciousness

Boehm quotes Winston Royce from his classical paper on the waterfall model written in 1970 [210], "In order to procure a $5 million hardware device, I would expect a 30-page specification would provide adequate detail to control the procurement. In order to procure $5 million worth of software, a 1500 page specification is about right in order to achieve comparable control." Over the last four decades, the behavior richness of software has exceeded the data or control richness. Agile software development methods recognize the limitations of the human mind in its capacity to see and freeze the details in advance. They view detailed acceptance tests not just as testing artifacts but also as executable requirements. They differ from the traditional waterfall model essentially by continuously evolving and detailing the specifications iteratively and incrementally creating just enough documentation for the situation at hand in a just-in-time manner. Empirical methods have become very popular in software engineering, hence, **ability to gather data and its systematic analysis** have become very important for software developers. After spending enough energy exactly detailing the specifications and/or acceptance tests, the algorithm/computation design process becomes a much simpler task.

Further, software developers need to follow and comply with policies, procedures, checklists, safety and security measures, and standards. In their thinking and expressions, software developers need to show attentive considerations for context, scope, boundaries, interfaces, assumptions, scalability, and constraints. Often serious oversights occur during the systems analysis phase, resulting in wrong, inconsistent, or incomplete requirements, poor usability, and poor test planning. Software bugs, often requiring costly rework, are also usually caused due to oversight over seemingly minor details.

Metzger has catalogued various types of *skill based errors* in software that occur either because of '*inattention*' *or* '*over-attention*' by the developers [157]. As per Metzger, the *inattention*

failures category includes psychological error subcategories of 'interrupt sequence, start another sequence,' 'interrupt sequence, omit step,' 'interrupt sequence, repeat step,' 'sequence interrupted, loss of control,' 'multiple matches, incorrect choice,' 'multiple sequence active, step mixed,' and 'sensory input interferes with active sequence.'

The *over-attention failures* arise because of human memory failure and manifest themselves as omission, repetition, and reversal. The errors in this category include psychological error subcategories of 'forgetting the goal,' 'order memory error,' 'spacing memory error,' 'coordination memory error,' 'remembering incorrectly,' and 'not remembering.'

Further, Metzger recommends that *like a detective*, debugging requires the developers to focus on facts, pay attention to unusual details, gather facts before hypothesizing, use a system for organizing facts, state facts to someone else, start by observing, avoid guessing and following emotionally comfortable hypotheses, keep a log of observations, assumptions, hypothesis, and experiments, and follow look-once-and-look-well strategy.

Good software needs to have mechanisms for ensuring data consistency, fault tolerance, and graceful handling of exceptions.

Code analysis, performance tuning, quality assurance, standard and regulatory compliance, program comprehension, code archaeology, large teamwork, geographical distribution, legacy systems, contractual constraints, risk engineering, data or technology migration, and disaster recovery require very careful attention to minute details. Procedures of version control, configuration management, requirement tracing, defect tracing, and document tracing also require an attentive eye for details. Hence, software developers need to have the habit *of repeated verification and careful monitoring*. For ensuring traceability, they need to regularly organize and maintain records of their work. They need to develop the *habit of seeking and bringing clarity, precision, accuracy, completeness, and consistency in work, its documentation as well as record*.

Some Theoretical Perspectives on Attention

Cognitive psychologists have been studying the phenomenon of attention for several decades. Galotti gives an excellent account of their findings on this aspect [196]. We give a brief summary in Annexure AN6. In the 1980's, Anne Treisman showed that *perceiving individual features takes little effort or attention, whereas gluing features together into a coherent object requires more*. Software development is basically about gluing a large number of abstractions related to application domain as well as programming environment into a coherent system. Hence, it requires a significantly higher level of attention.

As per classical Indian philosophy, the Raja Yoga system deals with attention and concentration. Yogi Ramacharaka (real name William Walker Atkinson) wrote a commentary on this system [211]. In this commentary, he wrote that the word 'attention' is derived from the Latin words "*ad tendere*," meaning "to stretch toward." It involves focusing of mind's entire energy upon the object, observing every detail, dissecting, analyzing, and drawing every possible bit of information about the object. Attention is a prerequisite of good memory, and it also affords the powers of association. It enables one to combine, associate, classify, etc., and thus create new knowledge and expressions. It sharpens all other mental faculties.

According to Raja Yoga [211], after voluntary attention is firmly fixed, and held upon an object, the mind will "do the rest." *Voluntary attention is a very good substitute for genius, and unlike genius, it can be sharpened through practice and perseverance.* Attention requires thinking of, and doing, one thing at a time. This habit is learnt through practice. In order to excel, one has to "immerse oneself" in the work. In order to discover more details about an object, one needs to engage in *several iterations of (re)examinations and evolutionary expressions. Critique of the work products of earlier iterations in the light of the re-examination of the object (problems), progressively reveals newer details and affords new opportunities for richer descriptions and other expressions*.

Given the knowledge intensive nature of software development, and software development being viewed as a continuous learning task [120] [148], it is no surprise that the evolutionary methods of software development are manifestations of this approach.

Pedagogic Perspective

In **our recently concluded survey,** "Software developers - (How) Did your college help you in your development?" (Table A10.2 (ii) , Appendix A10), a large fraction of *71% felt that as compared to all other kind of academic engagements, their student projects did much better to develop their attention to details.* This was followed by research literature survey and mentoring juniors (37% each), laboratory work (35%), laboratory work, and industrial training (33%). *Discussions with others and traditional knowledge transmission oriented lectures were found to be least effective* in this regard by the respondents.

In our proposed framework of pedagogies of engagements in software development education, active (critique) as well as reflective engagements in Sections 8.3.1 and 8.3.3 respectively contribute to the development of this competence.

Section 5.2: Software Developers' Education for Development of Critical and Reflective Thinking

What is Critical Thinking?

John Dewey (1909), considered by many as the father of modern critical thinking, posited that *critical thinking involves active, persistent, and careful consideration of a belief or supposed form of knowledge in the light of the grounds which support it and the further conclusion to which it tends*.

Tama [212] saw it as "*a way of reasoning that demands adequate support for one's beliefs and an unwillingness to be persuaded unless support is forthcoming*."

American Philosophical Association posited that *critical thinking is purposeful, self-regulatory judgment that results in interpretation, analysis, evaluation, and inference, as well as explanation of the evidential, conceptual, methodological, criteriological, or contextual considerations upon which that judgment is based* [213].

What is Reflective Thinking?

Kottkamp [219] defined reflection as "*a cycle of paying deliberate attention to one's own action in relation to intention… for purpose of expanding one's opinion and making decisions about improved ways of acting in the future, or in the midst of the action itself*."

Importance of Critical and Reflective Thinking for Software Development

In a study [191], almost unanimously, i.e., 99.3% of 1023 experts rated 'abstract thinking and *reasoning*' as an important element of human intelligence. *In our 2009 survey* on required competencies for software developers, twenty software professionals assigned critical and reflective thinking an average rating of 2.6 on a scale of 0-4. A majority of 55% of these respondents recommended it to be a critical or very important competency with respect to the requirements of software developers' multi-faceted professional activities.

With reference to Appendices A2 and A3, 'critical and reflective thinking' of software developers relates to the following competencies:

1 Reasoning: quantitative and verbal, and critical thinking: ability to question, validate, and correct the purpose, problem, assumptions, perspectives, methods, evidence, inference, reliability, relevance, criteria, and consequences
2 Analytical skills
3 Listening skills
4 Quality consciousness and pursuit of excellence
5 Constructive criticism
6 Research skills: methods of mathematical research, engineering research, design research, and social science research
7 Reflection and transition between ladders of reflection. Meta-cognition
8 Self-acceptance, self-regulation, self-awareness, self-improvement: strength to resist instant gratification in order to achieve better results tomorrow. Being honest and forthright about one's own limitations of competence. Tendency to avoid false, speculative, vacuous, deceptive, misleading, or doubtful claims. Faith in reason and review, inclination for verification and validation, respect for facts and data. Awareness and regulation of automatic thoughts
9 Sensitivity towards global, societal, environmental, moral, and ethical issues, and sustainability
10 Entrepreneurship, sense of mission, perseverance, commitment, self motivation, dedication. Adaptability, flexibility, open-mindedness, and ability to multi-task

Like all professionals, software developers also examine and interpret situations as per some established paradigms of their profession. Software development is well recognized as a knowledge-based activity. Hence, the behavior and performance is controlled by conscious logical and analytical reasoning [157]. Such reasoning is invoked by beginners who start performing a task, or by experienced persons who face a novel situation. In either case, working at this level, we make two kinds of errors either because of resource limitation of the conscious mind, or because of incomplete/incorrect knowledge. These errors are dominated by extrinsic factors, and are difficult to detect and correct.

With specific reference to software debugging, Metzger cites research based findings regarding errors in logical and analytical reasoning. Table 5.1 gives a summary of this collation. This table is included to support our proposed framework of pedagogical engagements (Table 8.5).

Table 5.1: Some common errors in logical and analytical reasoning

1	Misdirected focus – tendency to focus on interesting rather than logically important aspects of the problem.
2	Storage limitation - storage capacity of conscious mind is extremely limited; hence, the presentation of problem can have a great impact on the ability to store all the relevant information as conscious mind reasons through a problem.
3	Information availability - people give too much weight to facts that readily come to their mind, and have a tendency to ignore information that is not readily accessible.
4	Hypothesis persistence - preliminary hypothesis formed on the basis of incomplete data early, in the problem solving process are retained in the face of additional, more complete data available later.
5	Selective support - people are often overconfident of their information. They justify their plans by focusing on their information and often ignore information that does not support their plan.
6	Limited reviewing – people do not consider all the aspects during review. Even when they do, they fail to see the aspects as an integrated whole.
7	Inadequate data – people are very likely to draw conclusions from inadequate data.
8	Multiple variables – people tend to predict extreme values for partially related variables.
9	Misplaced causality – people are likely to judge causality based on their perception of the similarity between a potential cause and its effect.
10	Dealing with complexity – people have trouble thinking about complex processes that occur over time, and prefer to deal with a single moment. They also have difficulty in dealing with nonlinearity and multiple side effects.
11	Decision and probability – people don't make good decisions in circumstances that require assessing probabilities.

Only *critical thinking can help in controlling such errors in logical and analytical reasoning at various stages of software development*. Further, Metzger recommends that like a detective, debugging requires the developers to *reason based on facts, validate assumptions, eliminate*

alternative hypotheses, reason inductively as well as deductively, and consider all interpretations of the fact that seem to be relevant.

Hazzan and Tomakyo [124] highlight the importance of ability of *reflection* for software developers. *Evolutionary software development approaches including agile methods* draw their strength from the possibility of continuous reflection. Reflection helps in building new perspective. The highest level of SEI's Capability Maturity Model (CMM) level 5 (Optimized level) is characterized by focus is on continually improving process performance through both incremental and innovative technological changes/improvements. Such improvements can only be facilitated by reflective thinking. Similarly, the highest (5^{th}) level of People CMM (P-CMM optimizing level) also focuses on continuous improvement of workforce competence through reflection on the quantitative management activities established at maturity levels 4.

Some Theoretical Perspectives on Critical Thinking

Minger's Framework for Critical Thinking

In 2000, Minger [216] proposed a framework for critical thinking with special reference to management education. Because there are many subjective aspects related to software development, we find it relevant for the purpose of software developers as well. The four levels of this framework are as follows:

i. Critique of rhetoric: argument analysis by checking for logical fallacy, soundness, and validity.
ii. Critique of tradition: critical attitude towards actions in organizations, cultures, traditions, and assumptions that underpin these beliefs.
iii. Critique of authority: being skeptical of one dominant view.
iv. Critique of objectivity: being skeptical of information and knowledge, recognition that information and knowledge is never value free, and are continuously reshaped by the structures of power within a situation. Implies the meta-cognitive process in critical thinking.

These levels are included in proposed framework of pedagogical engagements in software development education (Table 8.5).

Paul's Model of Critical Thinking

Paul sees it as a mode of thinking - about any subject, content, or problem - in which the thinker improves the quality of his/her thinking by skillfully analyzing, assessing, and reconstructing it. Paul proposed a taxonomy of Socratic questioning to facilitate critical thinking [214]. It included *six categories of questions*: (i) questions of clarification, (ii) questions that probe assumptions, (iii) questions that probe reasons and evidence, (iv) questions about viewpoints or perspectives, (v) questions that probe implications and consequences, and (vi) questions about the question. This model has been extended and also applied to *engineering reasoning* [215].

As per this model, the *elements for critical thinking* are: *purpose, question at issue/problem to be solved, concepts, information, assumptions, inferences, interpretations, points of view, implications, and consequences*. *We also add the elements of context, criteria, and method to this list.* This model also lists some *standards for critical examination* of the elements. These include *clarity, specificity, relevance, logical, significance, consistence, breadth, depth, accuracy, precision, fairness, and completeness*. Critical thinking involves the processes of *identifying, analyzing, synthesizing, evaluating, reviewing, and considering the elements in the light of the abovementioned standards*. As per Paul, repeated engagements in these processes result in the development of the intellectual traits required for critical thinking.

Paul's extended model is included in our proposed framework of pedagogical engagements in software development education as a checklist for guiding critical thinking during various stages of software development (Section 8.3.1).

Some Theoretical Perspectives on Reflective Thinking

Critical thinking about ideas, object and world is not sufficient for creating meaningful systems. In his classic book, Barnett [217] describes his notion of 'critical being' as including thinking, self-reflection and action: "critical persons are more than just critical thinkers. They are able critically to engage with the world and with themselves as well as with knowledge." He *identified three domains of criticality: knowledge and ideas (critical reason), the experience of self (critical reflection) and the action in the world (critical action).*

Moon [216] cites Ford (2005) who differentiated between the levels of pre-criticality, criticality with the agenda of others without much challenge to the given frameworks, and *criticality to one's own agenda*. This calls for reflective thinking. As per the multiple intelligence theory of Gardner, *reflection is associated with the intra-personal intelligence*. Costa and Kallick [203] also emphasized on the ability to reflect to evaluate the productiveness of our own thinking.

In 1979, **Bateson** proposed a model of logical categories of learning. He viewed that progressively deeper levels of learning require *change of action, assumptions, or context and commitment*. The *first level* of learning is about making *minor fixes or adjustments in action*. The *second level* of learning requires reflection to *challenge one's beliefs and assumptions*. This facilitates new insights for changing the rules and making major changes. The *third level* of learning requires even deeper reflection to bring about a *shift in understanding our context, values, point of view, and commitments*.

Schön [218] defined reflective practice as the practice by which professionals become aware of their implicit knowledge base and learn from their experience. He introduced the following three notions:

1. Reflection in action: reflect on behavior as it happens, so as *to optimize* the immediately following action.
2. Reflection on action: reflecting after the event, to review, analyze, and evaluate the situation, so as to gain insight for improved practice in future.
3. **Ladders of reflections**: action, and reflection on action make a ladder. Every action is followed by reflection and *every reflection is followed by action in a recursive manner*. In this ladder, the products of reflections also become the objects for further reflections. This is included in our proposed framework of pedagogical engagements in software development education (Table 8.9).

Further, Schön [220] posited that the mental habit of reflection and ability to move across the ladders of reflections is central to professionals' approach to their work. He saw 'design' as 'reflection in action' in which changing a given situation takes precedence over the interest of understanding it. He also observed that for a designer, the phenomena/situation continues to

change during their work. Table 5.2 summarizes some of the key observation of Schön, in this regard.

Table 5.2: Some key aspects of Schön's perspectives on 'design' as 'reflective action'

Designers begin with situations that are at least partially uncertain, ill-defined, complex, and incoherent. Designers construct and impose a coherence of their own. Subsequently they discover consequences and implications of their constructions – some unintended – which they appreciate and evaluate, sometimes leading to reconstruction of initial coherence – a reflective conversation with material of a situation. They spin out a web of moves, consequences, implications, appreciations, and further moves. Each move is a local experiment that contributes to the global experiment of reframing the problem. Moves create new problems to be described and solved. Moves have expected/or unexpected consequences in many design domains and implication bindings on later moves. In this process, designer reflect in three dimensions: 1. The domains of languages in which the designers describe and appreciate the consequences of their moves, e.g., use, technology, form, cost, scale, character, representations, quality, standards…. 2. The implications they discover and follow. Designers evaluate their moves in terms of: a. Desirability of their consequences. b. Conformity to/violation of implications of earlier moves. c. Their appreciation of new problems or potentials they have created. 3. Their changing stance towards the situation with which they converse: Can/might, should/must, what if, unit/total, moves/appreciation of outcomes, and tentative adaption of strategy/commitment.

Relating software engineering to Schön's work on reflective thinking and professions (1987), Hazzan and Tomakyo [124] posit that mental habit of reflection and ability to move across the ladders of reflections are closely associated with software engineering processes. They also give examples of such ladders of reflection for soft engineering tasks. Further, one of the key principle in the *agile manifesto* is, "at regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly."

Proposing a model for reflective design, Phoebe Sengers et al [221] recommend that designers should use reflection to uncover and alter the limitations of design practice, to re-understand their own role in the technology design process, and to support users in reflecting on their work and lives. Stones [222], Ginsburg [223], and Lasley [224] have identified the following *elements of reflection: (i) practical experience, (ii) meaningful knowledge base of subject, context, and users, (iii) interaction with others, (iv) philosophical awareness and understanding of what constitutes good practice, and (v) strong problem solving skills.* These elements are embedded in various dimensions of our proposed framework of pedagogical engagements in software development education (Section 8.3).

Reflection is not an automatic activity. It requires controlled thinking. Students do not usually automatically reflect well upon their actions and tasks in various assignments. This limits not only the quality of their assignments, but their overall learning as well. A small post-assignment, reflective activity can amplify their learning from the same assignments.

Borton [225] proposed a three-level model for reflection through three stem questions: *what?, so what?, and now what?* Many other later frameworks by Gibbs (1988), John (1994), Smyth (1989), and Kim (1999) are manifestations of this framework. *Borton's model is included in our proposed framework of pedagogical engagements in software development education* (Section 8.3.3).

Pedagogic Perspective

Many studies have showed that multi-paradigm disciplines like humanities, social sciences, and psychology had a positive influence on students' self reported growth in critical thinking skills. However, Li et al [226] have found that self perceived gains of students' *critical thinking skills most significantly depended upon the degree of their integration into the academic and social community of the university rather than their discipline of study* The other significant influencing factors were found to be the quality of lower division courses. Gender and quality of advising were found to be insignificant factors in this regard. The quality of teaching was found to be a very significant factor for influencing their academic integration. *The quality of curriculum was found to be the most significant factor for influencing their social integration.*

In **our recently concluded survey,** "Software developers - (How) Did your college help you in your development?" (Table A10.2 (ii) , Appendix A10), half of them felt that as compared to all other kind of academic engagements, their student projects did much better to develop their critical and reflective thinking. Critical and reflective thinking was also felt to improve through engagements of thinking oriented lectures (48%), discussions with faculty and other students (44% each), and research literature survey (42%).

Engagements in homework, knowledge transmission oriented lectures, written examinations, and industrial training were felt to be least effective with respect to development of critical and

reflective thinking. Adding an element of reflection after all their engagements can enhance the perceived effectiveness of many low rated engagements as well. In our proposed framework of pedagogical engagement, reflective engagements seek to achieve this goal. We have also discussed some such instructional interventions discussed in Section 9.1.3.

Section 5.3: Software Developers' Education for Development of Creativity and Innovation

What is creativity?

In a study [191], a large fraction of 59.6% among 1023 experts rated 'creativity' as an important element of human intelligence. It was rated much above some other elements like 'goal directedness' or 'achievement orientation.' Costa and Kallick [203] have included 'Creating, Imagining, and Innovating,' as one of sixteen mental habits that characterizes intelligent people when they are confronted with problems, the resolution to which are not immediately apparent. As per Sternberg's theory of tri-archic intelligence, creative ability along with practical and analytical abilities together define human intelligence. He explained creativity as the ability to *apply problem solving processes to novel and unfamiliar problems.*

Divergent thinking, i.e., the ability to generate new ideas is at the core of creative ability.

Creativity is generating new thoughts, invention is transforming the creative thoughts into novel tangible ideas, and innovation is the first novel application of those ideas in a specific context. Osche [227] sees creativity as bringing something into being that is original (new, novel, unusual, unexpected) and valuable. She posited that the most important criterion was the willingness of creative people to work hard and put in the extra time necessary to turn out a quality product in a given domain. Albert Rothenberg associated creativity with, 'Janusian thinking,' i.e., the ability to conceive and hold two or more contradictory or opposite thoughts simultaneously. He also posited that the creative process is a matter of continually separating and bringing together, bringing together and separating, in many dimensions: affective, conceptual, perceptual, volitional, and physical.

While, mystical view of creativity attributes it to divine inspirations, pragmatic view believes that some techniques can stimulate creativity. Psycho-dynamic perspective posits that it arises from the tension between *conscious reality* and *unconscious drives*. Social-personality attributes creativity to personality variables, motivational variables, and the socio-cultural environment. Evolutionary approaches suggest that like the process of evolution, blind generation of a large number of ideas should be followed by selective retention. Confluence approach seeks to integrate various perspectives.

Importance of Creativity for Software Development

In our survey of fifty-seven software professionals (Table 4.2), 66% respondents included 'innovation and research' as one of the most important activities that must be included in the main goals for new curriculum for the future generation of software developers. **In another survey conducted by us in 2009** on required competencies for software developers, twenty software professionals assigned creativity and innovation an average rating of 3.0 on a scale of 0-4. A large majority of 80% of these respondents recommended 'creativity and innovation' to be a critical or very important competency with respect to the requirements of software developers' multi-faceted professional activities.

With reference to Appendices A2 and A3, creativity and innovation of software developers also relates to the following:

1 Design skills
2 Imagination: storyboarding, extrapolation, visualization, cognitive flexibly: ability to transfer and models of solutions of one situation/field to another, multi-perspective thinking, lateral thinking, inductive thinking, out-of-box thinking, unstructured thinking
3 Complex problem solving
4 Research skills: use and integrate methods of mathematical research, engineering research, design research, and social science research
5 Experimentation skills
6 Entrepreneurship

Usually, software design projects require more than a synthesis of previously learned knowledge. Design is primarily an inductive process. This process of reasoning is non-deductive: there is no closed pattern of reasoning to connect the needs, requirements, and intentions with a form of software. To succeed as software designers, computing students need to be well trained in inductive reasoning.

Software has grown much beyond the simple interfaces to information. Much of software is increasingly becoming concerned about user's experience.

In the last two decades, there has been an increasing attention on user interface design with a focus on user experience. Software companies need creative people in order to do the high-level design of new innovative software products. They need creative minds to design the new procedures and tools that make the development of new, ever-more-complicated software applications easier. A reductionist and linear thinking give evolutionary incremental advancements; revolutionary advancements come from non-linear and holistic thinking, and intuition. Software companies require both kinds of mindset. Non-linear thinking is necessary for generating the ideas to break current boundaries. However, linear mindset is necessary for executing these ideas.

With reference to the complex problem solving discussed in section 4.5, the aspects of problem solving or decision making process in which creativity can be applied are the following: (i) restructuring the problem/decision task, (ii) generating alternatives, and (iii) selecting decision criteria and strategy, and evaluating alternatives [230]. These three are included in our proposed framework of pedagogic engagement (Table 8.5).

Restructuring the problem/decision task requires holistic perception of problem/issue and involves several iterations of redefining the problem and/or goals. *Hence, systems thinking discussed in Section 6.3 and reflective thinking discussed in Section 5.2 play a very crucial role in finding creative solutions.* One of the software engineers, we interacted with, commented, *"a problem is not a problem until it is revised again and again."* Another expressed that *"problem*

definition is a thing which people generally don't take interest in. People start running for the different solutions without even having a mere idea of the problem." An entrepreneur reflected, *"problem formulation does not end, at least till a solution is achieved, and sometimes it just goes on and on by improving upon the found solution."*

Sternberg's propulsion theory of creativity

As per Sternberg's propulsion theory of creativity [229], creative contributions are attempts to propel a field from wherever it is to wherever the creator believes the field should go. He proposed following four-level taxonomy of creative contributions.

1. The lowest level consists of *paradigm preserving contributions* that leave the field where it is through replication.
2. The next creative level is of *paradigm forwarding contributions* that move the field forward in the direction it already is going. This movement may be forward incrementation or advance forward incrementation.
3. A higher level of creative contributions is *paradigm rejecting*. Such creations move the field in a new direction from an existing or preexisting point. It involves *redirection or reconstruction*.
4. The highest levels of creative contributions are also paradigm rejecting. This rejection is not to redirect the field from an existing old point, but to restart the field in a new place, and move in a new direction from there. It requires *re-initiation and/or integration*. Inter-disciplinary approaches stimulate such thinking.

These levels are included in our proposed framework of pedagogical engagements in software development education (Table 8.5).

Pedagogic Perspective

Several techniques have been developed for stimulating the mind for generating alternative ideas. These include Osborne's checklist, SCAMPER (Substitute, Combine, Adapt, Modify or Magnify, Put-to-another-use, Eliminate, Rearrange or Reverse), and Edward de Bono's concept of lateral thinking and 'po' (provocative operation) emphasizing on suppose, possible, hypothesize, and poetry, etc., [231]. *Brainstorming* also helps a great deal creative thinking.

Altshuller [232] studied hundreds of thousands of patents, and proposed a powerful Theory of Inventive Problem Solving (TRIZ/TIPS). This theory identified 40 recurring principles that were repeatedly being applied by the inventors in different fields. Table 5.3 gives a brief list of these 40 principles. Subsequently, this technique has become very popular among a large number of researchers. Since 1996, 'The TRIZ journal' is being published every month at triz-journal.com. *TRIZ principles have also been found to be metaphorically manifested in software design* [233-234]. Researchers have also attempted to extend these principles by adding some more principles that are especially relevant for software, especially because of its material-less nature [235]. Some of these additional principles include *metaphor, scope, evolution, privacy, usability, synchrony*, etc.

Table 5.3: Principles of Theory of Inventive Problem Solving (TRIZ/TIPS)

1. Segmentation	11. Prior cushioning	21. Rushing through	31. Porous materials
2. Taking out	12. Equi-potentiality	22. Blessing in disguise	
3. Local quality	13. The other way round	23. Feedback	32. Colour change
4. Asymmetry	14. Spheroidality or curvature	24. Intermediary	33. Homogeneity
5. Merging	15. Dynamics	25. Self-service	34. Discarding and recovering
6. Universality	16. Abundance	26. Copying	35. Parameter change
7. Nested doll	17. Another dimension	27. Cheap short-lived objects	36. Phase transition
8. Anti-weight	18. Mechanical vibration	28. Mechanics substitution	37. Thermal expansion
9. Preliminary anti-action	19. Periodic action	29. Pneumatics and hydraulics	38. Strong oxidants
10. Preliminary action	20. Continuity of useful action	30. Flexible shells and thin films	39. Inert atmosphere
			40. Composite materials

Kowalick identified *seventeen secrets of inventing new products* [236]. Some of these are (i) the real problem to be solved is rarely the same as the problem initially posed, (ii) technical systems often have many functions, some of which are useful, and others that are useless or even harmful, (iii) pruning a technical system is one of the highest forms of creativity, and (iv) solving technical design conflicts by making tradeoffs is not as useful as stating the objective in the form of a 'contradiction,' and meeting the contradictory requirements by design.

Metaphors and Analogies

Altshuller [232] also made the following observations: (i*) problems and solutions were repeated across industries and sciences*, (ii) patterns of technical evolution were repeated across industries and sciences, and (iii) innovations used scientific effects outside the field where they were developed. Moreover, since software serves multiple industry verticals, there is ample scope for

cross-pollination of ideas and best practices. Cross-industry and multi-domain exposure fosters creative thinking, if one has an open mind. Using metaphors helps designers and developers to think around user goals and assumptions. *Understanding customer expectations in terms of another common device or appliance that everyone uses may help them to design a better product interface that improves user adoption and reduces training time*. However, metaphors and analogies need to be used with care as they come from a specific context, and hence, can sometime lead to serious misunderstandings in changed circumstances. *The education program must encourage the development of metaphorical thinking*. *Arts and Literature related courses can make a huge contribution in developing such thinking.*

In **our recently concluded survey,** "Software developers - (How) Did your college help you in your development?" (Table A10.2 (ii), Appendix A10), a large fraction of 82% felt that as compared to all other kind of academic engagements, their student projects did much better to develop their creativity and innovation. This was followed by thinking oriented lectures (53%), research literature survey and discussion with other students (45% each), laboratory work and discussion with faculty (39% each), and mentoring juniors (31% each). Written examinations, knowledge transmission oriented lectures, and homework were found to be least effective in this regard by the respondents.

Amoussou et al [237] have identified and collated the *following activities for enhancing creativity* and design in computing courses: (i) *reflections on sources of inspiration* including brainstorming techniques, (ii) *reflections on bias* that may affect creativity and design, (iii) identify and define the steps of the design process and provide design examples, (iv) identify and define *criteria and constraints*, (v) practice *methods of evaluating* options, (vi) reflect on norms of communication, and (vii) *discuss ethics* within the context of design.

Lassig [238] has proposed to adapt a balanced view about *six environmental conditions to inculcate computing students' creativity*. These are: (i) *a supportive and nurturing environment that also provides obstacles and challenges*, (ii) *some constraints* are helpful for novel/unfamiliar tasks that are to be performed with limited knowledge/skills, (iii) evaluation generally inhibits creativity, when it must be done, the criteria should be clear, self evaluation

can also facilitate creativity, (iv) if the task is not too difficult, competition can stimulate a person who is initially not very motivated, however, if the task is difficult or the person is already motivated, competition can create anxiety and inhibit creativity, (v) enthusiastic cooperation does not automatically lead to more creative ideas, and (vi) role models are helpful for enhancing creativity, only when they encourage independent thinking.

We have included theories and techniques on creativity and innovative thinking in the course content of two undergraduate elective courses: (i) human aspects for information technology and (ii) software arteology. Many students have reported that it helped them to expand their creative thinking for their software projects. *Later in Section 8.3.1., we support our proposed framework of pedagogical engagements in software development education with techniques of SCAMPER, lateral thinking, 40 TRIZ/TIPS principles and further extensions, as well as the activities collated by Aoussou et al and the environmental conditions suggested by Lassig, as discussed above.*

Section 5.4: Chapter Conclusion

In this chapter we argued that the multifaceted basic competence for software developers can only be used and refined with the help of habits of mind that drive competence. These habits include attention to detail, critical and reflective thinking, and creativity and innovation. As per our studies discussed in this chapter, student-centric pedagogical activities, especially projects have been found to be most effective for development of these habits. The only way to inculcate these habits is by engaging them in such tasks that require them to use these habits. Only through repeated usage can these be enhanced. Development of these habits has to be put as a core learning outcome of all courses. We further discuss this issue in the seventh, eighth, and ninth chapters. In the next chapter, we discuss competency conditioning attitudes and perspectives that help in enhancements and meaningful application of these habits.

CHAPTER 6: SOFTWARE DEVELOPERS' EDUCATION FOR DEVELOPMENT OF COMPETENCY CONDITIONING ATTITUDES AND PERSPECTIVES

Senge, an eminent thinker on system thinking posited that true learning should be transformation of spirit and mind, not merely an accumulation of information or knowledge. According to Bigg's 3P model (involving three stages of Presage, Process, and Product) [239], students' perceptions of their learning environment, in light of their motivations and expectations, determine how situational factors influence their approaches to learning and learning outcomes. Research indicates that students' learning strategies, academic performance, understanding, and academic integration, are linked to their *attitude and epistemological perspectives*. These *attitude and perspectives* may either enhance or constrain the scope and nature of their learning [240-241]. These vary according to age, past performance, and contextual factors like home environment, pre-college schooling experience, college experiences, and educational level. Attitudes and perspectives also affect a professional's motivation and ability to practice. Hence, it is most *important to make efforts to help students to form enabling attitudes and perspectives.*

The following list enumerates the recommended attitudes and perspectives especially with reference to the requirements of the profession of software development:

1. Curiosity
2. Decision making perspective
3. Systems-level perspective
4. Intrinsic motivation to create/improve artifacts

In the following sections we discuss the rationale as well as explore theoretical and empirical grounds of these traits in multiple disciplines.

Section 6.1: Software Developers' Education for Development of Curiosity

Importance of Curiosity for Software Developers

In our 2009 survey on required competencies for software developers, twenty software professionals assigned 'curiosity' an average rating of 3.15 on a scale of 0-4. An overwhelming majority of 90% of these respondents recommended it to be a critical or very important competency with respect to the requirements of software developers' multi-faceted professional activities.

With reference to Appendices A2 and A3, curiosity of software developers also relates to the following:

1 Interest in 'how things work' and 'how to create things that work,' interest in the power of technology,
2 Ability to see things as they are, observation skills
3 Broader understanding and interests
4 Respect for the classic authors of the great books
5 Openness to constructive criticism
6 Value and readiness for lifelong learning.
7 Active listening skills
8 Ability to develop a very good understanding of domain specific vocabulary, its semantics, and established thinking patterns
9 Experimentation skills
10 Knowledge of contemporary issues and business practices
11 Knowledge of physical and natural world. Intercultural knowledge
12 Mentoring
13 Research skills
14 Self-awareness
15 Inclination for verification and validation, respect for facts and data

Curiosity is a highly important trait for software developers, and they need to proactively remain ready and engaged in *lifelong learning* because of following reasons:

1 The applications domains are highly diverse and continuously evolving, consequently software developers have to continuously learn more about these *domains*, mostly through self-learning, and work experience often without any long term formal education in the concerned domains

2 Various technological innovations and changing social trends are continuously and rapidly reshaping user expectations, understanding these *continuously evolving expectations* is very crucial for software developers

3 The *development technology and platforms keep changing* constantly often without proper documentation and examples, hence, the developers need to explore the useful enhancements and changes themselves again usually without much formal training

4 The developers usually have to *understand other developers' work* in order to extend, debug, maintain, integrate and/or re-engineer it

5 Creation of "simple and idiot-proof system interfaces" requires them to be *curious about how an average person approaches technology*, and

6 There can be *unintended consequences and risks* of creating software inappropriate or at odds to its real purposes.

Metzger recommends exercising curiosity during the debugging process to locate other defects of the same root cause, and also defects of other kind [157].

In olden days the software developers main focus was on learning how computing systems work so that they can be efficiently utilized for meeting well known and understood computing needs. Fast and reliable internet access, multimedia rich client, and mobile computing have opened new possibilities for exploring hitherto unknown computing needs. Hence, today's software developers need to be deeply interested in learning not only about the power of information and software technology, but also *needs and even possibilities of human beings*.

Expanding user expectations, changing user processes, evolving domain knowledge and understanding of users' needs, growing power of information technology, and rapid transformation of development platforms make software development a highly iterative and

evolutionary process. In order to properly respond to these factors, the developers need to have open mindedness. Their work needs good observation skills and strong ability to see things as they are. Hence, software developers also need to have broader understanding and interests. *Only highly curious software developers are able to develop very good understanding of domain* specific vocabulary, its semantics, procedures, and established thinking patterns.

Active listening is crucial for requirement analysis and all other form of knowledge sharing with various stakeholders in the process of software development. *Only a curious mind can be an active listener*.

Some Theoretical Perspectives on Curiosity

What is Curiosity?

David Hume explained curiosity in his *Treatise of Human Nature* as "that love of truth, which is the source of all our enquiries." Brand interpreted Hume's work on curiosity [242]. Benedict [243] viewed *curiosity as a sign of the rejection of the known as inadequate*. Further, she posits that curious people seek and manifest new realities and reshape their own identities and their products. Reio and Callahan view curiosity as a *state of emotional arousal*, induced by a conceptual conflict or uncertainty that induces information seeking or exploratory behaviors to relieve the uncertainty. It results in the restructuring of knowledge structures or learning [244]. Annexure AN8 gives some more important theoretical perspectives on curiosity: Arnone [245] and Peterson et al [246].

While curiosity is a *state* commonly experienced by all people, it is also a *trait* which is much more typical of some people than others [247]. With reference to the importance of curiosity, Einstein said, *"I have no special talents. I am only passionately curious."* Curiosity increases learners' attention. Curious people can challenge their views of self, others, and the world with an inevitable stretching of information, knowledge, and skills. Curiosity is closely associated with love for learning which is necessary for systematically mastering new skills and bodies of knowledge through formal education or self-learning.

Curiosity is a *fundamental motivational component for all openness facets including openness to experiences and open mindedness.* Open mindedness involves *multi-perspective thinking and suspension of judgment.* Only in the state of open-mindedness one is able to recognize one's misconceptions and the limitations of one's knowledge. Curiosity gives one the ability to weight all evidence with fairness, and if required, change one's mind in the light of new evidence. It is recognized as *a source of critical thinking and also creativity*.

Curiosity stimulates *an inquiry within the existing framework* that leads to acquisition of more information. A higher level of curiosity can also stimulate an *inquiry about the framework itself*, and results in evolution of perspective. At such level, a curious mind can get engaged in evolving a larger meaning in life beyond the immediate and short term interests of the self. Research suggests that curiosity is an important process for *psychological well-being* [248].

Diversity of Curiosity

A reinterpretation of Anderson and Krathwohl's taxonomy of knowledge types [134], Carson's taxonomy of knowledge types [249], and also Gardner's theory of 'multiple intelligence' [155] help us to understand the *variations of the curiosity of different persons*. Depending upon their interest and abilities, persons may have their strengths or weaknesses with respect to the categories of all these classification systems. From the perspective of Anderson and Krathwohl's taxonomy of knowledge types, persons may differ with respect to their (i) factual curiosity: inquisitiveness about factual knowledge, (ii) conceptual curiosity: inquisitiveness about conceptual knowledge, (iii) procedural curiosity: inquisitiveness about procedural knowledge (mental and psychomotor), and also (iv) meta-cognitive curiosity: inquisitiveness about meta-cognitive knowledge. *Software developers need to have curiosity of all these types*. In addition, in order to develop useful software, they also need to have a high level of contextual curiosity: inquisitiveness about the evolving context and expanding context of software technology.

Using Carson's taxonomy [249] as the lens to differentiate between different types of curiosities, we can see the categories of (i) empirical curiosity: inquisitiveness about the environment and experiences, (ii) rational curiosity: inquisitiveness about abstractions, relations, and quantities, (iii) conventional curiosity: inquisitiveness about manmade conventions e.g. language, notation,

protocol, rule, law, standard, guidelines, procedures, etc., (iv) conceptual curiosity: inquisitiveness about concepts, theories, patterns, design, (v) cognitive curiosity: inquisitiveness about mental procedures, algorithms, heuristics, (vi) psychomotor curiosity: inquisitiveness about body control, (vii) affective curiosity: inquisitives about emotional and aesthetic aspects, (viii) narrative curiosity: inquisitiveness about understanding human condition with human perspective, and (ix) spiritual curiosity: inquisitiveness about the spiritual (not to be confused as religious) side of human experience and life. *All these curiosities, except the last one, are beyond any doubt highly relevant to software developers' work*. It can also be argued that spiritual curiosity helps in overall growth of any person and helps them to understand larger purpose and meaning of life, which helps them to deal with work related dilemmas and issues of responsibility.

Levels of Curiosity

Epistemological beliefs of the learner about 'what is knowledge' and 'what are the roles of a learner, teacher, and peers in the learning process' influence their curiosity as well as learning process. In 1970's, Perry [250] proposed a nine stage model of cognitive and moral development. The initial five stages are purely cognitive, whereas ethical aspects also get integrated in the later four stages. These nine micro level stages are also broadly grouped into four macro level stages. At the level of 'dualism,' people believe things are right or wrong and have faith and commitment to truth and knowledge as stated by genuine authorities. At the second macro level stage of 'multiplicity,' the diversity in thinking is recognized, but the person does not feel the need to commit to any specific belief or mode of thinking. The third macro-stage is 'relativism.' At this stage, the person sees the context sensitivity of knowledge. The final macro-stage is 'commitment,' at which the learners feel the need to take positions and commit to them.

As per Perry's model, the movement through this stage is not automatic and progressive. One can undergo a long term pause at some position, or escape the progression by developing competence in some specific field, or even regress to lower position without one's awareness. Felder and Breta [251], as well as West [252], provide a comparison between Perry's model and

some other similar models proposed by Belenky et al in 1986, Baxtor Magolada in 1992, and King and Kitchener in 1994.

We have re-interpreted Perry's model of intellectual and ethical development as a nine stage model of development of curiosity. Our re-interpretation is given in Table 6.1.

Table 6.1: Re-interpreting Perry's nine stage model of intellectual development as nine stage model of curiosity development

Dualism

1. Basic Dualism: Persons at this cognitive level believe that right solutions (knowledge and also values) to all problems are already known to 'genuine' authorities. Their curiosity is limited to learning right and specific solutions (facts and formulas) from authorities.
2. Full Dualism: Persons believe that solutions to all problems are already known to authorities, but some 'genuine' authorities may differ. Their curiosity is even more strongly focused on learning only the right and specific solutions from authorities by ignoring all other perspectives.

Multiplicity

3. Early Multiplicity: Persons believe that all problems are solvable. Further they think that even if 'genuine' authorities do not know the solutions to all problems, they know the right ways to find the correct solutions. Their curiosity expands to learn the right concepts and specific procedures, and ways of finding the correct solutions from authorities.
4. Late Multiplicity: Persons believe that some problems are unsolvable. Their curiosity is expanded to know what different experts say about such problem. However, they believe that one can choose any solutions for such problems as per one's choice because there are no non-arbitrary bases to determine what is right.

Relativism

5. Contextual Relativism: Persons believe that all solutions must be evaluated in context and relative to their support by real evidence and logic. Their curiosity expands to learn to differentiate between weak and strong evidence, and to learn the analytic methods to evaluate solutions in the light of context, logic, and evidence.
6. Pre-Commitment: Persons start to see the need of integrating intellect with ethics for finding solutions in a contextual relativistic world. Their curiosity expands to learn to explore alternatives in open-ended problem solving, to make judgments based on personal and articulated standards, and be open to changing circumstances. However, persons at this level do not yet well consider or feel committed to their standards.

Commitment

7. Initial Commitment: Person makes actual commitments in personal directions and values as standards for open-ended problem solving and decision making.
8. Challenge to commitment: Persons experience the implications of their chosen commitments and standards, and also explore the issues of responsibility. Their curiosity expands to learn to evaluate the consequences and implications of their commitments, and to resolve conflicts.
9. Developing commitments: Persons develop a sense of self in both commitments and style and realize that commitment is an evolving activity. Their curiosity expands to learn to evolve and unfold their commitments in an ongoing manner.

Does Education Arouse Curiosity?

Interpreting Hume's view on curiosity, Brand concludes [242] that *education is not so much about imparting the content of an inquiry, but has more to do with inquiry, process, activity, and finally a sense of pride that comes from ownership*. He also observed that unlike content,

curiosity is elicited rather than imparted. He quotes Hume, ‘‘*What is easy and obvious is never valued; and even what is in itself difficult, if we come to the knowledge of it without difficulty, and without any stretch of thought or judgment, is but little regarded."*

Commenting on the inadequacy of modern education methods to promote curiosity, Einstein said, *"It is, in fact, nothing short of a miracle that the modern methods of instruction have not entirely strangled the holy curiosity of inquiry."* Studies show that most under-graduates enter college at a Perry level 3, and graduate at a level 4 showing an average advancement only by 1/3 of a unit on a nine-point scale in four years [253]. Longitudinal studies have also shown that in the first three years, there is not much forward movement in the engineering students' level as per Perry's nine stage mode [254-255]. To understand the reasons of this phenomenon, in 2005, we carried out an empirical study through a detailed questionnaire on nature of questioning in the class. Twenty-nine undergraduate students of computer science and engineering and information technology gave their responses. A summary of their responses is given in Appendix A9.

Appendix A9 suggests that usually the *classroom teaching is not oriented towards arousing or raising the level of curiosity*. *Consequently, we posit that higher education must motivate students to raise the levels of their curiosity on this hierarchy. Hence, Perry's model (Table 6.1) is included in our proposed framework of pedagogical engagements in software development education* (ref: Section 6.5 and Table 8.2, second column).

Enabling and Inhibiting Factors

Peterson et al [246] give an overview of the research on *enabling and inhibiting factors* that influence curiosity. *Novelty, complexity, uncertainly, and conflict* may work both ways depending upon the person's appetite. Arnone [245] identified six instructional elements that can arouse curiosity**:** *incongruity, contradictions, novelty, surprise, complexity, and uncertainty*. Arnone [245] also suggested some instructional strategies for fostering curiosity among students.

Peterson et al posited that perceived probability that the knowledge is attainable, and perceived probability that personal resources can be expanded by integrating new knowledge, determine

the level of curiosity. The fueling factors also include increased knowledge and awareness of knowledge gaps in areas that are personally meaningful and engaging [246]. Impediments include anxiety, overconfidence, excessive self focused attention, dogmatism, low cognitive resources, internal pressures like guilt and fear, external pressures like threat, punishment, and tangible rewards or pathological conditions. *These suggestions are embedded in our proposed framework (Section 8.2.1).*

Pedagogical Implications

In our recently concluded survey, "Software developers - (How) Did your college help you in your development?" (Table A10.2 (iii), Appendix A10), a large fraction of 66% felt that as compared to all other kind of academic engagements, their student projects did much better to develop their 'curiosity.' This was followed by research literature survey (62%), thinking oriented lectures (42%), laboratory work (38%), discussions with faculty, discussion with peers, and industrial training (36% each), and mentoring juniors (32%). Written examinations and discussion with others were found to be least effective in this regard by these respondents.

Given the nature of the problems, software developers need to solve, computing *students need to be repeatedly engaged in asking questions like*: (i) What (else) can be technology enabled? (ii) (ii) How can we do this using available and forth coming technology? (iv) What resources are needed? (v) Is this approach efficient, effective, and/or appropriate? (vi) What is the scalability and sustainability of this approach? (vi) What are unintended consequences and risks? Further, *negatively phrased question (why not?...)* are also equally important.

Hence, in order to arouse curiosity of various kinds and at various levels, the education process has to be necessarily made *student-centric* where they learn to *ask variety of questions as per Anderson and Krathwohl's taxonomy, Carson's taxonomy, and also Perry's levels*. Repeated and continued engagement in *challenging questions, open-ended problem solving and projects, collaboration, community work, mentoring, etc*., offer such opportunities. Finally, in order to continuous develop and evolve students' intellect, and develop their zeal for excellence and elegance, their *curiosity in great works of literature and arts* should also be developed.

Section 6.2: Software Developers' Education for Development of Decision Making Perspective

Importance of Decision Making

Decision making is about *choosing intelligently among less than perfect possibilities.* Professional decisions are broadly classified in three categories according to their scope: (i) *strategic decisions* concern general direction, long term goals, philosophies, and values; least structured and most imaginative; most risky and with most uncertain outcome, (ii) *tactical decisions* support strategic decisions; tend to have medium range, medium significance, with moderate consequences, (iii) *operational decisions* support tactical decisions; are structured and often made with little thought; impact is immediate, short term, short range, and usually low cost; can be preprogrammed, pre-made, or set out clearly in policy manuals.

With respect to software development, the developers broadly need to take decisions on two issues:

(i) What is to be (to visualize the product) and

(ii) How to deliver what is certain to be in the product.

These can be viewed as product decisions, project decision, and process decisions. In this context, it is very important that operational decisions in all these categories are consistent with tactical and strategic decisions.

In our 2009 survey on required competencies for software developers, twenty software professionals assigned decision making skills an average rating of 2.85 on a scale of 0-4. A large majority of 75% of these respondents recommended decision making to be a critical or very important competency with respect to the requirements of software developers' multi-faceted professional activities. Recently, we also concluded a poll among software professionals that was conducted for over one year. The respondents were asked to identify the weakest area addressed by engineering education in computing related disciplines. The offered choices were conceptual knowledge, decision making ability, learning ability, procedural knowledge, and thinking ability.

With reference to Appendices A2 and A3, decision making competence of software developers also relates to the following:

1 Perseverance and commitment
2 Integrity and authenticity
3 Accountability and responsibility
4 Project planning and management, project scoping, estimation, process planning and management
5 Entrepreneurship
6 Persuasion and negotiation skill
7 Sense of urgency and stress management

Fifty-eight professional responded to our poll. Around 30% of the respondents' age was above 35 years, and around 50% were in the age group of 25 to 34. 65% of the respondents worked for large or enterprise size organizations and remaining worked for small or medium size organizations. Responsibility allocation among the respondents varies as 64% in engineering, 12% in consulting, 8% in academics, and 4% each in creative, marketing, and operations. The distribution of their choices is as follows:

Decision making ability	41%
Thinking ability	24%
Procedural knowledge	15%
Conceptual Knowledge	15%
Learning ability	3%

A very large fraction of the responding software professionals consider that the weakest contribution of engineering education in computing related disciplines is in the area of developing students' decision making ability.

Decision making requires a decision making perspective which is complementary but independent of intrinsic motivation to create/improve artifacts, curiosity, and systems level perspective. It is done in the light of one's personal as well as organization's values, and is highly affected by one's sensitivity and awareness of socio-economic and other broader concerns. Decision making perspective requires taking decisions based on information and

evaluation of alternatives against objectives. In order to strengthen students' decision making ability, software development education has to sensitize its students to multi-dimensional aspects and also some well known techniques of decision making.

The decision making process requires software teams to blend short term as well as long term perspectives. Long term perspective focuses on sustainability that includes concerns for stability, efficiency, and scalability. Often senior management is found expressing their concern about new software developers' tendency to rush the problem by making a solution that addresses the operational problems of the customer without looking for strategic solutions.

Decision Deficiencies

Salas and Klein [256] have identified *five forms of decision deficiencies*: (i) *aim deficiency* occurs when a decision fails to meet a decision makers explicitly stated aim, (ii) *need deficiency* occurs when a decision maker fails to meet the actual need(s) in a given situation, (iii) *aggregate outcome deficiency* occurs when, collectively, all the outcomes of a decision (even beyond aim and need) leave the decision maker worse off than some effective reference, (iv) *competitor deficiency* occurs when, in aggregate, a decision is inferior to some competing alternative, and (v) *process cost deficiency* occurs when the cost of arriving at a decision is very high. *This model is included in proposed framework of engagements (Table 8.6b).*

Some Theoretical Models about Decision Making

Elaborating upon the social and creative dimension of decision making, Allwood and Selart [257] have emphasized on the importance of *restructuring the decision task through many iterations* of problem redefinition as well as goals, and also reformulating the problem space, seeing the broader context. *Their suggestion about iteration is integrated in our proposed framework (Table 8.5).* They also recommend *viewing decision making as a synthesis rather than analysis activity*, and insist on building a more holistic relational mental model. To generate alternatives, they recommend generating a large pool of alternatives by focusing attention on more unusual aspects of problem situation. As per them the alternatives need to be evaluated by integrating intuition and insight with logic and analysis.

Ullman posits that in real-life there are no right decisions but only satisfactory decisions. *Decision making is about finding the best possible satisfactory decisions* [258]. He posits that it is not an event or an action, but a process of repeatedly finding out what-to-do next. He has defined robust decision as the best possible choice, one found by eliminating all the uncertainty possible within available resources, and then choosing with known and acceptable levels of satisfaction and risk. He also posits that decision management is determining what-to-do-next with the available information in order to make most robust decision as part of standard work processes, and documenting the results for distribution and reuse. He has attributed *information uncertainty to factors like knowledge limitations, incompleteness, approximations, viewpoint differences, terminology imprecision, inconsistency, and information's evolving nature*. He suggests that *decision making requires effort for uncertainty management* to make the best possible use of the uncertainty that cannot be eliminated.

Further, he has identified some ***decision making challenges***. These are *conflicting interpretations, conflicting priorities, incomplete understanding of the criteria of evaluation and risks of each alternative, and absence of good decision making strategy.* In order to develop decision making competence, students need to be given practice in decision making through such challenging situations. *Hence, we include these challenges in our framework of pedagogical engagements (Table 8.6b).*

Seyedjavadein and Fahimi have recommended the use of TRIZ principles, cited in section 5.3, to generate alternatives during decision making [259]. As operational level decisions focus is on simplification and efficiency, the decision maker should seek the most *positive alternatives* which would add to the value of the system. For strategic level decision making, they recommend seeking positive *alternatives* which would add to the value of the system, while avoiding the *threats* to the system. For safety level decisions, seeking potentially negative alternatives which would damage the system is important in order to prevent them. For security level decisions, one needs to seek the *most negative alternatives* which would damage the system seriously in order to prevent them at any cost.

Taxonomy of decision making

Rowe and Boulgarides [260] have designed a two dimensional taxonomy of decision making styles with respect to management education. They have identified four different styles of decision making. The four styles differ from each other mainly in two dimensions: (i) need for high structured-ness vs tolerance for ambiguity, and (ii) focus on technical aspects vs focus on people and their needs. The summary of these four styles is given in Table 6.2. *With reference to decision making in software development, we posit that the software developers need to integrate these styles. This perspective is included in our proposed framework of pedagogical engagements in software development education* (Ref: Section 6.5 and Table 8.5).

Table 6.2: Four decision styles proposed by Rowe and Boulgarides

i. Directive style: This style is characterized by a low tolerance for ambiguity and rational way of thinking. It uses limited data and considers limited alternatives. This style is good for such technical issues that require lower cognitive complexity and have short range impact. It is especially suitable for implementing operational objectives by using rules and procedures in a systematic, efficient, and satisfactory way. It is more suitable for seeking acceptance and avoiding conflicts.
ii. Behavioral style: This style is characterized by a low tolerance for ambiguity and intuitional way of thinking. Like directive style, it also uses limited data and considers limited alternatives. This style is good for such people related issues that require lower cognitive complexity and have short range impact.
iii. Analytic style: This style is characterized by a high tolerance for ambiguity and rational way of thinking. It involves consideration of large amount of data from multiple sources, and evaluation of multiple alternatives. This style is suitable for such challenging technical issues that require focus on long range, and creativity. Analytics decision making is particularly useful for situations that require significant effort of analysis, planning, and forecasting.
iv. Conceptual style: This style is characterized by a high tolerance for ambiguity and intuitional way of thinking. Like analytic decision style, it also involves consideration of large amount of data from multiple sources, and evaluation of multiple alternatives. This style is suitable for such challenging people related issues that require focus on long range, and creativity. Conceptual decision making is particularly useful for situations that require exploring new options, initiating new ideas, forming new strategies, being creative, taking risks, people oriented-ness, and ethical considerations.

Becker and Connor [261] have found that immediate gratification values are significantly related to the tendency to use a directive decision-making style. *Delayed gratification* values are related to a preference for a conceptual style. *Competence values* are related to a directive style, while *conscience values* are related to a behavioral decision-making style. *Self-constriction values* (self-controlled, responsible, logical, and obedience) are related to a behavioral style, while *self-expansion values* (broad-minded, cheerful, and imaginative) are related to a conceptual decision-making style.

In a laboratory experiment, software project managers with *directed and analytics* decision making style were found to respond better to performance measure criteria of time to complete project plan, completeness of initial project plan, and variances in a project plan, and scope change in a project plan [262]. *Behavioral style* decision making responded better to change of end date in a project plan.

The traditional engineering education model strengthens directed and analytics styles which are apt for taking decision regarding how to deliver what is certain to be in a product. However, *with respect to taking decisions regarding visualizing and defining the product to be, conceptual style has to be strengthened*. The evolutionary approaches to software development share many similarities like people orientation, openness, trust, and shared goals with conceptual style of decision making.

With reference to decision making in software development, we posit that the software developers need to be able to integrate the four decision making styles identified by Rowe and Boulgarides, Table 6.2. *Hence, we include this in our proposed framework of engagements (Table 8.5)*

PrOACT and PROACTIVE approaches

Hammond et al have created a framework for effective decision making, *PrOACT (Problem, Objectives, Alternatives, Consequences, and Trade-offs)* [263]. As per PrOACT, decision making consists of *eight elements – formulating the problem in terms of its context and essential elements, clarifying key objectives with priority to serve as decision criteria, creating alternatives using creative thinking, identifying consequences with accuracy and completeness, clarifying trade-offs, uncertainty, risk tolerance, and linked decisions*. The last three elements are not necessarily involved in all situations, but help clarify the decisions in volatile and evolving situations. Since, software development is a highly evolving situation, these aspects become very important for software developers' decision making. Hammond et al view decision making as a multidimensional task with analytical, psychological, social, cultural, and intuitive processes.

Hunink has extended PrOACT model for medical decision making in the face of uncertainty and resource constraints [264]. As per this extension, **PROACTIVE** approach includes: *defining the Problem, Reframing the problem from multiple perspectives, focusing on the Objectives, expanding the Alternatives, considering the Consequences and associated chances for each alternatives, identifying the Trade-offs involved, Integrating the evidence and values, optimizing the Value of interest, and Exploring uncertainty*. We consider it propose that it can be helpful software development related decision making. *We strengthen our proposed framework of pedagogical engagements with this model (Ref: Table 8.6b).*

Decision Oriented Model of Software Processes

Toffolon and Dakhi [265] have proposed a decision oriented model of software processes. As per this model, the software development decision making is taking decisions with respect to *four subspaces related to software projects: (i) problem space, (ii) solution space, (iii) construction space, and (iv) operation space*. The decisions in these four spaces are driven by two broad categories of purposes:

(i) decisions to *manage complexity and risk*, the two essential characteristics of software, and

(ii) decisions to *reduce the negative impacts of two kinds of accidental characteristics of software, i.e., uncertainty and complications.*

This model is included in proposed framework of engagements (Table 8.6b).

Decision Making for Risk Management in Software Projects

Risk has been viewed as the probability of suffering losses while pursuing goals due to factors that are unpredictable or beyond [266]**.** Risks can be internal or external. Internal risks arise because of inadequacies in process capability (including core and support functions), and organizational structure. External risks are caused by uncertainties in external conditions. *Risk management* requires a systematic approach of reducing the harms due to risks, making the project less vulnerable and product more robust. *It is very important aspect of decision making.*

Boehm [267], one of the pioneers of software risk management field described it as comprising of two functions (i) *risk assessment: identification, analysis, and prioritization;* (ii) *risk control:*

management planning, resolution, and monitoring. SEI's has also identified six elements of software risk management: *identify, analyze, plan, mitigate, track and communicate.*

Importance of Risk Management in Software Development Education

In one of **our recently concluded survey** among software professionals, fifty-seven professionals responded. The respondents professional experience distribution is as follows: (i) around 15% with more than 15 year experience, around 10% with between ten to fifteen years of experience, around 40% have five to ten years of experience, and the remaining with less than five years experience. A good fraction of *36% respondents recommended that risk planning and mitigation must be included in the main goals for new curriculum* for the future generation of software developers.

Software Risk Categorization Schemes

Boehm identified the top ten risks items. The top four in this list were personnel shortfall, unrealistic schedule and budget, wrong function and properties, and wrong user interface. According to Brian A Will, the top most risks include creeping software requirements, requirement gold plating, low quality of released software, and unachievable schedule.

Keil et al provided categorization framework for software project risks [268]. They categorized these risks into *four quadrants*. The *first quadrant* risks relate to *customers and users*. These risks have a high level of perceived importance but a low level of control possibility. Hence, mitigation is essentially done by increasing users' participation and commitment to the software project. The *second quadrant* risks relate to *ambiguities and uncertainties about scope and requirements*. These risks have high perceived importance as well as a high level of control potential for project managers. The *third quadrant* risks relate to *execution* that has moderate perceived importance but high level of control is possible. The *last quadrant* risks relate to *environment* and have moderate perceived importance as well as low control possibility for project managers.

Wallace and Keil have further classified fifty software risks into these four categories [269]. They also analyzed the effect of these risks on process and product outcome. They have

concluded that for project managers, minimizing and managing the execution, scope and requirement related risks are critical from both perspectives. Further, they observed that in situations where product outcomes are more important than time and budget, the risks related to users and customers also become very critical.

SEI has proposed two taxonomies. First, they catalogued and classified one hundred and ninety-four risks into the *three broad level categories* [270] of *product engineering, development environment, and program constraints* Later, SEI proposed another *taxonomy for software development risks for high-performance computing scientific/engineering applications.* This taxonomy classifies the sources of software development risks into the *three broad categories* of *development cycle risks, development environment risks,* and *programmatic risks.*

Pandian has given a distribution of software development risks. As per his analysis, 70% risks are internal and only 30% are external. *Project risks* account for 30%, *product risks* for 30%, and *process risks* for 40%. In the process risks, the most vulnerable areas are related to human resource and requirement issues, and least vulnerability is found to exist in coding and testing [266]. Georgieva et al have provided a survey of software risk assessment methods [272].

We include risk assessment for identifying, analyzing, and prioritizing project, product, and process risks in our proposed framework (Table 8.6a). We also recommend the use of SEI taxonomies as checklists.

Ethical Decision Making

Professional decision making ability is not only related to technical competence only. Instead of just technical competence, intelligence, or creativity, it is related to professional wisdom. **Sternberg** [229] has defined *wisdom as the application of tacit as well as explicit knowledge, as mediated by values, towards the achievement of a common good through a balance among (a) intrapersonal, (b) interpersonal, and (c) extra-personal interests over the (a) short term and (b) long term to achieve a balance among (a) adaptation to existing environments, (b) shaping of existing environments, and (c) selection of new environments.*

Boyle [273] has proposed a six-stage process of ethical decision-making for computing professionals. The *first stage is about moral perception* and personal knowledge of the moral good, which depends upon the ability to recognize that an ethical problem exists and that a person has some personal responsibility to respond. The *second stage is of the moral discernment* and personal ability to think logically, which enables a person to state the ethical problem clearly. The *third stage is of moral resolution* and personal ability to analyze complexities of the stated problem, in order to arrive at an individual position which is justifiable to one's self conscience. The *fourth stage is of moral assessment* and personal ability to assess one's freedom. According to Boyle, computing professionals must be aware of new developments, particularly in the context of the history of technology in the computing field, in order to handle the new freedoms properly. The *fifth stage deals with moral decision* and personal knowledge of one's duties. The *last stage is of moral action* and personal willingness to follow one's intellect.

IEEE-ACM code of ethics for software engineers provides directions and guidelines for all these stages, except the second and last stage. The second and third stage depends upon the critical thinking ability, and the last stage depends upon one's value system. Boyle sees the entire process as circular, such that the moral actions of one cycle shape the moral perception for next cycle. *We strengthen our proposed framework of pedagogical engagements with this model (Table 8.6b).*

Pedagogical Perspective

In **our recently concluded survey,** "Software developers - (How) Did your college help you in your development?" (Table A10.2 (iii), Appendix A10), a large fraction of *77% felt that as compared to all other kind of academic engagements, their student projects did much better to develop their decision making skills*. This was followed by laboratory work (38%), mentoring juniors and industrial training (35% each), thinking oriented lectures (31%), and discussions with other students (31%). *Traditional knowledge delivery oriented lectures and written examinations were found to be least effective in this regard* by the respondents. *Further, 90% and 71% respondents respectively felt that as compared all other engagements, student projects and industrial training did much better to enhance their project planning and management skills.*

All other engagements were felt to be ineffective in this regard. We can conclude traditional form of engineering education, misses the opportunity to develop students decision making thinking and perspective.

Students need to learn to take decisions related to product, process, and project. Such ability can be developed by developing their decision thinking perspective. Development of such perspective requires exposure to *wider contexts, and also reflection on senior professionals' decisions taken in tricky situations.*

In addition, the education programs must also engage students in professional decision making in real-life like situations. Typical academic engagements like traditional lectures, short assignments, written examinations, and textbook oriented exercises do not create such engagement. *Student-centric learning engagements like semester long group projects offer a great potential to give them opportunities to take and improvise their decisions*.

Decision thinking is not automatic, but controlled thinking [274]. Students' decision making ability can only be developed by developing their decision making perspective. In this section we have discussed some important models and tools that can help in developing their decision perspective. The decision oriented models suggested by (i) Toffolon and Dakhi as well as (ii) Boyle (with respect to IEEE-ACM code of ethics) are can be very useful for students. Students should to be exposed to the decision deficiencies identified by Salas and Klein and decision challenges identified by Ullman. As suggested by Allwood and Selart, they must be required to iterate over the decision tasks. The student engagements should require them to integrate the four decision styles suggested by Row and Boulgerides. They must also be engaged in risk management of product, project, and process risks. SEI taxonomies can be used as checklists. *We include these in our proposed framework of pedagogical engagements (Tables 8.5 and 8.6b).*

Further, literature on decision making offers some excellent *general purpose techniques* that have been used in various professions. Some of these are: *Pareto analysis, paired comparison, T-Chart, decision matrix, grid analysis, PMI (Plus, Minus, and Interesting), decision Tree, six thinking hats, star-bursting, step-ladder, and Delphi*. All these techniques are essentially

manifestations of the core idea of decision making as a process of making choices. These involve generation of alternatives and evaluating their consequences. These and other such techniques can also be very effectively used by software developers for taking effective decisions during various activities of software development. Hence, computing students should be well exposed to these techniques through their curriculum. *These techniques are used to supports our proposed framework of pedagogic engagements/ software development education is on our future agenda (Ref: Table 8.6b).*

Finally, as per our exploratory study of students' software projects, we have found that normally student projects do not expose them with many typical risks in software projects. The most common risks in student projects are due to lack of their proficiency with development tools and/or open source, and unrealistic estimates. They do not get exposed to other typical software project risks discussed above. *We hypothesize that the student projects also need to be viewed and administered from an additional perspective of exposing them to common real-life software project risks.* Typical projects designed in protected academic setting often do not achieve this goal. *In future extension of our work, we plan to carry forward this idea and propose an appropriate model of administering and designing student projects.*

Section 6.3: Software Developers' Education for Development of Systems-level Perspective

Merriam Webster's Collegiate Dictionary, tenth edition, defines 'system' as "*a regularly interacting or interdependent group of items forming a unified whole*." This, however, is only a partial view of systems. Thinkers of various disciplines like engineering, management, science, economics, sociology, political science, etc., have contributed significantly to understanding systems as well as systems thinking. According to Meadows, *a system is more than the sum of its parts, its part are simultaneously interconnected in multiple directions, it has a purpose, it produces its own behavior over time and its response to external triggers and/or forces is a characteristic of itself.* In real-life these responses are usually very complex [275]. Checkland [275a] identifies the *four classes of systems*: *natural systems, designed physical systems, designed abstract systems, and human activity systems*.

"Systems thinking" is seeing wholeness, seeing interrelationships rather than individual things. Isolated knowledge by a group of specialists generated in a narrow field has no value in itself, only its synthesis with the rest of the existing knowledge gives it a meaning [277]. Solovey [279] found the eleven laws of system thinking proposed by Senge [277] to be applicable to software development. Annexure AN9 gives these laws.

Importance of Systems Thinking for Software Development

In our 2009 survey on required competencies for software developers, twenty software professionals assigned 'systems-level perspective' an average rating of 2.95 on a scale of 0-4. A large majority of 85% of these respondents recommended it to be a critical or very important competency with respect to the requirements of software developers' multi-faceted professional activities.

Usually software programs are complex systems. They are executed on computing environments that are examples of complex systems. Software is usually a critical subsystem of a larger technical and/or organizational/social system. Further, the development life cycle of software is another example of a very complex social system. In the context of software development, holistic understanding of the problem and solution paves the way for a robust implementation. Metzger recommends the inclusion of gestalt understanding for debugging tasks [157]. As software-based systems have grown larger, more complex, and need inter-disciplinary inputs, the capacity of systems thinking has become crucial for software developers. Concerns like user-centeredness, reuse, integration with existing subsystems, legacy systems, quality, cost, security, availability, and maintainability make it imperative to develop systems thinking.

With reference to Appendices A2 and A3, systems-level perspective of software developers also relates to the following:

1 Ability to accommodate himself to others, empathy, *"be the customer" mentality* - genuine interest in understanding what other people are trying to accomplish and based on this understanding think about creating technical solutions to help them reach their goals. Genuine interest in understanding "why to create software" and the broader context of

software systems. Cognitive task analysis. Appreciation of unstated requirement and ability to identify these. *Listening skills,* approachable, and respect for people. *Ability to work in homogeneous, multi-disciplinary, multi-locational and multicultural teams*. Ability to work under supervision and constraints, Understanding of the impact of personal character and behaviors on others.

2 Ability to see the self as bound to all humans with ties of recognition and concern. Seek help from other, *Ability to help and assist others, mentoring, commitment to others' success. Sensitivity towards global, societal, environmental, moral, ethical and professional issues, and sustainability*. Respect for the intellectual property of others. Work ethics.

3 Organizational skills.

4 Quality, *cost,* and security consciousness, pursuit of excellence, intellectual *accountability and responsibility*, intellectual *integrity,* intellectual courage, strength of conviction: assertive without being aggressive. Commitment to systematic documentation of the work. Recognize and act upon the need to consult other experts, especially in matters outside their area of competence and experience. Commitment to the fulfillment of needs of all users and persons who get affected by the technological solutions. Eagerness and inclination to understand the unintended consequences of creating software inappropriate or at odds to its real purposes. Commitment to health, safety, dignity, and welfare of the users and also the people who will be affected by their systems. Sensitivity towards constraints like economic disadvantage and physical disabilities that may limit software accessibility.

5 Self-acceptance, self-regulation, self-awareness, self-improvement: strength to resist instant gratification in order to achieve better results tomorrow. Being honest and forthright about one's own limitations of competence. Tendency to avoid false, speculative, vacuous, deceptive, misleading, or doubtful claims. Faith in reason and review, inclination for verification and validation, respect for facts and data. Awareness and regulation of automatic thoughts.

6 'Big picture' view, holistic and multi-perspective thinking, knowledge integration, consideration for multilateral viewpoint, and user-centeredness. Process and rule-oriented mindset. Tolerance to ambiguity and risk. Ability to understand and also build upon other's work. Ability to work such that others can easily understand and build upon.

7 Perseverance and commitment.

8 Complex problem solving skills

9 Analytical thinking

10 Design skills

Systems Engineering Perspective

Frank and Waks [280] have given a multifunctional comprehensive definition and explanation of engineering systems thinking. Further they have also given the characteristics of engineers who are able to demonstrate such thinking. This definition links multiple options of seven facets. The definition is given in Table 6.3. We use this to strengthen our framework of pedagogical engagements (Table 8.6a).

Table 6.3: Multifaceted definition of engineering systems thinking by **(**Frank and Waks, 2001)

The engineering systems thinking of a/an *[facet A – specialization field]* engineer who deals with a system of a *[facet B- complexity level]* level of complexity involves the ability to understand *[facet C – systems aspects and implications]* and the *[facet D - interrelationship] [facet E- interconnections]* and to *[facet F – functional domain]* without *[facet G – constraints].*
Options for *[facet A – specialization field]*: (i) electrical, (ii) electronics, (iii) computers, (iv) software, and (v) others
Options for *[facet B- complexity level]*: (i) very low, (ii) low, (iii) intermediate, (iv) high, and (v) very high
Options for *[facet C – systems aspects and implications]:* (i) understanding the whole system, (ii) understanding the synergy of the system, (iii) understanding the contribution of components of the system, (iv) understanding the system from multiple perspectives, (v) understanding the implications of modification to the system, and (vi) understanding a new system immediately upon presentation
Options for *[facet D - interrelationship]*: (i) interaction between, (ii) hierarchy of
Options for *[facet E- interconnections]:* (i) internal subsystems, (ii) external neighboring systems
Options for *[facet F – functional domain]:* (i) locate system failures, (ii) outline failure solution, (iii) analyze/dismantle system to individual components, and (iv) synthesize/design subsystems linkages to a whole
Options for *[facet G – constraints]:* (i) need to understand details for understanding the whole, (ii) refraining from multitasking, and (iii) "getting lost" when dealing with system issues or acting in a non-familiar professional environment

Levels of systems thinking

Sanford observed that our upbringing, and particularly our education, has trained our thought patterns to follow a segmented and reductionist path. The new capability to see and to think in terms of systems thinking also starts with being able to "envision" relationships and structural components of nested whole ways of thinking. There are *five levels of systems thinking: closed, cybernetic, complex adaptive, developmental, and evolutionary* [281]. We use the lower levels of Boulding's levels (Annexure AN9, Table AN9.4) to extend this hierarchy as depicted in Table 6.4. We also merge the levels of closed and cybernetic systems into a single category. *This*

modified ladder is integrated in our proposed framework of pedagogical engagements in software development education (Table 8.2, fourth column).

Table 6.4: Levels of systems thinking (derived from Boulding and Sanford)

1	Pre-structural thinking: seeks to understand, analyze, build, evaluate, and maintain the components
2	Structural thinking: seeks to understand, analyze, build, evaluate, and maintain static structures and frameworks involving various components.
3	Clockworks thinking: seeks to understand, analyze, build, evaluate, and maintain predetermined motion.
4	Closed systems thinking: seeks to understand, analyze, build, evaluate, and maintain mechanisms, seek stabilization within tolerance and standards by allowing limited access and exchange with systems outside their boundaries in spite of richer interactions through a feedback based control.
5	Complex adaptive systems thinking seeks to understand, analyze, build, evaluate, and maintain effectiveness of open systems in the context of a continuously dynamic and evolving environment.
6	Developmental systems thinking: seeks improvement by uncovering the full potential and expression of the unique essence of any entity or system, including the greater system of which we are a part. It involves re-conceptualization of the values by exploring the core value, core process, and core purpose interactively looking beyond themselves.
7	Evolutionary systems thinking: this is generative field of evolving systems, it requires looking at the entire value chain and context and beyond what they serve.

Shifting the Focus for Systems Thinking

Capra [282] had proposed *five criteria for systems thinking* in natural sciences, as given in Table 6.5. The first two criteria refer to our view of nature's complexity. In addition, the next three criteria refer to our epistemological beliefs and uncertainty.

Table 6.5: Shifting the focus for systems thinking (Capra's criteria)

1	Shift the focus from parts to whole, the properties of the parts can be understood only from the dynamics of the whole; part is merely a pattern in an inseparable web of relationships
2	Shift the focus from structures to process; every structure is a manifestation of an underlying process, the entire web of relationships is dynamic.
3	Shift from the objective science to epistemic science, the understanding of the process of knowledge has to be included explicitly in the description of natural phenomena.
4	Shift from building to network as metaphor of knowledge, there the material universe is seen as a dynamic web of interrelated events, there are no fundamental entities whatsoever: constants, laws, or equations, none of the properties of any part of this web is fundamental, they all follow from the properties of the other parts, and the overall consistency of their interrelations determines the structure of the entire web.
5	Shift from truth to approximations, all scientific concepts and theories are limited and approximate, scientists do not deal with truth but with limited and approximate description of reality.

Blaauw's Principles of System Architecture

Blaauw [282a] has identified eight principles of good architecture. These include *consistency, orthogonality, propriety, parsimony, transparency, open ended-ness, generality, and completeness*. Deliberate usage of these principles for evaluation of software architectures can

help a great deal to improve them. *These are included to support our framework of pedagogic engagements (Table 8.6b).*

Soft Systems Methodology for Solving Soft Problems

The famous waterfall model of Structured Systems Analysis and Design Method (SSADM), is based on Checkland's [284] Soft Systems Methodology (SSM) that was developed in late 1980's for *solving soft problems*. *Soft problems are such problems that have multiple stakeholders with divergent values, beliefs, philosophies, interests, and also views about what the problem is.* This iterative approach consists of seven distinct stages given in Table 6.6. Recently, Jacobs [285] proposed an approach to applying systems thinking. This is also included in Table 6.6.

Table 6.6: Systems thinking approaches by Checkland and Jacobs

Checkland's Stages of Soft Systems Methodology for Solving Soft Problems	Jacobs' approach for applying systems thinking
1 Define and understand the problem situation (i.e., nature of the process, key stakeholders, etc.), 2 Express the problem situation through rich pictures, 3 Select how to view the situation from various perspectives and produce root definitions, 4 Build conceptual models of the system requirements to adequately address each of the root definitions, 5 Compare the conceptual models to the real world expression, 6 Identify feasible and desirable changes to improve the situation, and 7 Develop recommendations for taking action to improve the problem situation.	1 Explore the event/problem from multiple perspectives without jumping to solutions, 2 Track the situation over a period of time and identify patterns and trends of behavior that go below the surface, 3 Look for systemic structures such as interrelationships in the patterns and trends, balancing and reinforcing feedback, and delays, also understand the mental models that are driving these patterns, and 4 Create new mental models to introduce change into the system, track and evaluate the effects of the changes, and identify unintended consequences and decide what needs modification.

Software as Socio-technical Systems

The criteria identified by Capra and characteristics proposed by Sweeney and Meadows are highly relevant for software developers. In his system theory, Senge [286] argues that we often complicate the nature of the problem, because we tend to treat problems as if we are outsiders, rather than treating the problems and ourselves as one. He further posits that systems thinking also aims at integrating one's insights into the inner systems and visions of the outer systems, and we need to transform the technical mode of working into the spiritual pursuit of work ethics. As *many software systems are socio-technical systems and the software development systems are*

essentially social systems, Senge's perspective is even more relevant in the context of software development.

In our 2009 survey on required competencies for software developers, *twenty software professionals assigned 'accommodate oneself to others' an average rating of 3.1 on a scale of 0-4. A large majority of 80% of these respondents recommended it to be a critical or very important* competency with respect to the requirements of software developers' multi-faceted professional activities. In order to do cognitive task analyses, software developers need to have a genuine interest in understanding what other people are trying to accomplish, and based on this understanding think about creating technical solutions to help them reach their goals. In order to identify unstated requirements, they need to have a genuine interest in understanding 'why to create software' and also the broader context of software systems. Development of this kind of genuine interest requires the virtue of empathy as manifested in 'be the customer/user mentality' and tolerance to ambiguity. Often, software developers have to work in large teams of developers that are temporally and often geographically distributed and even culturally diverse. This requires the developers to have the ability to understand and also build upon other's work and also the ability to work such that others can easily understand and build upon.

In another survey conducted us of fifty-seven software professionals (Table 4.1), *65% of our respondents included 'group work, people management, and leadership' as one of the most important activities* that must be included in the main goals for new curriculum for the future generation of software developers. This ability also requires an attitude and ability to *'accommodate oneself to others.'* Such ability also makes one more approachable.

Ethical Aspects of Systems Thinking

Hoffman saw empathy as the key to moral motivation [287]. **In our 2009 survey** on required competencies for software developers, *twenty software professionals assigned 'see the self as bound to all humans with ties of recognition and concern' an average rating of 2.65 on a scale of 0-4. A majority of 60% of these respondents recommended it to be a critical or very important* competency with respect to the requirements of software developers' multi-faceted professional activities.

Commitment to the fulfillment of needs of all users avoiding unintended consequences on safety, dignity, health, and welfare of the users and also the people who will be affected by their systems requires them to have empathy and a commitment for accountability and responsibility. Deeper sense of responsibility comes from attitude and ability to *'see the self as bound to all humans with ties of recognition and concern.'* This ability requires sensitivity towards global, societal, environmental, moral, ethical and professional issues and sustainability, as well as, respect for work ethics and intellectual property of others. The concern for sustainability requires sensitivity towards constraints like economic disadvantage and physical disabilities that may limit software accessibility. *Sternberg's definition of wisdom, discussed under the theme of ethical decision making in section 6.2, is very relevant in this context.*

Such well grounded sense of responsibility can facilitate self regulation to resist instant gratification (in order to achieve better results) and also avoid false, speculative, vacuous, deceptive, misleading, or doubtful claims about their competence, products, and services. It strengthens intellectual integrity, intellectual courage, and gives a strength of conviction. This ability to see the self as bound to all humans when combined with the ability to accommodate oneself with other strengthen one's ability to seek and provide help, participate as mentors/mentees/supervisor/supervise, have commitment to others' success, and be assertive without being aggressive. All these are very important for successful group work.

Cultivating systems thinking

Senge [278] developed a toolbox for cultivating systems thinking. This are given in table 6.7.

Table 6.7: Senge's toolbox for cultivating systems thinking

1	Learning how to draw systems maps, including a. the interaction between cause and effect, b. dynamic loop, c. system feedback perspectives, d. systems problems,
2	Learning how to describe reinforcing loops,
3	Learning how to describe balancing loops, and
4	Learning how to describe delays.

Software systems analysis and design techniques

Several semi-formal and formal techniques have been developed for software systems analysis and design. They offer powerful representation tools for data and behavior of software systems. *Data representation techniques* include conceptual data modeling techniques, knowledge representation techniques, ontologies, etc. *Behavior representation techniques* include FSM, State-chart, State Nets, Petri Nets, etc. We strongly recommend the frequent and repeated use of many of these semi-formal techniques in computing courses. *We include these techniques to support our proposed framework (Table 8.6b).*

Meta-Framework for Systems Engineering

Haskin [288] has proposed a *meta-framework for systems engineering*. The 6C's in her framework are: *Comprehension, Communication, Coordination, Cooperation, Collaboration, and Continuity*.

Comprehension needs listening, empathy and broader general knowledge. She posits that these six C's sit in the context of *Code of ethics*. Hence, we see that *systems thinking require a higher maturity level of not only cognitive development, but also emotional and moral development*.

Kohlberg [289] proposed a six stage model of human development based on their moral reasoning. Table 6.8 gives a summary of this model. *This is part of our proposed framework of pedagogical engagements in software development education (Section 6.5 and* Table 8.2, fifth column).

Table 6.8: Kohlberg's six stage model of human development

A.	**Pre-conventional level (Egocentric) (Self-centered)**
1.	Obedience and punishment: the moral reasoning is motivated by avoiding anticipated punishment.
2.	Individualism and Reciprocity: the moral reasoning is motivated by self interest.
B.	**Conventional Level (Socio-centric) (Conservative)**
3.	Interpersonal conformity: the moral reasoning is motivated by avoiding anticipated disapproval of others by 'looking' nice to them.
4.	Social systems and "Law and order": the moral reasoning is motivated by avoiding anticipated dishonor or institutionalized blame and desire for maintaining social order.
C.	**Post-conventional (Onto-centric) (Progressive)**
5.	Social Contract: the moral reasoning is motivated by concern of self-disrespect and broader social welfare.
6.	Universal ethical principles: the moral reasoning is motivated by maintaining respect and dignity of all by emphasizing human values and rights.

Moral development of a person is closely linked with the person's value orientation. Spini [290] refers to Schwartz [291] who saw people's values as their motivational constructs for deciding their actions. Schwartz value categories are discussed in Annexure AN9. Our exploratory survey of undergraduate computing students showed that, by and large, the responding *students felt that most of their peers lacked the values of self-direction, benevolence, and universalism*. However, a more systematic study is required on this aspect. Nevertheless, development of benevolence and universalism is crucial for developing the abilities to '*accommodate oneself to others*' and '*see the self as bound to all humans with ties of recognition and concern*.' Both these abilities are identified as key aspects of systems thinking. Further, development of *self-direction* is very important for arousing the intrinsic motivation to create/improve artifacts (discussed in the next section). Hence, the development of these values has to be addressed by software development education.

Pedagogical Perspecive

In **our recently concluded survey,** "Software developers - (How) Did your college help you in your development?" (Table A10.2 (iii) , Appendix A10), a large fraction of *68% felt that as compared to all other kind of academic engagements, their student projects did much better to develop their 'systems-level perspective.'* This was followed by research literature survey (46%), laboratory work (34%), and industrial training (32%). *Written examinations were found to be least effective in this regard* by these respondents.

In this survey (Table A10.2 (iii), Appendix A10), a large fraction of 64% felt that as compared to all other kind of academic engagements, their student projects did much better to develop their '*accommodate oneself to others*.' This was followed by mentoring juniors (56%), discussions with other students (46%) and industrial training (38%). Written examinations and lectures were found to be least effective in this regard by these respondents.

Further, in the same survey (Table A10.2 (iii), Appendix A10), a large fraction of 51% felt that as compared to all other kind of academic engagements, their student projects did much better to develop their attitude to '*see the self as bound to all humans with ties of recognition and concern*.' This was followed by mentoring juniors (45%), industrial training (41%) and

discussions with faculty, students, and others (33%, 31%, and 29% respectively). Written examinations and traditional form of knowledge delivery oriented lectures were found to be least effective in this regard by these respondents.

In this section, we discussed several models and tools that can be used to develop systems-level perspective of computing students. We derived a new ladder of systems thinking based on Boulding and Sanford ladders. Capra's suggestion for shifting the focus is very helpful in developing the mindset for systems thinking. Blaauw's principles for systems Architecture are excellent guidelines for systems analysts and designers. Systems thinking approaches by Checkland and Jacobs, and Senge toolbox are also very helpful for inculcating the habit of systems thinking. Kohlberg's six stages can act as ladders of moral development to take care of the moral aspects of systems thinking. Finally, as discussed in the previous section, use of the risk assessment techniques is also very helpful for developing systems thinking perspective. *We include all these models and tools in our proposed framework of pedagogical engagements (Table 8.6).*

Section 6.4: Software Developers' Education for Development of Intrinsic Motivation to Create/Improve Artifacts

In our 2009 survey on required competencies for software developers, *twenty software professionals assigned 'intrinsic motivation to create/improve things' an average rating of 2.9 on a scale of 0-4. A majority of 65% of these respondents recommended it to be a critical or very important* competency with respect to the requirements of software developers' multi-faceted professional activities.

With reference to Appendices A2 and A3, 'intrinsic motivation to create/improve artifacts' of software developers also relates to the following:

1 Design skills
2 Creativity and idea initiation
3 Complex problem solving
4 Entrepreneurship, initiative taking, enjoys challenges, sense of mission, perseverance, result orientation, commitment, self motivation, dedication. Adaptability, flexibility, open-mindedness, and ability to multi-task

5 Research skills

6 Experimentation skills

7 Readiness for lifelong learning

Intrinsic Motivation

According to Webster's Dictionary, motivation is "the psychological feature that arouses an organism to action;" and "the reason for the action." Psychologists have carried out extensive research on various aspects of motivation. Motives influence one's perception, cognition, emotion, and behavior [292]. Annexure AN10 summarizes the perspective of Aristotle, Descartes, James and McDougall, and Murray on this issue.

In 1943, *Maslow proposed his famous theory of hierarchy of human needs* [141a]. After later extension, his theory classifies human needs in a hierarchical structure of levels given in Table 6.9. He viewed that a person attempts to satisfy basic needs before directing behavior toward satisfying upper-level needs. According to him, people have a need to grow to move up the hierarchy of needs. The satisfied needs cease to motivate and unsatisfied needs can cause frustration, conflict, and stress. *We view that higher education must motivate students to raise the levels of their needs on this hierarchy. This hierarchy is part of our proposed framework of pedagogical engagements in software development education* (Ref: Section 6.5 and Table 8.2, third column).

Table 6.9: Maslow's Hierarchy of Human Needs

1. Biological and physiological needs,
2. Safety needs,
3. Belongingness and love needs,
4. Esteem needs,
5. Cognitive needs,
6. Aesthetic needs,
7. Self actualization needs,
8. Transcendence needs

Annexure AN10 includes some later perspectives by several researchers like Herzberg, Vroom, Alderfer (ERG theory), and Reis [294].

Ryff and Singer [245] have identified *six factors for psychological wellbeing: self acceptance, positive relations with others, autonomy, environmental mastery, purpose in life, and personal*

growth. Hence, satisfaction of higher level needs as per Maslow's model, motivator factors as per Herzberg's theory, or growth needs as per ERG theory is necessary for wholesome experience and happiness in life. Enrichment and advancement of needs from low-level to higher level is not automatic. Satisfaction of lower level needs does not automatically facilitate upward movement of motivation factors. In 1980's, Deci and Ryan proposed 'self determination theory' to suggest that humans have three innate psychological needs: autonomy, competence, and relatedness [296-298].

Motivation for Creativity

As per Sternberg, *motivation behind creativity is to go beyond what is known*. Sternberg [299] saw *motivation* at the centre of the processes that result in the development of expertise such that it affects meta-cognitive as well as knowledge acquisition activities. It also evolves as a result of learning and thinking. Ambile [300] proposed that the *intrinsically motivated state is conducive to creativity, whereas the extrinsically motivated state is detrimental*. Creativity research has found that *personal autonomy is a core characteristic of creative people*. Autonomous people consider their behavior as emerging from themselves, and may stay more deeply and creatively engaged in what they are doing. Self determined people may be more open to possible analogies or intuitions that are relevant to the problem with which they are concerned. They may also devote more conscious attention to problems that genuinely interest them [301].

Cognitive orientation theory [302] sees *motivation for creativity as a function of beliefs of four types (about goals, norms, oneself, and general beliefs about others and reality)* concerning themes identified as relevant for creativity. Their findings have shown that *there are attitudes and personality tendencies that promote creativity*. As per this study, the high and low creativity architecture students showed significant differences in the following themes:

i. feeling it is incumbent upon them to activate and use their talents and unique abilities
ii. interest and no discomfort in regard to views which differ or contradict their own
iii. daydreaming a lot
iv. demanding a lot from themselves
v. not in need of firm framework or strict regulations
vi. tendency to do original things

vii. tendency to delve deeply into what one deals with and examine it from all points of view
viii. thinking about things in one's own way, and not necessarily as one has been taught
ix. thinking and doing one's own thing even with no support from others
x. concern with the functionality of what one does
xi. ability and tendency to invest a lot of effort

Advancing this work, Caskin and Kreitle [303] have concluded that the belief system of *highly creative students of architecture and engineering disciplines put a lot of emphasis on self - its uniqueness, development, and expression. The second major factor is maintaining openness to the environment without endangering inner directedness*. They found that self beliefs as well as goal beliefs supporting creativity are higher in students of architecture and there are no significant differences with respect to their general and norm related beliefs. Architectural students scored higher than engineering students in the following groupings:

i. Self-development: investing in one-self and developing oneself; taking advantage of opportunities for promotion and learning; developing skills; not satisfied with any achievement but seeking more.
ii. Emphasis on the inner world: more interested in what takes place within oneself than in what occurs outside and in others; making efforts to learn about him/herself; feeling contradictions within the inner world; emphasizing the importance of fantasy
iii. Inner-directedness: making efforts to succeed in circumstances in which others tend to fail; lack of support from others does not affect self-confidence or self-esteem; clarity about one's goals.
iv. Emphasizing one's uniqueness: experiencing one's uniqueness; feeling that he/she has unique talents; developing and highlighting one's uniqueness; understanding things in his/her own way; being different from others; seeing things differently from others; making original things.
v. Functioning under conditions of uncertainty: liking ambiguity and uncertainty; liking to take risks; liking jobs in which not everything is clear; functioning even if he/she cannot control every detail of the process.

vi. Self-expression: expressing emotions outwardly; creating something personal; expressing thoughts, views, and skills; externalizing feelings; being loyal to own feelings and ideas; speaking with others about oneself.

vii. Non-functionality: able to work even if sees no immediate benefit; does not believe that every idea can be implemented in practice; compromising in regard to practicality and functionality; readiness to act even if functionality is not clearly stated, or not clearly requested from the start.

On the other hand, engineering students had higher scores in the following groupings:

i. Freedom in functioning: does not need a rigid framework of rules defining the situation and the conditions; unable to function according to the instructions of others; functions by intuition; need of freedom in thinking and acting; acting because he/she wishes and not because he/she ought to.

ii. Being receptive to the environment: extracts something from the environment even if it offers only a few stimuli (openness to the environment); absorbing from the environment as much as possible, not selectively; curious to learn a lot about every domain.

iii. Demanding from oneself: does not withdraw in the face of difficulties; striving for perfection, getting to the level of excellence one determines for oneself; high demands from oneself; Investing without limits; able to renounce comfort and pleasure.

Both the groups showed similar results with respect to their beliefs about contribution to the society: make something important and significant, even if it does not contribute to self-promotion; feeling that one can promote the general welfare; devoting time and effort to society; readiness to invest a lot to help people.

Pedagogical Perspective

In **our recently concluded survey,** "Software developers - (How) Did your college help you in your development?" (Table A10.2 (iii), , Appendix A10), a large fraction of *74% felt that as compared to all other kind of academic engagements, their student projects did much better to develop their 'urge to create/improve things' and open mindedness. This was followed by research literature survey (58%), thinking oriented lectures (54%), discussions with students and faculty (50% each), mentoring juniors (44%), and laboratory work (42%). Written examinations,*

traditional knowledge delivery oriented lectures, and homework were found to be least effective in this regard by the respondents. One of the main purposes of education is to sensitize and help its beneficiaries to enrich and nourish their intrinsic motivation towards growth oriented needs of cognition, aesthetics, self actualization, and transcendence needs.

Further, software development work requires significant design effort. In order to create interesting software, the developers need to first become intrinsically motivated interesting persons. Due to their nature, design problems cannot be solved by retrieving already existing solutions or by applying a routine process. Consequently, *it is very important that software development education programs create such conditions that ignite intrinsic motivation among its students for creating/improving things.*

Love for challenges, habit of perseverance, concentration, and initiative taking depend upon intrinsic motivation. Computing students also deserve to be self motivated to enjoy the pleasure of creative tasks for its own sake rather than for the associated extrinsic rewards. However, the above study clearly exposes a strong weakness of traditional engineering education in this aspect. It does not help the students much to evolve their attitudes and belief in support of creativity. *Hence, if computing students' intrinsic motivation for creativity needs to be enhanced for creating conditions for self actualization through creation, their education process needs to be significantly enriched, perhaps even by borrowing elements from architecture or design education that relatively more strongly encourages their students for seeking self-development, uniqueness and self-expression.*

Intrinsic motivation for creativity is very difficult to develop through educational interventions. Repeated engagements in self reflection, collaboration, and a creativity supporting educational environment, as discussed in Section 5.3 is likely to help. *Over-emphasis on external rewards like grades is detrimental to inculcating the intrinsic motivation.* We have not been able to suggest any concrete pedagogical models in this regard. *There is a need for more research to find suitable solutions for this goal.*

Section 6.5: Chapter Conclusion

In this chapter, we have discussed the rationale of three traits that are classified as the most critical attitudes, perceptions, and values for software developers: motivation to create/improve things, curiosity, and systems perspective. We have examined some philosophies, models, theories, suggested procedures, and empirical results from multiple disciplines to understand the deeper meaning of these traits. *In addition to the opening up the possibilities of upward movement along the professional development ladder proposed in Table 4.7, we also propose that development of the required attitudes, perceptions, and values for software developers must be kept on the top of the agenda of software development education programs.* In order to achieve this goal, software development education programs must first aim to facilitate students' movement to the higher levels of each of the following dimensions:

1. **Cognitive development: Perry's model**, and others, discussed in this chapter suggest the levels of development along this dimension (Table 6.1).
2. **Personal need perception development: Maslow's model**, and others, discussed in this chapter can be used as reference for understanding the levels of development along this dimension (Table 6.9).
3. **Levels of Systems Thinking:** Derived from **Boulding and Sanford (Table 6.4).**
4. **Moral development: Kohlberg's model** opens up the possibilities of understanding the levels of development along this dimension (Table 6.8).

Table 8.2 juxtaposes these models. Further, with reference to decision making in software development, we posit that the software developers need to integrate the four decision making styles identified by **Rowe and Boulgarides** (Ref: Table 6.2 and Table 8.5).

CHAPTER 7: THE PHENOMENON OF 'LEARNING'

In the traditional form of engineering education [16], based on teacher-centeric one-to-many learning, the teacher is seen as the source of information. Success in learning is often seen as the reproduction or direct application of what the teacher has taught. Such instruction, in which abstraction precedes the instantiation and concretization, helps students in *developing skills in deductive reasoning* and succeeds in *creating a knowledge-base as an inventory of concepts. It also trains students in linear thinking.* However, Projects like SUCCEED [304] have encouraged the participant campuses to move away from straight lecturing and individual homework, and to adopt more learner-centered instructional methods.

In second chapter, we discussed about a benchmark study focusing on the analysis of successful practices in engineering education in ten leading European and U.S. universities, 'Successful Practices in International Engineering Education' (SPINE) [78a] (Annexure AN11). The SPINE report indicates that *engineering graduates, even from these leading universities, have not rated the effectiveness of lectures and pedagogical and didactic skills of the teaching staff at a very high level*. There are significant differences in assessment of these parameters by faculty and engineers. *Faculty's assessment of these parameters is found to be inflated.*

Section 7.1: Empirical Investigations for Assessing Effectiveness of Educational Methods with Respect to the Requirements of Software Development

In this section, we discuss some quantitative and qualitative surveys conducted by us among computing students, software professionals, and engineering faculty.

Section 7.1.1: Empirical Studies on Effectiveness of Teaching Methods and Educational Experiences of Computing Students and Software Developers

Effective Teaching Methods: SPINE like Survey of Software Professionals (2004-05)

In 2004-05, we administered a SPINE (Annexure AN11) like survey (Appendix A1) among Indian engineers and managers working in Indian and multinational IT companies to obtain their

perceptions on the importance of forty-nine parameters of engineering education. Eight teaching methods (*group projects, homework/out-of-class assignment, industrial training/internship, lecture, projects, practical training, seminars,* and *written projects/studies)* assessed by the SPINE were used for evaluation by Indian respondents. As can be seen in Table 7.1, *group project, projects, and practical training* have been rated as *more effective teaching methods* than lectures.

Table 7.1: Importance of teaching methods as rated by Indian engineers and managers working in Indian and multi-national IT companies

No	Teaching Method	Category
1	Group Projects	Pivotal
2	Project	Pivotal
3	Practical Training	Pivotal
4	Industrial Training /Internship	Obligatory
5	Lecture	Obligatory
6	Seminars	Obligatory
7	Written projects/studies	Obligatory
8	Homework/Out-of-class assignment	Complementary

It is hypothesized that the low importance assigned to lecture as a teaching method can be attributed to perceived lack of contribution of conventional lectures in the development of most important engineering competencies and professional skills. Engineering faculty is strongly encouraged to use *pedagogies of engagement in their lectures* in order to lay the foundations of deep learning through their lectures.

The respondents from the Indian IT industry have rated the importance of group work at a much higher level than the respondents of SPINE. This relatively higher importance of '*group project'* as a teaching method for the IT industry as compared to the larger engineering industry is because of the *special nature of the software development activity*. The homework/out-of-class assignments have been rated at a much lower level by Indian respondents. Possibly *low quality routine home assignments* and unchecked plagiarism are responsible for this response.

Effectiveness of Teaching Methods: Survey of Software Developers (2009)

In 2009, this study was further extended and refined by refining and adding a few more teaching methods. Through the online global community LinkedIn.com, and online surveying tool surveymonkey.com, we conducted a survey, "Software developers - (How) Did your college

help you in your development?" among working software professionals. *We asked them to rate various educational experiences of college studies with respect to their direct/indirect contribution for respondent's later technical/professional/academic activities in terms of skill, knowledge, problem solving methodology, mindset, thinking, habits, values, etc*. The details of this survey are discussed in Appendix A10 (Table A10.1).

Usually, the traditional educational systems and approach over-emphasize three educational methods: *(i) knowledge transmission oriented lectures, (ii) homework and tutorials, and (iii) written examination and required preparation.* Very interestingly, as shown in Table A10.1, (Appendix A10), *these methods were found to be the least valuable* by our respondents for contributing to the development of their skill, knowledge, problem solving methodology, mindset, thinking, habits, values, etc., for their later technical/professional/academic activities. These were the only three methods that were found to have an average rating of less than 2 on a scale of 0 to 4. That means that *a good number of our respondents found only some or none of these methods to be helpful in their multidimensional development*.

Project work, laboratory work, discussions with other students, thinking and work oriented lectures, and teaching peers/juniors were rated as the most valuable educational experiences. All these experiences are learner-centric, whereas the least rated three experiences are essentially teacher-centric. These findings further validated our earlier SPINE-like study discussed above.

Effectiveness of Teaching Methods-II: Effect on Desired Competencies

In this survey, we had also asked them to rate the effectiveness of these pedagogical engagements for developing specific competencies, as discussed in Chapter 3. The details of this survey are discussed in Appendix A10. The results of this survey have also been discussed in twelve competency specific sections of Chapters 4 to 6. Table 7.2 provides the summary of the results of this part of this survey. The details are given Appendix A10, part A1.

Table 7.2: Perceived effectiveness of pedagogical engagements with respect to enhance of competencies: perceptions of software professionals

"Software developers - (How) Did your college help you in your development?" Summary of Table A10.2, (Appendix A10)

S.No	Competency (Table 8.1)	Ranked list of most effective pedagogical engagements (selected by half or more respondents)	Least effective pedagogical engagements (selected by less than 25% respondents)
1	**Technical competence** (including analytical, design, implementation, debugging)	**Projects (84%)** and Laboratory work (65%)	Discussion with others (9%), Written exams (16%), Knowledge transmission oriented lectures (18%), and Discussion with faculty (19%)
2	**Communication competence**	**Discussions with other students (84%), Mentoring juniors (71%),** Discussions with faculty (69%), and Discussion with others (51%)	Home work (8%), Research literature survey (8%), Laboratory work (8%), Written examinations (12%), Knowledge transmission oriented lectures (12%), Thinking oriented lectures (20%), and Project (22%)
3	**Domain competence**	Projects (61%), Research literature survey (51%), and Knowledge transmission oriented lectures (51%)	Discussions with others (18%)
4	**Complex problem solving competence**	**Projects (79%),** Laboratory work (59%), and Thinking oriented lectures (51%)	Discussions with others (6%) and Knowledge transmission oriented lectures (18%)
5	**Computational thinking competence** (including abstract and algorithmic thinking)	Projects (64%) and Thinking oriented lectures (49%)	Discussions with others (5%), Discussions with other students (20%), Written exam (20%), Discussions with faculty (21%), Mentoring juniors (22%), industrial training (22%), and Knowledge transmission oriented lectures (24%)
6	**Attention to details**	**Projects (71%)**	Discussions with others (6%), Knowledge transmission oriented lectures (18%), Discussions with faculty (22%), Thinking oriented lectures (22%), and Discussions with other students (24%)
7	**Critical and reflective thinking**	Projects (50%)	Homework (10%), Knowledge transmission oriented lectures (14%), Written examinations (14%), Industrial training (18%), Discussions with others (22%), and Laboratory work (24%)
8	**Creativity and innovation**	**Projects (82%)** and Thinking oriented lectures (53%)	Written examinations (4%), Knowledge transmission oriented lectures (8%), Homework (18%), and Discussion with other (22%)
9	**Intrinsic motivation to create/improve artifacts**	**Projects (74%),** Research literature survey (58%), Thinking oriented lectures (54%), Discussions with students (50%), and Discussions with faculty (50%).	Written examinations (6%), Knowledge transmission oriented lectures (14%), Homework (16%), and Discussion with other (24%)
10	**Curiosity**	Projects (66%) and Research literature survey (62%)	Written examinations (12%) and Discussion with other (14%)
11	**Decision making perspective** (including project planning, and management skills)	**Projects (90%), Industrial training (71%)**	Knowledge transmission oriented lectures (13%), Written examinations (13%), discussion with others (13%), and Home work (15%)
12	**Systems-level perspective** (including ability to see himself as bound to others, and also ability to accommodate himself to others)	Projects (58%) and Mentoring other students (51%)	Written examinations (10%, Knowledge transmission oriented lectures (10%), Homework (15%), Thinking oriented lectures (17%), Research literature survey (19%), and Discussion with others (24%)

Effectiveness of Teaching Methods: Survey of Students (2009)

The findings of Table 7.1 were also further validated through an almost similar survey among the final year (seventh semester) computing students at Jaypee Institute of Information Technology. Our earlier two surveys showed that *projects were the most valuable educational experience* with reference to later professional activities. Hence, we asked the students to rate the effectiveness of their earlier educational experiences with respect to its contribution on their final year project.

We asked them to *rate the following educational experiences of the last 3+ years with respect to their direct/indirect contribution for this project in terms of skill, knowledge, problem solving methodology, mindset, thinking, habits, etc.* There was a slight modification in the list of the educational experiences. Since, as a department, we have been using all the methods listed in Table 7.3, we dropped the last option of 'rarely/never experienced during college studies' in this survey. The respondents, who did not respond to some option, were treated as 'no comments' for that educational experience with a numeric value of zero. The first five options were used for this survey. We received a total of 210 responses. Table 7.3 shows the summary results of this survey. The details are discussed in Appendix A10, part B.

Table 7.3: Effectiveness of educational experiences for competency enhancement of computing students

Teaching Methods	Rating Average (0-4)
1. Minor project-I/Minor project-II of 3rd year	2.8
2. Mini projects as part of specific courses	2.8
3. Laboratory work (during laboratory classes)	2.7
4. Industrial Training	2.5
5. Developmental work (for laboratory classes)	2.5
6. Discussions with faculty	2.4
7. Literature survey oriented assignments	2.2
8. Discussions with peers/seniors	2.1
9. Lectures	1.9
10. Tutorial	1.8
11. Written examination and required preparation	1.6
12. Mentoring juniors	1.5

Broadly speaking, the result of this survey also reconfirms the supremacy of projects and laboratory work as the best educational experiences with reference to their contribution for final

year project in terms of skill, knowledge, problem solving methodology, mindset, thinking, habits, etc. In the same context, it also reconfirms the inadequacy of lecture, tutorial (homework), and written examination and required preparation. All these are teacher-centric activities. It is very interesting to note that the students find discussions with faculty as very useful for their project, where their response for lecture is very poor.

This result in Table 7.3 has one significant variation with respect to the result of Table A10.1, (Appendix A10). The lowest rating of mentoring juniors is attributed to the fact that a good number of the respondents gave no comments for this experience. Mentoring juniors is a student-centric activity for the senior students. The details of this scheme are discussed in Section 9.2.3.2. A large fraction of 58% of the students found that their experiences in mentoring of juniors were either extremely useful, mostly useful, or many were useful with reference to their project work. The effect of 'mentoring the juniors/peers' experiences on enhancement of specific competencies as perceived by working professionals has been discussed in the fourth, fifth, and sixth chapters. As per the report of the faculty, nearly 50% of the final year students very seriously participate in the mentoring program. We can interpret that *most of those who had enthusiastically participated in the mentoring program, found that experience useful even for their final year project.*

These three surveys show that teacher-centric educational activities like lecture, written exam, homework, tutorial, etc., do not significantly contribute to the development of students' skill, knowledge, problem solving methodology, mindset, thinking, habits, etc. The development of these virtues is attributed by our respondents to student-centric activities like projects, laboratory work, discussions, literature survey, teaching (mentoring), etc.

Section 7.1.2: Empirical Examination of Software Development Education Through Bloom's Taxonomy

Several authors have given a summary and commentary on Bloom's taxonomy [133] [305-306]. This taxonomy continues to be extensively used for course and assessment design by several computer sciences education researchers. For example, on December 20th, 2009, ACM Digital

library showed 407 papers referring to Bloom's taxonomy out of which 214 papers were published 2007 onwards.

The simplest level, '*Knowledge,*' exhibits previously learned material by recalling facts, terms, basic concepts, and answers. The '*Comprehension*' level demonstrates understanding of facts and ideas by organizing, comparing, translating, interpreting, giving descriptions, and stating the main ideas. '*Application*' is about solving problems by applying acquired knowledge, facts, techniques and rules in a different way. '*Analysis*' represents the act of examining and breaking information into parts by identifying motives or causes, making inferences and finding evidence to support generalizations. '*Synthesis*' aims at compiling information in different ways by combining elements in new patterns or proposing alternative solutions. '*Evaluation*' is about presenting and defending opinions by making judgments about information, validity of ideas, or quality of work based on a set of criteria. These upper three levels are considered to represent higher-level cognitive activities that require and develop mental faculties of problem solving.

Anderson and Krathwohl modified Bloom's taxonomy by adding another dimension of knowledge types: factual, conceptual, procedural, and metacognitive. They renamed the earlier hierarchy of levels from nouns to verbs [134]. They also swapped the position of the uppermost two levels. *However, the highest level of evaluation involves designing of criteria and also considerations of larger context, human values, and ethics. Hence, it is appropriate to keep it at the highest level. However, in the absence of either of these two aspects in evaluation, it reduces to a higher level of analysis only.* The summary of Bloom's revised taxonomy of cognitive processes for our purpose is as follows: (i) remember, (ii) understand, (iii) apply, (iv) analyze, (v) create, and (vi) evaluate.

The main aim of this study conducted by us in 2003 [11-12], was to empirically understand the degree to which the formal components of the traditional teaching-learning-evaluation process in engineering education succeed in creating opportunities for enhancing various competencies. Appendix A11 gives the further details of this study.

Table 7.4 (copy of Table A11.3, Appendix A11) tabulates the numeric ratings for six Bloom levels where large values indicate high ranks by most of the respondents.

Table 7.4: Comparison of Bloom level-specific normalized consolidated ratings

Bloom's Cognitive levels	What students think they get	What students get in examinations	What students think works well for them	What professional engineers recommend
Remember	0.24	0.36	0.04	0.09
Understand	0.24	0.16	0.11	0.10
Apply	0.22	0.40	0.13	0.10
Analyze	0.14	0.04	0.15	0.19
Create	0.14	0.05	0.46	0.38
Evaluate	0.02	0.00	0.11	0.15

Table 7.5 (copy of Table A11.4, Appendix A11) gives the correlation coefficients between these three ratings (each can be viewed as an arrays of 6 elements).

Table 7.5: Correlation between different consolidated ratings

	What students think they get	What students get in examinations	What students think works well for them	What professional engineers recommend
What students get in examinations $L'_{Ri\text{-}Exam}$	0.77		-0.25	-0.57
What students think works well for them $L'_{Ri\text{-}student\text{-}II}$	-0.22	-0.25		0.96
What professional engineers recommend $L'_{Ri\text{-}professional}$	-0.38	-0.57	0.96	

Correlation

In Table 7.5, a high correlation of 0.77 is observed between the perception of fifty 2nd year computing students, and the data collected from the fifteen question papers that were administered to around 1200 students of different seniority in electronics, computing, and biotech disciplines.

These results imply that in spite of the differences in disciplines, subjects, and seniority, there is not much difference in the cognitive level of activities that engineering students are engaged in. The professional engineers place a high emphasis (combined rating of 0.71) on engaging the students in activities to analyse, create, or evaluate that require higher-order cognition as compared to the emphasis (combined rating of 0.29) on simpler activities to remember, understand, and apply requiring lower level cognition. Interestingly, most of the engineering students experience more effective learning (combined rating of 0.72) when they are engaged in activities that require higher-order cognition as compared to the much lower perceived effectiveness (combined rating of 0.28) of the learning that occurs as a result of their engagement in activities requiring lower order cognition.

This demonstrates that most of the engineering students' preferred learning style is in alignment, and having a very high correlation of 0.96, with the recommendations of the professional engineers. However, the prevailing practice among the majority of engineering educators demonstrates an opposite preference leading to negative correlation of -0.22 and -0.25 with the preferred learning style of their students, and -0.38 and -0.57 with professional engineers' recommendations respectively.

Can Language Change Thinking?

Obviously, the higher education system needs to make a deliberate and committed effort to put students into activities that will encourage, motivate, and force them to apply genuinely higher level cognitive skills. The kind of activities that a typical engineering student is generally engaged in does not help in enhancing ill-defined problem solving. It is clear that most of the activities that students get formally engaged in as part of the teaching learning-evaluation process promote rote-learning and conformity. If the structure of language offers a key to the structure of thought, we will know the educational ethos has changed when we see a predominant use of different verbs by faculty. Further, it may well be that a deliberate shift in the verbs used in teaching (and consequently a rethinking of the assignments being given) could be an important step in fostering ill-defined problem-solving among engineering students.

This study also partially explains the findings of our first three study in this section about effective teaching methods and educational experiences, where the software professionals as well as computing student respondents, rated teacher-centric activities like lecture, written examinations, and tutorial etc., as the least effective teaching methods.

Section 7.1.3: Qualitative Study of Effective Lectures

The following qualitative study examines the issue of effective teaching methods in the limited context of lecture classrooms by collecting, documenting, and analysing the anecdotes about the most effective lectures as recalled by engineering students at different levels, and also by faculty members.

Section 7.1.3.1: Perceptions of Computing Students at Senior and Junior Levels

Some anecdotes about the most effective lectures as recalled by undergraduate engineering students specializing in computing disciplines students, at two different levels of seniority, were collected and documented. In all, 110 anecdotes about the most effective lectures have been collected. The first collection of 28 anecdotes was elicited from students who had completed four to six semesters of study work, and also a month long industrial training. Table A12.1 (Appendix A12) gives excerpts of some selected samples from this collection. The second set of 82 anecdotes was collected from students at the beginning of their 3rd semester. Table A12.2 (Appendix A12) lists excerpts of some of these anecdotes.

A closer look at these two tables reveals interesting patterns in the learning preferences of these two categories of respondents. Out of twenty-eight anecdotes from senior students, an overwhelming majority of twenty-five anecdotes associate most effective lectures with at least one form of active or collaborative activity like problem solving, group work, discussions, critique, and so on. This preference also confirms the earlier finding [11-12] even in the limited context of a lecture classroom. On the other hand, out of eighty-two anecdotes from junior students, nearly half, i.e., thirty-eight did not make a special mention to any form of the active or collaborative activity in the most effective lecture as recalled by them.

This difference can be explained in the light of difference in the degree of their exposure to different lecturing techniques. The senior group of students were exposed to more forms of lecture formats as they did courses with a larger number of faculty members. Through some courses, senior students were already exposed to classroom engagement in problem solving, group work, and other forms of active learning. On the other hand, the junior students were not so much exposed to such lecture classes. They were possibly mainly trained through the conventional method of teacher-centric knowledge transmission oriented lectures. This difference in the responses of two groups suggests that unless students get exposed to intense active and collaborative learning, they remain satisfied with teacher-centric conventional lectures with occasional interaction in the form of seeking clarification. Their learning preferences undergo a major change after they get exposed to some techniques of active and collaborative learning during lectures.

Section 7.1.3.2: Perceptions of Faculty Members in Engineering Institutes

Anecdotes about the most effective lectures as recalled by the faculty members were collected, documented, and analysed. The faculty members recalled and contributed anecdotes from their experience both as student and also as teachers. In all, 142 anecdotes of such best lectures have been collected. The first set of 99 anecdotes was collected from engineering faculty's recalled experiences from their student life. Table A12.3 (Appendix A12) shows some of these anecdotes. Many of these faculty members also contributed the fourth group of 43 anecdotes about their most effective lectures as teachers. Some of these anecdotes are listed in Table A12.4 (Appendix A12).

The anecdotes of teachers' most favourite lectures from their student life, tabulated in Table A12.3, shows a pattern which is skewed in favour of active and collaborative classrooms. Approximately 80% of the anecdotes have a specific mention of some such classroom engagements. Further, an overwhelming majoring of the faculty members (94%) considered those lectures as their best in which they are able to put the students into one or the other kind of activities like problem solving, seeking clarifications, design, group work, and so on.

There is a great structural similarity between the favourite lectures mentioned in the anecdotes offering most effective learning in all four tables of Appendix A12. The best practices as

narrated by different groups have a common pattern which is skewed towards usage of 'action and interaction' in lecture classrooms. However, in spite of similar perceptions about the structure of most effective lectures, use of such effective practices is far from common. This gap can be attributed to one or more of following factors:

1. Lack of sufficient exposure to best practices during the faculty's student life.
2. Lack of training in using best practices after joining the profession.
3. Lack of belief in the necessity of best practices.
4. Lack of confidence in the sustainability and success of best practices.

It hypothesized that lack of sufficient exposure during their student days and subsequent lack of training after joining the teaching profession are the basic reasons for this gap. Most of the lectures attended by faculty members during their student life were possibly devoid of practices of action and interaction. Occasional experiences with some of these practices, though result in positive memory of such lectures, such experiences are possibly not frequent or strong enough to make the necessity of such practices as part of their belief system about the teaching-learning system. On joining the profession, no training is provided to them to help them to fill up this gap. They never formally learn about teaching-learning principles, models, or practices and they don't get the opportunity to strengthen their confidence in the sustainability and success of such practices.

Section 7.1.4: Quantitative Study of Effective Lectures

The following quantitative analysis among senior undergraduate and postgraduate engineering students, attempts to more closely understand and identify the attributes of this gap among common and effective practices from their perspective. In this process it also attempts to validate the first part of the proposed hypothesis about the lack of faculty's exposure to effective practices of action and interaction in the lecture classes during their student life through an understanding of experiences of current students, some of whom will grow as faculty in future. The second part of the hypothesis about lack of training for faculty is examined and discussed in the next section.

Section 7.1.4.1: Perspective of Computing Students

Based 252 anecdotes from students and faculty (Appendix A12), a list of fourteen non-exclusive lecture properties was prepared as possible attributes of different lecture formats. These were used for a quantitative survey among students. This survey is discussed in Appendix A13. These attributes have been used to propose typology of learning environments as follows:

1. Passively engaged student: The student only listens and does not add any content to the discourse.
2. Reactively engaged student: The student reacts and asks for some clarifications without adding any other type of content to the discourse.
3. Actively engaged student: The student gets individually engaged in some kind of problem solving activity, and adds some content to the discourse.
4. Collaboratively engaged student: The student proactively collaborates with others to solve problems and adds content to the discourse in the lecture classroom.

Table 7.6 (copied from Table A13.3, Appendix A13) shows the summary of results of this survey.

Table 7.6: Attribute category-wise consolidated ratings by computing students

Lecture format attribute category	Most effective for learning (A)	Least effective for learning (B)	Most often used (C)	Least often used (D)
1 Passively engaged student	0.48	**1.61**	**1.68**	0.43
2 Reactively engaged student	0.48	0.39	0.48	0.30
3 Actively engaged student	4.39	0.77	0.66	3.57
4 Collaboratively engaged student	**5.00**	0.57	0.27	**4.82**

The last two columns of Table 7.6 confirm the first part of hypothesis made in the last section that during student life, *there is lack of exposure to usage of action and interaction in the lecture classrooms*. The mismatches between corresponding values in columns A and B on one hand, and columns C and D on the other hand, are very significant. This is further consolidated in Table 7.7 that shows the correlation coefficients between A, B, C, and D columns of Table 7.6.

Table 7.7: Correlation matrix between attributes of different lecture formats based on computing students responses

	Most effective for learning (A)	Least effective for learning (B)	Most often used lecture format (C)
Least effective for learning (B)	-0.79		
Most often used lecture format (C)	-0.69	**0.99**	
Least often used lecture format (D)	0.50	-0.75	-0.78

Table 7.7 shows that there is a very high negative correlation of -0.79 between the attributes of the most effective and the least effective lecture formats. This implies that attributes of the most effective and the least effective lectures have great dissimilarities. Also, there is a very high negative correlation of -0.78 between the attributes of the most often used and the least often used lecture formats. This signifies that there is a very significant contrast in the attributes of the least often used and the most often used lecture formats. This suggests that there are great dissimilarities in the lecture formats, and also the attributes of the lecture format have an impact on learning. There is a positive correlation of 0.50 between attributes of the most effective and the least often used lecture formats. This implies that *attributes that make a lecture format most effective are used least often.*

Most disturbing is the very high positive correlation of 0.99 between the attributes of least effective and most often used lecture formats signifying that the *attributes that make lectures least effective are most often used by faculty members*. This observation is further strengthened by the very high negative correlation of -0.75 between the attributes of the least effective attributes and the least often used lecture formats indicating that the *attributes that make lectures least effective are most used by teachers*. Similarly, there is a negative correlation of -0.69 between the most effective attributes and the most often used lecture formats. This means that the *attributes that make lectures most effective are not most often used by teachers.* This suggests that senior undergraduate students report better learning in active and collaborative lectures that offer opportunities for creative thinking, problem solving, group work, analysis, design, and so on.

Hence, *in order to increase the effectiveness of lectures in courses for computing students, there is a need to increase a teacher's awareness about the students' preferred learning styles.*

Further, the role of the lecture needs to expand from the traditional teacher-centric content delivery to a student-centric content exploration, discovery, and creation.

A detailed analysis of Table A13.2 (Appendix A13) shows the following:

1. A very high majority (80% to 90%) of respondents feel that the most often used lecture format has the following attributes:
   i. During the lecture, the main purpose of a teacher's presentation is to explain the textbook.
   ii. Lecture classroom is primarily a place for careful listening to a teacher's presentation and prepare class notes.

   It can be noted that both of these attribute belong to the first category of attributes, and promote passivity in students' in-class behavior.
2. Only a very small minority of 11% of the respondents identified 'explaining and interpreting textbook' as a attribute of the most effective lecture formats. On the other hand, a large majority of 91% respondents felt that the lectures that focus on 'explaining and interpreting textbook' are least effective.
3. Only a very small minority of 36% of the respondents identified 'careful listening and preparing notes' as a attribute of the most effective lecture formats. On the other hand, a large majority of 70% respondents felt that the lectures that focus on this attribute are least effective.
4. A majority (approximately 50% to 77%) of the respondents felt that the lecture format that they found to be the most effective in terms of its impact on learning outcomes has the following distinguishing attributes:
   i. It encourages and demands students to do on-the-spot creative thinking (75%).
   ii. It encourages and demands students to do in-class-group-work (64%).
   iii. It encourages and demands students to engage in on-the-spot discovery (63%).
   iv. It encourages and demands you to in-class create conceptual designs (59%).
   v. It encourages and demands students to in-class analyze presented information (59%).
   vi. It encourages and demands students to get on-the-spot practice of problem solving as an individual (57%).

vii. It encourages and demands students to on-the-spot communicate their creations to the entire class (50%).

Interestingly, all these lecture attributes belong to the category of active and collaborative engagements.

From this study, we can deduce that much to the dislike the students, they are a passive (could be very careful) recipient of information in the engineering lecture classroom. Interaction, if any, in the lecture classrooms is also passive in nature. It remains limited to seeking clarifications in a teacher's presentation. Students are rarely engaged in activities that give them the opportunity of actively and collaboratively expressing their ideas. These lectures do not contribute to effective learning. *Respondents report most effective learning when they are engaged in activities that give them the opportunity of 'individual action and collaboration.'*

In the light of this study, we can also provide an explanation of *working engineers' perception* of low importance of lecture as a teaching method. In their experience also, the most often used lecture format may have been least effective for their learning, as is the case with current students. Hence, *low importance attached to lecture by them* can be attributed to lack of action and active interaction in lecture classrooms.

Section 7.2: Reflections About the Phenomenon of 'Learning'

The mechanism of learning has been attracting the attention of thinkers in philosophy, psychology, education, and also computer science. Annexure AN12 briefly summarizes some important theoretical perspectives [206] [307-332] about the phenomenon of learning.

Behaviorists see learning as a relatively permanent change in behavior due to experience, and concentrate on control of the external environment.

Cognitive psychologists perceive it as a relatively permanent change in mental associations due to experience, and believe that humans are capable of insight, perception, and attributing meaning.

Social psychologists view it as a social enterprise, depending upon interaction between learner and his/her socio-cultural environment.

Humanists emphasize the development of the whole person, and place emphasis on the affective domain.
Constructivism stresses that all knowledge is context bound, and that individuals make personal meaning of their learning experiences through internal construction of reality.

Neuroscience ascribes 'learning' to the brain's ability to change its structure. Though learning is natural, it is not automatic. It is driven by voluntary and/or involuntary efforts made in response to stimulating experiences. *Such stimulating experiences create 'cognitive dissonance' [327] and 'learning contexts' by inducing recognition of inadequacy of existing meanings.* These contexts catalyze the activation of operating learning processes. ***Learning is a natural multi-faceted process that helically progresses through making and rendition of meaning at progressively deepening levels.*** Meaning making and rendition processes *unfold in a multi-dimensional space* of physical world, community, culture, psycho-motor, cognition, emotion, attitude, and values.

Humans continuously make meanings about the external world, inner self, and the relationship of the two. Experiences are interpreted as mental objects by the human mind to create an individual's meanings. *Mental objects* include thoughts, ideas, concepts, impressions, percepts, rules, images, notions, scripts, schemas, and so on. The combined strength of deductive, inductive, convergent, divergent, linear, nonlinear, critical, and creative thinking processes, as well as intuition, drive this interpretation. Symbols, notations, language, diagrams, and concept-maps are used to represent and create these objects.

We create meaning at different levels in different contexts. These levels range from *superficial symbolic levels to deeper conceptual and revelational levels*. A disjoint ensemble of inflexible and incoherent superficial meanings results in surface learning. Deep learning requires the learners to create integrated, coherent, and trans-contextually transferable meaning at deeper conceptual and revelational levels. Ability to apply, blend, and regulate thinking processes governs coherence, accuracy, richness, interconnectedness, and representations of mental objects, and hence, the level of meanings. Deeper meanings are characterized by richer representations. *At the deepest levels of learning, meanings related to self, get well integrated*

with the meanings related to the external world. Prior meanings may expedite, impede, or even block the progress of an individual's meaning making processes.

We render our meanings in abstract forms like models and theories, and concrete forms like artifacts, e.g., software and processes at varied levels of sophistication. Meaningful and creative renderings manifest learners' deeper integrated meanings and refined rendering skills. Meaning making continues during rendition, and rendering skills themselves are refined through practice and newer meanings. *The level of meaning, and also the form and sophistication of rendering, depend upon the richness of context and strength of operating processes of learning as well as learners' nature, nurturing, and intrinsic motivation.*

An individual's value orientation and interests shape his need perception. Many of our efforts made for fulfilling our needs and other experiences create 'learning contexts' [333] by inducing recognition of inadequacy of existing meanings. An individual's value orientation, perceived needs, intrinsic motivation, and flow of emotions trigger, drive, and direct their meaning making process and efforts. Community and culture significantly influence value orientation, perceived needs, intrinsic motivation, and flow of emotions. Further, community and culture also provide the ground for creating shared meaning.

Repeatedly reinforced meanings, cultural norms, and social expectations affect the meanings about the inner self. Meanings related to inner self have strong influence on personal values, interest, attitude, intrinsic motivation, goals, and even perspective. Changes of self-related meanings affect individual's efforts, and also their meanings about external world. Consequently, a *practice of critical self-reflection on self-related meanings strengthens self-regulation of meaning making, and increases the efficacy of learning processes*.

Wisdom is an outcome of trans-contextual meaning integration, self-awareness, openness based on awareness of competency limitations, and a concern for collective and sustainable well-being.

Section 7.3: Implications for Software Development Education

Formal education and training programs are man-made interventions to foster human development through *synthetic learning contexts*. Consequently the efficacy of education and

training programs primarily depends upon the richness of the synthetic learning context and efficiency of the activated learning processes. Learners' nature, nurturing, and intrinsic motivation also affect the efficiency and efficacy of their learning processes.

Training is usually concerned with skill development. Education on the other hand has wider goals of also nurturing the mind in higher levels of cognitive as well as affective domains. Training programs mainly aim to channelize students' efforts and 'ability to learn' to develop their 'competencies.' Educational programs on the other hand aim to enhance self-awareness, expand the perspective, intuition, and intellect. They are also expected to contribute in making students 'learn to make efforts' and 'learn to learn.' Educational programs have a wider goal of cultivating 'valuable competencies' to develop wise and competent citizens.

Though education and training have a seemingly different focus, they share a symbiotic relation. Training programs can offer a fertile ground for creating educational contexts, that in turn contribute back to enhance the efficiency and efficacy of training programs. Both, high competence and wisdom require deep learning.

As learning facilitators, teachers are essentially students' experience and engagement designers. Societal and institutional environment and expectations, as well as teachers' skill, knowledge, experience, personal perspective, and even value system guides them in this design process. All institutes and teachers have their explicit or implicit educational philosophy. This philosophy plays no lesser important role than the subject expertise in student engagement.

In Chickering and Gamson's classic paper [334], *seven principles of good practice in undergraduate education* were elaborated. These are: (i) encourage contact between students and faculty, (ii) develop reciprocity and cooperation among students, (iii) encourages active learning, (iv) gives prompt feedback, (v) emphasize time on task, (vi) communicate high expectations, and (vii) respect diverse talents and ways of learning.

During the process of undergraduate education, the students grow from late adolescents to early adults. This growth is characterized by considerable transformation and evolution of self-awareness and value orientation. These transformations automatically influence perceived

interests, needs, attitude, and intrinsic motivation. Sometimes these transformations can be very swift, and cause the students to experience short-term loss of emotional well-being and self-regulation. All these changes affect their meanings and also meaning making process. Training and educational programs ought to offer a spectrum and stream of stimulating learning contexts.

Researchers used Perry's nine stage model to measure the intellectual development of engineering students. *They found that there was not any significant change in Perry position for them in their first three years; however, a growth of approximately one* Perry position was observed between the 3rd and 4th years [255]. The authors attributed this change to increased opportunity of real-life industrial exposure, group activity, and project work in the last year.

In order to stimulate deep learning, education programs need to create and offer such learning contexts that *induce forwarding levels of meaning-deficits, enabling flow of emotions, rich set of mental objects and representations, enhanced self-awareness, multifarious perspectives, and persistent practice of meaning integration.* Disciplines of educational psychology, adult development, curriculum design, and instructional design offer a very rich set of theories and models on these aspects.

Section 7.4: Student Engagements for Facilitating Deep Learning through Higher Education

The National Survey of Students Engagement (NSSE) is a very large scale study encompassing hundreds of thousands of students, and thousands of faculty members, of hundreds of institutes in various disciplines conducted over several years in USA. Table 7.8 catalogues the NSSE recommended *pedagogic engagements for deep learning under the categories of higher-order, integrative, and reflective learning*. The NSSE study [335] showed that, as compared to most other disciplines, engineering students experience relatively higher involvement in higher-order learning. It also showed that engineering, physical sciences, as well as business students report much lesser engagement in integrative and reflective learning. Integrative learning requires students to relate the content of one subject with another. As per the findings of the NSSE survey, social sciences, arts and humanities, and some other professional disciplines like medicine and architecture, create a much higher level of students' engagement in integrative as well as reflective learning.

Table 7.8: Selected catalogue of learning engagements for deep learning from the NSSE study

Higher-order learning engagements:
1. Applying theories or concepts to practical problems, or in new situations.
2. Analyzing the basic elements of an idea, experience, or theory, such as examining a particular case or situation in depth, and considering its components.
3. Synthesizing and organizing ideas, information, or experiences into new, more complex interpretations and relationships.
4. Making judgments about the value of information, arguments, or methods, such as examining how others gathered and interpreted data, and assessing the soundness of their conclusions.
Integrative learning engagements:
5. Working on a paper or project that required integrating ideas or information from various sources.
6. Including diverse perspectives in class discussions or writing assignments.
7. Putting together ideas or concepts from different courses when completing assignments, or during class discussions.
8. Discussing ideas from your readings or classes with faculty members outside of class.
9. Discussing ideas from your readings or classes with others outside of class (students, family members, co-workers, etc.).
Reflective learning engagements:
10. Learning something from discussing questions that have no clear answers.
11. Examining the strengths and weaknesses of your own views on a topic or issue.
12. Trying to better understand someone else's views by imagining how an issue looks from his or her perspective.
13. Learning something that changed the way you understand an issue or concept.
14. Applying what you learned in a course to your personal life or work.
15. Enjoying completing a task that required a lot of thinking and mental effort.

Section 7.4.1: Curriculum Integration

According to Ausubel's 'assimilation learning theory,' meaningful learning occurs through the process of linking or integrating new ideas or concepts with previous knowledge [340-341]. Curriculum needs to be viewed like a living organism, with most of the important subsystems functioning even at a very early stage of its development. Most of the subsystems get developed very rapidly very early in living organisms. For example, in case of a human embryo, the heart start beating in the fifth week of pregnancy, and primordial forms of liver, pancreas, lungs, and stomach are evident in the sixth week. By the eight week the hind brain, elbows, and testes/ovaries are visible. After very rapid development of all necessary systems, living organisms grow by acquiring sophistication to existing systems, rather than acquiring new systems during further developmental stage of its life. The living organisms simultaneously grow in multiple dimensions in a continuously integrated manner.

Computing is a continuously evolving field, *there is always a pressure to introduce courses to address contemporary trends*. However, the traditional *computing curricula take computing*

more like a vertical discipline, rather than like an integrated network of ideas, concepts, technologies, processes, and methods. The foundation courses do not make an attempt to include the basic exposure to contemporary aspects in an integrated manner. In traditional computing curricula, students are usually exposed to these contemporary aspects only in advanced level courses, sometimes only through electives. This creates the possibility of a large number of computing students to complete their undergraduate program without learning anything about one or more of contemporary aspects. Such an approach also fails to correlate the real experiences of students with the curriculum. For example, the current generation of students grows using software that is rich in graphics and multimedia and also web and mobile-enabled. They are also aware of the security risks of software systems.

Drake [345] cites many studies on integrated curriculum with various reported benefits: *increased learning, greater personal growth, increased ability to apply concepts, better understanding, increased student motivation, increased student cooperation, enhanced confidence, enhanced sense of responsibility, increased use of higher thinking skills, and improved quality of work*. The computing curricula, however, take an unnatural route for educating the students by ignoring this aspect of continuous integration. *Repeated exposures to integration are necessary for developing an integrated perspective of the curriculum*. Hence, most students do not get sufficient opportunity to view the curriculum as a closely integrated system. Even the curriculum recommendations of several professional and academic bodies like ACM and IEEE treat computing curricula as fragmented or loosely coupled systems and elaborate a list and sequence of courses, topics and also minimum hours for each topic. *These recommendations do not elaborate any specific micro-level integration goals or strategies.*

Brown and Nolan [342] developed a *continuum for curriculum integration*: integration through correlation between subjects, integration through common themes and ideas, integration through the practical resolution of issues and problems, and integration through student-centered inquiry. Beane [343] identified *three dimensions of curriculum integration*: integration of experience, social integration, and integration of knowledge (and skills). Fogarty [344] identified ten models of curriculum integration that fall into *three general categories*: *integration within single disciplines, integration across several disciplines, and integration within and across learners*.

Building upon various earlier works on curriculum integration with more specific focus on school education, Harden [346] proposed a taxonomy of curriculum integration with reference to the specific context of medical education. We find it very suitable for designing integrated computing curriculum as well. Harden has structured this taxonomy as an eleven stage ladder given in Table 7.9. *We use selected elements of Harden's taxonomy in our proposed framework of pedagogic engagements (Table 8.8).*

Table 7.9: Harden's taxonomy of curriculum integration

Subject based teaching
1. Isolation: Integration is not explicitly facilitated and is left to students themselves.
2. Awareness: Teachers avoid duplication across subjects. Integration is left to students themselves.
3. Harmonization: Teacher may make some explicit connections within the subject area to other subject areas.
4. Nesting: Content from different subjects may be infused to enrich the teaching of one subject.
Higher levels of integration
5. Temporal co-ordination: Related topics in different subjects are taught *concurrently but separately*.
6. Sharing: Overlapping concepts of different subjects are used as organizing elements *for joint teaching of shared concepts* in complementary subjects.
7. Correlation: An integrated teaching session, course, project, assignment is introduced in addition to the subject-based teaching to bring together related topics.
8. Complementary program: The integrated sessions now represent a major feature of the curriculum. Running alongside the integrated teaching are scheduled opportunities for subject-based teaching.
9. Multi-disciplinary: New courses are developed around integrating themes, problems, or issues. The courses may include a structured body of knowledge but which transcends subject boundaries. The theme or problem is the focus for the learning, and the subjects demonstrate how they contribute to the students' understanding of the theme or problem. The subjects *give up a large measure of their own autonomy*.
10. Inter-disciplinary: Content of many subjects, is combined into a new course. There may be *no reference to individual disciplines or subjects*, and hence a loss of the subject or discipline specific perspectives.
11. Trans-disciplinary: The curriculum transcends the individual disciplines. The focus with trans-disciplinary integration for learning, however, is not a theme or topic selected for this purpose, but the field of knowledge as exemplified in the real world.

Contemporary technologies like *Web, Mobile, Multimedia,* and *Security* and professional practices like *Estimation, Systems Design, Open Source Usage,* and *Debugging* are highly pervasive in software development. In Section 9.2.1, we propose a model for enriching the existing courses through *multi-level infusion* of these selected elements of *contemporary technologies* and *professional practices*. This infusion can be incrementally carried out across most of the existing core courses. Consequently, even before studying the dedicated courses on these topics, the students are fairly well exposed to all these areas in their junior level courses, either as an extension, development tool, process, or application domain. In principle, this is similar to the fourth stage of 'nesting' in Harden's eleven stage curriculum integration ladder. In

addition to increasing *cognitive flexibility* [206], multi-level infusion of selected topics enhances *open-mindedness and integrative thinking*. It also nurtures the habit of *reflection.*

While multi-level infusion helps to integrate the computing curriculum, some integrative capstone courses can further strengthen the unification of some important computing concepts, and also integrate the computing concepts with other disciplines. Integrative capstone courses help in strengthening nonlinear, integrative and systems thinking, and flexible learning. They also have the potential to integrate, and hence, consolidate the already learnt material, and also provide a stronger foundation for further studies. They also have the potential to integrate, and hence, consolidate the already learnt material, and also provide a stronger foundation for further studies. In Section 9.2.2, we discuss some integrative capstone courses, visualized and administered by us.

Section 7.4.2: SOLO Taxonomy

Biggs and Collis [329] proposed a five-level Structure of the Observed Learning Outcome (SOLO) Taxonomy in terms of increasing complexity. As per this taxonomy, the lower three levels: *'pre-structural,' 'uni-structural,' and 'multi-structural'* are about quantitative increase in details of the response. The upper two levels: *'relational' and 'extended abstraction'* are about its qualitative transformation through integration, extension, and abstraction. The first level indicates complete lack of comprehension and understanding.

Brabrand and Dahl [347] examined intended learning objectives of more than six hundred science courses (including computer science) at two Danish Universities, and found that the average SOLO level of intended learning objectives varied from 2.8 to 3.4 for undergraduate students, and between 2.9 to 3.8 for postgraduate students. Aggregating all the disciplines, nearly 70% of courses' intended learning objectives aimed to achieve only third SOLO level. For some disciplines, this was as high as 80%. Overall, only a little more than 10% intended learning objectives targeted for fifth SOLO level, and for some disciplines this was even lesser than 5%. The realized objectives would perhaps be even lesser than the intended ones.

Hence, we can conclude that students' most common engagement in higher education is not only at the lower levels of Bloom's taxonomy [9][10], it is also at the lower levels of the SOLO taxonomies. In the last four years, few papers in the ACM SIGCSE, and very few in IEEE conferences have made some reference to the SOLO taxonomy. Further, very few papers at the ACM SIGCSE and IEEE educational conference and transactions refer to any theoretical framework on curriculum integration. We include SOLO taxonomy in our proposed framework of pedagogic engagements (Table 8.8)

Section 7.4.3: Collaborative Learning

A serious shortcoming of the typical undergraduate engineering education process is neglecting to train students to work with other programmers. Usually in a four-year engineering course, collaborative effort and teamwork is done only in the later years, i.e., in the third and fourth year. Even then, the collaboration is driven more by division of labor due to size rather than complexity. Its core purpose is not about engaging them in collaboration.

In 2009, the first semester class of the M.Tech. (CSE) students at Jaypee Institute of Information Technology, coming from various colleges of various Indian universities, indicated that they had not experienced much of group work during their undergraduate computing studies, except for their final year projects. In our empirical study of anecdotes about the best lectures as recollected by senior students and faculty members, both groups recollected that their best lectures involved activities like group work, discussions, critique, and so on [348]. A majority of senior undergraduate computing students felt that the classes that engaged them in some form of group work were most effective for their learning, whereas nearly half of them also felt that group work is the least often used pedagogical approach in classes. Many a times, the assignment or project is designed after forming the groups in the classroom, when the teacher knows the strength of a particular group. Such approaches do not necessarily develop teamwork skills. Researchers have felt the need for a way to facilitate students to work together with clearly defined boundaries [349].

Group work has a very wide range of possible forms, ranging from simple task division between two members, to intense community collaborations. At the simplest level, it may be in the form

of coordination between members for handling simple situations with clear task boundaries requiring minimum intra-group communication. Alternatively, it may take the form of cooperation for handling complicated situations with well-defined, but marginally overlapping subtask boundaries, and mild intra-group communication. Group work in its most sophisticated form requires collaboration for handling complex situations with evolving subtask boundaries and intense intra-group communication. The group size, distance, and professional as well as cultural diversity are other important aspects that bring further variations of these three forms of group work. Interestingly, software development involves all these forms of group work. However, sometimes the word collaboration is used to mean all these forms of group activities. 'Interpersonal intelligence,' one of the eight forms of multiple forms of intelligences identified by Gardner [155], is essential for developing the ability to work effectively with others. Discussions, and collaborative activities like projects, have been found to be effective for developing this form of intelligence.

Interestingly, social psychologists view the act of learning as a social enterprise, depending upon interaction between learner and his/her socio-cultural environment. Collaborative learning focuses on the role of peer work for educational success. Vygotsky in his seminal social development theory proposed that social interaction plays a fundamental role in the process of cognitive development [350]. One of key assumptions of Bruner's model of constructivist learning is a social enterprise [351-351a]. Pask's Conversation Theory [320] is founded on the idea that learning occurs through conversations with instructors or peers [352]. As per the Social Learning Theory of Bandura, gaining insights into others' practices can be a valuable experience [320].

Bextor Magolda's 'epistemological reflection model' [353] elaborates upon the evolutionary stages of learners' perceptions of peer's role in learning. At the first stage of 'absolute knowing,' it is limited to sharing material, and explaining what they have learnt to each other. The next stage of 'transition knowing' is characterized by periodic active exchanges among peers. At the third stage, 'independent knowing,' peers share views, and serve as source of knowledge. The learners at the final stage of 'contextual knowing' enhance learning via quality contributions.

Other models of intellectual development like Perry's nine stage model [250] also highlight a similar evolution in learners' perception of peer's role in learning.

Lave and Wenger proposed *Situated Learning Theory*, with the central idea that learning involves a process of engagement in a '*community of practice*.' From the perspective of this theory, learning is not seen as the acquisition of knowledge by individuals, but as a process of ***social*** participation [354]. *Collaborative Problem Solving* [156] has been found effective for developing content knowledge in complex domains, problem-solving and critical thinking skills, and collaboration skills.

Learning in a collaborative environment is a process that could be subject of two different perspectives: individual effort and social sharing of knowledge [355-356]. However, this sharing of knowledge is useful only if students are engaged in collaborative activities**.** Engagement in collaborative activities causes individuals to master something that they could not do before the collaboration [356]. Salmons [357] proposed six levels of collaborative e-learning, as given in Table 7.10.

Table 7.10: Salmon's levels of collaborative e-learning

1. Solo: no collaboration 2. Dialogue: simple exchange of information 3. Peer review: reviewing others' work 4. Parallel Collaboration: dividing the task in the beginning, and finally integrating individuals' work 5. Sequential Collaboration: building upon each other work 6. Synergistic Collaboration: doing it together in a synergistic manner

We include this model in our proposed framework of pedagogic engagements (Table 8.10).
Preston [381] summarizes that collaborative learning research has already established two things: (1) the effectiveness of having students work together, and (2) the critical attributes common to successful collaborative learning approaches. It is important to design exercises in such a way that the solution requires co-authoring.

Dillenbourg [355] elaborated on collaborative learning as follows: "*… a situation in which particular forms of interaction among people are expected to occur, which would trigger learning mechanisms, but there is no guarantee that the expected interactions will actually*

occur. Hence, a general concern is to develop ways to increase the probability that some types of interaction occur." According to Dillenbourg [355], in order to maximize the likelihood of the specific forms of interaction as he had mentioned in his definition of collaborative learning, there are four conditions to accurately set a collaborative context. These are given in Table 7.11. These conditions can be very helpful in designing effective instructional interventions for pedagogical engagement in computing courses.

Table 7.11: Dillenbourg's four conditions for collaborative learning

1. Set up the initial conditions: This involves taking decisions about group formation. It is difficult to set up initial conditions that guarantee effective collaborative work. At this stage the faculty is required to take decision about the size, heterogeneity, geographical, and temporal placement of peers, i.e., face-to-face co-location, side-by-side co-location, geographically dispersed locations, collaboration technology (if any), selecting peers, etc. 2. Over-specify the collaboration contract with a scenario based on roles: The collaboration contract can be specified by setting up differences among learners either by triggering conflictual interactions, or assigning complementary roles, or limiting their access to different parts of information. 3. Scaffold productive interactions by encompassing interaction in the medium: This encompasses specifying interaction rules in face-to-face or technology enabled collaboration. 4. Monitor and regulate the interactions: The teacher may decide to directly facilitate, monitor, and regulate the face-to-face or technology supported collaboration among learners. Alternatively, a mechanism or a tool may be developed for self-regulation by peer learners.

We include Dillenbourg's four conditions in our proposed framework of pedagogic engagements (Section 8.3.4).

Section 7.4.3.1: Pair Programming

The most common and widely used form of collaborative programming is pair programming. Pair programming is a situation in which two programmers work side-by-side, designing and coding, while working on the same algorithm. As Chaparro [372] paraphrases Cockburn & William [373] and Williams & Kessler [374] a relevant aspect of pair programming is that it transforms what used to be an individual activity into a cooperative effort. Typically there are two roles in pair programming: the driver who controls both the computer keyboard and the mouse, and the navigator who examines the driver's work, offering advice, suggestions and corrections to both design and code.

It was originally designed for production rather than the educational environment. Research shows that that a pair or a group working together in solving a programming exercise minimizes

the cognitive load [378]. Agile methods like eXtreme Programming include it as a common practice. It has also been used in educational settings, with the reported benefits [379-380] of program quality, programming speed, learners' enjoyment, etc. Pair programming exposes code to constant review and reduces defect rate. Pair programming also enhances technical skills, improves team work, and considered to be more enjoyable for the participants [349] [375] [383-384]. Nagappan et al [381] reported an improvement in pairs' grades on examinations over students who programmed individually. Researchers have used various methods of group formation [349] [378-379] [381-382].

Domino et al [385] reported better performance and satisfaction outcomes using face-to-face pair programming, as compared to its virtual setting. They also found that limiting the extent of collaboration can be effective, especially when programmers are more experienced. Sfetsos et al [386] have shown better performance and collaboration-viability for pairs with heterogeneous personalities and temperaments. Brereton et al [387] report the results of a systematic literature review of ten empirical studies. They conclude that pair programming may improve undergraduate students' pass and retention rates on programming modules, and is likely to improve their confidence in their work and their attitude towards programming. Lui and Chen [388] reported a software process fusion (SPF) process in a real software production situation by alternating between pair as well as solo programming. They found that the longer team members work alone, the more code patterns they develop for reuse later in pairs.

Many practitioners have also felt that pair programming was not sustainable, and they had to take breaks from pairing. Some have also reported a loss of self-confidence and the development of a reluctance to program alone [147]. These are very serious deterrents for introducing it in the educational setting.

The pair programming method, as defined and practiced in available literature, has a few other limitations and weaknesses with respect to the educational context. In pair programming, the pair sits on the same computer all the time and takes turns to write the code. This technique has a major disadvantage when one of the students in a pair is dominant or faster compared to the other student. Pair programming, as reported so far in the literarure, does not completely satisfy

Dillenbourg's four conditions (Table 7.11). In many cases the collaboration was not monitored, and lead to a major disadvantage where students complained that it was difficult for them to be free at the same time. In VanDeGrift's [375] experiment, nearly half of the students felt that it was extremely difficult to schedule time for meetings. At times, students have also complained of unreliable partners [390], and the possibility of being paired with a parasite [349], suggesting that pair programming does not provide a scenario based on well-defined roles.

We have refined the approach to pair programming. Our intervention is discussed in Section 9.2.3.1.

Section 7.4.4: Cross-level Peer Mentoring

The code of ethics of all engineering and computing societies put highest emphasis on social welfare. As per 'adult learning theory' [320], adults are motivated to learn by six factors. In addition to external expectation, personal advancements, and cognitive interest, these factors also include building social relationships, engaging in social welfare, and stimulation, i.e., contrast from routine work. Service learning, when integrated in the regular curriculum, satisfies all these factors. Hence, it has become a well accepted approach of experiential education that combines curriculum with meaningful service. Many educators see it as a way to enhance professional and interpersonal skills of students. It is found to be particularly useful for enhancing their sense of civic responsibility, and preparing a ground for lifelong civic engagement. Penn State University has even started a peer reviewed International Journal for Service Learning in Engineering (IJSLE) that can be accessed at http://www.engr.psu.edu/IJSLE.

With increasing demand of higher education in general, and software development education in particular, the class size has become larger, and it continues to increase. Universities find it increasing difficult to build enough capacity to provide a teacher's long term individual attention to all students. This seriously affects the quality of the laboratory work and hands-on practice in computing courses. Few highly motivated students are able to overcome this limitation by building their own network among working professionals and/or senior students. However, the majority finds it difficult to cope up with the challenging laboratory work without support from more experienced persons.

Based on his studies, Bloom argued that about *90% of the tutored students achieved the level reached by the highest 20% of the students* of a conventional class of 30 students. Further, with respect to higher mental processes: problem solving, application of principles, analytical thinking, and creative thinking, the average tutored student was above 98% of the students of the later group [393]. Because faculty cannot simultaneously guide so many students, they also develop a reluctance to increase the amount and complexity of the laboratory work. Hence, in real practice, at many institutes, the students do not get enough challenge and practice in software development.

The concept of mentoring the juniors, during and after qualification, is a well established practice in many professions like medicine, nursing, law, chartered accountant, social work, school teaching, etc. Mentoring has been defined as *a process for the informal transmission of knowledge, social capital, and psychosocial support perceived by the recipient as relevant to work, career, or professional development; mentoring entails informal communication, usually face-to-face and during a sustained period of time, between a person who is perceived to have greater relevant knowledge, wisdom, or experience (the mentor) and a person who is perceived to have less (the protégé)* [394].

Organizations use it for widening of skills base and competencies in line with their strategic goals, and find it a cost effective form of personal development. It also improves teamwork and cooperation in organizations. *Mentees get benefitted* by mentor's support in many ways: analysis and reflection, problem solving, self-confidence and ability to take risks, acceptance of criticism, as well as broadened horizons and maturity.

Teaching has been well recognized as one of the most effective engagements to learn. Learners create deeper understanding for themselves by teaching others. *Mentors also draw several benefits:* improved awareness of own learning gaps, ability to give and take criticism, leadership, organizational and communication skills, ability to challenge, stimulate and reflect, and stimulation [395].

Since the 1970s, University of Missouri-Kansas City has been spearheading an academic assistance program, Supplemental Instruction (SI) that utilizes peer-assisted study informal sessions. The sessions are facilitated by 'SI leaders,' students who have previously done well in the course, and who attend all class lectures, take notes, and act as model students [396]. Researchers have reported that 75.8% of medical schools in USA had near-peer tutoring programs [397]. Lockspeiser et al have studied several of the earlier studies on the practice of near-peer teaching in medical schools, and also documented experiences of students and their seniors who worked as near-peer-teachers at University of California, San Francisco [398]. Reported benefits in earlier studies and in their documentation include: reduction in the dropout rates, improved academic performance, alternate explanations, and enhanced confidence of junior students.

Topping and Ehly argue that peer assisted learning works well for the tutees because it offers them easy access (quantity and immediacy), and it can also enhance their motivation and confidence while tutors develop a sense of pride and responsibility [399]. Peer tutors engage in explaining, answering questions, correcting tutee errors, manipulating different representations, etc. This provides them opportunities not just for rehearsing their knowledge but also for reflective knowledge building by recognizing and repairing their own knowledge gaps and misconceptions, integrating new and prior knowledge, and also generating new ideas [400].

Since 2005-06, University of Calgary, Canada has been running a college-wide inter-disciplinary three credit course '*Collaborative Learning and Peer Mentoring*' for fourth year undergraduate students [401]. This course includes weekly discussion sessions of 2.5 hours, and 40 hours of practical experience of 'curricular peer mentoring.' The students of this course collaborate with a '*host instructor'* to serve as cross-level peer mentor for a '*host course*,' which they have already taken. Peer mentoring activities include a combination of in-class and out-of-class activities in the host course. Enthusiastic students are offered a second opportunity to mentor through another course '*Advanced Peer Mentoring*.' These students are also engaged in mentoring the new peer mentors.

Section 7.4.4.1: Possibility of Cross-level Peer Mentoring in Software Development Education

In 2009, a feedback was collected from working professionals' online community LinkedIn. They were asked the question "How did you benefit from your first experience as a mentor/coach/guide" and also requested to critique the idea of making mentoring compulsory for all final year students. Twenty-seven professionals (excluding our own alumni) belonging to diverse countries and age group responded, and also commented on the idea. An overwhelming majority of these respondents very enthusiastically supported the idea and also identified many very significant learning outcomes from their own first mentoring experience. Some of the identified learning outcomes from their personal experiences are given in Table 7.12.

Table 7.12: Software professionals' reflections about advantages of first mentoring experience

1.	… it was the best two years of my time at IBM. Every day I went to work and came home with a smile on my face ...
2.	… I learned how valuable diversity was to the success …
3.	… we get multiple perceptions…
4.	… Are you able to explain which is the best idea? and which is not? Can you explain concepts that initially are beyond the other person? These are crucial skills and mentoring helps to develop and sharpen them.
5.	…keep on learning by inventing new ideas… .
6.	…I learnt more about myself, decision making process, individual differences, and of course communication skills…
7.	… my learning grows exponentially by coaching or guiding someone
8.	… great sense of satisfaction … fresh perspective to your own outlook ... learn how to manage interactions and figure out how to deal with people
9.	"Help" is a fundamental button for homo-sapiens …
10.	I found that I was required to look within myself and develop patience and empathy…
11.	… by sharing what they know, it forces them to think about things critically so that they can explain it to someone else. I have found personally that mentoring forces me to grow, and usually benefits me more intellectually than the recipient

Hence, we include mentoring in our proposed framework of pedagogical engagements (Table 8.5). We discuss the details of our intervention in Section 9.2.3.2.

Section 7.5: Chapter Summary

In this chapter, we discussed some surveys conducted by us among software developers, students, and also faculty. We also collected and analyzed a large number of anecdotes of effecting learning experiences. We established that *teacher-centric educational experiences like lecture, written examination, and homework are least effective for developing required competencies. We also found that student-centric educational experiences like projects, laboratory work, and discussions, active and collaborative engagements in lecture classes, teaching, and mentoring juniors/peers give much deeper learning experiences to computing students.* We concluded that students experience deeper learning in active and collaborative environments.

In the light of our own empirical studies, and various existing theoretical perspectives, we presented our reflections on the phenomenon of learning. Finally, we also discussed several studies about designing student engagements for facilitating deeper learning.

In this backdrop, *we propose a novel unified framework of pedagogic engagements in software development education in the next chapter. We have chosen and integrated some very useful theories and models in this unified novel framework. This framework can be used for designing interventions for instructional reform in computing courses for developing students' multi-dimensional competencies with respect to the requirements of software development. In the ninth chapter, we shall discuss some such interventions designed and tested by us.*

CHAPTER 8: A FRAMEWORK OF PEDAGOGIC ENGAGEMENTS IN SOFTWARE DEVELOPMENT EDUCATION

In this chapter, we propose a unified framework of pedagogic engagements in software development education. This framework can be used for designing interventions for instructional reform in computing courses, for developing students' multi-dimensional competencies with respect to the requirements of software development. To summarize the objectives for our framework, we first consolidate the key findings discussed in the earlier chapters. In the third chapter, we concluded that in addition to solving *ill-defined problems*, it is imperative *to develop diverse types of thinking skills, comprising whole-brain activity, among software professionals.*

In the Table 3.2, we designed a three-tier taxonomy of core competencies for software developers. In order to provide a ready reference, this is reproduced as Table 8.1.

Table 8.1: Three-tier taxonomy of core competencies for software developers (Ref: Table 3.1)

Basic Competencies	**Competency Driver-Habits of Mind**	**Competency Conditioning Attitudes and Perspectives**
1. Technical competence 2. Computational thinking competence 3. Domain competence 4. Communication competence 5. Complex problem solving competence	6. Attention to details 7. Critical and reflective thinking 8. Creativity and innovation	9. Curiosity 10. Decision making perspective 11. Systems-level perspective 12. Intrinsic motivation to create/improve artifacts

Appropriate proficiency in these competencies is required to carry out the activities identified in Table 4.1. We have already discussed the meaning of these competencies with the help of several multi-disciplinary findings and recommendations in the context of software development in the fourth, fifth, and sixth chapters. *Development of these competencies is intertwined with multi-dimensional professional and human development of software developers. In the earlier*

chapters, we have identified and argued in favor of five complementary perspectives for this multi-dimensional development.

In Table 4.7, we unified the *ladders of professional competence* proposed by the Gordon Institute, Dreyfus brothers, and Denning. The professional development to the highest two levels in this ladder is completely beyond the scope of formal educational. Hence, we drop these two levels from educational agenda. Though the professional development at 6^{th} and 7^{th} levels in this ladder also primarily depends upon the professional experiences of the concerned person, the formal educational can help in laying the right foundations for achieving these levels.

Further, in Section 6.5, we concluded that software development education programs should also aim to facilitate students' movement to higher levels in the *ladders of cognitive development, motivation (need-perception) development, levels of systems thinking (Table 6.4), and moral reasoning development* (Table 6.8). The highest three levels in the Perry's Model of cognitive development (Table 6.1), and the highest level in the Maslow's model of motivation (Table 6.9) are difficult to achieve through formal college education. However, college education must nurture the mental habits as well as the attitudes and perceptions to facilitate later development to these stages. All *these five ladders of professional and human development are juxtaposed in Table 8.2.*

Table 8.2: Five-dimensional ladder of professional and human development

Levels of development of professional competence (Gordon Institute, Dreyfus and Dreyfus, and Denning) (ref: Table 4.7)	**Levels of cognitive development** (Perry) (ref: Table 6.1)	**Levels of motivation (need-perception) development** (Maslow) (ref: Table 6.9)	**Levels of systems thinking** (derived from Boulding and Sanford) (ref: Table 6.4)	**Levels of development of moral reasoning** (Kohlberg) (ref: Table 6.8)
1. Unconscious incompetence 2. Conscious incompetence 3. Novice (beginner) 4. Advanced beginner 5. Entry-level Professional (competent) *6. Proficient professional* 7. *Expert*	1. Dualism 2. Multiplicity 3. Contextual relativism 4. Pre-commitment *5. Initial Commitment* *6. Challenge to commitment* 7. *Developing commitments*	1. Biological and physiological needs 2. Safety needs 3. Belongingness and love needs 4. Esteem needs 5. Cognitive needs 6. Aesthetic needs 7. Self-actualization needs *8. Transcendence needs*	1 Pre-structural thinking 2 Structural thinking 3 Clockworks thinking 4 Closed systems thinking 5 Complex adaptive systems thinking 6 Developmental systems thinking 7 Evolutionary systems thinking	1 Obedience and punishment 2 Individualism and reciprocity 3 Interpersonal conformity 4 Social systems and 'law and order' 5 Social contract 6 Universal ethical principles

Comprehensive professional development needs an upward movement on these five ladders. Hence, *we term it as a five-dimensional ladder of professional and human development.* Deep learning is necessary for development of twelve competencies and five-dimensional professional and human development.

Section 8.1: Three-dimensional Knowledge Domain for Designing Computing Courses

In Table 4.1, we identified and ranked the most important professional activities that must be included in the main goals for a new curriculum for the future generation of software developers. These activities are rank-listed in first column of Table 8.3.

In the absence of a comprehensive model for knowledge categorization, course designers over-emphasize some kind of theoretical knowledge, and do not give sufficient attention to contextual, meta-cognitive, and empirical aspects. We propose a novel three-dimensional model for the knowledge domain for designing computing courses.

Anderson and Krathwohl categorized knowledge domain for any subject into four types: *factual, conceptual, procedural, and meta-cognitive* [134]. These categories have been used by many educators to design their courses. *The curriculum and course designers do not give sufficient attention to meta-cognitive aspects*. While this classification may be acceptable for designing courses for school education, with reference to higher education, especially professional software development education, we find this to be inadequate. *It does not explicitly address the knowledge related the context and empirical world.*

The context of a subject has to be understood in terms of application domain of the subject knowledge, knowledge about the likely consequences of such application, and also the professional responsibilities. Often these consequences are multi-dimensional, sometimes even transcending the initial imagination of technology as well the domain specialists. Ropohl [336] also highlighted the importance of contextual knowledge in technology. In the context of software, it is even more important. *We extend the knowledge domain categories proposed by Anderson and Krathwohl to make it more suitable for designing the computing courses, by adding the fifth category of contextual knowledge and also have further sub-categories of theoretical and empirical knowledge for each of these five categories*.

Routio [337] sees the world of knowledge in two broad categories of theory and empiria. We leverage this conceptualization to refine these five knowledge categories. This gives us a total of *ten types in the knowledge categories* for designing the course content for any subject in software development education.

The course-designers need to view the details of the activities listed in the first column of Table 8.3 from the perspective of these ten categories and enrich the courses. Table 8.3 gives a schematic representation of this model, which offers a rich spectrum of knowledge types for designing the content of computing courses. *Table 8.3 should be used as checklist for designing the curricula and also the course content of computing courses in software development education.*

Table 8.3: A novel three-dimensional taxonomy of knowledge domain for designing computing courses

Dimension 1 (important software development activities, ref: Table 4.1)	**Dimension 2** (extension of knowledge domains listed by Anderson and Krathwohl)	**Dimension 3** (Routio's categories of knowledge)
1. Algorithm/Computational Procedure/Component and Interface Design 2. Application/Product/System Design/Prototyping 3. Product/Requirement Definition and Specification/Requirement Engineering/Visualization/Consulting 4. Code Analysis, Program Comprehension, Re-documentation 5. *Innovation and research* 6. Application, Component Development/System Integration 7. *Group work, people management, and leadership* 8. Estimation and Costing, Project Scheduling 9. Product/Process Quality Assurance and Control 10. Validation and Verification (Testing) 11. Technical Documentation, Presenting Ideas and Insights 12. Test Design 13. User Interface Design 14. User Acceptance, End-user Documentation, Deployment and Roll-out, Customer Support 15. Security Architecture Design, Architecting, Component Selection 16. Project Monitoring and Control 17. Tools and Technology Selection and Evaluation 18. Usability/Value/Impact Analysis 19. Resource Planning and Management, Staffing and Team Development 20. Risk Planning and Mitigation 21. Build and Release, Configuration Management	1. Factual Knowledge – Specifics: terms, details, and elements 2. Conceptual Knowledge – classifications and categories, principles and generalizations, theories, models and structures (structural rules) 3. Procedural Knowledge – cognitive and psychomotor, skills and algorithms, techniques and methods, criteria for using these, implicit technical knowhow, explicit functional rules (what to do to achieve a certain result in a given situation) 4. Contextual Knowledge – application domain, risks, uncertainties, consequence-centric, professional responsibilities 5. Meta-cognitive Knowledge – strategic (general strategies for learning and thinking), cognitive tasks (including contextual and conditional), self-knowledge	1. Theoretical knowledge domain – Concepts, models, theories, principles, laws, frameworks, theorems and lemmas, methods, taxonomies, templates, patterns, formal languages, guidelines, rules, checklists, standards, code of ethics, etc. 2. Empirical knowledge domain – Existing artifacts and systems, stakeholder and other people's needs, perceptions and aspirations, and tools.

Further, as discussed in third chapter and also in Section 4.1, many characteristics like significant work in new development in every project, discrete abstractions, complex interactions among a very large of components, inherent invisibility, large groups of developers, continuous evolution, etc., make software highly vulnerable to errors. Because of *lack of attention and/or misconceptions,* errors (bugs) creep in during various stages of software development. *Hence, there is a need to place special emphasis on debugging activity at various stages.*

Section 8.2: Two Core Principles Related to Learning

We ground our framework of pedagogic engagements in two core principles associated with learning – cognitive dissonance and cognitive flexibility.

Section 8.2.1: Cognitive Dissonance

Curiosity is the most fundamental requirement for 'learning.' As discussed in Section 6.1, *incongruity, contradictions, novelty, surprise, complexity, and uncertainty can arouse curiosity* [245-246]. Further, the fueling factors also include increased knowledge and *awareness of knowledge gaps in areas that are personally meaningful and engaging. The impediments to curiosity include anxiety, overconfidence, excessive self focused attention, dogmatism, low cognitive resources, internal pressures like guilt and fear, external pressures like threat, punishment, and tangible rewards or pathological conditions* [246].

Cognitive Dissonance Theory [327] postulates the following:

a. Humans are sensitive to inconsistencies between actions and beliefs.
b. Recognition of an inconsistency results in *cognitive dissonance*, and motivates an individual to resolve the dissonance.
c. Dissonance can be resolved in one of three ways: change in beliefs, change actions, *or change perception of actions*.

Based on the cognitive dissonance theory, it has been shown by *Structured Design for Attitudinal Instructions* [321] that instruction can be designed to create short term dissonance. This *dissonance facilitates the learners to first recognize the need to change attitude*, and then they should be guided through progressive changes to resolve the dissonance.

In a similar approach, Kort et al [326] view learning as a spiral process of construction and de-construction (of misconceptions) phases through positive as well as negative emotions. Recognition of mis-conception is preceded by moderate negative emotions.

Non-threatening levels of perceived meaning-deficits generate manageable cognitive load [338] an enabling flow of emotions, and positive incongruence [339]. When the positive incongruence

is within an individual's 'threshold', it *supports learners to sustain their motivation, enjoyment, and efforts.*

However, *perceived inadequacy or overloads of meaning-deficit can create long-term negative emotions* such as anxiety, fear, boredom, frustration, humiliation, dejection, and so on. Continued long term continuation of such sustained negative emotions slow down learners' efforts, and may also lead to completely withdrawal. Stronger negative emotions are felt when the perceived meaning deficit relates more closely to self. This 'threshold' depends upon the learner, learning context, culture, and community. Hence, *in order to help the learners to develop their ability to learn, and also the ability to solve ill-defined unfamiliar problems, the prime aim of higher education has to be to increase this threshold.*

The traditional teacher-centric lectures do not create much or any dissonance among learners. This thesis proposes to transform software development education by creating gradually increasing levels of dissonance for short periods, and then the teacher should guide the students to progressively resolve the dissonance to higher levels of learning.

Section 8.2.2: Cognitive Flexibility

The ability to 'transfer' what learners have learned in a context, to different, even unique situations is referred to as 'cognitive flexibility' [206]. *Cognitive flexibility is closely associated with complex problem solving competence, as well as creativity and innovation.* As per the *Cognitive Flexibility Theory*, the way learners are taught has a significant influence on how flexible the learners will be when they need to use the acquired knowledge.

Cognitive Flexibility Theory posits that the traditional linear teaching may be ineffective for ill-structured knowledge domains. *Aptitude-Treatment Interaction* [319] posits that highly structured treatment is good for low-ability students but hinders high-ability students. Spiro and Jehng recommended that in order to enhance cognitive flexibility, the information must be presented in a variety of ways. They suggested encouraging the flexibility by allowing learners to develop their own knowledge representations to adapt knowledge for future use in different types of situations. With reference to ill-structured learning domains, they also strongly

advocated for presenting multiple representations and different thematic perspectives on the same information. Further, they also recommended that in advanced knowledge domains, interconnectedness of ideas must be emphasized.

Section 8.3: Four-dimensional Taxonomy of Pedagogic Engagements in Software Development Education

The course content of computing courses can be designed with the help of the above proposed novel three-dimensional taxonomy of knowledge domain in Table 8.3. However, *deeper learning resulting in enhancement of competencies through professional and human development is a consequence of various kinds of engagement with the chosen content.*

In fourth to seventh chapters, we discussed the results of our survey, "Software developers - (How) Did your college help you in your development?" (Appendix A10, Summary in Table 7.2). In this survey we had examined the perceptions of software professionals about the effectiveness of various pedagogical engagements on specific competencies listed in Table 8.2. These examined pedagogical engagements included – lecture (knowledge transmission oriented/ thinking oriented), tutorial, home work, written exam, projects, laboratory work, research literature review, industrial training, discussion with faculty, discussions with peers, and mentoring/teaching other students. Table 8.4 gives a further summary of this survey.

Table 8.4: Perceived effectiveness of pedagogical engagements with respect to enhance of competencies: perceptions of software professionals
"Software developers - (How) Did your college help you in your development?" Summary of Table 7.2

SNo	Competency (Table 8.1)	Ranked list of most effective pedagogical engagements (selected by half or more respondents)
1	Technical competence	**Projects (84%)** and Laboratory work (65%)
2	Communication competence	**Discussions with other students (84%), Mentoring juniors (71%),** Discussions with faculty (69%), and Discussion with others (51%)
3	Domain competence	Projects (61%), Research literature survey (51%), and Knowledge transmission oriented lectures (51%)
4	Complex problem solving competence	**Projects (79%),** Laboratory work (59%), and Thinking oriented lectures (51%)
5	Computational thinking competence	Projects (64%) and Thinking oriented lectures (49%)
6	Attention to details	**Projects (71%)**
7	Critical and reflective thinking	Projects (50%)
8	Creativity and innovation	**Projects (82%)** and Thinking oriented lectures (53%)
9	Intrinsic motivation to create/improve artifacts	**Projects (74%**), Research literature survey (58%), Thinking oriented lectures (54%), Discussions with students (50%), and Discussions with faculty (50%).
10	Curiosity	Projects (66%) and Research literature survey (62%)
11	Decision making perspective	**Projects (90%), Industrial training (71%)**
12	Systems-level perspective	Projects (58%) and Mentoring other students (51%)

Further, in the seventh chapter, *we established that teacher-centric educational experiences like lecture, written examination, and homework are least effective for developing these competencies* (Table 8.1) to professionally and confidently participate in activities associated with software development (Table 8.3, 1st column). *We also found that student-centric educational experiences like projects, laboratory work, and discussions, active and collaborative engagements in lecture classes, teaching, and mentoring juniors/peers give much deeper learning experiences to computing students.* We concluded that students experience deeper learning in active and collaborative environments.

In these studies, integrative as well reflective engagements, were subsumed within the larger category of active engagements.

However, NSSE Survey [335] separated these as – higher order, integrative, and reflective. As per NSSE recommendations, the collaborative engagements are subsumed within integrative engagements. In section 8.1, we argued that a comprehensive taxonomy of knowledge domain with well distinguished categories is expected to help the course designers to enrich their

courses. In the similar manner, a comprehensive taxonomy of pedagogical engagements can help the educators to design a rich variety of interesting and meaningful engagements to create learning contexts for their students.

Hence, in our framework of pedagogic engagements, we propose four dimensions of engagements – active, integrative, reflective, and collaborative. Figure 8.1 gives an overview of this model.

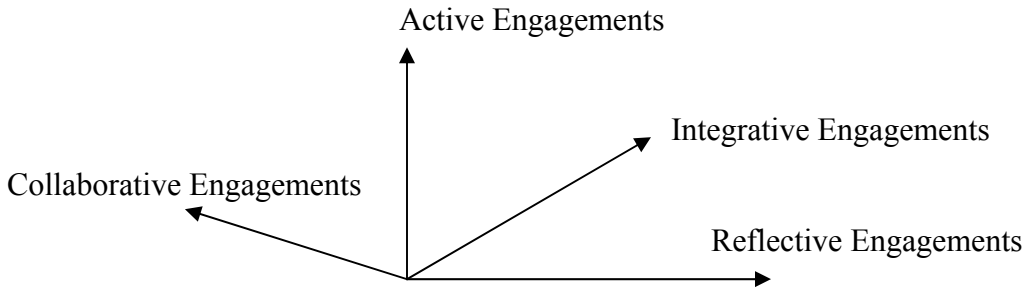


Figure 8.1: A schematic view of four-dimensional taxonomy of pedagogic engagements

Active engagements

The student gets *individually engaged* in *some kind of problem solving activity and proactively adds content to the discourse of learning*. As in Section 4.5 we discussed that software problems are complex ill defined problems. In the same section, we also discussed that and performance in well defined problem solving is not correlated with performance in solving ill defined problems solving. Hence, active engagements in software development education have to provide good experience in solving *complex ill defined problems*. Further, given the high importance of systems-level perspective as discussed in Section 6.3, the students need to analysis and solve these problems *in the light of systems-level perspective*.

The problem situation creates cognitive dissonance for students. Students take appropriate actions to resolve their dissonance. It requires them to actively seek information and apply their knowledge to analyze, create, critique, or decide in following types of tasks (Table 7.8, higher order learning engagements):

1. Applying theories or concepts to practical problems, or in new situations.
2. Analyzing the basic elements of an idea, experience, or theory, such as examining a particular case or situation in depth, and considering its components.

3. Synthesizing and organizing ideas, information, or experiences into new, more complex interpretations and relationships.
4. Making judgments about the value of information, arguments, or methods, such as examining how others gathered and interpreted data, and assessing the soundness of their conclusions.

Complex ill-defined problems create higher levels of cognitive dissonance in a larger variety of knowledge areas. Hence, active engagements to solve such problems create even higher levels of cognitive flexibility and deeper learning.

Integrative engagements

Solving complex ill-defined problems usually requires inclusion and integration of various ideas and diverse perspectives. Such engagements cause higher levels of cognitive dissonance and help in developing systems-level perspective and creativity. Typically, the student is required to perform following types of tasks (Table 7.8, integrative learning engagements):

1. Working on a paper or project that required integrating ideas or information from various sources.
2. Including diverse perspectives in class discussions or writing assignments.
3. Putting together ideas or concepts from different courses when completing assignments, or during class discussions.
4. Discussing ideas from your readings or classes with faculty members outside of class.
5. Discussing ideas from your readings or classes with others outside of class (students, family members, co-workers, etc.).

Reflective engagements

Evolution of beliefs, attitudes, values, perspectives, assumptions, and mental habits are essential desired learning outcomes. Kottkamp [219] defined reflection as *"a cycle of paying deliberate attention to one's own action in relation to intention… for purpose of expanding one's opinion and making decisions about improved ways of acting in the future, or in the midst of the action itself."* Reflective engagements require students to think deeply to evaluate and refine/transform their own approach and views. Such engagements cause *highest levels of cognitive dissonance* and result in deepest learning. Typically, the student is required to perform following types of tasks (Table 7.8, reflective learning engagements):

1. Learning something from discussing questions that have no clear answers.
2. Examining the strengths and weaknesses of your own views on a topic or issue.

3. Trying to better understand someone else's views by imagining how an issue looks from his or her perspective.
4. Learning something that changed the way you understand an issue or concept.
5. Applying what you learned in a course to your personal life or work.
6. Enjoying completing a task that required a lot of thinking and mental effort.

Collaboratively engagements

The student proactively collaborates with others to solve problems. All the above mentioned tasks under active, integrative, and reflective engagements can be performed collaboratively.

Main Theoretical Foundations for the Four Dimensions of Engagements

Bloom's taxonomy, discussed in Sections 2.10 and 7.1.2, can be used as a basic hierarchy of *active engagements*. Harden's taxonomy of curriculum integration (Table 7.9) and SOLO taxonomy (Section 7.4.2) provide the base for designing a ladder of *integrative engagements.* Schön's model of ladders of reflections and Borton's model of reflective thinking, both discussed in Section 5.2, provide vital axis for *reflective engagements*. Salmon's levels of collaboration (Table 7.10) and Dillenbourg's four conditions for collaborating (Table 7.11) provide us the foundations for designing our ladders for *collaborative engagements*.

In our framework of pedagogic engagements, we integrate these forms of engagement into a four-dimensional taxonomy of pedagogic engagement. We use Bloom's revised taxonomy, SOLO taxonomy, Schön's model of ladders of reflections, and Salmons' taxonomy of collaborative e-learning as the main axes of the four dimensions of this novel unified taxonomy of students' engagements. Further, all these dimensions are further enriched and extended, restructured, or enriched with the help of some other very important conceptualizations related to learning. In order to facilitate deep learning among students, we need to regularly engage them at higher levels of all the four dimensions of our taxonomy.

Section 8.3.1: Dimension 1 - Levels of Active Engagements (Extension of Bloom's Taxonomy)

Bloom's Revised Taxonomy

We have created a ladder of active engagements by integrating Bloom's revised taxonomy, Sternberg's propulsion theory of creativity, Minger's framework of critical thinking, and Rowe

and Boulgarides taxonomy of decision styles, in a novel manners. Table 8.5 gives an overview of these levels. We strongly recommend the need for incorporating iterations at the higher levels in this ladder.

Anderson and Krathwohl renamed the Bloom levels from nouns to verbs [134]. They also swapped the position of the uppermost two levels. However, since highest level of evaluation involves designing of criteria/standards and may also requires considerations of larger context, human values and ethics. Hence, it is appropriate to keep it at the highest level. In fact, some create activities may require lower cognitive effort than evaluate, whereas some of them will be based upon serious evaluation. Hence, in order to avoid simplistic classification, we propose to keep create and evaluate at the same level.

Adding a new level: Mentoring

The effectiveness of students' active engagements as mentors has been discussed in fourth, fifth, sixth, and seventh chapters. Further, a large number of fresh software engineers within first two years of starting their professional career, start getting the responsibilities to lead new engineers. Consequently, it becomes very important for the education programs to develop the ability to mentor. With reference to Table A3.2 (Appendix A3), mentoring has been identified as desirable ability fresh software developers. Hence, with respect to designing active engagements in software development education, we extend Bloom's taxonomy by including the tasks to 'mentor' the junior students. For the purpose of senior students' engagements as mentors, we propose two sub-levels of mentoring: (i) coaching for specific skills, and (ii) mentoring the projects. These are included in Table 8.5. Students with positive experiences as mentees are found to be more enthusiastic and serious with mentoring activities. Hence, in the junior levels, it is necessary to give them positive experiences as mentees. *We discuss our experiments with cross-level peer mentoring in Section 9.2.3.2.*

The summary of our adaption and extension of Bloom's taxonomy for our purpose is as follows:

1. Remember: recognizing, recalling
2. Understand: interpreting, exemplifying, classifying, summarizing, inferring, comparing, and explaining

3. Apply: executing, implementing
4. Analyze: differentiating, organizing, attributing, checking, critiquing using existing criteria

5A. Create: generate, plan, and produce

5B. Evaluate: Critiquing based on self-designed criteria, Deciding in the light of larger context, human values and ethics

6. Mentor: coaching juniors for skills and providing guidance in their projects

Table 8.5: Levels of active engagements (first of four dimensions of our taxonomy of pedagogic engagements) (derived from Bloom's revised taxonomy, Sternberg's propulsion theory of creativity, Minger's framework of critical thinking, and Rowe and Boulgarides taxonomy of decision style.)

1. Remember
2. Understand
3. Apply

Several iteration of analysis/create/evaluate are recommended in student assignments

4. Analyze: Iterative
- 4.1. Analyze data
- 4.2. Analyze problems
- 4.3. Analyze complex ill defined problems
- 4.4. Analyze systems

5. Create and Evaluate: Iterative

5A. Create	**5B. Evaluate**	
	5BA. Critique	**5BB. Decide**
5A.1. Paradigm preserving: replication, adaption 5A.2. Paradigm forwarding: forward incrementation, advance forward incrementation 5A.3. Paradigm rejecting: 5A.3A. Paradigm redirection, Paradigm reconstruction 5A.3B. Paradigm re-initiation	5BA.1. Critique of Rhetoric 5BA.2. Critique of Tradition 5BA.3. Critique of Authority 5BA.4. Critique of Objectivity	5BB.1 Directive decision 5BB.2 Behavioral decision 5BB.3 Analytic decisions 5BB.4 Conceptual decisions

6. Mentor
- 6.1 Coach for skill development
- 6.2 Project mentor

Enrichment of 'Analyze,' 'Create,' and 'Evaluate' Levels

Using Bloom's taxonomy in its original or revised form for deciding the learning objectives of school education is perfectly fine. Recently, a lot of engineering or software development education research also has been based on these models. Given the nature of the work of software developers, we take a position that a much higher emphasis has to be placed on 'analyze,' 'create' and 'evaluate' levels.

In Section 7.1.2 we discussed our findings. Our respondents from software industry recommended that more than 70% pedagogic engagements of computing students should be at these three levels. Our findings in Section 7.1.2 also showed that these levels are not sufficiently addressed in engineering/software development education. Hence, in order to fill this gap, there is a need to further refine these levels in order to enhance educators' understanding of the pedagogic possibilities. We postulate that such an expansion of these levels into sub-ladders will help the computing educators design appropriate learning objectives and instructional interventions for their courses. We also suggest that an evolutionary approach is necessary in this process. Hence, we also recommend the use of several iterations in such engagements. Reflective engagements of our third dimension of this taxonomy of engagements will further enhance the value of such iterations.

Further, we also include various models to support student engagement at upper levels of this extension of Bloom's taxonomy. These are depicted in Table 8.6.

Enrichment of 'Analyze'

Sub-levels

The analysis phase is the most important phase in software development. The purpose of analysis in software development is for solving complex ill-defined problems that usually require system analysis. *Systems analysis is very important requirement for software developers*. The analytical habits of software developers have to be necessarily inclined towards using systems thinking for complex ill-defined problem solving. Consequently, we propose to create sub levels of 'analyze.' *We differentiate between the analysis of data, problems, complex ill defined problems, and that of systems*. Through this differentiation, we propose to *enhance the emphasis on engaging*

students in systems analysis for ill-defined problem solving. We also propose to enhance engagements in puzzle solving and qualitative data analysis.

Approaches

Table 8.6a gives the overview of our proposed approach for supporting 'analyze' level engagements. For this level of student engagements, we strongly encourage the practice with various kinds of data analysis techniques (quantitative and qualitative), puzzle solving techniques (discussed in Section 4.5 and Annexure AN6), mathematical modeling for diverse application domains, and reasoning: deductive, inductive, and analogical. We also strongly recommend the use of various complex ill-defined problems solving techniques (Table 4.9). With reference strengthening analytical skills for debugging, we include the Root-cause analysis techniques, as suggested by Metzger [157], and discussed in mentioned in Section 4.1.

Table 8.6a: Some selected models for supporting student engagements at Analyze level

4. Analyze: Iterative *These techniques are also to be used for subsequent **create/evaluate** levels as well*

Data Analysis	Problem Solving	Complex Ill-defined Problem Solving Techniques	Systems Thinking
Quantitative data analysis Qualitative data analysis	Puzzle Solving: Generate and test, Means-end analysis, Working backwards, Backtracking Analogical reasoning Mathematical modeling: Reasoning: Deductive, Inductive, Analogical	Flow charts Concept mapping Systems diagrams SWOT analysis Appreciation 5 Why's Cause and effect diagram Affinity diagrams Appreciative inquiry – 4D approach Root-cause analysis techniques: Cause and event charting Faulty tree analysis.	Definition of system engineering by Frank and Waks Sternberg's definition of Wisdom Senge's toolbox Systems thinking approaches: (i) Checkland (ii) Jacobs Capra's criteria for systems thinking Software systems analysis and design techniques: Data representation techniques: conceptual data modeling techniques, knowledge representation techniques, ontologies, etc. Behavior representation techniques: FSM, State-chart, State Nets, Petri Nets, etc. Risk assessment: Identification, analysis and prioritization of process, product, & project risks Checklists: SEI taxonomies of software risks

With respect to enhancing systems-level perspective, we include the Systems engineering definition by Frank and Waks (Table 6.3), Sternberg's definition of wisdom (Section 6.2, under the theme of Ethical decision making), Senge's toolbox (Table 6.7), systems thinking approaches

(suggested by Checkland as well as Jacobs) (Table 6.6) and Capra's criteria for systems thinking (Table 6.5). We emphasize the use of various software systems analysis techniques to represent data (conceptual data modeling techniques, knowledge representation techniques, ontologies, etc.) and behavior (FSM, State-chart, State Nets, Petri Nets, etc.). Finally, in this context, we also include risk assessment (identification, analysis and prioritization) of process, product, & project risks. Risk taxonomies prepared by SEI can be very useful checklists in this process.

Enrichment of 'Create'

Sub-levels

With reference to the expansion of the level of 'create' from a single level into a sub-ladder, we find Sternberg's taxonomy of creative contributions as a useful source that can be used by educators. This has not yet been used by software development education researchers. Sternberg's taxonomy of creative contributions [229], discussed in section 5.3, includes four levels. We have included these four levels as sub-levels of create in Table 8.5.

Approaches

Table 8.6b gives the overview of our proposed approach for supporting 'create' level engagements. Through their varied educational experiences, computing students' engagements at the level of 'create,' should produce both artifacts as well as theoretical constructs. In such creative engagements, they should be encouraged to forward and also challenge/reject the paradigms rather than remain limited to creating the artifacts and theoretical constructs within the existing paradigms. Mere teaching of existing paradigms in traditional style will not help us achieve this objective. They have to learn to forward and challenge/reject existing paradigms. The discipline of architecture, design, and arts, etc., place a higher emphasis on such engagements, some such practices can be used to enrich the culture of software development education.

As discussed in Section 5.3, problem solving or decision-making tasks, offer maximum creative possibilities during the phases of problem restructuring, alternative generation, criteria definition, and alternative evaluation. Hence, these phases need to be given more attention for designing student engagements. In the same section, we discussed some techniques, e.g., SCAMPER,

lateral thinking and 'po', 40 TRIZ/TIPS principles (Table 5.3) and extensions, etc. These can be very helpful for stimulating creative thinking. Further, as discussed in Section 6.4, creative persons place higher emphasis on self – its uniqueness, development, and expression, as well as on openness to the environment. Development of these traits is essential for developing intrinsic motivation for creating/improving artifacts and/or systems. These are included in Table 8.6.

Many complex ill-defined problem solving techniques as well as systems thinking tools mentioned under 'analyze,' continue to be useful for design engagements at this level.

Further, as discussed in Section 5.3, the activities collated by Aoussou et al [237] and the environmental conditions suggested by Lassig [238] can be can be used for designing students' creative engagements. The engagement in other three dimensions – integrative, reflective, and collaborative, are particularly helpful for finding creative solutions.

Enrichment of 'Evaluate'

Similarly, we also expand both the sub-levels of 'evaluate,' using some established theoretical models in a different context. Table 8.6b gives the overview of our proposed approach for supporting 'evaluate' level engagements.

Table 8.6b: Some selected models for supporting student engagements at Create and Evaluate levels

The techniques listed for analysis are also to be used for these levels as well

4. Create and/or Evaluate: Iterative

5A. Create

Main opportunities of creativity
Restructuring the problem/decision task
Generating alternatives
Selecting decision criteria and strategy, and
Evaluating alternatives

Cognitive tools for creative and inventive thinking
SCAMPER
Lateral thinking, and 'po'
40 TRIZ/TIPS Principles and extensions
Emphasis on self – its uniqueness, development, and expression
Emphasis on openness

5B. Evaluate

5BA. Critique

Common errors of logical and analytical reasoning (Table 5.1)
Misdirected focus
Storage limitation
Information availability
Hypothesis persistence
Limited reviewing
Inadequate data
Multiple variables
Misplaced causality
Dealing with complexity

Paul's Extended Checklist for Critique	
Elements for Critical Thinking	Standards for Critical Thinking
Purpose,	Clarity
Problem to be solved or question at issue,	Specificity
Concepts,	Relevance
Information,	Logical
Assumptions,	Significance
Inferences & interpretation,	Consistence
Points of view,	Breadth
Implications & consequences	Depth
Context	Accuracy
Criteria	Precision
Method	Fairness
	Completeness

Blaauw's principles of system architecture
Consistence
Orthogonality
Propriety
Parsimony
Transparency
Open endedness
Generality
Completeness

5BB. Decide

Decision deficiencies
Aim deficiency
Need deficiency
Aggregate outcome deficiency
Competitor deficiency
Process cost deficiency

Decision challenges
Conflicting interpretations,
Conflicting priorities,
Incomplete understanding of the criteria of evaluation and risks of each alternative,
Absence of good decision making strategy

Decision oriented model of software processes	
<u>Subspaces</u>	<u>Purposes</u>
Problem	Manage complexity and risk
Solution	
Construction	Reduce negative impacts of uncertainty and complications
Operation	

Decision making approaches
PROACTIVE approach
Boyle's Ethical decision making process (wrt ACM-IEEE code of ethics for Software Engineers)

Techniques
Pareto analysis,
Paired comparison
T-Chart
Decision matrix
Grid analysis
PMI (Plus, Minus and Interesting),
Decision Tree
Star-bursting
Step-ladder
Group technique: Six thinking hats, and *Delphi.*

Sub-levels of 'Critique'

In section 5.2, we discussed skill based in software occur either because of 'inattention' or 'over-attention' by the developers [157]. We also discussed that voluntary attention is a very good substitute for genius, and unlike genius, it can be sharpened through practice and perseverance. We also discussed that in order to discover more details about an object, one needs to engage in several iterations of (re)examinations and evolutionary expressions. Critique of the work products of earlier iterations in the light of the re-examination of the object progressively reveals newer details and affords new opportunities for richer descriptions and refinement.

The sublevel of 'critique,' is expanded into four levels using Minger's framework proposed for critical thinking with reference to management education [216], discussed in section 5.2. We find it relevant for the purpose of software developers as well. The four levels of this framework are included in our framework to expand the critique engagements.

Approaches for Critique

Paul's extended model of creative thinking, discussed in Section 5.2, is powerful instrument for carrying out the critique at all these levels. Hence, this model is also incorporated as a checklist for critical thinking in our framework. Further, in the same section, we also discussed Metzger's collation of research based findings regarding errors in logical and analytical reasoning (Table 5.1). This list is also included to guide critical thinking in our proposed framework of pedagogical engagements (Table 8.6b).

Paul's extended model does not bring some of the important elements that are necessary for critique of systems. Hence, we also include *Blaauw's principles of systems architecture*, discussed in Section 6.3, as criteria for critique of systems.

Enrichment of 'Decide'

The sub-level of 'decide,' is expanded into four levels using the taxonomy of decision styles proposed be Rowe and Boulgarides, discussed [260]. They identified four kinds of decision styles that are suitable for different kinds of problems. These styles are directive, behavioral,

analytic, and conceptual (Table 6.2). *We posit that the software developers need to integrate the four decision making styles*

Two decision making frameworks discussed in Section 5.2, *PROACTIVE* [264] and *Boyle's six stage process of ethical decision making* for computing professionals [273] are powerful instrument for carrying out the critique at all these levels. In order to take meaningful decisions, students need to be exposed to various types of software risks, as briefly suggested in Section 6.2.

Engineering/computing students do not get sufficient experience in 'critique' and also in 'decision making,' especially in analytic and conceptual style decisions, that require *decision making in ambiguous situations that require collection of large amounts of data and evaluation of a large number of alternatives.* Such decision making can be facilitated by the use of *decision making techniques discussed in Section 6.2.* The last two of these techniques, six thinking hats and Delphi require a collaborative approach.

A Richer Hierarchy of Active Engagements

None of these abovementioned three main models (Sternberg [229], Minger [216], or Rowe and Boulgarides [260]) used for expanding the sub-levels of 'create,' 'critique,' and 'decide,' have so far been used by software development education researchers. In February 2010, a search of the ACM SIGCSE digital library and also the IEEE digital library, did not a show even a single paper referring to these models. With this integration, Table 8.5 gives a *richer hierarchy of possible active engagements. In order to enhance opportunities of deeper learning among students, the educational programs must ensure repeated engagements at upper levels in this hierarchy.*

These higher levels of engagements need to be applied with reference to the three-dimensional knowledge domain (Table 8.3). The objects of study and deliverables of engagements for these levels must include a good variety of knowledge categories. Both the theoretical as well empirical worlds need to studied and also inflected through students engagements. Consequently, these engagements must also ensure a good mix of *convergent, assimilative,*

divergent, and accommodative activities as per **Kolb's model** (Table 4.3). *All the core competencies, identified by us, can be nurtured through such diversified higher-level active engagement. The three competency driver–habits of mind, as well as competency conditioning attitudes and perspectives, will be especially strengthened through such engagements.*

Section 8.3.2: Dimension 2- Levels of Integrative Engagements (Extension of SOLO Taxonomy)

In Section 7.4.2, we discussed a five-level Structure of the Observed Learning Outcome (SOLO) taxonomy [329]. As per this taxonomy, the lower three levels: 'pre-structural,' 'uni-structural,' 'multi-structural' are about quantitative increase in details of the response. The upper two levels: 'relational' and 'extended abstraction' are about its qualitative transformation through integration, extension, and abstraction. The first level indicates complete lack of comprehension and understanding. As we are not using SOLO taxonomy as a standalone hierarchy, *we drop its first and fifth level in our adaption of integrative engagements.* The fifth level of the SOLO taxonomy is addressed by combining the relational level of the SOLO taxonomy with the 'create' level of our first dimension.

Orbits of Integration

In Table 7.9, we outlined Harden's taxonomy of curriculum integration with reference to the specific context of medical education. We find it very suitable for designing integrated computing curriculum as well.

Further, in Section 4.3, we discussed about Biglan's classification of academic disciplines (Table 4.2) that classifies the disciplines with the help of three bi-level axes: (i) soft vs hard, (ii) pure vs applied, and (iii) life vs non-life [170-173a]. Each of these three axes classifies the larger cube containing all the disciplines, into two cuboids. Any two of these axes, create four quadrants of disciplines, and all three categorize the disciplines into eight octants. Engineering and computer science belong to the octant of non-life, hard, and applied disciplines. The difficulty of integrating two or more disciplines depends upon the degree of similarity between those disciplines, as per Biglan's classification.

We propose a novel approach of refining all the levels of curriculum integration, proposed by Harden (ref: Table 7.9), further into four sub-levels of first, second, third, and fourth orbit integration. We explain these four sub-levels in Table 8.7.

Table 8.7: Discipline integration sub-levels based on Biglan's classification of disciplines

1. **First orbit integration:** The integrating disciplines share the same category along all the three bi-level axes, as identified in Biglan's classification. With reference to computer science, first orbit integration implies that all other involved disciplines also belong to non-life, hard, and applied category, e.g., civil engineering, telecommunication engineering, mechanical engineering, chemical engineering, electrical engineering, etc.
2. **Second orbit integration:** The integrating disciplines share the same category along any two of the three bi-level axes. At this level of integration, at least one of the concerned disciplines must belong to the other different category along any one of the three axes. With reference to computer science, second orbit integration implies that at least one of the other involved discipline belongs to (i) life, hard, and applied category, e.g., agriculture, psychiatry, medicine, pharmacy, dentistry, horticulture, etc., or (ii) non-life, soft, and applied category, e.g., finance, accounting, banking, marketing, journalism, library and archival science, law, architecture, interior design, crafts, arts, dance, music, etc., or (iii) non-life, hard, and pure category, e.g., mathematics, physics, chemistry, geology, astronomy, oceanography, etc.
3. **Third orbit integration:** The integrating disciplines share the same category along only one of the three bi-level axes. At this level of integration, the concerned disciplines must belong to the other categories along any two of the three axes. With reference to computer science, third orbit integration implies that at least one of the other involved discipline belongs to (i) life, hard, and pure, e.g., biology, biochemistry, genetics, physiology, etc., (ii) life, soft, and applied category, e.g., recreation, arts, education, nursing, conservation, counseling, HR management, etc., or (iii) non-life, soft, and pure category, e.g., linguistics, literature, communications, creative writing, economics, philosophy, archaeology, history, geography, etc.
4. **Fourth orbit integration:** The integrating disciplines do not share the same category along any of the three bi-level axes. At this level of integration, the concerned disciplines must belong to the other categories along all the three axes. With reference to computer science, fourth orbit integration implies that at least one of the other involved disciplines belongs to life, soft, and pure category, e.g., psychology, sociology, anthropology, political science, area study, etc.

Since the application domains of software developers belong to all kinds of disciplines, software developers must develop their ability of integrating their disciplinary knowledge of computing with the disciplinary knowledge of any other discipline. *The task of integration between those disciplines that are quite divergent from each other as per Biglan's classification is far more challenging and much more creative, as compared to the inter-disciplinary integration between closer disciplines.*

Sub-levels of Relational Level of SOLO Taxonomy

We feel that the 'relational' level of SOLO taxonomy, 'relational,' can be interpreted by educators at various levels. Without the relational approach and multi-disciplinary and inter-disciplinary integration, complex real-life problems and systems cannot be analyzed, designed, or evaluated effectively. Hence, *we refine the 'relational' level of the SOLO taxonomy into a*

ladder with the help of the chosen elements from Harden's taxonomy (Table 7.9). We further refine the multi-disciplinary, inter-disciplinary, and trans-disciplinary levels in this ladder, into four sub-levels each using our novel four-level ladder of the first orbit, second orbit, third orbit, fourth orbit integration, proposed above.

Through this arrangement, we propose a new hierarchy of levels of integrative engagements for designing instructional interventions in computing courses. Table 8.8 gives this hierarchy.

Table 8.8: Levels of integrative engagements
(second of four dimensions of our taxonomy of pedagogic engagements)
(derived from SOLO taxonomy, Harden's taxonomy of curriculum integration, and Biglan's classification of disciplines)

1. **Uni-structural:** One or a few aspects of the same topic are picked up.
2. **Multi-structural:** Several aspects of the same topic are treated as if they were separate, different ideas not integrated coherently.
3. **Relational:** different ideas (from a topic, subject, discipline, many disciplines, and many disciplines with real-life context) are integrated coherently.
   - i. Intra-topic relational
   - ii. Intra-subject relational
   - iii. Multi-subject intra-disciplinary relational
   - iv. Inter-subject intra-disciplinary relational
   - v. Multi-disciplinary relational: perspectives of individual disciplines are retained, and disciplines use a black-box interface oriented approach for applying other discipline's material and perspective.
     - Four sub-levels from first to fourth orbit (Table 8.7) multi-disciplinary relational
   - vi. Inter-disciplinary relational: discipline-specific perspectives are given-up to create an open synergy. The material of different disciplines is approached and integrated in an open manner, rather than with a black-box interface oriented approach.
     - Four sub-levels from first to fourth orbit (Table 8.7) inter-disciplinary relational
   - vii. Trans-disciplinary relational: the focus is on real-world problems transcending disciplinary boundaries.
     - Four sub-levels from first to fourth orbit (Table 8.7) trans-disciplinary relational

Cognitive flexibility [206], and hence, complex problem solving competence, domain competence, and creativity and innovation are especially nurtured as a result of such higher level integrative engagements. Further, they also help in enhancing systems-level perspective. Computing educators need to design engagements and instructional interventions to facilitate this. Currently, this is a serious weakness of engineering education as found by the NSSE survey discussed earlier in Section 7.4.

With reference to course design, we advocate for following three approaches of integration in courses:

1. *Multi-level Infusion (interventions discussed in Section 9.2)*
2. *Intra-disciplinary Integrative Capstone courses (interventions discussed in Section 9.2.2)*
3. *Interdisciplinary Integrative Capstone courses (interventions discussed in Section 9.2.2)*

Section 8.3.3: Dimension 3 - Levels of Reflective Engagements

In Section 5.2, we discussed about reflective thinking. Bateson's model of logical categories of learning [328], discussed in Section 5.2, suggests that deepening levels of learning require change of action, assumptions, or context and commitment. The first level of learning is about making minor fixes or adjustments in action. The second level of learning requires reflection to challenge one's beliefs and assumptions. This facilitates new insights for changing the rules for making major changes. The third level of learning requires even deeper reflection to bring about a shift in understanding our context, values, point of view, and commitments. Further, Schön in his work on reflective thinking and professions [125] [220], discussed in Section 5.2, posited that the mental habit of reflection and ability to move across the ladders of reflections is central to professionals' approach to their work. These habits are also closely associated with software development work. Agile methods like eXtreme Programming draw their strength from the possibility of continuous improvement through reflection.

Based on these two models, we propose the levels of reflective engagements for computing students. These engagements will require the students to review and rethink about the products, processes, assumptions, and value of all their engagements at different levels of all the other dimensions of our four-dimensional taxonomy of pedagogical engagements. These reflections should encourage them to reflect about the reflections as well, i.e., creating ladders of reflections. *Borton's three-stage model [225] of deliberately thinking about 'what,' 'so what,' and 'now what,' discussed in Section 5.2, has to be used in some form at each such stage and ladder of reflection*. This reflection will show new insights to them. The reflection process will help them to improve their work, practices, habits, and perhaps even revise their value systems. In fact, there is no better way to help them to revise their value system with reference to the professional responsibilities they need to handle in accordance with the suggested codes of professional conduct, practice, and ethics. In Table 8.9, *we propose a new model for representing these levels of reflective engagements.*

Table 8.9: Levels of reflective engagements

(third of four dimensions of taxonomy of pedagogic engagements)
(derived from Bateson's logical categories of learning and Schön's ladders of reflection)

Pre-reflection: No reflection 1st order Reflection: Product Reflection- creating ladders of reflections around the results and products of their other engagements 2nd order Reflection: Process Reflection- creating ladders of reflections around the processes in their other engagements 3rd order Reflection: Assumption Reflection- creating ladders of reflections around the assumptions behind the products and/or processes in their other engagements 4th order Reflection: Value Reflection- creating ladders of reflections around their self-beliefs and value system that influences their assumptions, goals, and role in their other engagements Borton's model of reflection for all levels What? So what? Now what?

Reflective engagements help in sharpening critical and reflective thinking which, in turn, has a cascading effect on all the other core competencies. Reflection is not an automatic activity. It requires deliberate engagement. Currently, lack of reflective engagements is a serious weakness of engineering education as found by the NSSE survey discussed in Section 7.4. *In order to inculcate the ability to learn, we strongly recommend that small reflective exercises must follow most of students' assignments.* Computing educators need to design reflective engagements and instructional interventions to inculcate the habit of reflection. *Reflection is embedded in our intervention of project centric evolutionary teaching, discussed in Section 9.1.2. In section 9.1.3, we discuss some reflective engagements visualized and administered by us. Further, in Section 9.3, we also discuss, our experiment with conducting reflective workshop on pedagogy for engineering faculty*

Section 8.3.4: Dimension 4 - Levels of Collaborative Engagements

The group methodology promises to facilitate collaboration, promote mentorship, and enhance collective ownership of code. We assert that whenever group work engages the learners to evaluate, adapt, transform, extend, and integrate their individual ideas to co-generate newer collective ideas, it creates stimulating conditions for learning at higher levels of the other three dimensions. We also postulate that such group work also enhances attention to details, critical and reflective thinking, and also creativity and innovation. Further, it also stimulates students to reflect and evolve their perception of peer's role in learning. Hence, properly designed group

work offers the potential to contribute in students' cognitive development as per Perry's, and also Bextor Magolda's, intellectual progression models in more than one way.

In Section 7.4.3, we discussed about various theoretical perspectives about collaborative learning. In Table 7.10, we outlined *Salmon's proposed levels of collaborative e-learning* [357]. These levels are used in Table 8.10, as proposed levels of collaborative engagements for our framework of pedagogic engagements. The collaborating units can be individuals or groups. Individuals may collaborate in small or large groups at different levels as per Salmon's ladder. Similarly *small sub-groups may also collaborate with other sub-groups at different levels*. For example, in a particular situation, the intra-subgroup collaboration may be carried out at synergistic levels, whereas the inter-group collaboration may take place in dialogue, peer review, parallel, or sequential mode. *Hence, we add sub-levels of intra-group and inter-group collaboration of all levels except the first and last in Salmon's levels*.

At all these levels, Dillenbourg's four conditions of collaborative learning (Table 7.11) need to be satisfied to draw learning benefits from these collaborative engagements. In Section 9.2.3.1, we discuss our approach of collaborative pair and quadruple programming, which combines all the features of this model. Cross-level peer mentoring, discussed in 9.2.3.2, also gives the mentors an opportunity to engage in cross peer review. Collaboration is also embedded in all forms of problem centric teaching discussed by us in Section 9.

Table 8.10: Levels of collaborative engagements
(last of the four dimensions of taxonomy of pedagogic engagements)
(Derived from Salmon's levels of collaborative e-learning and Dillenboug's four condition)

1 Solo: no collaboration
2 Dialogue: simple exchange of information
Intra-group, Intergroup
3 Peer review: reviewing others' work
Intra-group, Intergroup
4 Parallel Collaboration: dividing the task in the beginning, and finally integrating individuals' work
Intra-group, Intergroup
5 Sequential Collaboration: building upon each other work
Intra-group, Intergroup
6 Synergistic Collaboration: doing it together in a synergistic manner

Dillenboug's four condition (Table 7.11)
1. Set up the initial conditions
2. Over-specify the collaboration contract with a scenario based on roles
3. Scaffold productive interactions by encompassing interaction in the medium
4. Monitor and regulate the interactions

We suggest that these levels must also be integrated with the levels of the other three dimensions of our four-dimensional taxonomy of pedagogic engagements. The current practice is that in the name of the collaborative work, the students are usually not engaged in the highest two level of collaborative engagement as per salmon's levels. Further, they also do not normally experience inter-group collaborations. We strongly suggest that in performing their engagements discussed so far, the students must also be engaged at the higher levels of collaborative engagement as given in Table 8.10. *Higher level collaborative engagements will enhance cognitive flexibility [206] and systems-level perspective.*

Section 8.4: Chapter Summary

Students need to carry out and reflect upon multi-subject, multi-disciplinary, inter-disciplinary relational analysis, creation, and evaluation. On few occasions, they should also preferably get engaged in some form of trans-disciplinary relational analysis, creation, and evaluation, and subsequent reflection. It must also be ensured by the faculty that the students get many opportunities to integrate computing knowledge with a large variety of disciplines, especially those that belong to divergent categories as per Biglan's classification. In order to further deepen their 'learning,' reflective engagements, especially at higher levels of Table 8.9, are also necessary. Finally, in all these engagements, there should be enough opportunities for higher-level collaborative engagements (Table 8.10). *We strongly recommend that small reflective exercises must follow most of their assignments.*

Our proposed framework of pedagogic engagements in software development education is grounded in (a) core activities of software development, and (b) distinguishing characteristics of software development profession. It includes –

1 three-tier taxonomy of twelve core competencies,
2 five-dimensional ladder of professional and human development,
3 three-dimensional perspective of the knowledge domain of software development,
4 two core principles (cognitive dissonance and cognitive flexibility) for facilitating learning, and

5 a four-dimensional taxonomy of pedagogic engagements (active, integrative, reflective, and collaborative) over '3' for developing '1' and '2.'

We postulate that higher level engagements in the first three dimensions will create the necessary learning conditions by creating 'cognitive dissonance.' Higher level engagements in integrative and collaborative dimensions will create cognitive flexibility.

The core competencies in our three-tier taxonomy are likely to be sufficiently addressed by higher level pedagogic engagements in all these four dimensions. It is neither sufficient, nor recommended to only use these dimensions in an isolated manner. Through their undergraduate education, students must be repeatedly required to carry out such comprehensive tasks that engage them at the higher levels of multiple dimensions, sequentially, or even better, simultaneously. In the next chapter, we discuss some interventions developed by us manifesting some aspects of this framework. The development of the framework and these interventions has been an intertwined and highly spiral process.

CHAPTER 9: SOME INTERVENTIONS FOR ENHANCING THE QUALITY OF SOFTWARE DEVELOPMENT EDUCATION

In the previous chapter, we developed a framework of pedagogic engagements. This framework can be used for designing a large variety of instructional interventions to immerse students in a four-dimensional hierarchy of active, integrative, reflective, and collaborative pedagogical engagements. In this framework, we included two core principles for facilitating deep learning: cognitive dissonance [327] and cognitive flexibility [206]. *Cognitive dissonance is about the conditions that are necessary for learning and cognitive flexibility is about the mastery of some subject matter*. Instructional interventions designed with the help of this framework can help the faculty to create conditions of cognitive dissonance and flexible learning of the subject. In this chapter, we discuss some such interventions developed by us. As stated in the previous chapter, *development of the theoretical framework and these empirical interventions has been an intertwined process*.

Learning primarily happens because of learners' engagement in various activities relevant to the content, rather than mainly depending upon content's transmission from external sources. Student-centric active learning offers the freedom of different kind of activities for different students. Depending upon their prior experience and interests, students can choose or even define their activities. The author has applied some in-class active learning techniques such as think-pair-share, share your experiences with the project and assignments, design a small algorithm, and so on and also advocated a strategy of activity based flexible credit definition as one component of learner-centric education [358].

Baumgartner [359] has proposed a framework for viewing teaching as a designed activity, and has observed that teachers employ a diverse range of coaching and mentoring strategies like 'teacher as guide, 'teacher as project manager,' and 'teacher as troubleshooter' in open-ended learner-centric classrooms to support students during their design process. It has been suggested

that an open-ended approach encourages a diversity of views and perspectives, and also makes a critical reflection on observations and experiences possible.

Section 9.1: Increasing Cognitive Dissonance through Problem-centric Approach in Software Development Education

The most natural way to *create dissonance* (Section 8.2.1) would be to lead learners *through a problem-centric approach*. We elaborate upon three types interventions based on problem-centric approach: *Inquiry Teaching*, *Project- inclusive Teaching, and Reflective Teaching/Assignments.* In *all these interventions, we try to engage the students at higher levels of all the four-dimensions of our framework of pedagogical engagements: active, integrative, reflective, and collaborative (Section 8.3).*

Section 9.1.1: Inquiry Teaching in Software Development Education

The lecture format in which abstraction precedes the instantiation and concretisation, helps students in developing skills in linear thinking and deductive reasoning, and also succeeds in creating a knowledge-base as an inventory of unutilised concepts. It however fails miserably to give direct and guided practice in inductive reasoning and lateral thinking. Bruner and other constructivists [30-31] recommend that instruction should allow the learner to discover principles for themselves through active dialogue. Instead of aiming to teach some general rules and theories, inquiry teaching aims to teach how to discover the general rules and theories. Inquiry teaching is particularly effective in exposing learners' misconceptions. *It is particularly suited for developing curiosity, self-learning, analytical skills, humbleness, inductive and lateral thinking skills, and hence, in facilitating deep learning.*

Inquiry teaching revolves around questions. This will require the teachers, and also the students, to ask many more questions in their classes. We have developed a new model, SERO, for designing inquiry teaching oriented lectures. This mode has been tried out in some courses, viz., Data Structures, Computer Graphics, Orientation to Engineering, etc.

Section 9.1.1.1: SERO Model for Inquiry Teaching in Software Development Education

The discourse in the lecture classroom can be viewed as a story telling artifact. The objective of this artifact is to create a meaningful learning experience and knowledge structures for every learner. The discourse in a large number of lectures is designed as a closed artifact that primarily sees the students as consumers. A fundamental challenge for designers in the new millennium is to design open systems and artifacts by inventing and designing a culture in which humans can express themselves and engage in personally meaningful activities [317]. Open systems and artifacts must evolve, they cannot be completely designed prior to use. They must evolve at the hands of the users, and they must be designed for evolution. The dichotomy of designer and user has to be eschewed. *Seeding, Evolutionary growth, and Reseeding (SER) has been proposed as a conceptual framework for designing sustainable, open, and evolutionary systems* [318] [360].

A seed is the initial state of a system that is intended to evolve. The evolutionary growth phase is one of unplanned evolution as the seed is used by the members of a community to do work. Reseeding is a deliberate effort to organize, formalize, and generalize knowledge created during the evolutionary growth phase. Courses as seeds have been proposed as a promising model to evolve and enrich courses by allowing students to act as active contributors, and not just as passive consumers [361].

The genesis of any story experience is Emotional Movement [362]. Users crave emotional engagement and stimulation. Situated inside the context of lecture classroom, every learner (user) is the author of his own personal meaning. Meaning is the product of interaction between the observer and the system, the content of which is in a state of flux, of endless change and transformation [363]. In Poetics, Aristotle suggested that a well constructed plot must be a whole having beginning, middle, and end [364]. Movement Oriented Design (MOD) views a story as an ensemble of 'story units' in which a 'story unit' has three parts, the Begin, Middle, and the End (BME) [365]. Begin lays the groundwork, hooks the user, imploring to find out more. Middle carries the main story message, conveys the core meaning. End terminates the story, concludes the current story, and/or links to the next.

As per the SERO model, every lecture is delivered as a series of SER blocks, and concluded with a learning Outcome. **Seed** is the fresh idea or question from a teacher which is generally not an obvious derivative of an earlier idea. **Evolution** has been used to label the active learning phase in the class involving individual thinking, group work, discussions (among student groups of varying size, and also between the students and teacher), and solving problems that require thinking in terms of analysis, synthesis, and/or evaluation. **Reseed** is being used to label the phase of formalizing the informal ideas generated during the evolution stage, and deriving another seed as a derivative of this evolution.

Students usually have greater motivation to learn in the context of solving a problem, than if the content is delivered out of context [366]. The seeding phase in SERO based lectures offers good opportunity to create context. Situated in this context, the content is developed during the evolution phase through problem solving activities.

The teacher makes a deliberate attempt not to deliver generalized content without the context or before problem solving. Instead, the generalisations are presented as a natural fallout of the theorising process through solution-unification during the reseeding phase to conclude the evolution phase. In this model, the teacher has to support the students during evolution phase individually or in smaller groups and only some time the entire class.

The teacher needs to be the centre of attention of the entire class only during the limited period of **seed** and **reseed** stages, and occasionally during the **evolution** stage, as and when the need arises. Sometimes the evolution phase may also become teacher-centric, as the teacher may occasionally decide to demonstrate the problem solving process with some specific case(s), rather than engaging the students in problem solving because of the lack of sufficient background with the students or time constraints. However, the problem solving characteristic of the evolution phase remains unchanged. At the end, the learning outcomes are summarized and an assignment is announced. This assignment forms the reseed for the next class.

Usually there are not many seeds in a lecture, only reseeds. Most of the time is used in evolution and active learning. This model has been tried out successfully in many courses, even with a

large number of students. Figure 9.1 shows pictures of one such class during the evolutionary growth of a concept through group exercise. Appendix A14 gives a summary of two such lectures, one each in computer graphics and data structures.

Figure 9.1: Group exercise during the evolutionary phase of SERO style lecture

Experience

SERO style lecture classes were found to be highly engaging and useful by motivated undergraduate students. However, many other students, who were mainly motivated by examination oriented study, did not find these classes very useful for them.

Challenges for Inquiry Teaching in Software Development Education

The success of Inquiry Teaching mainly depends upon students' active participation in the inquiry process. It requires, and also furthers, the *transformation of students' perception about their own role in the process of learning from an information receiver to an active contributor to meaning making*. However, for many students, their old habits formed through prior experiences with exposition based teaching, can hinder their enthusiastic participation as an active learner in the classroom, especially in large and unresponsive classes. Such students find inquiry teaching to be unsatisfactory, and miss the opportunity of not only deep but also surface learning. Therefore, it is most important to sensitize students to this method of learning in their early courses. *For maximizing the benefits of inquiry teaching, students need to 'learn to learn' through this method.*

Developing Habit for Inquiry Learning in Software Development Education through Puzzle Solving

Solving a puzzle is another example of inquiry learning. Puzzle solving activity demands that the teachers start their sessions with problems rather than concept. Many software companies include puzzle solving in their selection criteria of new software developers. Puzzle solving sharpens critical thinking and problem solving ability, and offers a higher potential to develop many of the multifaceted thinking skills. Therefore, we *redesigned the delivery strategy of the first computing course by starting it with puzzle solving activity, even before the introduction of the basic syntax of any programming language.* In this course, at two different campuses of JIIT, over eight hundred first year engineering students were distributed in six lecture sections and twenty tutorial sections. More than twenty faculty members were involved in delivering lectures and running weekly tutorial classes. All the concerned faculty members (Prakash Kumar, Alok Agarwal, Vikas Saxena, Shikha Mehta, Anshul Gakhar, and Chetna Debas) agreed to the proposal that instead of teaching programming or computer basics, we should start solving puzzles. For over a month, various kinds of puzzles were discussed in the lecture and tutorial classes. The puzzles were collected and chosen by the concerned faculty members. As per the feedback from the faculty, these classes were highly active and collaborative. Even in the post-lunch sessions, students very enthusiastically came to these classes.

Faculty members felt that puzzle solving activity improved students' logical thinking ability, which is at the core for designing computer programs. *A large number of* students have reported multi-dimensional benefits in terms of enhancements in logical, creative, multi-perspective, and out-of-box thinking, attention, focus, concentration, patience, comprehension, urge for creation, etc.

The faculty members expressed that these were the most active and collaborative classes they had ever attended or conducted. It showed them the benefits of active and collaborative inquiry oriented classes. Encouraged by the positive results of this trial, we have now infused puzzled solving in two more courses in the current semester. In 'data structures' (2^{nd} semester) and 'fundamentals of algorithms' (4^{th} semester) courses, all teachers have happily dedicated the first

one to two weeks solely to puzzle solving. A more structured research in needed on infusing puzzle solving in software development education.

Section 9.1.2: Project- inclusive Teaching in Software Development Education

According to various surveys discussed in chapters four to seven, we found that projects were the most effective teaching methods with respect to enhancing various competencies relevant to software development. Semester-long project experience helps in developing multidimensional competencies in all the dimensions. Hence, semester-long projects have the potential to facilitate deeper learning in many significant ways. However, projects are usually conceived as a culmination activity of learning something. It is assumed that only after completion of conceptual learning and acquiring practical skills, some project can be attempted. *Usually in Indian universities, semester-long project work is not included as part of the regular computing courses. This limits the effect of the courses in terms of developing their competencies*.

Project-Inclusive Regular Courses

The constructivist paradigm of *project-inclusive teaching* challenges this assumption. Rather than viewing a *project* as the culmination activity, *it is viewed as the instrument of creating richer context for learning the subject matter*. It also opens many new challenges for the faculty. They have to guide the students in formulating and completing their projects. Simultaneously, they also have to manage the learning process. Hence, project-inclusive teaching also offers a higher level of creative opportunity for the faculty as well.

However, as the traditional textbooks are normally not written with this objective in mind, the *project- inclusive course teaching requires a change in delivery strategy*. We have tried to enhance the quality of several undergraduate computing courses by project inclusion. This attempt has given us the confidence that that it is usually possible to plan and deliver the courses with a central focus on the semester-long project work of the students. *Two different models, viz., project-centric evolutionary instruction and project-oriented instruction, have been proposed* for achieving this goal.

Project-centric Evolutionary Teaching in Software Development Education

Project centric evolutionary teaching offers *active, integrative, reflective as well as collaborative engagements* as per our frame work discussed in Section 8.3.

During the course of this research, project-centric evolutionary teaching was evolved for Enterprise Application Development [367]. Recently, it has been further expanded to many other courses like object oriented programming, database management, web application engineering, software engineering, and information systems.

In project-centric teaching, *we reverse the traditional teaching methodology* in which conceptual learning is followed by practice assignments, and only sometimes project work. In our scheme, at the beginning of the course, the teachers first help and guide the students to formulate the initial project problem. Examples and templates are used to complete this task. Since it is not possible for the teacher to discuss every project in the large class, the instructor then selects some of these projects to forward the subsequent classroom discussions. They try to define the initial and simplistic project scoping and specification for one or two projects through classroom discussion. The students follow a similar process to complete these tasks for their projects. Teacher guide the students to *incrementally enhance their project scope later* in the semester, essentially to create the context for the forthcoming concepts and topics of the subject matter. *They refine the project scope before introducing any new topic.*

The teacher has to bring in the concepts after setting the context. Conceptually, this model has some similarity to zero inventory manufacturing practice. The learners are not given a large inventory of unused concepts. The concepts are introduced only after creating the need for its use with reference to students' semester-long project.

We have developed the conceptual schema for defining the main characteristics of student projects' evolution in project-centric evolutionary teaching of object-oriented programming, software engineering, database management systems, web application engineering, enterprise software development and information systems. All these schemas have also been tried in real courses by concerned faculty members. Appendix A15 gives the stages of evolution of the defining characteristics of student projects in different computing courses.

Progressive evolution of the subject matter in the evolving context of a project is the hallmark of this approach. Appropriate concepts, theories, technologies, procedures, and tools are introduced as per the needs of each stage of project evolution. These stages are flexible enough to accommodate new concepts, theories, technologies, procedures, and tools at each stage. Metaphorically, it looks at content delivery as a natural process of a small but complete bud blossoming into a complete flower, rather than like the traditional but unnatural compartmentalized additive advancement. As per this model, most of the projects have high similarity with respect to technological issues. The application domain becomes the main differentiating factor for different student projects. Hence, observation and review of other students' projects also gives an opportunity to expose the students to a variety of application domains.

This model is very suitable when the student projects can be planned to use most of the concepts of the subject matter. This model is not suitable if the objective is to have the students to carry out their projects in different areas of the subject matter, and the projects are required to be differentiated based on their technological aspects, rather than application domains.

Project-Oriented Teaching

While engagement in semester-long projects is highly beneficial for ensuring deep learning in courses, project-centred teaching has its own difficulties as well as limitations. In many courses, it is very difficult, and perhaps not even desirable, to plan technically similar semester-long student projects encompassing most of the topics of the particular subject matter. In such cases, it is better to plan projects on different topics. In order to leverage the advantage of peer learning, care has to be taken to evenly distribute the students' projects over all the main topics. However, this scheme imposes some challenges regarding synchronising the project activity with the content delivery in the class. Either most students are not able to start their projects early in the semester or they have to start the project without any instructional support on the project topic. In order to partially overcome this limitation, a *two-level content delivery scheme* has been tried out in some computing courses like Microprocessors and Microcontrollers, Operating Systems, Computer Networks, and Complier Design.

As per this scheme, the entire course is delivered in two phases. In the first phase, lasting approximately two to three weeks, an extended introduction of the course gives a comprehensive macroscopic view of the entire subject matter. During this phase, the major issues, relevance, and typical project possibilities with respect to all topics are presented to the students. The objective of this phase is to help students broadly understand the subject matter, see the inter-connections between different topics, and *also identify their project topics as well as formulate their project problems. Thereafter, the students start working on their projects.*

In the second phase, the topics are sequentially picked up for in-depth classroom discussion.

If students' projects are evenly distributed over all the main topics, topic related projects can be easily leveraged to provide the context and enrichment for detailed discussion on the topic, and also give a partial flavour of project-centric teaching for every topic.

Section 9.1.3: Creating Conditions for Reflective Engagements in Software Development Education

Reflection is not an automatic activity. Students do not usually automatically reflect well upon their actions and tasks in various assignments. This limits not only the quality of their assignments, but their overall learning as well. A small post-assignment, reflective activity can amplify their learning from the same assignments. Borton's frame-work for reflective thinking [325] discussed in Section 5.2 and Section 8.3.3, includes three questions: 'what, so what, and now what?'

We have successfully deployed this framework to enhance the learning value of many assignments. For example, for the past three years, we have been asking the students to maintain a log (PSP style), of their time and programming errors in software laboratories. More details about infusion of PSP (time estimation as well as bug) are discussed in Sections 9.2.1. Many students were finding it to be a wasteful activity. We realized that because they never referred to their logs, they saw no benefit of creating such logs. Hence, recently we introduced an element of mid-semester reflective exercise, where a group of few students jointly review their logs and

write a reflective report using Borton's framework. This exercise helped the students to draw and see the benefits of maintaining a log. They saw what kind of errors they were commonly making, and how they compared with other peers. This helped them in improving their programming skills.

For the last several years, we have been asking the final-year students to submit a reflection report on their final year project. After completing the project, they are required to give an additional report answering the questions given in Table A16.1 (Appendix A16). Further, in the main project report, we have added elements that require reflective thinking. These include project specific reflective review of quality assurance procedures, debugging, risk recovery, and error and exception handling techniques.

In 2009-10, we have specifically tried to inculcate reflective thinking through reflective engagements in several courses. For example, in three elective courses for the 8th semester undergraduate students, 'software documentation,' 'software construction,' and 'software risk engineering,' delivered at a fast pace in three weeks, reflection has been used very strongly. In each of these three courses, the students were required to write a report, reflecting on their 7th semester project in the light of subject knowledge of the respective courses. They were required to suggest strategies to improve their project's specific aspects that were related to the subject matter of these specific subjects. Table A16.2 (Appendix A16), gives the problem statement of the assignment (25% credit) in these three courses. Further, even in the final exam, question(s) were asked to make them reflect upon this work. These were finally designed by the concerned faculty members in consultation with the author of this thesis. *In future, we plan to create more templates for infusion of critical thinking and reflection in many computing courses.*

In an ongoing elective course, 'software arteology,' reflection is being infused in all assignments. The assignments require them to essentially reflect upon published literature, professional's experiences, peer's experiences, or their own experience. Further, at the end of every assignment, they are also required to write a small report specifically addressing the issues as per 2nd and 3rd questions in Borton's model. Table A16.3 gives two sample assignments in this course. The students are required to submit their assignments, only after a peer review. This also brings some

elements of collaboration and reflection on others' work. The second-last and last sub-questions in each assignment are based on the 2nd and 3rd question respectively as per Borton's framework. The responses of the last sub-question, in each assignment are particularly very interesting, where students are expressing what they learnt by doing the specific assignment, and what they plan to change in future. Table A16.4 (Appendix A16) gives a few sample responses.

We conclude that reflective engagements are highly effective for creating deeper learning. More work is required to reflective assignments in all courses and tasks.

Section 9.2: Increasing Cognitive Flexibility through a Multifaceted Integrated Approach in Software Development Education

In order to engage the students at higher levels of integrative engagements, the second dimension of our four-dimensional engagement taxonomy discussed in Section 8.3, and to impart cognitive flexibility with special reference to software development, an integrated approach to software development education is necessary. In order to achieve this objective, we have visualized and administered *three types of interventions for instructional reform*, viz., *Multilevel Infusion* of key technologies (web, multimedia, mobile, and security) and professional practices (systems design, estimation, open source, and debugging), *Integrative Courses* and *Group and Community Learning*.

Section 9.2.1: Multilevel Infusion for Continuous Integration in Software Development Education

As mentioned in Section 7.4.1, the details of multi-level infusion of various technologies and professional practices are discussed below. This kind of '*multi-level infusion' offers 'Inter-subject intra-disciplinary relational' engagement to the students as per Table 8.8.* In the following discussion, we also refer to the feedback received from mentors (Appendix A17). The details of our intervention to engage senior level students as cross-level mentors are discussed later in section 9.2.3.2. It may be noted that every mentor was mentoring only one host course.

Multi-level Infusion of Web Technology

Web technology is integrated in several introductory courses. The first programming laboratory courses, introduces HTML, before any practice with programming. In the data structures course, they do some programming assignments around HTML files. JDBC is introduced in database course. Some web search and page ranking algorithms are included in the fundamental of algorithms course. Information systems course focuses on building web-enabled information systems. Computer network courses starts from the topmost layer of the protocol stack, leveraging students' familiarity with the web, and goes deeper to lower layers. Table A17.1 (Appendix A17) gives a summary of the feedback received from the mentors regarding infusion of web technology in different junior level courses.

Multi-level Infusion of Multimedia Technology

Multimedia technologies are infused in many introductory courses. In the first programming course, students learn to use basic graphics and sound functions. In the data structures course, deep practice of recursion is given with the help of several examples of graphical fractals. They also learn some basic data structures for simple geometric objects. The data structures for building simple games like snake-and-ladders, and ludo, etc., are also discussed. Database systems course insists on creating databases with multimedia objects. Graphics API's are used in object-oriented programming as well. In the algorithms course, students are required to write programs for algorithms visualization and also implement simple games, using decision trees. In the web application engineering course, hypermedia design patterns are introduced. Table A17.2 (Appendix A17) gives a summary of the feedback received from the mentors regarding infusion of the multimedia technology in different junior level courses.

Multi-level Infusion of Mobile Technology

Aspects of mobile computing have also been infused in some introductory courses. For example, in the operating systems course, an overview of mobile operating systems is given. J2ME is included in database systems and web application engineering. The course on information systems includes Android. The courses on computer organization, microprocessors, and computer architecture also bring some discussion about mobile platforms. Table A17.3

(Appendix A17) gives a summary of the feedback received from the mentors regarding infusion of the mobile technology in different junior level courses.

Multi-level Infusion of Security Aspects

Recently, some attempts [368-370] have been made to incorporate security aspects in computing courses. We give an elaborate model of infusing security related aspects in every semester of the first three years. We have chosen traditional computing courses for infusing the selected security aspects without overloading the students or compromising on the main topics of the core course. Appendix A18 gives the details of this model [370a]. Some of the topics indicated for each course, can be easily integrated by interested faculty.

Some features of this model have already been tested in our courses. Infusion of security aspects in our courses, including some laboratory assignments has been highly appreciated by final year students who are mentoring the laboratories of first three year courses. Table A17.4 (Appendix A17) gives a summary of the feedback received from the mentors regarding infusion of security aspects in different junior level courses.

Multi-level Infusion of Systems Design Aspects

In order to lay an emphasis on systems design, some improvisations like necessity of design diagramming, evolutionary project scoping (in many courses), and necessity of following design guidelines and standards (in some courses), have been visualized and administered. For helping design diagramming habits for analysis and design of systems, a new graphic notation, *Concept Map*, for representing software systems has been developed and administered in some courses. The details of this graphic notation are given in Appendix A19.

Faculty and students have found this concept mapping technique to be very useful. It has been used several times in the data Structures course. Recently, it has also been deployed in some other advanced level courses to compliment the standard notation of UML. Table A17.5 (Appendix A17) gives a summary of the feedback received from the mentors regarding infusion of the some of the system design related aspects in many aspects in different junior level courses.

Multi-level Infusion of Estimation Tools

In order to develop estimation skills, the programming process related data, as adapted from the Personal Software Process (PSP) [371], has been administered in many computing laboratory courses of the first to third year. Initially, the students show a lot of unwillingness to record the PSP logs of their progress. It is perceived as an unnecessary burden that has nothing to do with their programming tasks. However, a reflection after some practice makes many of them self-realize the benefits of using it with respect to their programming practice. In April 2009, the students were asked to write their comments on the use of the PSP in their computing laboratory. This was an open-ended feedback. A majority of the students have reported benefits in terms of programming efficiency enhancement, defect rate reduction, activity record, and reflection. Many of them have also reported benefits in terms of improvement in estimation and planning skills. Table 9.1 gives a summary of the feedback received from students regarding the perceived benefits of maintaining PSP logs in their computing laboratory courses.

Table 9.1: Benefits of PSP as perceived by Students

Benefits of PSP perceived by Students	2nd Semester students, 109 responses	4th Semester students, 91 responses	6th Semester students, 75 responses
Programming Efficiency Enhancement	57%	59%	76%
Defect Rate Reduction	37%	37%	59%
Activity Record and Reflection	32%	39%	59%
Estimations and Planning	25%	27%	31%

Based on this feedback, we realized that *Humphrey's format of PSP logs is not good enough for enhancing the estimation skills of undergraduate students. Hence, we have modified it for achieving higher gains in estimation as well. In our new PSP format, students are required to write their estimated time for completing their programming assignments.* Students are also required to *revise their estimates after every stage* of the software development: analysis, design, and implementation. Finally, they *also record the actual time* for completing their assignments. With this continuous engagement with estimation, they become sensitive to its importance. Further, every revision in estimates makes them more careful while making future estimates. Table A17.6 (Appendix A17) gives a summary of the feedback received from the mentors regarding infusion of the estimation related tools and techniques in different junior level courses.

Multi-level Infusion of Open Source

Open source has been infused in many ways. Students are encouraged to search, select, and include/modify open source code in their projects: mini projects in all courses, minor projects, and final year project. Since 2008, all final year students are required to give an additional report on how they used and/or modified the open source code in their project. Open source is regarded as published literature, and all literature survey oriented assignments allow and often insist on inclusion of open source survey, e.g., in advanced data structure (M.Tech. course), all students were required to survey open source in their chosen application domain, and catalogue the data structures used in chosen code. In the first programming course, the 1st semester undergraduate students start their programming laboratories with introductory Python, even before they use C. Core Java is taught in the object-oriented programming course. MySql and JDBC are used in the database course. Linux is used in the Unix laboratory course. The web application engineering course includes PHP, Java Script, XML, J2ME, etc. Linux is used in Operating systems course.

In 2008, for their 5 credit minor project-I, all 5th semester computing students were required to enhance an open source project in the area of software engineering (any phase). In 2009, all 5th semester computing students built database driven websites using open source add-ons like crawlers, security API, J2ME, etc. Since 2007, program comprehension and re-engineering have been included in the 5th semester software engineering course. In 2008, all 6th semester B.Tech (IT) students used Wonderland for creating database integrated virtual worlds in their second minor project. J2EE, Android SDK, XMS API, Ajax, Open XLS, and Drupal are being used in the information systems course. NS2 simulator is used in the computer networks lab. The elective course, computer graphics, emphasized the use and/or extension of open source game engine, e.g., Box2D, Box3D, J-monkey, Ogre-M, etc. Table A17.7 (Appendix A17) gives a summary of the feedback received from the mentors regarding infusion of the open source in different junior level courses.

Multi-level Infusion of Debugging

Debugging is generally thought of as an implicit activity for software development. Students are expected to detect and fix their buggy code. Due to this expectation, students typically have limited experience in bug detection. Not only this, the debugging experience they get is purely

by chance and not by design. As part of assignments, students can be asked to identify and correct code section containing buggy code. The computing curriculum recommendations by professional societies [1] [50] [52] [113] also include debugging. To address this issue, assignments for bug detection and removal should be given. However, we feel that the typical delivery of computing courses does not enforce students to have debugging experience in a systematic manner. To bridge this gap between suggested curriculum and its actual implementation, we propose some guidelines for assignments. Bug detection with and without use of debugging tools should be inculcated among students.

In order to infuse debugging experience, we have prepared a taxonomy of software bugs with an objective of designing debugging related assignments in various computing course. This taxonomy has been summarized in Appendix A6. We take a view that software bugs are results of misconceptions about specific topics in specific courses. Debugging assignments can be given during delivery of courses based on topics included in our taxonomy of bugs. The taxonomy can be used as an input for generating these assignments. Bugs related to a particular course, as mentioned in our taxonomy, should be experienced by students during the course delivery. This can be done through assignments such as bug detection, bug generation, comparative study of debugging tools, and program comprehension of existing debugging tools, creating simple debugging tools for specific bugs and enhancing existing debugging tools. Additionally, students can be asked to maintain a bug log for every programming assignment as prescribed by Humphrey [371].

The log format suggested by Humphrey has the following parameters: (i) date of bug detection, (ii) sequential numbering of bug, (iii) bug category, (iv) phase of SDLC in which the bug was injected, (v) phase of SDLC in which the bug was removed, (vi) time spent in finding and fixing the current bug, (vii) bug number for the bug whose fix resulted in current bug, (viii) brief description of the bug, mentioning its reason. *We propose that following parameters should also be incorporated in the bug log: (i) behavioural manifestation: the symptoms of running system that helped in finding bug, and (ii) techniques and tools that helped in fixing bug.*

Reflecting on the data from the bugs log will help the students in systematizing their art of debugging. In our experience of introducing this log in laboratory work of various computing courses like introduction to programming, data structures, object-oriented programming, database management systems, algorithm, software engineering, information systems, and compiler design has been encouraging. A large number of student have felt that it helps them to improving their programming and debugging skills. However, a good number of students, especially at the first year level have found this to be too time consuming. While a two-third majority of second and third year students appreciated the benefit of maintaining this log, only one-third of the first year students found it to be useful. Table 9.1 summarizes the benefits, perceived by the students. Table A17.8 (Appendix A17) provides a summary of mentors' feedback on PSP logs in some computing laboratories. Appendix A20 gives some more proposed interventions in this regard.

Some more aspects being considered for multi-level infusion in software development education

In collaboration with various faculty members, we continue to strengthen the infusion of the abovementioned eight elements: web technology, multimedia technology, mobile technology, security aspects, design aspects, estimation aspects, open source, and debugging in various introductory computing courses. Now, we have also started working towards designing appropriate models for infusion of selected elements of the following important issues related to software development:

1. Software documentation
2. Software quality
3. Software risks managements
4. Advanced level programming techniques
5. Formal methods in software engineering

Some elements of these have already been infused in the final year project deliverables. More work is needed to offer courses using the approach of *'multi-level inter-disciplinary infusion.'*

Section 9.2.2: Integrative Capstone Courses in Software Development Education

As discussed in Section 7.4.1, integrative capstone courses can help in strengthening nonlinear, integrative and systems thinking, and flexible learning.

Courses for Intra-disciplinary Integration of Diversified Computing Topics

We have made some attempts to design some advanced level computing courses like 'multi-dimensional data structures,' 'systems programming,' and 'graph algorithms and applications' to bring integration of otherwise widely spread computing concepts. *These integrative courses offer 'Inter-subject intra-disciplinary relational' engagement level as per Table 8.8.*

Data structures design is a pervasive computing concept, and design of application specific data structures is a crucial software development activity. The basic course on data structures helps in creating a general purpose foundation for most of the later computing courses. In this basic course, usually some common linear and non-linear data structures are introduced in multiple application contexts. Later, the courses on algorithms try to further build up this generic understanding. Many of the specialized computing courses can be modeled as data structures, algorithms, and methods. Limited space in the curriculum does not give the opportunity to take many such specialized courses. An integrative course on 'has been created to deepen problem solving ability with a special focus on various domains involving n-dimensional, multimedia, and spatio-temporal data. This course leverages the advantages of basic foundation data structures course and multilevel infusion. It offers students an opportunity to learn about some important computation issues related to various areas of computing, e.g., computer graphics, image processing, multimedia, GIS, robotics, data mining, mobile computing, bio-informatics, VLSI Layout, etc., in a single course. More importantly, they also understand and explore the reusability of many multi-dimensional data structures across application domains. This approach enables flexible learning.

Similarly, the course on '*graph algorithms and applications*,' attempts to contextualize the graph based algorithms in a variety of application area. The course on '*systems programming*' leverages the learning in microprocessors, operating systems, compiler design and computer networks.

Courses for Inter-disciplinary Integration with Selected Elements of Human Sciences

Over the years disciplines have evolved such that they have been separated not only in terms of underlying factual, conceptual, and procedural knowledge, in theoretical as well as empirical space, but also in terms of research questions, perspectives, meta-cognition, and methodologies. In modern times, most of the interesting developments are taking place at the edges of the disciplines. The disciplines are getting integrated not only in terms of content but also perspectives and methodologies. A trans-disciplinary approach is required for solving most large real-life problems. Hence, the integration of seemingly disconnected disciplines of human knowledge offers very exciting learning opportunities. Here we elaborate upon some experiences in designing and delivering some courses that try to contextualize and integrate computing with selected elements of human sciences. These human science concepts have been carefully chosen based on their relevance and importance for enhancing some core competencies. We have made an attempt to create a fourth orbit inter-disciplinary integration (Table 8.8) with some traces of trans-disciplinarity in these courses.

Theory of Knowledge, Learning, and Research

As discussed in Section 3.7, Armour [120] [148] viewed software development as a learning activity rather than a production activity, and advocated that software developers need more training in learning, and knowledge structuring mechanisms, rather than in software itself. The course of 'theory of knowledge, learning, and research' attempts to addresses this requirement. The students are exposed to a spectrum of theories related to human learning and thinking. All these theories are also used for reflecting about learning in general, and also with specific reference to software development. It enhances the understanding of their own learning process, and also helps them identify their misconception about learning. These theories and models help sharpen students' *questioning skill, critical thinking, and reflective thinking*. It helps the students to become *better learners*. Understanding of the diversity of learning styles also prepares them with enhanced *ability of self-regulation*, *ability to accommodate themselves to others and also makes better prepared for understanding of domains' thinking processes.* They are motivated to view software development as a learning and critical thinking activity.

Human Aspects for Information Technology

Another multi-disciplinary integrative course is 'human aspects for information technology.' It aims to explore the humanistic grounds for information technology and software development. Students analyze the required competencies for specific activities of software development. The students evaluate the information technology, and also the activity of software development with respect to multi-dimensional aspects of social welfare and professional decision making. Various activities of this course help the students to understand the meaning and importance of professional *responsibility*. They are engaged in collecting and analyzing professional dilemmas of practicing software developers in the light of theories of moral reasoning and human development. They learn about technological disasters and failures of software systems.

Various codes of professional ethics for engineers are also analyzed in this course. All these experiences help the students to understand the meaning and importance of professional responsibility. In this backdrop, the models of critical thinking are used for analyzing the ethical issues with respect to ongoing developments in information technology.

Finally, a module on creative thinking and inventive problem solving is integrated with this background (as per Table 8.6b). Selected models of creative thinking and inventive problem solving are used for designing ethically sensitive technological solutions and services. Further, there is strong tradition of formally teaching 'Research Methodology' in many non engineering disciplines. Such content is not usually offered in engineering disciplines. However, some programs of information systems offer such courses. The research in the field of software development combines the research methods of engineering as well as social sciences. In this course, various qualitative as well as quantitative research methods are discussed with the help of illustrative examples from the published research literature in software development. It also helps in enhancing the *critical and integrative thinking, analytical skills, and also self-learning.*

Section 9.2.3: Group and Community Oriented Engagements in Software Development Education

In the following two sub-sections, we report our experiments with two different models of group and community learning. *Collaborative pair and quadruple programming prepare the students*

for teamwork and benevolence, while cross-level peer mentoring is aimed for preparing them for larger organizational concerns, universalism, and responsible citizenship of larger communities. The students reported several benefits of collaborative pair programming like enhancement of *problem solving skills, efficiency, quality, trust, and teamwork skills*. Further, it provides experience in *reading and understanding foreign code, writing code for others' understanding, and integrating one's code with foreign code*. Both forms of group and community oriented engagements help in enhancing students' *sense of accountability and responsibility, ability to accommodate themselves to others, to see themselves as bound to all humans with ties of recognition and concern, as well as multi-perspective and creative thinking.*

Section 9.2.3.1: Collaborative Pair and Quadruple Programming

Using our framework, we have transformed the popular concept of pair programming, to make sure that both the students in a pair necessarily collaborate, build upon each other's work, and also do an equal amount of similar work. Our adapted implementation of 'collaborative pair programming' is based on facilitating higher levels of collaboration using Dillenbourg's four conditions of collaborative learning as discussed earlier (Table 8.10). Table 9.2 shows Dillenbourg's four requirements for maximizing collaborative learning, and how we implemented each in our study.

Table 9.2: Application of Dillenbourg's principles

Dillenbourg's requirement	Our implementation
1. Set up the initial conditions.	Pairs of students without any programming experience were formed by faculty in the beginning of the semester.
2. Over-specify the collaboration contract with a scenario based on roles.	In each laboratory session, the members of each pair were first required to individually complete two different programming tasks. On completion of both their individual tasks, they worked together to solve a more complex problem that was designed as an extension of both their individual problems (ref: Table A21.1, Appendix A21).
3. Scaffold productive interactions by encompassing interaction in the medium.	The pair members were not allowed to interact for completing their individual tasks. However, if one of the pair member completed his/her task much in advance, and the other member felt the need of peer's support even for completing his/her individual task, the laboratory instructor allowed them to do so by assigning a small penalty of marks to the second member.
4. Monitor and regulate the interactions.	For every group of thirty students, at least two faculty members and one teaching assistant were available for clearance of doubts and monitoring.

Each exercise fulfilled the purpose of making the students work both individually and in a team. Table A21.1 (Appendix A21) shows a few sample exercises [389]. The concerned teaching faculty validated these assignments before administering the same to students. Appendix A21 gives some more details about the setting up of experiment.

For all the laboratory instructors and teaching assistants, the most common observations were to find paired programmers brainstorming far more than individual programmers, suggesting alternate implementations during evaluations, approaching instructors for doubts lesser than individual programmers, and having more details like null checks and memory checks in their programs. It led the students to check their thinking and reason their decisions, they examined and discussed their ideas with others, and evaluated other's statements and solutions. They modified their own programs to fit in their code in the new but similar situation presented by the combined task question, and at the same time also acted as evaluators for their partner's programs. We believe that this experience trained them for reading and building upon others' code in future. The instructors also felt that student pairing also helped in improving the effectiveness of teacher student interaction in the labs.

Based on the results and the feedback from the students and instructors, some of the evident advantages of collaborative programming that we could bring out effectively in our course were: *efficiency, trust and teamwork, problem solving skills, and quality.* By the end of the semester, inexperienced-paired programmers reduced the relative performance gap from 40% to 10%, and performed at the same level as the experienced-solo programmers during the final examinations.

In 2009, we administered this form of collaborative programming in the laboratories of the object-oriented programming course for more than 350 students of the third semester of B.Tech. (CSE/IT). These students have already had two semesters of programming experience, but had not experienced pair programming in their earlier courses. Based on the same model, the assignments were designed by five colleagues teaching this course. In the laboratories of this course, we also had approximately thirty final year students as regular visitors, who act like mentors of third semester students. The details of mentoring program are discussed in the next

sub-section. Approximately 70% of these mentors have felt this form of collaborative programming to be extremely valuable for students' long term as well as short term gains. Another 20% have also found it to be valuable, and also felt that this instructional intervention is worth the extra effort by students. These mentors have felt that it exposes students to observe different ways of programming, improve their style, and also encourages weak programmers to learn to program.

Collaborative Quadruple Programming

Our approach combines all the levels of collaboration proposed by Salmon (Table 8.10). The regular two-stage fixed-partner pair programming model has been further enriched by occasional extension into a three-stage semi-fixed partner quadruple programming model. For the purpose of occasional extension into a three-stage model, the laboratory class of thirty students is divided into four categories A, B, C, and D. Like the collaborative pair programming mode, the students first complete their different individual tasks. One student of category A and one student of category B then make pairs, and collaborate to modify, adapt, and integrate their individual work to complete a larger and more complicated task AB. Students of categories C and D also pair to complete another larger and complicated task CD. On completion of their individual tasks, the pair partners test each other's work. If needed, the faster students can also help their partners after completing their own individual tasks. The members of these pairs are fixed for the entire semester, and they are advised to progressively evolve and follow their own coding guidelines through the semester. Finally, in each laboratory session, one AB pair collaborates with a CD pair to complete the final complex task that requires adaptation, modification, reuse, and integration of the work done for their individual and/or pair tasks. These partnerships between pairs are not fixed for the semester. On completion of their combined AB task, the fastest AB pair partners with the fastest CD pair after they have also completed theirs. Gradually, other pairs are also grouped into quadruples. The pairs that are not able to complete their pair tasks are not allowed to carry out the next level of quadruple task. Appendix A22 gives one such assignment for 'J2EE,' based on this model.

Our approach of collaborative pair programming has resulted in benefits like enhancement of *problem solving skills, efficiency, quality, trust, and teamwork skills*. Further, it provides

experience in *reading and understanding foreign code, writing code for others' understanding, and integrating one's code with foreign code*. We have also observed that paired laboratory experience is especially advantageous to inexperienced programmers. Another advantage that was evident from the students' responses to the feedback sessions was that paired programmers were motivated to work collaboratively even outside the class, although this was not demanded or suggested by us. Consequently, we conclude that *this form of collaborative pair programming positively influences all the dimensions of our competency taxonomy*, and also does not suffer from the disadvantage of developing reluctance developed for solo programming, as was reported by some practitioners of regular form of pair programming.

Section 9.2.3.2: Cross-level Peer Mentoring in Software Development Education

As per our framework, *Table 8.5, mentoring experiences gives the highest levels of active engagement*. *It also gives an opportunity to the mentors to review the work of others, giving experience of third level of collaborative engagement (Table 8.10).* Further, it also creates conditions for integration (Table 8.8) and reflective engagements (Table 8.9) for mentors. Hence, in our view, *mentoring offers a wholesome learning opportunity to the mentors.*

During 2005 to 2008, a total of one hundred and sixty-four final year undergraduate students were engaged in mentoring their junior students' laboratories as part of their formal assignment in '*learning sciences'* or '*theory of knowledge, learning, and research*.' They also correlated their real mentoring experience with various learning theories and proposed designs for e-learning systems for specific modules of host courses. In 2008, through the facilitation of *software engineering* course, a total of two hundred and five third-year students were engaged as project mentors for junior students' mini projects.

In 2007-08, selected forty students of another fourth year elective course, *software engineering management*, were engaged to mentor juniors' second year combined project in object-oriented programming and database management systems as part of their own activity: project management practice. In 2008-09, all two hundred students of this course group-mentored the third-year five credit minor projects. These final-year students mentored the juniors' projects to build tools in diverse areas of software engineering. They submitted weekly mentoring reports.

As per the feedback received from the faculty of these courses, nearly 65-70% mentors provided good help to mentees. The three faculty members of the facilitating course software engineering management felt that *mentoring assignment provided their students a better understanding of the role of human factors in software engineering*, *improved their project management, team management, leadership skills and also helped them to improve their problem understanding and problem solving abilities.*

Based on our earlier positive experiences, very encouraging feedback from industry, and consultation with faculty members of the Department of CSE and IT, *in 2009, more than three-hundred final year B.Tech (CSE) and B.Tech (IT) students were compulsorily engaged to mentor approximately fourteen hundred juniors' laboratories and projects at any of the three lower years*. Mentoring was considered as an integral part of their day-to-day work for mentors' own year-long final year capstone project that is assigned more than 10% credit of the entire B.Tech. program. Nearly forty faculty members, who are also the project supervisors of these final year projects, agreed to keep 10 marks (out of the supervisor's quota of 35 marks) earmarked for day-to-day work of the first semester of the final year.

Multiple Benefits of Cross-level Peer Mentoring

The feedback received from host faculty, facilitating faculty, mentee students, and mentor students during different stages of this scheme's implementation has been positive. In 2007, a survey was conducted among the CSE and IT department's faculty members. Twenty-six faculty members responded. *More than 40% faculty members felt that this model of cross-level curricular peer mentoring significantly helped many students*. Another 26% felt that it marginally helped many students, and the remaining were of the view that it was marginally helpful for few students. Most of them felt that it provided benefits to mentees as well as mentors.

In their opinion, mentees got benefits like increased level of instructional and doubt clearing help, increased opportunities for one-to-one out of the class help, improved programming skills, improvement in problem solving approach, and increased comfort level. The other benefits in their view included healthier cross-level relationships between cross-level students, and also

increased confidence of the mentors. Few faculty members also expressed their concern about the risk of increased spoon feeding of the juniors and discipline. *Except for one, all other faculty members expressed their desire to continue the scheme.*

In 2007, a feedback survey was jointly conducted through *facilitating and host faculty* among the second year students of a host course. *Two hundred and seventeen students gave an average rating of 3.3 to more than forty final-year mentors based on the extent of help provided by them on a scale of 0 to 4*. While the juniors felt the benefit of more easily accessible and friendly guidance, their *mentors also reported several learning outcomes from this engagement: increased pride, and hence, enhanced motivation for more challenging work in their final year project, insights for leadership and project management issues, exposure to people related aspects in software engineering, handling quality and late delivery, and enhanced interpersonal skills.*

As the seniors guide the juniors, and also help them in debugging their work, it gives them the practice of reading and comprehending foreign code. It gives the opportunity to *refresh their basics, and also enhances their knowledge,* by asking more questions related to 'how,' 'why,' and 'why not.' It helps to visualize the same concepts from another perspective. This deepens and consolidates their learning, and helps appreciate the interrelationship of advanced level courses with junior level courses. Mentors have reported several other benefits for themselves: *experiencing joy and satisfaction, enhanced confidence, improved understanding of self and others, appreciation of diversity, development of patience, empathy, multi-perspective and out of box thinking, improvement of analytical and debugging skills,* as well as *enhancement of communication, collaboration, leadership and decision making skills.* In the second semesters of 2007-08 and 2008-09, when the mentoring facilitating courses were not operational, many students of the final year, and also the third year, volunteered to mentor the juniors' laboratories without any credit.

Mentors provide support in various ways. The mentors of "Introduction to Computer Programming" have reported to help their mentees in removal of syntactical errors, problem understanding, programming logic development, mapping it to programming language

constructs, debugging, providing study resources, helping, project formulation, etc. Some of them have attempted to work at a deeper level by trying to help their mentees to develop a better approach towards programming problems. More than 70% of these mentors have claim that in order to mentor, they have revised the old content of the host subject, and also learnt the new content that has been added for the juniors through self study. For example, nearly all mentors of the introduction to programming course revised their C language skills, and also learnt Python that has been recently introduced in this course. Every week, before meeting the mentees in the scheduled laboratory time, they prepare themselves well with mentee's specific programming assignments. More than 70% responding mentors claimed to provide regular support to few of their mentees even outside the scheduled contact time. Some motivated mentors have taken some special initiatives like creating online communities of their mentees, regularly holding discussion with their mentees after the scheduled hours. Mentors are also discussing their mentee's problems with other mentors. Some of their comments regarding their own learning gains through mentoring of juniors are given in Appendix A24.

Some of these students have felt that mentoring does well to productively engage their mind better than many other conventional education experiences like lectures, tutorials, and even written examinations. In their view, *mentoring is specifically effective for engaging their mind in the following types of thinking:*

Thinking required paying attention to minute details.

1. Thinking required learning application of some theory, concept, model, tool, procedure, or method.
2. Thinking required critiquing something, and also designing the criteria for the same.
3. Reflection upon personal and others' experience/work/ideas to evaluate/improve it or to identify some pattern/model.

Reflections of Former Cross-level Mentors (Alumni)

To understand the learning gains of mentoring experience, the *alumni of Jaypee Institute of Information Technology* has been approached to give their feedback on their mentoring experiences of juniors' laboratories and projects. This survey conducted by us 2009, is discussed in Appendix A22. The results of their feedback show that, in terms of its effect on all

competencies in our taxonomy, an overwhelming majority of responding alumni members who had got involved in mentoring during their undergraduate program perceived mentoring to be more/most effective as compared to other academic experiences. Many of them found that in comparison all other academic experiences, it was the most effective experience in terms of its effect on development of several competencies. The respondents felt its most significant effect on development of competencies like: *accountability and responsibility, communication skills,* and *ability to accommodate self to others*. Its positive effect on several other competencies is also significantly higher than several other academic experiences. These competencies include: *curiosity with humility, attention to details, critical and reflective thinking, decision making skills, problem solving, creativity and innovation,* and *analytical/design/debugging skills.* More than half of these respondents also mentioned that they are still in touch with their own erstwhile mentors.

Reflections of Final year Cross-level Mentors

In another survey, *conducted in 2009, among the more than three hundred final year mentors,* an overwhelming majority of nearly *95% respondents have felt that mentoring juniors is resulting in their own multi-dimensional learning of various kinds that will be useful for their future career.* Only 15% mentors did not find their mentoring experience to be useful with respect to their final year project. Around 70% of them considered mentoring experiences to be extremely, mostly, or many times useful in terms of its direct or indirect contribution of knowledge, skill, mindset, thinking, habits, problem solving methodology, etc., for their final year project. The mentors of second and third year level host courses considered it to be directly useful for their final year project. They have reported *learning benefits like revision of the subject, sharpening of skills, and improvement of project planning and people related skills like understanding of multiple perspectives, listening skills, group work, leadership,* etc. They also feel that it *has increased their patience, empathy, sense of responsibility, etc.* Some felt that this experience will help them in competitive examinations, placement interviews, or getting teaching assistantship during higher studies. Interestingly, some of them are very excited to discover their hidden teaching talent and interest.

Why Does Cross-level Mentoring Benefit the Mentors?

As per the *cognitive flexibility theory* [206] revisiting a subject with different issue questions makes the learnt matter more easily transferrable to unfamiliar problem situations. Mentoring gives senior students an opportunity to revisit an earlier course from a different objective, higher level of maturity, and richer background of various other related courses. *Mentoring juniors for their laboratories and projects gives a wholesome experience to the mentors.* It engages them in rehearsal as well as elaboration of the host subject's concepts, technical skills, and applications. The act of explaining the subject to juniors requires the mentors to create novel examples, analogies, and expressions. In additional to advising their mentees on doing their assigned problems, many motivated mentors often also design additional problems for them. The act of guiding them in project formulation, scoping, and design helps them to validate their own project experience in various courses. Mentors also often help the juniors in debugging, and some time marginally even in implementation. Many of them have felt that in terms of SOLO taxonomy, earlier they had usually approached the subject from a quantitative perspective with limited focus on inter-linkages between different concepts. The mentoring experience facilitated them to review the same subject from a qualitative perspective at relational level focusing on integrating varied concepts.

Mentoring very frequently creates cognitive dissonances [327] for the mentors. In the process of resolving these dissonances, mentors get engaged in reflection about the subject matter and also about their own thinking habits, attitudes, beliefs, and even values. This reflection created opportunities for deeper learning and transformation. Many mentors have reported that mentoring became their turning point. Mentors have reported several other benefits for themselves: *experience of joy and satisfaction, enhanced confidence, improved understanding of self and others, appreciation of diversity, development of patience, empathy, multi-perspective and out of box thinking, improvement of analytical and debugging skills, as well as enhancement of communication, collaboration, and also leadership and decision making skills.* Some faculty members have observed that sometimes even those students, who had not performed well in their course as regular students, in the later semester, take their mentoring task in the same course very seriously and do an excellent job.

Hence, we conclude that cross-level curricular peer mentoring has multi-dimensional effect on mentees as well as mentors. Instead of viewing it as a strategy to partially overcome faculty shortage for junior level courses, it should be viewed as a necessary educational experience for seniors that help them in enhancing several of their own competencies.

Section 9.3: Reflective Workshop on Pedagogy for Engineering Faculty

The author has also conducted some workshops for engineering faculty on effective teaching process. The experiences of one such workshop, 'effective lecture,' are briefly discussed here. It was conducted in 2004, for the faculty members of three engineering institutes. The session was attended by faculty members of varied experience, and diverse departments of science, engineering, and humanities. At the beginning the workshop, the faculty members were asked to fill up a form to rate the importance (most important/important/not important) of 16 attributes of a lecture. After this few anecdotes collected earlier were shared with them. Then, they were requested to recall and briefly write their own anecdotes about the two most effective formal lecture classes attended by them a student. Then they were also required to recall their own most effective lecture classes as faculty members. They were required to mutually share their anecdotes within pairs. Subsequently, they were required to publically share some of these anecdotes. Faculty members showed a great enthusiasm to share their anecdotes. Finally, they were required to re-rate the same sixteen attributes. Fifty-four faculty members coming from different institutes, departments, qualification level and experience level exercised their option to give their responses to the author. Table 9.3 summarizes these responses.

Table 9.3: Comparison of pre- and post-workshop consolidated ratings by faculty

Lecture Format attribute	Fraction of respondents who rated the attribute as most important at the beginning of the workshop **(A)**	Fraction of respondents who rated the attribute as most important towards the end of the workshop **(B)**
a. careful listening	20.37%	15.09%
b. explain textbook	1.85%	3.77%
c. seek on-the-spot clarifications	42.59%	60.38%
d. seek clarifications	18.52%	18.87%
e. problem solving	38.89%	60.38%
f. creative thinking	66.67%	83.02%
g. in-class-group-work	**22.22%**	**60.38%**
h. create conceptual designs	**31.48%**	**69.81%**
i. analyze presented information	64.81%	67.92%
j. communicate your creations to neighbor students	14.81%	30.19%
k. communicate your creations to the entire class	29.63%	41.51%
l. critique	12.96%	24.53%
m. evaluate	33.33%	39.62%
n. discover	57.41%	66.04%
o. real-life example	72.22%	73.58%
p. contemporary issues	31.48%	41.51%

The difference in the two ratings, collected at the beginning and end of this 90 minute session, are very significant. While at the beginning of the session, only 22% respondents considered in-class-group-work as the most important attribute of lectures, 60% respondents rated this attribute as most important towards the end of workshop. Similarly, the fraction of the respondents who rated in-class conceptual design as one of the most important attributes also increased from 31% to 70%. Significant enhancement in favor of other attributes of problem solving, creative thinking, on-the-spot seeking the clarifications, communicate with the neighbor, communicate to entire class, critique, discover, and contemporary issues can also be seen. No theories of education, pedagogy, or communication were discussed in this very short duration workshop of 90 minutes. With the help of this reflective workshop, a significant change in faculty's thinking was measured. This experiment shows that properly designed reflective engagements can be highly effective for changing the attitude, beliefs, and/or values.

Section 9.4: Chapter Summary

In this chapter, we have discussed several instructional interventions tried by us. All these interventions were administered in a chosen set of existing computing courses. Some new courses have also been developed in the process. Inquiry teaching has been tried out in some core courses. It was found that many students are not able to change their earlier learning habits, and hence, could not experience the advantages of deeper learning using this technique. Hence, in order to develop inquiry learning habit, puzzle solving has been integrated as the first component of the introductory programming course. Initial results are very encouraging. Future research is required for its impact analysis, and also to investigate the applicability of inquiry teaching in the context of different computing courses. Both forms of project-inclusive teaching have been adapted in many computing courses. More systematic studies are required to validate the effectiveness of the model in the context of specific computing courses.

A new graphic notation for modeling the software problems has been developed and infused in some courses. In order to develop estimation skills, the process data as adapted from PSP has been infused in many laboratory courses. In order to infuse debugging experience, taxonomy of software bugs has been prepared with an objective of designing debugging related assignments in various computing courses. In collaboration with various faculty members, we continue to strengthen the infusion of eight elements: web technology, multimedia technology, mobile technology, security aspects, systems design aspects, estimation aspects, open source, and debugging in various introductory core computing courses. This is bringing deeper integrated learning, higher levels of enthusiasm, and challenge in the courses.

Further, some new courses have been designed to strengthen the integrative thinking. Some of these courses make an attempt to integrate several computing areas, while some other make an attempt to integrate computing content with human sciences. A new form of collaborative learning have been proposed and tried out. A novel approach of collaborative pair and quadruple programming has been proposed. A novel form of collaborative learning, cross-level peer mentoring, has been evolved, tested, and scaled up. The results of sample tests were found to very encouraging.

We discussed our experience in conducting reflective workshops on pedagogy for engineering faculty. A significant shift in faculty's beliefs about the active and collaborative learning was noticed. More work needs to done in designing teachers' training programs on pedagogy. We intend to use our framework to design many such workshops to motivate the teachers to use aspects of our framework in their teaching.

We also discussed the impact of many of these interventions in terms of feedback from students and alumni, and our experience in conducting a faculty development program. All these interventions are manifestations of some aspect(s) of the framework proposed by us in previous chapter. It may be noted that most of these instructional interventions were developed, refined, and administered during the course of this study before the development of the final framework proposed in the previous chapter. *Our experiences with all these interventions have helped a great deal in formulating the thought process for development of the framework.*

CHAPTER 10: SUMMARY AND FUTURE SCOPE OF WORK

Summary

In this study, we have proposed a three-tier taxonomy of twelve competencies for software development education. It includes five basic competencies, three 'competency driver-habits of mind,' and four 'competency conditioning attitudes and perspectives.' The five basic competencies are: (i) technical competence, (ii) communication competence, (iii) domain competence, (iv) complex problem solving competence, and (v) computational thinking competence. The three 'competency driver-habits of mind' are: (i) attention to details, (ii) critical and reflective thinking, and (iii) creativity and innovation. The 'competency conditioning attitudes and perspectives' include: (i) intrinsic motivation to create/improve artifacts, (ii) curiosity, (iii) decision making perspective, and (iv) systems-level perspective.

We have reviewed the educational research literature to examine its applicability for developing these competencies through appropriate interventions for instructional reform. We have done many empirical (qualitative and quantitative) studies among students, faculty, and professionals, to find out the preferred approaches of learning and effective pedagogical techniques. Our empirical studies suggest that didactic approaches of teaching are ineffective. Students experience much deeper learning in active, integrative, reflective, and collaborative constructive environment.

Hence, we have proposed a comprehensive unified framework of pedagogic engagements. *Our proposed framework of pedagogic engagements in software development education is grounded in* (a) core activities of software development, and (b) distinguishing characteristics of software development profession. It includes - (i) three-tier taxonomy of twelve core competencies, (ii) five-dimensional ladder of professional and human development, (iii) three-dimensional perspective of the knowledge domain of software development, (iv) two core principles (cognitive dissonance and cognitive flexibility) for facilitating deep learning, and (v) a four-dimensional taxonomy of pedagogic engagements (active, integrative, reflective, and collaborative) over (iii) for developing (i) and (ii).

We have also discussed some instructional interventions developed by us, manifesting some aspects of our framework. These interventions were administered in a chosen set of existing computing courses. Some new courses have also been developed in the process. The development of the framework of pedagogic engagement, and these interventions for instructional reform of software development education, has been an intertwined and highly spiral process. Large classes offer a huge challenge. There is a need to explore the possibility of a complete revamp of the software development education and curriculum through our framework. While some interventions have been successfully tested with large classes, others were not as successful for large numbers. For example, the use of inquiry teaching in lecture classes offers huge benefits to learning oriented students, it has not been found to be as attractive to exam oriented students.

Future Scope of Work

We have discussed our experience in conducting reflective workshops on pedagogy for engineering faculty. More work needs to done in designing teachers' training programs on pedagogy. We intend to use our framework to design many such workshops to motivate the teachers to use aspects of our framework in their teaching [402].

We hope that our proposed framework of pedagogic engagement in software development education will help the community of software development educators and researchers to create a variety of interventions that will help in extending the 'Software Engineering Body of Knowledge' (SWEBOK) to 'Software Development Education Body of Knowledge' (SDEBOK).

The curriculum, syllabus, and textbooks often ignore many professional as well as pedagogical aspects. Our proposed framework of pedagogic engagements of software development education, offers the potential to redesign the instructional material for all computing courses. Systematic projects can be initiated in this direction.

Reflection has been found to be a highly effective pedagogical engagement. However, its use in computing courses is not very popular. Future work is required to systematically incorporate this aspect in student assignments in all computing courses and projects.

Systems-level perspective is one of most important competencies for software developers. The development of system-level perspective depends upon students' engagement with a curriculum and courses that are themselves designed with this perspective. *The curriculum as well all courses need be redesigned as systems*, where not only the computing course, but also the other courses, offered by other departments for computing students, will also be well integrated into a single whole. Our approach of multi-level infusion offers a way out. This will also help in increasing *domain sensitivity* and expertise of computing students.

Project centric evolutionary teaching offers active, integrative, reflective, as well as collaborative engagements as per our frame work. Future work is required for using this approach in different computing courses.

A novel approach of *collaborative pair and quadruple programming* has been proposed. The results of sample tests were found to very encouraging. Further work is required to examine the impact, and investigate ways of pervasively integrating it into all computing courses. More research is required to create different types of collaboration models in the context of different computing courses and projects.

Multi-level infusion opens a new way of transforming the computing courses. In collaboration with various faculty members, we continue to strengthen the infusion of eight elements: web technology, multimedia technology, mobile technology, security aspects, systems design aspects, estimation aspects, open source, and debugging in various introductory core computing courses. This is bringing deeper integrated learning, higher levels of enthusiasm, and challenge in the courses. We have also started working towards designing appropriate models for multi-level infusion of selected elements of software documentation, software quality, software risks managements, advanced level programming techniques, and formal methods of software

engineering. More work is required to develop detailed instructional material using this approach.

Cross-level mentoring has been found to highly effective wholesome engagement for senior students. More work is required to integrate this approach within the educational systems. Many new ways of forging collaborations between senior and junior level students need to be invented to create a collaborative community of co-learners.

We also believe that the proposed framework and our research approach are fairly comprehensive, reusable, and robust. Designers of *educational programs for other professions* can also adapt this framework and methodology.

More research is needed in developing new models and exemplars for offering multi-dimensional engagement to the users of online education and e-learning programs [403-408]. *Our framework of pedagogic engagements can be suitably adapted to create a framework of pedagogic engagements in e-learning and online environments.*

REFERENCES

[1].The Joint Task Force on Computing Curricula, IEEE Computer Society and ACM, Characteristics of CS graduates, Computing curricula, 2001, retrieved from http://www.computer.org/portal/cms_docs_ieeecs/ieeecs/education/cc2001/cc2001.pdf, last accessed on October 15, 2005.

[2]. Jalote P., The success of the SPI efforts in India, Software Quality Professional, Vol 3, No. 2, March 2001, retrieved from http://www.cse.iitk.ac.in/users/jalote/papers/IndiaSPI.pdf, last accessed on October 15, 2005.

[3] Task Force on Meeting the Human Resource Challenge for IT and IT enabled Services, Report and recommendations, Ministry of Communication and Information Technology, Government of India, pp 15, 2003,

[4] The Times of India, Learn to work, Editorial, India, June 23, 2005.

[5] NASSCOM-KPMG, Strengthening of HR for the IT services and ITES sector, p. 38, 2003.

[6] Wilkinson J., Re-engineering competency-based education through the use of a multimedia CD-ROM: A matter of life or death, Industry and Higher Education, IP Publishing Ltd., Volume 16, Number 4, pp. 261-265, August 1, 2002, retrieved from http://www.ingentaconnect.com/content/ip/ihe/2002/00000016/00000004/art00008, last accessed on October 15, 2005.

[7] Stephen, W. D., National and Global Imperatives in Engineering Education, Australasian Journal of Engineering Education, Vol. 7, No. 1, , 1996, retrieved from http://elecpress.monash.edu.au/ajee/vol7no1/director.htm, last accessed on Jan 5, 2006.

[8] Bullen F., Waters D., Bullen M. and de la Barra B. L., Incorporating and developing graduate attributes via program design, 15th Annual AAEE Conference, pp 29-39, 2004.

[9] Felder R. M., Does engineering education have anything to do with either one: Toward a systems approach to training engineers. R.J. Reynolds Industries Award Distinguished Lecture Series, North Carolina State University, 1982, retrieved from http://www.ncsu.edu/felder-public/Papers/RJR%20Monograph.pdf, pp 6, last accessed on October 16, 2005.

[10] Brown A. and Rudolph H., Educating engineers for the 21st century, Proceedings of 15th Annual AAEE Conference, pp 106-113, 2004.

[11] Sanjay Goel, What is high about higher education: Examining engineering education through Bloom's taxonomy, The National Teaching & Learning Forum, Vol. 13, pp 1-5, Number 4, 2004.

[12] Sanjay Goel and Nalin Sharda, What do engineers want? Examining engineering education through Bloom's taxonomy, Proceedings of 15th Annual AAEE Conference, pp173-185, 2004.

[13] R. M. Felder & R. Brent, The intellectual development of science and engineering students Part 1: Models and challenges, Journal of Engineering Education, USA, 93 (4), pp 269-277, 2004.

[14] Rapaport W. J., William Perry's scheme of intellectual and ethical development, 2004, retrieved from http://www.cse.buffalo.edu/~rapaport/perry.positions.HTML, last accessed on October 27, 2005.

[15] National Academy of Engineers, Educating the engineer of 2020: Adapting engineering education to the new century The National Academies Press, 2005, retrieved from http://books.nap.edu/catalog/11338.HTML, last accessed on October 18, 2005.

[16] Paulsen M.B., Peseau B.A. A practical guide to Zero Based Curriculum Review, Innovative Higher Education, Vol. 16, No. 3, Human Science Press, Inc., pp 211- 221.

[17] Woods D.R., Felder R.M., Rugarcia A. & Stice J.E., The Future of engineering education. III. Developing Critical Skills, Chem. Engr. Education, 34(2), pp 108-117, , 2000retrieved from http://www.ncsu.edu/felder-public/Papers/Quartet3.pdf, last accessed on October 16, 2005.

[18] The Steering Committee of the National Engineering Education Research Colloquies, "The national engineering education research colloquies," Journal of Engineering Education, Vol 95, No 4, 257-261, Oct. 2006.

[19] Michael C.Mulder, A Recommended Curriculum in Computer Science and Engineering, Computer. IEEE Computer Society, pp 72-75, December 1977.

[20] O. E. Dunn, Information technology a management problem, DAC '66: Proceedings of the SHARE design automation project, ACM, January 1966.

[21] ACM Curriculum Committee on Computer Science, An Undergraduate Program in Computer Science – Preliminary Recommendations, Communications of the ACM, pp 543-552, September 1965.

[22] ACM Curriculum Committee on Computer Science, Curriculum 68, Communications of the ACM, pp 151-197, March 1968.

[23] COSINE Committee of the Commission on Engineering Education, Computer science in electrical engineering, IEEE Spectrum, pp 96-103, March 1968.

[24] Michael C.Mulder Model Curricula for Four-Year Computer Science and Engineering Programs: Bridging the Tar Pit, Computer, IEEE Computer Society, pp 28-33, December 1975.

[25] M.E. Sloan, Evaluation of the Model Curriculum in Computer Science and Engineering, ComputerIEEE Computer society, pp 114-120, December 1977,.

[26] Engel, Gerald L, A Comparison of the ACM-C3S and the IEEE/CSE Model Curriculum Subcommittee Recommendations, Computer, IEEE Computer society, pp 121-123, December 1977.

[27] Daniel Teichroew, Education related to the use of computers in organizations, Communications of the ACM, pp 573-588, September 1971.

[28] R.L. Ashenhurst, A Report of the ACM Curriculum Committee on Computer Education for Management, Communications of the ACM, pp 363-398, May 1972.

[29] F.W. McFarlan and R.L. Nolan, M. Shaw (Ed), Curriculum Recommendations for Graduate Professional Programs in Information Systems: Recommended Addendum on Information Systems Administration, Communications of the ACM, pp 439-441, July 1973.

[30] K.A.Duncan, R.H. Austing, S. Katz, R.E. Pengov, R.E. Pogue, and A.I. Wasserman, Health Computing: Curriculum for an emerging profession, Proceedings of the 1978 annual conference, pp 277-288, December 1978.

[31] Kenneth I. Magel, Richard H. Austing, Alfs Berztiss, Gerald L. Engel, John W. Hamblen, A. A.J. Hoffmann, Robert Mathis, Recommendations for master's level programs in computer science: A Report of the ACM Curriculum Committee on Computer Science, Communications of the ACM, pp 115-123, March 1981.

[32] Jay F. Nunamaker, Educational Programs in Information Systems: a report of the ACM Curriculum Committee on Information Systms, Communications of the ACM, pp 124-133 March 1981.

[33] Jay F. Nunamaker, J. Daniel Couger, and Gordon B. Davis, Information systems Curriculum Recommendations for the 80s: Undergraduate and Graduate Programs: a report of the ACM Curriculum Committee on Information Systms, Communications of the ACM, pp 781-805. , November 1982

[34] Normal E. Gibbs and Allen B. Tucker, A Model Curriculum for a Liberal Arts Degree in Computer Science, Communications of the ACM, pp 202-210, March 1986.

[35] Henry M. Walker and G. Michael Schneider, A Revised Model Curriculum for a Liberal Arts Degree in Computer Science, Communications of the ACM, pp 85-95, December 1996.

[36] Liberal Arts Computer Science Consortium, A 2007 Model Curriculum for a Liberal Arts Degree in Computer Science, Journal on Educational Resources in Computing (JERIC), ACM, pp 1-34, June 2007.

[37] Association for Computing Machinery (ACM), Association for Information Systems (AIS), and The Computer Society (IEEE-CS), Computing Curricula 2005, retrieved from http://www.acm.org/education/curric_vols/CC2005-March06Final.pdf, last accessed on February 28, 2010.

[38] Anthony Ralston and Mary Shaw, Curriculum '78 – Is Computer Science Really that Unmathematical?, Communications of the ACM, pp 67-70, February 1980.

[39] Mary Shaw (ed.), The Carnegie-Mellon Curriculum for Undergraduate Computer Science, Springer-Verlag, New York, 1985.

[40] Alfs Berztiss, A Mathematically Focused Curriculum for Computer Science, Communications of the ACM, pp 356-265, May 1987.

[41] Anthony Ralston, The First Course in Computer Science Needs a Mathematical Corequisite, Communications of the ACM, pp 1002-1005, October 1984.

[42] Allen B. Tucker, Charles F. Kelemen and Kim B. Bruce, Our Curriculum Has Become Math-Phobic!, ACM SIGCSE Bulletin, pp 243-247, March 2001.

[43] Sukhen Dey and Lawrence R. Mand, Current Trends in Computer Science Curriculum: A Survey of Four-Year Program, Technical Symposium on Computer Science Education, Proceedings of the twenty-third SIGCSE technical symposium on Computer science education, Kansas City, Missouri, United States, ACM, pp 9-14, 1992.

[44] All India Council for Technical Education, Model Curriculum for Undergraduate Programme B.E./ B. Tech. in COMPUTER SCIENCE & ENGINEERING, 2000, retrieved from http://www.aicte.ernet.in/download/OnlineBooks/compsciandEngg.pdf.

[45] All India Council for Technical Education, Model Curriculum for Undergraduate Programme B.E./ B. Tech. in INFORMATION TECHNOLOGY, 2000, retrieved from http://www.aicte.ernet.in/download/OnlineBooks/it.pdf.

[46] Timothy C. Lethbridge, The relevance of software education: A survey and some recommendations, Annals of Software Engineering, Springer Netherlands, pp 91-110, March, 1998.

[47] Timothy C. Lethbridge, A survey of the relevance of computer science and software engineering education, . Proceedings of 11th Conference on Software Engineering Education, IEEE, pp 56-66, 1998.

[48] Timothy C. Lethbridge, What knowledge is important to a software professional?, Computer, IEEE, pp 44-50, 2000.

[49] The Joint Task Force on Computing Curricula, Computing Curricula 2001, IEEE Computer Society and ACM, 2001, retrieved from http://www.computer.org/portal/cms_docs_ieeecs/ieeecs/education/cc2001/cc2001.pdf, last accessed on October 15, 2005.

[50] Interim Review Task Force, Computer Science Curriculum 2008: An Interim Revision of CS 2001 Report, December 2008, Association for Computing Machinery and IEEE Computer Society.

[51] The Joint Task Force on Computing Curricula, Computing Curricula 2005: The Overview Report, Association for Computing Machinery, Association for Information Systems, and IEEE Computer Society, September 2005.

[52] The Joint Task Force on Computing Curricula, IEEE Computer Society and ACM, Software Engineering 2004: Curriculum guidelines for undergraduate degree programs in software engineering, 2004, retrieved from http://sites.computer.org/ccse/SE2004Volume.pdf, last accessed on October 15, 2005.

[53] Karl M. Fant, Computer Science Reconsidered: The invocation models of process expression, John Wiley & Sons, USA, pp 1-10, 2007.

[54] Richard H. Austing, Bruce H. Bernes, Della T. Bonnette, Gerald L. Engel, Gordon Stokes, Curriculum'78: Recommendations for the Undergraduate Program in Computer Science, Communications of the ACMpp 147-166, March 1979,.

[55] UNESCO-IFIP, A Model Curriculum in Computer Science, UNESCO, 1994.

[56] Richard H. Austing, Bruce H. Bernes, and Gerald L. Engel, A Survey of the Literature in Computer Science Education Since Curriculum'68, Communications of the ACM, pp 13-21, January, 1977.

[57] Michael Goldweber, John Impagliazzo, Iouri A. Bogoiavlenski, A. G. Clear, Gordon Davies, Hans Flack, J. Paul Myers, Richard Rasala, Historical perspectives on the computing curriculum (report of the ITiCSE '97 working group on historical perspectives in computing education, Annual Joint Conference Integrating Technology into Computer Science Education, The supplemental proceedings of the conference on Integrating technology into computer science education: working group reports and supplemental proceedings, Uppsala, Sweden, ACM, pp 94-111, 1997.

[58] Mingrui Zhang, Eugene Lundak, Chi-Cheng Lin, Tim Gegg Harrison, Joan Francioni, Interdisciplinary Application Track in an Undergraduate Computer Science Curriculum, SIGCSE'07, ACM, pp 425-429, March 2007.

[59] Michael C. Mulder and John Dalphin, Computer Science Program Requirements and Accreditation, Communications of the ACM, pp 330-335, April 1984.

[60] J.T. Cain, Professional Accreditation for the Computing Sciences, Computer, IEEE, pp 91-96, Januray 1986.

[61] Peter J. Denning, Douglas E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young, Computing as a discipline, Communications of the ACM, pp 9-23, January 1989.

[62] A. Joe Turner, A Summary of the ACM/IEEE-CS Joint Curriculum task Force Report: Computing Curricula 1991, Communication of the ACM, pp 69-84, July 1991.

[63] Anthony I. Wasserman and Peter Freeman, Software Engineering Concpets and Computer Science Curricula, Computer, IEEE, pp 85-91, June 1977.

[64] E.W. Dijkastra, David L. Parnas, W.L. Sherlis, M.H. van Emden, Jacques Cohen, R.W. Hamming, Richard M. Karp, and Terry Winnograd, Peter J. Denning (ed.), A Debate On Teaching Computing Science, Communications of the ACM, pp 1397-1414, December 1989.

[65] D.E. Conway, S.C. Dunn, and G.S. Hooper, BCS and IEE accreditation of software engineering courses, Software Engineering Journal, IEE, pp 245-248, July 1989.

[66] Gary Ford. The SEI Undergraduate Curriculum in Software Engineering, ACM SIGCSE Bulletin, March 1991, pp 375-385

[66a] Gary Ford, The Progress of Undergraduate Software Engineering Education, ACM SIGCSE Bulletin, pp51-58, December 1994.

[67] Thomas B. Hilburn, Iraj Hirmanpour, Soheil Khajenoori, Richard Turner, Abir Qasem, A Software Engineering Body of Knowledge Version 1.0, SEI, CMU, April 1999.

[68] Alain Abran, James W. Moore, Guide to the Software Engineering Body of Knowledge, IEEE, 2004.

[69] Tony Clear, Coupling and Cohesion Among Disciplines: Some Curriculum Paradigms, ACM SIGCSE Bulletinpp 14-16, December 1997,.

[70] Derek Cheung and Pun-Hon Ng, Science Teachers' Beliefs about Curriculum Design, Research in Science Education, , Springer Netherlands, pp 357-375, December 2000.

[71] Greg Scragg, Doug Baldwin, and Hans Koomen, Computer Science Needs an Insight-Based Curriculum, ACM SIGCSE Bulletin, pp 150-154, March 1994.

[72] Judith L. Gersting and Frank H. Young, Content + Experience = Curriculum, SIGCSE, ACM, pp 325-329, March 1997.

[73] Robert N. Carson, Why Science Education Alone is Not Enough, Interchange, Kluwer Academic Publishers, Netherlands, pp 109-120, April 1997.

[74] Faith Clarke, Han Reichgelt: The importance of explicitly stating educational objectives in computer science curricula. ACM SIGCSE Bulletin, 47-50, December 2003.

[75] Vaidyeswaran Rajaraman, Undergraduate Computer Science and Engineering Curriculum in India, IEEE Transactions on Education, pp 172-177, February, 1993.

[76] Aning, A. O., Lohani, V.K., Griffin H. Kampe, J.C. M., Aref, H., An Interdisciplinary Graduate Program in Engineering Education, iCEER, 2005, http://www.iaalab.ncku.edu.tw/iceer2005/Form/PaperFile/08-0016.pdf.

[77] Holmboe, C. et al , Research Agenda for Computer Science Education, In Proc. PPIG 13, G. Kadoda (Ed). Bournemouth UK, 2001, pp 207-223, , October, 2006, retrieved from http://www.ppig.org/papers/13th-holmboe.pdf.

[78] Passow Honor J., What Competencies Should Undergraduate Engineering Programs Emphasize? A Dilemma of Curricular Design that Practitioners' Opinions Can Inform, PhD Thesis, University of Michigan, 2008, retrieved from http://deepblue.lib.umich.edu/bitstream/2027.42/60691/1/hpassow_1.pdf, last accessed on December 25, 2008.

[78a] C. Bodmer, A. Leu, L. Mira & H. Rütter, SPINE: Successful practices in international engineering education, pp 92-102, 2002, retrieved from http://www.ingch.ch/pdfs/spinereport.pdf, last accessed on October 14, 2005.

[79] Bordogna J., Making Connections: The role of engineers and engineering education, The Bridge, Volume 27, Number 1 - spring 1997, retrieved from http://www.nae.edu/nae/bridgecom.nsf/weblinks/NAEW-4NHMPY?OpenDocument, last accessed on October 15, 2005.

[80] Dodridge M., Convergence of engineering higher education - Bologna and Beyond, Proceedings of the Ibero-American Summit on Engineering Education, 2003, retrieved from http://www.univap.br/iasee/anais/trabalhos/Dodridge-Convergence%20of%20Engineering%20Higher%20Education1.pdf, last accessed on October 14, 2005.

[81] Mason G., Engineering skills formation in Britain: Cyclical and structural issues, 1999, pp 9, retrieved from http://www.etechb.co.uk/reslib/Engineering%20Skills%20Formation%20in%20Britain%20-%20Cyclicale%20and%20Structural%20Issues.doc, last accessed on October 15, 2005.

[82] Hoscette J., Leading causes of failures in engineers - Career development, 2002, http://www.hi.is/~joner/eaps/es_1023f.htm, last accessed on October 15, 2005.

[83] Erlendsson J., Engineering graduates: Desirable characteristics, 2001, retrieved from http://www.hi.is/~joner/eaps/ds_chare.htm, last accessed on October 15, 2005.

[84] Sanjay Goel, Investigations on required core competencies for engineering graduates with reference to Indian IT industry, European Journal of Engineering Education, Taylor & Francis, UK, pp 607-617, October, 2006.

[85] Bailey, J. L. and Stefaniak, G., Preparing the information technology workforce for the new millennium, ACM SIGCPR Computer Personnel, Volume 20 , Issue 4, pp 4-15, 2002.

[86] Domelen, D. V., Problem-Solving Strategies: Mapping and Prescriptive Methods, Thesis, 1996, retrieved from http://www.physics.ohio-state.edu/~dvandom/Edu/thesis.HTML, last accessed on Jan 4, 2005.

[87] Gary, Krahn, "Inter-disciplinary Culture - a Result not a Goal", Proceedings of the Inter-disciplinary Workshop on Core Mathematics: Considering Change in the First Two Years of Undergraduate Mathematics, West Point, NY, 1999, Retrieved from http://www.dean.usma.edu/math/activities/ilap/workshops/1999/files/krahn.pdf.

[88] Bruner, J., The Culture of Education, Harvard University Press, Cambridge, MA, 1996.

[89] NASSCOM-KPMG, Task Force on Meeting the Human Resource Challenge for IT and IT enabled Services, 2003.

[90] ABET (Accreditation Board for Engineering and Technology), Criteria for accrediting engineering programs: Effective for evaluations during the 2005–2006 accreditation cycle, pp 2 & 3, 2004, retrieved from http://www.abet.org/Linked Documents-UPDATE/Criteria and PP/05-06-EAC Criteria.pdf.

[91] ABET (Accreditation Board for Engineering and Technology), Criteria for Accrediting Engineering Technology Programs: Effective for Evaluations during the 2005-2006 Accreditation Cycle, pp 5-7, 2004, retrieved from http://www.abet.org/Linked Documents-UPDATE/Criteria and PP/05-06-TAC Criteria.pdf, last accessed on October 31, 2005.

[92] ABET (Accreditation Board for Engineering and Technology), Criteria for Accrediting Computing Programs: Effective for Evaluations during the 2005-2006 Accreditation Cycle, pp 20, 2004, retrieved from http://www.abet.org/Linked Documents-UPDATE/Criteria and PP/05-06-CAC Criteria.pdf, last accessed on October 31, 2005.

[93] Engineering Council UK , UK Standard for Professional Engineering Competence, Chartered Engineer and Incorporated Engineer Standard, pp 5-11, 2003, retrieved from http://www.engc.org.uk/publications/pdf/ukspec_CE_IE_Standard.pdf, last accessed on Jan 5, 2006.

[94] Institution of Engineers, Singapore (IES), Engineering Accreditation Board: Accreditation Manual, pp 13-142004, retrieved from http://www.ies.org.sg/eab/accr_man.pdf, last accessed on Jan 5, 2006.

[95] Engineers Australia Accreditation Board, Engineers Australia policy on accreditation of professional engineering programs, pp 3-5, 2005, retrieved from http://www.ieaust.org.au/membership/res/downloads/P020%20Engineers%20Australia%20Policy%20on%20Accreditation%20of%20Professional%20Engineering%20Programs.pdf, last accessed on October 14, 2005.

[96] JABEE (Japan Accreditation Board for Engineering Education), Criteria for accrediting Japanese engineering, pp 1-15, , 2004, retrieved from http://www.jabee.org/english/OpenHomePage/e_criteria2004-2005(2).pdf, last accessed on October 15, 2005.

[97] Bell, T. E., Proven skills: the new yardstick for schools, IEEE Spectrum, September 2000, pp 63-67

[97a] Felder, R. M. and Brent R., Designing and Teaching Courses to Satisfy the ABET Engineering Criteria, Journal of Engineering Education, pp 7-25, January, 2003.

[98] Turner C. D. & Li W. Martinez A., Developing sustainable engineering across a college of engineering, Proceedings of American Society for Engineering Education Annual Conference & Exposition, 2001, retrieved from www.utep.edu/green/papers/asee2001.pdf, last accessed on October 11, 2005.

[99] Bigio D. & Schmidt J., A workshop of faculty development based on the underlying pedagogical issues of ABET EC 2000, 29th ASEE/IEEE Frontiers in Education Conference, pp 12a1:5-9, , 1999, retrieved from http://fie.engrng.pitt.edu/fie99/papers/1039.pdf, last accessed on October 14, 2005.

[100] Campbell D., Bunker J., Hoffman K. & Iyer R M., Processes in distilling course capability profiles, 15th Annual AAEE Conference, pp 57-67, , 2004.

[100a] Senini S. & Nouwens F., A design framework for developing technical competence professional skills and identity, 15th Annual AAEE Conference, 15th Annual AAEE Conference, 2004, pp 47-56, , September 2004.

[101] The Times of India, 9 Indians adorn MIT's top 100 innovators list, November 8, 2004, retrieved from http://timesofindia.indiatimes.com/articleshow/msid-916533,curpg-2,fright-0,right-0.cms, last accessed on October 11, 2005.

[102] Arya S. P., email posting, JIIT-placement e-group, 2005, retrieved from http://groups.yahoo.com/group/placement-jiit/message/957, last accessed on October 15, 2005.

[103] ValueNotes, R&D Outsourcing – The India edge: Key insights and success factors, Aug 2004, retrieved from http://www.researchandmarkets.com/reportinfo.asp?report_id=224141&t=e&cat_id=2, last accessed on October 15, 2005). This paradox needs some deeper analysis.

[104] National Board of Accreditation (NBA), AICTE, Accreditation parameter: Criteria and Weightages, pp 6-7, 2000, retrieved from http://www.nba-aicte.ernet.in/nba-aicte/accre/acc_8.pdf, last accessed on Jan 6, 2006.

[105] The Engineering Professors Council, The EPC engineering graduate output standards, EPC Occasional Paper, pp 7-8, Number 10, 2000.

[106] National Academy of Engineers, The engineer of 2020: Visions of engineering in the new century, The National Academies Press, pp 53-572005, retrieved from http://books.nap.edu/catalog/10999.HTML, last accessed on October 19, 2005.

[107] Rugarcia A., Felder R.M.,Woods D.R. & Stice J.E, The future of engineering education-I: A vision for a new century. Chem. Engr. Education, 34(1), pp 16-25, , 2000, retrieved from http://www.ncsu.edu/felder-public/Papers/Quartet1.pdf, last accessed on October 16, 2005.

[108] Cabrera A. F., Colbeck C. L., Terenzini P. T. Developing performance indicators for assessing classroom teaching practices and student learning: The case of Engineering, Research in Higher Education, Vol. 42, No. 3, Springer, pp 327-352, 2001.

[110] Stark, J. S., Lowther, M. A., & Hagerty, B. M. K., Faculty perceptions of professional preparation environments: Testing a conceptual framework, The Journal of Higher Education, 58(5), pp 530-561, 1987.

[111] The Joint Task Force on Computing Curricula, IEEE Computer Society and ACM, Curriculum guidelines for undergraduate degree programs in computer engineering, final report, pp 7, 2004, retrieved from http://www.acm.org/education/CE-Final Report.pdf, last accessed on October 15, 2005.

[112] The Joint Task Force on Computing Curricula, IEEE Computer Society and ACM, Characteristics of IT graduates, computing curricula: Information Technology Volume, Draft, pp 38-40, April 2005, retrieved from http://www.acm.org/education/IT_2005.pdf, last accessed on October 15, 2005.

[113] Association for Computing Machinery (ACM), Association for Information Systems (AIS), and Association of Information Technology Professionals (AITP), Model Curriculum and Guidelines for Undergraduate Degree Programs in Information Systems, , pp 14, 2004retrieved from http://www.acm.org/education/is2002.pdf, last accessed on Jan 6, 2006.

[114] Task Force on Meeting the Human Resource Challenge for IT and IT enabled Services, Report and Recommendations, Ministry of Communication and Information Technology, Government of India, 2003.

[115] Patrao M., Testing tenacity of IT students, DH Education, April 28, 2005, retrieved from http://www.deccanherald.com/deccanherald/apr282005/dheducation2022552005426.asp, last accessed on October 15, 2005.

[116] Chang I. F., Challenges to engineering education in the 21st century, 1998, http://www.hi.is/~joner/eaps/wh_enedx.htm, last accessed on October 15, 2005.

[117] Erlendsson J. Systemic engineering education reform, 2005, retrieved from http://www.hi.is/~joner/eaps/wh_enedx.htm, last accessed on October 29, 2005.

[118] Kelley R. and Caplan J., How Bell Labs creates star performers. Harvard Business Reviewpp 128-139, July-August 1993,.

[119] Turley Richard T. and Bieman James M., Competencies of Exceptional and Non-Exceptional Software Engineers, Journal of Systems and Software, 28(1):19-38, January 1995, Retrieved from http://www.cs.colostate.edu/~bieman/Pubs/turleyBiemanJSS95.pdf, last accessed on December 29, 2008.

[120] Philip G. Armour, The case for a new Business Model: Is software a product or a medium, Communications of ACM, USA, August, Vol. 43 No.8, pp 19-22, 2000.

[121] Connor H., Dench S. & Bates P., Skills dialogue: An assessment of skill needs in engineering, Department for Education and Employment, UK, 2002 retrieved from http://66.102.7.104/search?q=cache:IY7X0iPjfXIJ:www.qub.ac.uk/nierc/documents/Rwp60b.pdf+%22skill+deficiency%22+Engineering+Skills+Formation+in+Britain&hl=en, last accessed on October 15, 2005.

[122] Extreme Programming Explained, Ken Beck and Cynthia Andres, Addison Wesley, 2004.

[123] James Shore and Shane Warden, The Art of Agile Development O'Reilly Media Inc, Shroff Publishers and Distributors Pvt. Ltd., pp 354, 2008.

[124] Hazzan, O. and Tomayko, J., Reflection and abstraction processes in the learning of the human aspects of Software Engineering, IEEE Computer, pp. 39-45, June 2005.

[125] Schön Donald A., Educating the Reflective Practitioner: Toward a new Design for Teaching and Learning in the Professions, Jossey Bass Publisher. 1987.

[126] Sodiya A.s., Longe H.O.D., Onashoga S.A., and Awodele O., An improved assessment of personality traits in Software Engineering, Inter-disciplinary Journal of Information, Knowledge, and Management, Vol 2, Informing Science Institute, USA, pp 163-177, 2007.

[127] Bass Len, Clement Paul, Kazman Rick, and Klein Mark, Models for Evaluating and Improving Architecture Competence, Technical Report – CMU/SEI-2008-TR-006, Software Engineering Institute, Carnegie Mellon University, 2008.

[128] John Henry Newman, The Idea of a University Defined and Illustrated, Regnery Publishing, USA, 1999.

[129] Marrice Kogan and Stephen Hanney, Reforming Higher Education, Jessica Kinglsey Publishers Limited, UK, 2000.

[130] Martha Nussbaum, Education for Citizenship in an era of Global Connection, Journal of Studies in Philosophy and Education, Springer Netherlands, pp 289-303, July 2002.

[131] The National Leadership Council for Liberal Education & America's Promise, College Learning for the New Global Century, American Association of College and University, 2007, retrieved from http://www.aacu.org/leap/documents/GlobalCentury_final.pdf.

[132] Adela García-Aracil and Rolf Van der Velden, Competencies for young European higher education graduates: labor market mismatches and their payoffs, Journal of Higher Education, Springer Netherlands, pp 219-239, February 2008.

[133] Bloom Benjamin S. and David R. Krathwohl, Taxonomy of Educational Objectives: The Classification of Educational Goals, by a committee of college and university examiners. Handbook I: Cognitive Domain, New York, Longmans, 1956.

[134] Anderson, L., & Krathwohl, D. E., A Taxonomy for learning teaching and assessing: A revision of Bloom's taxonomy of educational objectives [Abridged]. New York: Addison Wesley Longman, Inc., 2001.

[135] A.L. Costa, Developing Mind: A resource book for teaching thinking, Association for Supervision & Curriculum Development; 3rd edition, December 2001.

[136] Kennedy, M. M., Inexact sciences: Professional education and the development of expertise. Review of Educational Research, Vol. 14, pp 133-167, 1987.

[137] Stark, Joan and Malcolm A. Lowther. Exploring Common Ground in Liberal and Professional Education, Armount, R. A. and B. S. Fuhrmann (eds.) Integrating Liberal Learning and Professional Education. New Directions for Teaching and Learning, No. 40, Winter. San Francisco: Jossey-Bass, pp 7-20, 1989.

[138] Marzano R. J., Pickering D. & McTighe J. Introduction, assessing student outcomes: Performance assessment using the dimensions of learning model, Association for Supervision and Curriculum Development (ASCD), 1993, retrieved from http://www.ascd.org/portal/site/ascd/template.chapter/menuitem.b71d101a2f7c208cdeb3ffdb62108a0c/?chapterMgmtId=a740a2948ecaff00VgnVCM1000003d01a8c0RCRD, last accessed on Oct 26, 2005.

[139] Sanjay Goel, Competency Focused Engineering Education with Reference to IT Related Disciplines: Is Indian System Ready for Transformation? Journal of Information Technology Education, Vol. 5, Informing Science Institute, USA, pp 27-52, 2006.

[140] Marzano, R. J., Designing a new taxonomy of educational objectives, Thousand Oaks, CA: Corwin Press, 2000.

[141] Kelly Coate, Curriculum, In Tight Malcolm, Ka Ho Mok, Jeroen Huisman, Christopher C. Morphew (Ed.), The Rutledge International Handbook of Higher Education, Routledge, USA, , pp 77-90, 2009.

[141a] A.H. Maslow, A Theory of Human Motivation, Psychological Review 50(4), pp 370-396, 1943, retrieved from http://www.salesjobs.ie/artman/uploads/theory_of_human_motivation_001.pdf.

[142] Philip G. Armour, Twenty Percent, Planning to fail on software projects, Communications of the ACM, pp 21-23, June 2007.

[142a] The Institute of Electrical and Electronics Engineers Inc., IEEE Standard Classification for Software Anomalies. New York, USA: IEEE Computer Society. 1993.

[143] James Miller, Triangulation as a basis got knowledge discovery in software engineering, Journal of Empirical Software Engineering, Springer, pp 223-228, February 2008.

[144] Whitehead Jim, Collaboration in Software Engineering: A Roadmap, Future of Software Engineering, (FOSE'07), IEEE Computer Society, pp 214-225, May 2007.

[145] Tiago Maurao Teixeria, Web collaboration for software engineering, MSc. Thesis, Universidade do Porto, Portugal, 2009.

[146] Whitworth Elizabeth and Biddle Robert, The Social Nature of Agile Teams, Agile 2007, IEEE Computer Society, pp 26-36, August 2007.

[147] Sharp Helen and Robinson Hugh, Some Social Factors of Software Engineering: the maverick, community and technical practices, Proceedings of the 2005 workshop on Human and social factors of software engineering, International Conference on Software Engineering, ACM, pp 1-6, 2005.

[148] Philip G. Armour , The Five orders of Ignorance: Viewing software development as knowledge acquisition and ignorance reduction, Communications of ACM, USA, Vol. 43 No.10, pp 19-20, October 2000.

[149] Timothy C. Lethbridge, Susan Elliott Sim and Janice Singer, Studying Software Engineers: Data Collection Techniques for Software Field Studies, Empirical Software Engineering, Volume 10, Number 3, Springer Netherlands, pp 311-341, July, 2005.

[150] Per Runeson and Martin Höst, Guidelines for conducting and reporting case study research in software engineering, Journal of Empirical Software EngineeringSpringer Netherlands, pp 131-164, April 2009,.

[151] Charles P. Snow, The Two Cultures, Cambridge University Press, UK, 1998.

[152] Kolb, David, Experiential learning: Experience as the source of learning and development, Englewood Cliffs, NJ: Prentice-Hall, 1984.

[153] Robert J. Sternberg, Beyond IQ: a triarchic theory of human intelligence, Cambrdige Univesrity Press, 1985.

[154] Ned Herrmann, The Whole Brain Business Book, McGrawHill, 1996.

[155] Gardner, Howard. "Frames of Mind: The Theory of Multiple Intelligences." New York, Basic Books, 1983.

[156] Aaron Fried, Karen Zannini, Don Wheeler, Yongjin Lee, and Jose Cortez, Instructional Design Theory Database Project, Syracuse University, 2005, retrieved from http://web.cortland.edu/frieda/ID/IDdatabase.HTML.

[157] Metzger, R.C., Debugging by Thinking: A multi-disciplinary approach, Hewlett Packard Development Company and Elsevier Digital Press, 2004.

[158] Paul T. Ward and Stephen J. Mellor, Structured Development for Real-Time Systems, Prentice Hall Professional Technical Reference, 1991.

[159] Interim Review Task Force, Computer Science Curriculum 2008: An Interim Revision of CS 2001 Report Association for Computing Machinery and IEEE Computer Society, December 2008

[159a] Vikas Kumar and Sanjay Goel, Software Bug Taxonomy for Effective Programming, Unpublished, 2009.

[160] Braintrack.com, Application Software Engineer Job Description, http://www.braintrack.com/colleges-by-career#computing-and-mathematics.

[161] Yang Xiaohu, Xu Bin, He Zhijun, Extreme Programming in global software development, CCECE 2004-CCGEl2004, IEEE, Niagara Falls, pp 1845-1848, May 2004.

[162] Thomas Chau, Frank Maurer, Grigori Melnik, Knowledge Sharing: Agile Methods vs. Tayloristic Methods.

[163] Alistair Cockburn, Learning From Agile Software Development – Part One, CROSSTALK - The Journal of Defense Software Engineering, Software Technology Support Center, Department of Defense, U.S. Government, pp 10-14, October 2002.

[164] Henrik Munkebo Christiansen, Meeting the Challenge of Communication in Offshore Software Development, Software Engineering Approaches for Offshore and Outsourced Development, First International Conference,

SEAFOOD 2007, Zurich, Switzerland, February 5-6, 2007. Revised Papers, Lecture Notes in Computer Science, Springer, Berlin, pp-19-26.

[165] Jack D. Becker, Robert G. Insley, Megan L. Endres, Communication skills of technical professionals: a report for schools of business administration, ACM SIGCPR Computer Personnel , Volume 18, Issue 2, USA, pp 3-19, April 1997.

[166] Guihua Li, Shawna Long, and Mary Ellen Simpson, Self perceived gains in critical thinking and communication skills: Are there disciplinary differences, Research in Higher Education, Vol. 40, No. 1, Springer, pp 43-60, February 1999.

[167] Henry A. Etlinger, A Framework In Which To Teach (Technical) Communication to Computer Science Majors, ACM SIGCSE, Houston, Texas, USA, pp 122-126, March 1-5, 2006.

[168] Alberto Sillitti, Martina Ceschi, Barbara Russo, Giancarlo Succi, Managing Uncertainty in Requirements: a Survey in Documentation-driven and Agile Companies, 11th IEEE International Software Metrics Symposium (METRICS), IEEE, USA, pp 10-17,September 2005.

[169] Shirley Booth, Learning to program: A phenomenographic perspective, University of Gothenburg, Sweden, 1992.

[170] Ruth Neumann, Disciplinarity, In Tight Malcolm, Ka Ho Mok, Jeroen Huisman, Christopher C. Morphew (Ed.), The Rutledge International Handbook of Higher Education, Routledge, USA, pp 487-500, 2009.

[171] Yonghong Jade Xu, Faculty Turnover: Discipline-Specific Attention is Warranted, Res earsch in High Educ ation. Springer, Vol. 49, pp 40–61, Feb 2008.

[172] Matthew Kwok, Disciplinary Differences in the Development of Employability Skills of Recent University Graduates in Manitoba: Some Initial Findings. Higher Education Perspectives, volume 1, issue 1, pp.60-77, 2004.

[173] Biglan, A., The characteristics of subject matter in academic areas, Journal of Applied Psychology, 57, 195–203, 1973.

[173a] Malaney, G. D., Differentiation in graduate education, Research in Higher Education, 25(1), pp 82–96, 1986.

[174] Michael B. Paulsen and Charles T. Wells, Domain differences in the epistemological beliefs of college students, Research in Higher Education, Vol. 39, No. 4, Springer, pp 365-394, August 1998.

[175] Kolb Alice Y. and Kolb David A., The Kolb Learning Style - Inventory version 3.1: 2005 Technical Specifications, Experienced based learning Systems, 2005.

[176] Charles L. Isbell, Lynn Andrea Stein, Robb Cutler, Jeffrey Forbes, Linda Fraser, John Impagliazzo, Viera Proulx, Steve Russ, Richard Thomas, and Yan Xu, (Re)Defining Computing Curricula by (Re)Defining Computing, Inroad SIGCSE Bulletin, Vol. 41, Number 4, pp 195-207, December 2009.

[177] Sanjay Goel, Om Vikas, Mukul Sinha, Guidelines for Masters in Archaeo-heritage Informatics, Indo US S&T Workshop on Digital Archeology, Musoorie, India, Invited paper, Nov 11-13, 2005.

[178] Gerald Weinberg, Rethinking systems analysis and design, Dorset House Pub. Co., USA, 1988.

[179] Winslow, Programming Pedagogy -- A Psychological Overview SIGCSE BULLETIN Vol. 28 No. 3, ACM, USA, pp 17-25, Sept. 1996.

[180] Robert Kowalski, Algorithm = Logic + Control, Communications of the ACM, Volume 22, Issue 7, ACM, pp 424 – 436, July 1979.

[181] Muller and Haberman, A course dedicated to developing Algorithmic Problem Solving Skills – Design and Experiment, 21st Annual Psychology of Programming Interest Group Workshop (PPIG 2009), University of Limerick, Ireland, June 24-26, 2009, http://www.ppig.org/papers/21st-muller.pdf.

[182] Charles L. Isbell, Lynn Andrea Stein, Robb Cutler, Jeffrey Forbes, Linda Fraser, John Impagliazzo, Viera Proulx, Steve Russ, Richard Thomas, and Yan Xu, (Re)Defining Computing Curricula by (Re)Defining Computing, Inroad SIGCSE Bulletin, Vol. 41, Number 4, pp 195-207, December 2009.

[183] J.M. Wing, Computational Thinking, Communications of the ACM, pp 33-35, March 2006.

[184] Michael Weigend, To Have or to Be? Possessing Data Versus Being in a State – Two Different Intuitive Concepts Used in Informatics, R.T. Mittermeir and M.M. Sysło (Eds.), Informatics Education - Supporting Computational Thinking, Third International Conference on Informatics in Secondary Schools - Evolution and Perspectives, ISSEP 2008 Torun Poland, Proceedings, Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg, pp. 151–160, July 1-4, 2008.

[185] Corrado Priami, Computational Thinking in Biology, In C. Priami (Ed.), Transactions on Computational System Biology VIII, Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg, pp. 63–76, 2007.

[186] Chris Thornton, Quantitative Abstraction Theory, Artifical Intelligence and Simulation of Behavior Journal 1(3), SSAISB, 2003, retrieved from http://www.cogs.susx.ac.uk/users/christ/papers/q-abstraction-theory.pdf, last accessed Dec. 18, 2009.

[187] J. Kramer, Is Abstract the key to computing?, Communications of the ACM, April 2007, pp 37- 42.

[188] Keiron Nicholson, Judith Good, Katy Howland, Concrete Thoughts on Abstraction, 21st Annual Psychology of Programming Interest Group Workshop (PPIG 2009), University of Limerick, Ireland, June 24-26, 2009, http://www.ppig.org/papers/21st-nicholson.pdf.

[189] Miller, G. A., "The magical number seven, plus or minus two: Some limits on our capacity for processing information". Psychological Review 63 (2), pp 81-97, 1956 retrieved from http://psychclassics.yorku.ca/Miller/.

[190] Anabela Gomes and António José Mendes, Problem solving in programming, 19th Annual Workshop of Psychology of Programming Interest Group, PPIG'07, Joensuu, Finland, July 2-6, 2007,

[191] Jerome M. Sattler, Assessment of Children: Cognitive Applications, Jerome M. Sattler Publisher, USA, 2001.

[192] Fred Nickols, Solution Engineering: Choosing the Right Problem Solving Approach, 2004, retrieved on January 30th 2010 from http://home.att.net/~nickols/makesdif.htm.

[193] David H. Jonassen, Toward a design theory of problem solving, Educational Technology Research and Development, Volume 48, Number 4, Springer Boston, pp 63-85, December, 2000.

[194] Gregory D. Sterling, Thomas M. Brinthaupt, Faculty and Industry Conceptions of Successful Computer Programmers, Journal of Information Systems Education, Vol. 14(4), 2003.

[195] G. Polya, How to solve it, Princeton University Press, 1945.

[196] Katgleen M. Galotti, Cognitive Psychology: In and out of Laboratory, Thomson Wadsworth, pp 359-381, 2004.

[197] Katya G. Stoycheva and Todd I. Lubart, The nature of creative decision making, In Carl Martin Allwood, Marcus Selart (Ed.), Decision making: social and creative dimensions, Kluwer Academic Publisher, Netherlands, pp 15-33, 2001.

[198] Fred Nickols, Four Tips for "Beefing Up" Your Problem Solving Tool Box – Part One, April 2009, retrieved on January 30th, 2010 from http://blog.smartdraw.com/archive/2009/04/21/four-tips-for-beefing-up-your-problem-solving-tool-box-part-one.aspx.

[199] Fred Nickols, Four Tips for "Beefing Up" Your Problem Solving Tool Box – Part Two, April 2009, retrieved on January 30th, 2010 from http://blog.smartdraw.com/archive/2009/04/27/four-tips-for-beefing-up-your-problem-solving-tool-box-part-two.aspx.

[200] Nonaka, I., A Dynamic Theory of Organizational Knowledge Creation, Organization Science, pp 14-37, February 1994.

[201] Leon E. Winslow, Programming Pedagogy - A Psychological Overview, ACM SIGCSE Bulletin, Vol. 28, No. 3, Sept 1996.

[202] P.J. Denning, The profession of IT: Career Redux, Communications of the ACM, pp 21-26, September 2002.

[203] A.L.Costa and B. Kallick, Discovering and Exploring Habits of Mind, Association for Supervision and Curriculum Development (ASCD), 2000.

[204] Vikki Fix, Susan Wiedenback, Jean Scholtz, Mental Representations of Programs by Novice and Experts, INTERCHI '93, ACM, pp 74-79, 1993.

[205] Rebecca Mancy, Norman Reid, Aspects of Cognitive Style and Programming, 16th Workshop of the Psychology of Programming Interest Group. Carlow, Ireland, April 2004.

[206] Spiro, R. J. & Jehng, J., Cognitive flexibility and hypertext: Theory and technology for the non-linear and multidimensional traversal of complex subject matter. In D. Nix & R. Spiro (eds.), Cognition, Education, and Multimedia. Hillsdale, NJ: Erlbaum, pp 163-205, 1990.

[207] Linda S. Gottfredson, Dissecting practical intelligence theory: Its claims and evidence Intelligence Volume 31, Issue 4, Elsevier,July-August 2003 , 2003.

[209] R. Hastie and R.M. Dawes, Rational choice in an uncertain world: The psychology of judgment and decision making, 2nd edition, Sage Publications, USA, 2010.

[210] Barry Boehm, A view of 20th and 21st Century Software Engineering, Proceedings of the 28th international conference on Software engineering, Shanghai, China , ACM, pp 12-29, 2006.

[211] Yogi Ramacharaka (William Walker Atkinson), Raja Yoga or Mental Development, The Yogi Publication Society, 1934, pp 97-122, retrieved from http://www.sacred-texts.com/eso/ryo/ryo07.htm on 18 December 2009.

[212] Huitt, W., Critical thinking: An overview. Educational Psychology Interactive, Valdosta, GA: Valdosta State University, 1998.

[213] Peter A. Facione, Critical Thinking: A Statement of Expert Consensus for Purposes of Educational Assessment and Instruction, The California Academic Press, 1990.

[214] Richard Paul, Critical Thinking: How to prepare students for a rapidly changing world, Sonoma State University: California. 1993.

[215] Richard Paul, Robert Niewoehner, Linda Elder, The thinker's guide to engineering reasoning, The foundation for critical thinking, 2006.

[216] Jennifer Moon, Critical thinking: an exploration of theory and practice, Routledge, pp 49-51, 2007

[217] Ron Barnett, Higher Education: A Critical Business, Open University Press /SRHE, 1997

[218] Schön D., The reflective practitioner, Basic Books: New York, 1983.

[219] Kottamp, R., Means of facilitating reflection, Education and Urban Society, 22.2, pp. 182-203, 1990.

[220] Schön D., Educating the Reflective Practitioner. Jossey-Bass: San Francisco, 1987.

[221] Phoebe Sengers, Kirsten Boehner, Shay David and Joseph 'Jofish' Kaye, Reflective Design, Proceedings of the 4th decennial conference on Critical computing: between sense and sensibility, Denmark, ACM, pp 49-58, 2005.

[222] Stones E., Reform in teacher education: The power and the pedagogy. Journal of Teacher Education, Vol. 45, Sage, pp 310-318, 1994.

[223] Ginsburg, M. B., Contradictions in teacher education and society: A critical analysis. New York: Falmer, 1988.

[224] Lasley, T.. Editorial. Journal of Teacher Education, SAGE, March - April 1998.

[225] Borton T., Reach, Teach and Touch. Mc Graw Hill, London, 1970.

[226] Guihua Li, Shawna Long, and Mary Ellen Simpson, Self perceived gains in critical thinking and communication skills: Are there disciplinary differences, Research in Higher Education, Vol. 40, No. 1, Springer, pp 43-60, February 1999.

[227] Osche, R., Before the gates of excellence: The determinants of creative genius. Cambridge, MA: Cambridge University Press, 1990.

[228] Robert W. Weisberg, Creativity: Understanding innovation in problem solving, science, inventions and the arts, John Wiley and sons, USA, pp 205-207, 2006.

[229] Robert J. Sternberg, Wisdom, intelligence, and creativity synthesized, Cambridge University Press, UK, pp 124-146, 2003.

[230] Katya G. Stoycheva and Todd I. Lubart, The nature of creative decision making, In Carl Martin Allwood, Marcus Selart (Ed.), Decision making: social and creative dimensions, Kluwer Academic Publisher, Netherlands, pp 15-33, 2001.

[231] Edward De Bono, Lateral thinking: creativity step by step, Harper & Row, USA, 1970.

[232] Kalevi Rantanen and Ellen Domb, Simplified TRIZ: new problem solving applications for engineers & manufacturing professionals, CRC Press, pp 129-210, 2002.

[233] Rea, K.C., TRIZ and Software - 40 Principles Analogies, Part 1, The TRIZ Journal. Sep, 2001, retrieved from http://www.triz-journal.com/archives/2001/09/e/index.htm.

[234] Rea, K.C., TRIZ and Software - 40 Principles Analogies, Part 2, The TRIZ Journal, Nov, 2001, retrieved from http://www.triz-journal.com/archives/2001/11/e/.

[235] Ron Fulbright, Teaching critical thinking skills in IT using PINE-TRIZ, Proceedings of the 5th conference on Information technology education, ACM pp 38-42, October 2004.

[236] James Kowalick, 17 Secreted of an inventive mind: How to conceive world class products rapidly using TRIZ and other leading edge creative tools, The TRIZ Journal, Nov, 1996, retrieved from http://www.triz-journal.com/archives/1996/11/b/index.htm.

[237] Guy-Alain Amoussou, Eileen Cashman, Steve Steinberg, Ways to Learn and Teach Creativity and Design in Computing Science, Proceedings of Science of Design Symposium, Humboldt State University, ACM, pp 12-13, March 2007.

[238] Carly J. Lassig, Promoting creativity in education -- from policy to practice: an Australian perspective, Proceeding of the seventh ACM conference on Creativity and cognition, ACM, USA, pp-229-238, October 2009.

[239] Biggs, J., Student approaches to learning and studying. Melbourne, Australia: Australian Council for Educational Research, pp 9, 1987.

[240] Michael B. Paulsen and Charles T. Wells, Domain differences in the epistemological beliefs of college students, Research in Higher Education, Vol. 39, No. 4, Springer, pp 365-394, August 1998.

[241] Marlene Schommer-Aikins, Orpha K. Duell, and Sue Barker, Epistemological beliefs across domain using Biglan's classified of academic disciplines, Research in Higher Education, Vol. 44, No. 3, Springer 347-366, June 2003.

[242] Walter Brand, Hume's Account of Curiosity and Motivation, The Journal of Value Inquiry, Volume 43 Number 1, Springer, pp 83–96, March 2009.

[243] Barbara M. Benedict, Curiosity: a cultural history of early modern inquiry, University of Chicago Press, pp 1-8, 2001.

[244] Thomas G. Reio, Jr. and Jamie L. Callahan, Affest, Curiosity, and socialization related learning: A path analysis of antecedents to job performance, Journal of Business and Psychology, Vol. 19, No. 1, Springer, pp 3-20, Fall 2004.

[245] Marilyn P. Arnone, Using Instructional Design Strategies to Foster Curiosity, 2003

[246] Christopher Peterson, Martin E. P. Seligman, Character strengths and virtues: A handbook and classification, Oxford University Press, USA, pp 125-141, 2004.

[247] David Beswick, An Introduction to the Study of Curiosity, Centre for Applied Educational Research, University of Melbourne, 10 May 2000, retrieved from http://www.beswick.info/psychres/curiosityintro.htm on Dec 27th. 2009.

[248] Todd B. Kashdan, Michael F. Steger, Curiosity and pathways to well-being and meaning in life: Traits, states, and everyday behaviors, Motivation and Emotion, Volume 31, Number 3 /, Springer, 159-173, September, 2007.

[249] Robert N. Carson, A Taxonomy of Knowledge Types for Use in Curriculum Design, Interchange, Vol. 35/1, Kluwer Academic Publishers, pp 59-79, March 2004.

[250] Perry, W. G., Forms of intellectual and ethical development in the college years. New York: Holt, Rinehart and Winston, 1970.

[251] Richard M. Felder and Rebecca Brent, The intellectual development of science and engineering students Part 1, Models and challenges, Journal of Engineering Education, ASEE, USA, pp 269–277, October 2004.

[252] Elise J. West, Perry's Legacy: Models of Epistemological Development, Journal of Adult Development, Vol. 11, No. 2, Springer, pp 61-70, April 2004.

[253] Center for Teaching and Learning, Helping our Students to Achieve Better Thinking, Nutshell Notes, Newsletter for Teaching Excellence, November, Idaho State University, 2005 retrived from http://www.isu.edu/ctl/nutshells/nutshell13-7.HTML.

[254] Michael J. Pavelich, Helping students develop high level thinking: Use of the Perry model, Procedings of Frontiers in Education (FIE), 1996, IEEE, pp 163-167, 2001.

[255] John Wise, Sang Ha Lee, Thomas A. Litzinger, Rose M. Marra, betsy Palmer, Measuring Cognitive growth in Engineering undergraduates: a longitudinal study, Proceedings of the American Society for Engineering Education Annual Conference and Exposition, ASEE, USA, 2001.

[256] Salas E. and Klein G.A., Linking expertise and naturalistic decision making, Lawrence Erlbaum Associates Inc, USA, pp 19-20, 2001.

[257] Carl Martin Allwood, Marcus Selart, Decision making: social and creative dimensions, Kluwer Academic Publisher, Netherlands, pp 18-20, 2001.

[258] David G. Ullman, Making Robust Decisions: Decision management for technical, business, and service teams, Trafford Publishing, Canada, 2006.

[259] S.R. Seyedjavadein, Amir Hossein Fahimi, Introduction of TRIZ to the Process and Levels of Decision Making, The TRIZ Journal, December 2005, http://www.triz-journal.com/archives/2005/12/08.pdf.

[260] A. Rowe and J. Boulgarides, Managerial Decision Making: A Guide to Successful Business Decisions, Macmillan, New York, 1992.

[261] Becker, Boris W. and Connor, Patrick E, Personal value systems and decision-making styles of public managers, Public Personnel Management; Spring2003, Vol. 32 Issue 1, pp 155-181, March 2003, http://goliath.ecnext.com/coms2/gi_0198-393074/Personal-value-systems-and-decision.HTML.

[262] Terry L. Fox, J. Wayne Spence, The effect of decision style on the use of a project management tool: an empirical laboratory study, The database for advances in information systems, Volume 360 Issue 2, ACM, pp 28-42, June 2005.

[263] Hammond JS, Keeney RL, Raiffa H. Smart choices: a practical guide to making better decisions Boston, Mass: Harvard Business School Press, 1999.

[264] M. G. Myriam Hunink, Decision Making in the Face of Uncertainty and Resource Constraints: Examples from Trauma Imaging, Radiology, Volume 235 Number 2, Radiological Society of North America Inc., pp 375-383, May 2005.

[265] Claudine Toffolon, Salem Dakhli, A decision oriented model of software engineering processes, Proceedings European and Mediterranean Conference on Information Systems 2007 (EMCIS2007), Polytechnic University of Valencia, June 24-26 2007, http://www.iseing.org/emcis/EMCIS2007/emcis07cd/EMCIS07-PDFs/702.pdf.

[266] C. Ravindranath Pandian, Applied software risk management: A guide for software project managers, Auerbach Publications, USA, 2007.

[267] Boehm, B.: Software Risk Management: Principles and Practices. IEEE Software, pp 32-41, January 1991.

[268] Mark Keil, Paul E. Cule, Kalle Lyytinen, and Roy C. Schmidt, A Framework for Identifying, Software Project Risk, Communication of the ACM/Vol. 41, No. 11, pp 76-83, November 1998.

[269] Linda Wallace and Mark Keil, Software project risks and their effect on outcomes, Communication of the ACM /Vol. 47, No. 4, pp 69 –73, April 2004.

[270] Marvin J. Carr, Suresh L. Konda, Ira Monarch, F. Carol Ulrich, Taxonomy-Based Risk Identification, Technical Report CMU/SEI-93-TR-6, ESC-TR-93-183, Software Engineering Institute, Carnegie Mellon University, USA, June 1993.

[271] Richard P. Kendall, Douglass E. Post, Jeffrey C. Carver, Dale B. Henderson, David A. Fisher, A Proposed Taxonomy for Software Development Risks for High-Performance Computing (HPC) Scientific/Engineering Applications, Technical Note, CMU/SEI-2006-TN-039, Software Engineering Institute, Carnegie Mellon University, USA, January 2007.

[272] Konstantina Georgieva, Ayaz Farooq, and Reiner R. Dumke, Analysis of the Risk Assessment Methods – A Survey, A. Abran et al. (Eds.): IWSM/Mensura 2009, LNCS 5891, Springer-Verlag Berlin Heidelberg pp. 76–86. 2009.

[273] Edward J. O'Boyle, An ethical decision-making process for computing professionals, Ethics and Information, , Springer Netherlands, pp 267–277, December 2002.

[274] R. Hastie and R.M. Dawes, Rational choice in an uncertain world: The psychology of judgment and decision making, 2nd edition, Sage Publications, USA, 2010.

[275] Donella H. Meadows , Thinking in Systems: A primer, Chelsea Green Publishing Company, USA, 2008.

[275a] Checkland Peter, Systems Thinking, Systems Practice, John Wiley & Sons, 1999.

[276] Boulding Kenneth, General systems theory - the skeleton of science, Management Science, 197-208, April 1956.

[277] Lars Skyttner, The Future of Systems Thinking, Systemic Practice and Action Research, Vol. 11, No. 2, pp193-205, April 1998.

[278] Senge, P., The Fifth Discipline: The Art and Practice of the Learning Organization, Currency Doubleday, New York, 1990.

[279] Andriy Solovey, 11 Laws of The System Thinking in Software Development, Software creation mystery, Jul 26th, 2007, retrieved from http://softwarecreation.org/2007/11-laws-of-the-system-thinking-in-software-development/.

[280] Moti Frank, and Shlomo Waks, Engineering Systems Thinking: A Multifunctional Definition, Journal of Systemic Practice and Action Research, Vol. 14, No. 3, Springer Netherlands, pp 361-371, June 2001.

[281] Carol Sanford , A Theory and Practice System of "Systems Thinking": With an Executive's Story of the Power of "Developmental" and "Evolutionary" Systems Thinking, 3rd International Conference on Systems Thinking in Management (ICSTM 2004), University of Pennsylvania, May 19 - 21, 2004, retrieved from www.interoctave.com/pdf/InterOctave_SystemsThinking_WhitePaper.pdf.

[282] F. Capra, Criteria of systems thinking, Futures, Volume 17, Issue 5, Elsevier, pp 475-478, October 1985.

[282a] George J. Klir and Doug Elias, Architecture of Systems Problem solving, Springer, 2003.

[283] Linda Booth Sweeney and Dennis Meadows, The Systems Thinking Playbook, Sustainability Institute, 2008.

[284] Checkland, P.B., Soft Systems Methodology, in J. Rosenhead and J. Mingers (eds), Rational Analysis for a Problematic World Revisited. Chichester: Wiley, 2001.

[285] Marty Jacobs, Systems Thinking: The Fifth Discipline of Learning Organizations, 2008, Systems In Sync.

[286] Mingfen Li, Fostering Design Culture Through Cultivating the User-Designers' Design Thinking and Systems Thinking, Journal of Systemic Practice and Action Research, Volume 15, Number 5, Springer Netherlands, pp 385-410, October, 2002.

[287] Martin Hoffman, The contribution of empathy to justice and moral development, In Nancy Eisenberg and Janet Strayer (Ed.), Empathy and its development, Cambridge University Press, pp 47-80, 1990.

[288] Cecilia Haskins, Using systems engineering to address socio-technical global challenges, Sixth Annual Conference on Systsems Engineering Research (CSER'08), LA, USA, April 2008, retrieved from http://cser.lboro.ac.uk/CSER08/pdfs/Paper%20111.pdf.

[289] L. Kohlberg, The psychology of moral development: the nature and validity of moral stages, Harper & Row 1984.

[290] Dario Spini, Measurement equivalence of 10 value types from the Schwartz value survey across 21 countries, Journal of cross-cultural psychology, Vol. 34 No. 1, SAGE Publications, USA, , pp 3-23, January 2003.

[291] Schwartz, S. H., Universals in the content and structure of values: Theoretical advances and empirical tests in 20 countries, In M. P. Zanna (Ed.), Advances in experimental social psychology, (Vol. 25, pp. 1-65). San Diego, CA: Academic Press, 1992.

[292] Steven Reiss, Multifaceted Nature of Intrinsic Motivation: The Theory of 16 Basic Desires, Review of General Psychology, Educational Publishing Foundation, Vol. 8, No. 3, 179–193, 2004.

[294] Steven Reis, Who Am I?: The 16 Basic Desires That Motivate Our Behavior and Define Our Personality, Tarcher, 2000.

[295] Carol D. Ryff and Burton H. Singer, Know thyself and become what you are: a Eudaimonic approach to psychological well-being, Journal of Happiness Studies, Jan 2006, Springer Netherlands, pp 13-39.

[296] Richard M. Ryan, Veronika Huta and Edward L. Deci, Living well: a self-determination theory perspective on eudaimonia, Journal of Happiness Studies, Volume 9, Number 1, Springer Netherlands, pp 139-170, January, 2008.

[297] Edward L. Deci and Richard M. Ryan, Hedonia, eudaimonia, and well-being: an introduction, Volume 9, Number 1, Springer Netherlands, pp 1-11 January, 2008.

[298] Deci, E. L., & Ryan, R. M. Intrinsic motivation and self-determination in human behavior, 1985, New York: Plenum.

[299] R.J. Sternberg, Intelligence as developing expertise, J. of Contemporary Educational Psychology, Elsevier, pp 359-375, October 1999.

[300] M.A. Collins, and T.M. Ambile, Motivation and Creativity, In Robert J. Sternberg (Ed.),, Handbook of Creativity, Cambridge University Press, UK, pp 297-312, 1999.

[301] Kennon M. Sheldon, Creativity and Self-Determination in Personality, Creativity Research Journal Vol. 8, No. 1, Lawrence Erlbaum Associates, pp 25-36, 1995.

[302] Hernan Casakin and Shulamith Kreitler, Motivational aspects of creativity in students and architects: implications for education, International conference on engineering and product design education, 4 -5, Universitat Politechnica de Catalunya, Barcelona, Spain, September 2008.

[303] Hernan Casakin and Shulamith Kreitler, Motivation for creativity in architectural design and engineering design students: implications for design education, International Journal Technol Desing Education, , Springer, Online published on Dec 1st, 2009.

[304] Brawner, catherine et al., A survey of faculty teaching practices and involvement in faculty development activities, http://www.ncsu.edu/felder-public/papers/survey_teaching-practices.pdf.

[305] Krumme Gunter, Major Categories in the Taxonomy of Educational Objectives, (Bloom 1956), 2002, Retrieved from http://faculty.washington.edu/krumme/guides/bloom.HTML.

[306] TALS (Effective Teaching in Agriculture and Life Sciences), Bloom's taxonomy: Lessons, 1998, Retrieved from http://www.ais.msstate.edu/TALS/unit1/1moduleB.HTML, http://www.ais.msstate.edu/TALS/unit1/1moduleC.HTML.

[307] Ostrow Jim, Service-learning for depth in a fluid world, Tomorrow's Professor, 2005, Retrieved from http://ctl.stanford.edu/Tomprof/postings/622.HTML.

[308] Honan, William H., The College Lecture, Long Derided, May Be Fading, Newyork Times, August 14, 2002.

[309] Fagen, Adam Paul, "Assessing and Enhancing the Introductory Science Course in Physics and Biology: Peer Instruction, Classroom Demonstrations, and Genetics Vocabulary", Ph.D. Thesis, Harvard University, 2003, http://mazur-www.harvard.edu/publications/Pub_405.pdf.

[310] Northwood M.D. and Northwood O..D., Problem-Based Learning (PBL): From the Health Sciences to Engineering to Value-Added in the Workplace, Global Journal of Engineering Education, Vol.7, No.2, pp 157-164, 2003. http://www.eng.monash.edu.au/uicee/gjee/vol7no2/Northwood.pdf.

[311] Woods, D.R., Problem-Based Learning: How to Gain the Most from PBL. Waterdown: Donald R. Woods Publisher, 1994.

[312] Merrill, M. D, Instructional strategies that teach. CBT Solutions, 1997, http://www.id2.usu.edu/Papers/Consistency.PDF.

[313] Fennimore, T.F. and Tinzmann , M.B. "What Is a Thinking Curriculum?", NCREL, Oak Brook, 1990, http://www.ncrel.org/sdrs/areas/rpl_esys/thinking.htm.

[314] Kearsley Greg & Shneiderman Ben, "Engagement Theory: A framework for technology-based teaching and learning", 1999, http://home.sprynet.com/~gkearsley/engage.htm.

[315] Ladyshewsky Richard K. and Ryan John, Reciprocal peer coaching as a strategy for the development of leadership and management competency, Teaching and Learning Forum, Edith Cowan University, Australia, 2002, retrieved from http://www.ecu.edu.au/conferences/tlf/2002/pub/docs/Ladyshewsky.pdf.

[316] Schank, Roger C. and Cleary Chip, Engines for Education, LEA Publishers, pp 27-31, 1995.

[317] Arias, E.G., Eden, H., Fischer, G., & Schraff, E. "Transcending the Individual Human Mind – Creating Shared Understanding through Collaborative Design", ACM Transactions on Computer Human-Interaction, 7(1), pp. 84-113, March 2000.

[318] Fischer, Gerhard, Meta-design: Beyond User Centered and Participatory Design, Proceedings of HCI International, Crete, Greece., 22-27 June, 2003 retrieved from http://www.cs.colorado.edu/~gerhard/papers/hci2003-meta-design.pdf.

[319] Aaron Fried, Karen Zannini, Don Wheeler, Yongjin Lee, and Jose Cortez, Instructional Design Theory database Project, Syracuse University, http://web.cortland.edu/frieda/ID/IDdatabase.html.

[320] Greg Kearsley, Explorations in Learning & Instruction: The Theory Into Practice Database, http://tip.psychology.org/index.html.

[321] Charles M. Reigeluth (ed.), Instruction Design Theories and Models: A New paradigm for Instruction Design, Routledge, 1999.

[322] Jonassen, D. H., Myers, J. M. & McKillop, A. M., From constructivism to constructionism: Learning with hypermedia/multimedia rather than from it, In B. G. Wilson (Ed.), constructivist learning environments: Case studies in instructional design. Englewood Cliffs, NJ: Educational Technology Publications, pp. 93-106, 1996.

[323] John Bransford, John D. Bransford, Barry S. Stein, The ideal problem solver: a guide for improving thinking, learning, and ..., W.H. Freeman, 1993.

[324] David Byrne, Charles C. Ragin, The SAGE Handbook of Case-Based Methods, SAGE Publications, UK, 2009.

[325] Merrill, M. D, Instructional Design Theory. Englewood Cliffs: Educational Technology Publications, 1994.

[326] Barry Kort and Rob Reilly, Theories for Deep Change in Affect sensitive Cognitive Machines: A Constructivist Model, Journal of Educational Technology & Society, Vol. 5 Issue 4, International Forum of Educational Technology & Society, USA, pp 56-63, 2002.

[327] L. Festinger, A theory of cognitive dissonance, Stanford University Press, Stanford, USA, 1957.

[328] Gregory Bateson, Steps to an ecology of mind, University of Chicago Press, USA, 2000.

[329] BIGGS J and COLLIS K, Evaluating the Quality of Learning: the SOLO taxonomy New York: Academic Press, 1982.

[330] Knowles Malcolm, "The Modern Practice of Adult Education: Andragogy versus Pedagogy", Association Press, New York, 1970.

[331] The Teacher's Guide, University of Tasmania, http://www.artschool.utas.edu.au/pigvision/packteachersguide.HTML.

[332] Managing Active Classrooms, UNICEF, 2000 http://www.unicef.org/teachers/teacher/manage.htm.

[333] M. Rauterberg, Framework for Information and Information Processing of Learning Systems, Echakrd falkenberg, Wolfganf Hesse, and Antoni Olive, Information System Concepts: Towards Consolidation of views, Proceedings of the IFIP international working group on Information Systems Concepts, Chapman and Hall, UK, pp 54-69, 1995.

[334] Chickering, A., & Gamson, Z., Seven principles of good practice in undergraduate education. AAHE Bulletin, pp 3-7, March 1987.

[335] George D. Kuh, et al, Exploring Different Dimensions of Student Engagement 2005 Annual Survey Results, National Survey Of Student Engagement, Indiana University, USA, 2005.

[336] Ropohl, G., Knowledge types in technology. International Journal of Technology and Design. Education, Springer Netherlands, pp 65 -72, January 1997.

[337] Routio, Pentti, Arteology: the science of artifacts, University of Art and Design, Helsinki, Finlad, http://www.uiah.fi/projects/metodi/154.htm.

[338] Sweller, J., Cognitive load during problem solving: Effects on learning, Cognitive Science, 12, 257-285, 1988.

[339] Csikszentmihalyi, M., Flow: The Psychology of Optimal Experience, Harper and Row, 1990.

[340] David Ausubel. Educational psychology; a cognitive view. Holt, Rinehart and Winston, New York, New York, 1968.

[341] Joseph Donald Novak, "Learning, Creating, and Using Knowledge: Concept maps as Facilitative Tools in Schools and Corporations", Lawrence Erlbaum Associates, pp 49-56, 1998.

[342] Brown, M.T. & Nolan, C.J.P., Getting it Together: Explorations in Curriculum Integration, Out of Class Activities and Computer Applications. Massey University, Palmerston North, 1989.

[343] Beane, J., Curriculum integration. Designing the core of democratic education, New York and London: Teachers College Press, Columbia University, 1997.

[344] Fogarty, R., Ten ways to integrate curriculum. Educational Leadership. 49(2), pp 61-65, 1991.

[345] Drake, S. M., Creating integrated curriculum: Proven ways to increase student learning. Thousand Oaks, CA, Corwin, 1998.

[346] Ronald M Harden, The integration ladder: a tool for curriculum planning and evaluation, MEDICAL EDUCATION, Vol. 34, Blackwell Science Ltd, pp 551-557, July 2000

[347] Claus Brabrand and Bettina Dahl, J. , Using the SOLO taxonomy to analyze competence progression of university science curricula, Journal of of Higher Education, 58, Springer, pp 531–549, February 2009.

[348] Sanjay Goel, Do Engineering Faculty Know What's Broken?, The National Teaching & Learning Forum, James Rhem & Associates, USA, Vol. 15, pp 1-10, Number 2, 2006.

[349] Cliburn, D.C., Experiences with Pair Programming at a Small College, Journal of Computing Sciences in Colleges, Pages: 20 - 29 Volume 19, Issue 1, October 2003.

[350] Goldfarb, Mary Ellen, The Educational Theory of Lev Semenovich Vygotsky (1896 - 1934), Edward G. Rozycki & M. F. Goldfarb and Associates, 2001, retrieved from http://www.newfoundations.com/GALLERY/Vygotsky.HTML.

[351] Bruner, J. S., Towards a theory of instruction. Cambridge Massachusetts: Belknap Press, 1966.

[351a] Driscoll, Marcy P., Psychology of learning for instruction, Allyn & Bacon, pp 227-245, 2004.

[352] Kutay Cat, Implementation patterns for supporting learning and group interactions, PhD thesis, University of New Soth Wales, Page, 19, 2005, retrieved from http://www.library.unsw.edu.au/~thesis/adt-NUN/uploads/approved/adt-NUN20060823.125823/public/02whole.pdf.

[353] Baxter Magolda, M., Knowing and reasoning in college: Gender-related patterns in students' intellectual development. San Francisco: Jossey-Bass, 1992.

[354] Smith, M. K., Learning theory, The encyclopedia of informal education, Infed, 2007, retrieved from http://www.infed.org/biblio/b-learn.htm#cite.

[355] Dillenbourg P., What do you mean by collaborative learning? In P. Dillenbourg (Ed) Collaborative-learning: Cognitive and Computational Approaches. Pergamon Oxford: Elsevier, pp.1-19, 1999.

[356] Lipponen, L. Exploring foundations for computer-supported collaborative learning, In G. Stahl (Ed.), Computer Support for Collaborative Learning: Foundations for a CSCL community. Proceedings of the Computer-supported Collaborative Learning Conference, Hillsdale, NJ: Erlbaum, pp. 72-81, 2002 .

[357] Janet Salmons, Expect Originality! using taxonomies to structure assignments that support original work, In T. S. Roberts (Ed.), Student plagiarism in an online world: problems and Solutions, Information Science Reference, IGI Global, USA, pp 216 – 217, 2008.

[358] Sanjay Goel, Activity based flexible credit definition, Tomorrow's Professor, 2003, http://ctl.stanford.edu/Tomprof/postings/513.HTML.

[359] Baumgartner, E. Designing inquiry: Contextualizing teaching strategies in inquiry-based classrooms. Annual Meeting of the American Educational Research Association, Montréal, 1999. http://www.designbasedresearch.org/reppubs/baum-AERA.pdf.

[360] Fischer, G., Seeding, Evolutionary Growth and Reseeding: Constructing, Capturing and Evolving Knowledge in Domain-Oriented Design Environments**,** Journal of Automated Software Engineering, Springer Netherlands, pp 447 – 464, October 1998.

[361] Fischer, G. dePaula, R., Ostwald, J., "Courses as Seeds: Expectations and Realities", Proceedings of The European Conference on Computer-Supported Collaborative Learning (Euro-CSCL 2001), Maastricht, The Netherlands, March 22-24, pp 494-501, 2001. [http://www.cs.colorado.edu/~gerhard/papers/ecscl2001.pdf.

[362] Sharda, Nalin, Combining the Art, Science and Technology of Multimedia with The Multimedia Creation Circles Paradigm, Preprint, 2004, http://sci.vu.edu.au/~nalin/MultimediaCreationCirclesPreprintSharda.pdf.

[363] Ascott, R., Is there Love in the Telematic Embrace? Art Journal: New York: College Arts Association of America. 49:3, pp. 241-7, 1990, retrieved from http://x.i-dat.org/~mp/DIGF/LM/PDF/TelematicEmbrace.pdf.

[364] Aristotle, "Poetics", 350 BC.

[365] Sharda, Nalin, Combining the Art, Science and Technology of Multimedia with The Multimedia Creation Circles Paradigm, Preprint, http://sci.vu.edu.au/~nalin/MultimediaCreationCirclesPreprintSharda.pdf, 2004.

[366] Miliszewska Iwona et al, "Transnational Education through Engagement: Students' Perspective", Informing Science, pp 165-173, June 2003, retrieved from http://ecommerce.lebow.drexel.edu/eli/2003Proceedings/docs/031Milis.pdf.

[367] Ritu Arora, Sanjay Goel, "Software Engineering Approach for Teaching Development of Scalable Enterprise Applications,", 22nd IEEE-CS Conference on Software Engineering Education and Training CSEET, , pp.105-112, February 2009.

[368] Linda Null, Integrating Security Across the Computer Science Curriculum, CCSC-Northern Eastern Conference,19, 5, pp 170-178, May 2004

[369] Blair Taylor, Shiva Azadegan: Moving Beyond Security Tracks: Integrating Security in CS0 and CS1, SIGCSE Bulletin Volume 40 , Issue 1, ACM, pp 320-324 March 2008.

[370] James Walden, Charles E. Frank: Secure Software Engineering Teaching Modules, InfoSecCD Conference, pp. 19-23, 2006.

[370a] Jolly Shah, Sangeeta Mittal, Sanjay Goel, An Approach for Infusing Security Aspects in Computing Curriculum, In progress,, 2009.

[371] Watt S. Humphrey, PSP: A self improvement process for software engineers, Addison-Wesley Professional, USA, 2005.

[372] Chaparro E. A., An Intelligent Cognitive Tool To Foster Collaboration In Distributed Pair Programming, 8th Human Centred Technology Postgraduate Workshop, University of Sussex, 2005. Retrieved from http://hct.fcs.sussex.ac.uk/Submissions/22.pdf.

[373] CockBurn, A. and Williams L., The cost and benefits of pair programming, Extreme Programming Examined, Page 223-243, ed. G. Succi and M. Marchesi, Addison-Wesley Longman Publishing Co., 2001. Retrieved from http://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF.

[374] Williams, L., Kessler, R., Pair Programming Illustrated, Boston, Massachusetts: Addison Wesley, 2003.

[375] VanDeGrift T., Coupling Pair Programming and Writing: Learning About Students' Perceptions and Processes, Proceedings of the Thirty-Fifth Technical Symposium on Computer Science Education (SIGCSE 2004), ACM Press, pp 2-6, 2004.

[376] Brusilovsky P., Kouchnirenko A., Miller P. and Tomek I. Teaching programming to novices: A review of approaches and tools. In T.Ottman, I.Tomek (eds.) Proc.of ED-MEDIA'94 - World conference on educational multimedia and hypermedia. Vancouver, Canada, pp 103-110, June 1994.

[377] Gogoulou A, Gouli E, Grigoriadou M, Samarakou M, Exploratory + Collaborative Learning in Programming: A Framework for the Design of Learning Activities, Proceedings of the 3rd IEEE International Conference on Advanced Learning Technologies, pp 350-351, 2003, retrieved from http://ieeexplore.ieee.org/iel5/8621/27318/01215118.pdf.

[378] Williams L.,Wiebe E., Yang K., Ferzli M., Miller C., In Support of Pair programming in the Introductory Computer Science courses. Computer Science Education, Volume 12, Issue 3, Swets & Zeitlinger, pp 197-212 September 2003.

[379] McDowell, C., Werner, L., Bullock, H., and Fernald, J., The Effects of Pair Programming on Performance in an Introductory Programming Course, Proceedings of the Thirty-Third Technical Symposium on Computer Science Education (SIGCSE 2002) , ACM Press, pp 38-42, 2002.

[380] Williams L. and Kessler R., Experimenting with Industry' s 'Pair programming' Model in the Computer Science Classroom, Journal on SW Engineering Education, December. 2000, Retrieved from http://collaboration.csc.ncsu.edu/laurie/Papers/CSED.pdf.

[381] Nagappan, N., Williams, L., Ferzli, M., Wiebe, E., Yang, K., Miller, C., Balik, S., Improving the CS1 Experience with Pair Programming, Proceedings of the Thirty-Fourth Technical Symposium on Computer Science Education (SIGCSE 2003), pp 259-362, ACM Press, 2003.

[382] Thomas, L., Ratcliffe, M., Robertson, A., Code Warriors and Code-a-Phobes: A Study in Attitude and Pair Programming, Proceedings of the Thirty-Fourth Technical Symposium on Computer Science Education (SIGCSE 2003), pp 363-367, ACM Press, 2003.

[383] Williams L. and Kessler R. All I Really Need to Know About Pair Programming I Learned in Kindergarten. Communications of the ACM, Volume 43, Issue 5, pp 108– 114, May 2000.

[384] Williams L. and Kessler R., Experimenting with Industry' s 'Pair programming' Model in the Computer Science Classroom, Journal on SW Engineering Education, Dec. 2000, Retrieved from http://collaboration.csc.ncsu.edu/laurie/Papers/CSED.pdf.

[385] Domino Madeline Ann, Collins Rosann Webb, Hevner Alan R. Controlled experimentation on adaptations of pair programming, Information Technology and Management, Springer, Volume 8, Number 4, pp 297-312 December, 2007.

[386] Sfetsos Panagiotis, Stamelos Ioannis, Angelis Lefteris, Deligiannis Ignatios, An experimental investigation of personality types impact on pair effectiveness in pair programming, Empirical Software Engineering, Volume 14, Number 2, Springer, pp 187–226, April 2009,.

[387] Brereton Pearl, Turner Mark, Kaur Rumjit, Pair programming as a teaching tool: a student review of empirical studies, Proceedings of 22nd Conference on Software Engineering Education and Training, IEEE, pp, 240-247, February 2009.

[388] Lui Kim Man and Chan Keith C.C., Software Process Fusion: Uniting Pair Programming and Solo Programming Processes, Q. In Wang et al. (Eds.): Proceedings of SPW/ProSim 2006, LNCS 3966, Springer-Verlag Berlin Heidelberg, pp. 115–123, 2006

[389] Sanjay Goel and Vanshi Kathuria A Novel approach for pair programming, Journal of Information Technology Education, USA, Accepted with revision, Revised copy submitted, 2009.

[390] Bevan, J., Werner, L., McDowell, C., Guidelines for the Use of Pair Programming in a Freshman Programming Class, Conference on Software Engineering Education and Training, Kentucky, IEEE Computer Society, pp 100-107, 2002.

[391] Williams, L., Kessler, R.R., Cunningham, W, Jeffries, R., Strengthening the case for pair programming, Software, Volume 17, Issue 4, IEEE, pp 19-25, Jul/Aug 2000.

[392] Jason A., Technical and Human Perspectives on Pair Programming, ACM SIGSOFT Software Engineering Notes Vol. 25, Number 5, pp 1-14, September 2004.

[393] Bloom Benjamin S., The 2 Sigma Problem: The Search for Methods of Group Instruction as Effective as One-to-One Tutoring, Educational Researcher 13 (6), pp 4–16, May 1984.

[395] Judy McKimm, Carol Jollie and Mark Hatter, Mentoring: Theory and Practice, 2007, retrieved from http://www.faculty.londondeanery.ac.uk/e-learning/feedback/files/Mentoring_Theory_and_Practice.pdf.

[396] University of Missouri-Kansas City, The international center for Supplemental Instruction website, Retrieved on September 23, 2009 from http://web2.umkc.edu/cad/SI.

[397] Moore-West, M., Hennessy, S. A., Meilman, P. W., & O'Donnell, J. F., The presence of student-based peer advising, peer tutoring, and performance evaluation programs among U.S. medical schools. Academic Medicine, vol. 65, pp 660–661, October 1990,.

[398] Tai M., Patricia O'Sullivan, Arianne Teherani, Jessica Muller, Understanding the experience of being taught by peers: the value of social and cognitive congruence, Adv in Health Sci Educ , Volume 13, Number 3, , Springer, pp 361–372, August, 2008.

[399] Topping, K., & Ehly, S. (eds). Peer-assisted learning. Mahwah, NJ: Lawrence Erlbaum Associates, 1998.

[400] Rod D. Roscoe and Michelene T. H. Chi, Tutor learning: the role of explaining and responding to questions, Instructional Science, Volume 36, Number 4, pp 321-350, July, 2008.

[401] Tania Smith, Integrating Undergraduate Peer Mentors into Liberal Arts Courses: A Pilot Study, Innovative Higher Education, volume 33 number 1, pp 49-63, Jun 2008,.

[402] Sanjay Goel, A proposal for a tutorial on enriching the culture of software engineering education through theories of knowledge and learning, Proceedings, 22nd IEEE-CS Conference on Software Engineering Education and Training, CSEET, , pp.279-282, February 2009.

[403] Sanjay Goel, Multimedia for Cultural learning, International workshop on Computer Applications in Archaeology, H.B. Bahuguna University, Sri Nagar, India, Invited paper, 2002.

[404] Siddharth Batra and Sanjay Goel, Digislim: A learning tool for logic level digital electronics, Computers in Education Journal, Vol XVIIII No 3, American Society of Engineering Education, USA, pp 17-27, July, 2009,

[405] Sanjay Goel and Mukul K. Sinha, Virtual Archaeolo-Heritage Exploratorium: A model design for School students, Indo-US S&T Forum Workshop on Digital Arcahaeology: A New Paradigm for Visualizing Past through Computing and Information Technology, India, Invited paper, Nov. 2005.

[406] Sanjay Goel, Anshul Jain, Priyank Singh, Saaransh Bagga, and Siddhartha Batra, Computer Vision aided Classification and Reconstruction of Indian Potteries, Indo-US S&T Forum Workshop on Digital Archaeology: A New Paradigm for Visualizing Past through Computing and Information Technology, India, Invited paper, Nov. 2005

[407] Sanjay Goel, A Model Design for Computer based Cognition Support Systems, International Conference on Multimedia in Humanities, IGNCA, 1998.

[408] Sanjay Goel, Design of Interactive Systems: Looking Beyond Cognitive domain, INCITE'07, EU-India co-operation in IT research Workshop, New Delhi, Invited talk, 2007.

APPENDIX

APPENDIX A1: SPINE-like Survey on Importance of Competencies

We administered a survey among Indian engineers and managers working in Indian and multinational IT companies to obtain their perceptions on the importance of forty-nine parameters of engineering education. *Twenty-three engineering and general professional competencies* were included in this list. Other parameters on *teaching methods*, *quality of education*, and *aspects of reputation of institutes* were the same as in the SPINE survey (Annexure AN11). Respondents were requested to assign numeric ratings to these parameters on a scale of 0 to 10, with 10 being the highest importance in terms of the parameter's criticality and potential contribution in preparing students for a successful professional career.

Fifty-four experts working in fifteen companies like ST Microelectronics, Infosys, HCL Technologies, Adobe Systems India Pvt. Ltd, Cadence, Tata Infotech, Syncata Ltd., and Computer Associates responded. The responding experts had industrial experience ranging from 1.5 years to 35 years with an average experience of 7.5 years, which is inferred to be slightly higher than the industry average, given the average age of employees in the Indian IT industry is only 27-30 years [5]. The Collection of these responses was spread over a period of approximately one year. The trends in the data were found to be quite stable in the present context of the Indian IT industry. Hardly any variation among the computed ranks was observed after the sample size exceeded thirty-five. Hence, the findings of this study can be considered as sufficiently reliable. Table A1.1 provides a statistical summary of the responses. Further, the Indian IT industry is heavily export driven, and generally the respondents work on projects for overseas clients. Consequently, the findings of this study have a global relevance, especially for those developing countries that are trying to boost up their export in software, and other IT services.

Engineering and General Professional Competencies

The respondents' response for some of the twenty-three competencies has been much more enthusiastic than many others. While the maximum response for all competencies was found to

be in the range of 9-10, the minimum response was more wide spread from 0 to 5. Table A1.1 reveals that the average ratings for these competencies vary from 4.7 to 8.2 with standard deviation varying from 1.5 to 2.3. It was seen that the maximum value for the importance of all competencies was 10 or 9, and minimum value for seventeen competencies varied from 0 to 2, with only six competencies getting the minimum importance rating of 3 or 4. It means that all competencies were considered to be most important by a few respondents, and absolutely unimportant by some others. Further, the standard deviation among the values of average ratings (Avg_R_j) was found to be only 1.02. Consequently, Avg_R_j alone could not be used as a good criterion for ranking the importance of different competencies. The opinion of respondents on the importance of some competencies was more uniform than others. Hence, a new figure of merit (FOM_j) has been defined in order to stretch out the distribution and classify the competencies in a reliable way. It is based on the three variables of average (Avg_R_j), standard deviation ($Stdev\_R_j$), and average of the least N responses ($Avg\_Min\_R_j$), as in eq. (1). Computations of figure of merit were performed with different values of N. The best distribution of FOM values with highest standard deviation was at N equal to 7. Hence, N equal to 7 was used for computation of $Avg\_Min\_R_j$.

Table A1.1: Importance of twenty-three core engineering and general professional competencies, as rated by Indian engineers and managers working in Indian and multi-national IT companies

No (j)	*Engineering Competency*	Average importance (Max. = 10) Avg_R_j	Standard deviation in the rated importance $Stdev\_R_j$	Average importance of the least 7 responses (Max. = 10) $Avg\_Min\_R_j$	Normalised Figure of Merit (Max. = 10) $NFOM\_R_j$	Category
1	Problem solving	8.2	1.5	5.6	10.0	Pivotal
2	Analysis/Methodological skills	8.0	1.6	5.0	8.8	Critical
3	Basic engineering proficiency	7.7	1.6	5.1	8.5	Critical
4	Development know-how	7.1	1.5	4.4	8.2	Critical
5	Teamwork skills	8.0	1.7	4.7	8.2	Critical
6	English language skills	7.6	1.7	4.6	7.6	Critical
7	Presentation skills	7.6	1.7	4.3	7.5	Critical
8	Practical engineering experience	7.1	1.7	4.3	7.3	Critical
9	Leadership skills	7.3	1.7	4.6	7.3	Critical
10	Communication skills	8.0	1.8	4.3	7.2	Critical
11	Ability to develop own engineering expertise	7.0	1.8	3.9	6.5	Obligatory
12	Research know-how	6.5	1.7	3.4	6.2	Obligatory
13	Ability to develop a broad general education	6.4	1.7	3.0	5.9	Obligatory
14	Awareness of environmental issues	6.3	1.8	3.3	5.7	Obligatory
15	Social skills	6.5	1.9	2.9	5.3	Obligatory
16	Specialized engineering proficiency	6.3	1.9	2.7	5.1	Obligatory
17	Project management skills	6.7	2.1	2.7	4.9	Desirable
18	Management of business processes and administration skills	6.5	2.1	2.6	4.6	Desirable
19	Sensitivity towards socio-economic aspects for sustainable technological development	6.0	2.0	2.0	4.2	Desirable
20	Finance	5.2	2.0	1.7	3.8	Complementary
21	Marketing	5.5	2.3	1.4	3.2	Complementary
22	Law	4.7	2.2	0.7	2.6	Complementary
23	Other language skills	4.8	2.3	0.4	2.4	Complementary

The overall average, Avg_R_j, represents the collective opinion of all respondents. A high value indicates the *necessity* of the competency. The average of the least seven responses, $Avg\_Min\_R_j$, represents a favorable rating by those seven respondents who were the least enthusiastic for j^{th} competency. Hence, even a moderate value of four for $Avg\_Min\_R_j$ is an

indicator of j^{th} competency's *indispensability*. A high standard deviation, $Stdev\_R_j$, represents highly divided opinion of the respondents. The proposed figure of merit (FOM_R_j) is inversely proportional to $Stdev\_R_j$, and is directly proportional to Avg_R_j as well as $Avg\_Min\_R_j$. Figure of merit (FOM_R_j) is then normalised with respect to the maximum value and the normalised figure of merit ($NFOM\_R_j$) is computed using eq. (2).

$$FOM\_R_j = (Avg\_R_j + Avg\_Min\_R_j)/Stdev\_R_j \quad (1)$$

$$NFOM\_R_j = 10*FOM\_R_j/Max\ \{FOM\_R_j:\ j = 1 \text{ to } 23\} \quad (2)$$

A high $NFOM\_R_j$ is possible only if Avg_R_j as well as $Avg\_Min\_R_j$ are high, and $Stdev\_R_j$ is low. On the other hand, a low computed value $NFOM\_R_j$ can be either be a result of low Avg_R_j, low $Avg\_Min\_R_j$, or a high $Stdev\_R_j$. $NFOM\_R_j$ improved the contrast among the ratings of the twenty-three competencies. The Normalised Figure of Merit ($NFOM\_R_j$) varies from 2.2 to 10. This is a good distribution with a standard deviation of 2.1, which is much higher than the standard deviation among the values of Avg_R_j. Table A1.1 enumerates these competencies in descending order of their normalised figure of merit ($NFOM\_R_j$). The competencies are then classified into the following five vertical categories depending on their normalised figure of merit ($NFOM\_R_j$):

i. Pivotal: nine or more.
ii. Critical: between seven and nine.
iii. Obligatory: between five and seven.
iv. Desirable: between four and five.
v. Complementary: less than four.

Teaching Methods

Eight teaching methods (*group projects, homework/out-of-class assignment, industrial training/internship, lecture, projects, practical training, seminars,* and *written projects/studies)* assessed by the SPINE study (Annexure AN11) were retained for evaluation by Indian respondents. Respondents were requested to assign numeric ratings (0-10) to these parameters in terms of their criticality, and potential contribution in preparing students for a successful professional career.

Table A1.2 enumerates these teaching methods in descending order of their normalised figure of merit (NFOM_Rj). The teaching methods are also classified into the following five vertical categories, as explained above.

All teaching methods were considered to be most important by a few respondents. As can be seen in Table A1.2, group project, projects, and practical training have been rated as more effective teaching methods than lectures.

Table A1.2: Importance of teaching methods as rated by Indian engineers and managers working in Indian and multi-national IT companies

No (j)	Teaching Method	Average importance (Max. = 10) Avg_R_j	Standard deviation in the rated importance $Stdev\_R_j$	Average importance of least 7 responses (Max. = 10) $Avg\_Min\_R_j$	Normalised Figure of Merit (Max. = 10) $NFOM\_R_j$	Category
1	Group Projects	8.0	1.3	5.6	10.0	Pivotal
2	Project	8.2	1.4	5.6	9.8	Pivotal
3	Practical Training	8.3	1.5	5.6	9.2	Pivotal
4	Industrial Training /Internship	7.6	1.8	4.3	6.5	Obligatory
5	Lecture	7.2	1.7	4.1	6.5	Obligatory
6	Seminars	7.0	1.7	3.9	6.3	Obligatory
7	Written projects/studies	6.8	1.7	4.0	6.2	Obligatory
8	Homework/Out-of-class assignment	6.1	2.1	2.3	3.8	Complementary

A closer examination of the data showed that approximately 20% of the respondents rated lecture's importance below or equal to 5, while the same fraction gave the rating of equal to or more than 9 on a scale of 10. It is interesting to note that the last three methods in the ranked list (Table A1.1) also have the largest variation. This low average importance rating, and also large variation in the ratings, of these teaching methods can be attributed to the nature of respondents' personal experiences during their student life. Most respondents may not have had a very positive experience with these methods, and only a few of them may have had the good fortune of attending good quality lectures and/or doing high quality written projects/studies or homework during their student life.

Appendix A2: A Comprehensive Distilled View on Desired Competencies

Summary of various recommendations about desired competencies of software developers (alphabetically ordered)

1. Ability to accommodate himself to others, empathy, "be the customer" mentality - genuine interest in understanding what other people are trying to accomplish and based on this understanding think about creating technical solutions to help them reach their goals. Genuine interest in understanding "why to create software" and the broader context of software systems. Cognitive task analysis. Appreciation of unstated requirement and ability to identify these. Listening skills, approachable, and respect for people. Ability to work in homogeneous, multi-disciplinary, multi-locational and multicultural teams. Ability to work under supervision and constraints, Understanding of the impact of personal character and behaviors on others.
2. Ability to apply knowledge, ability to integrate the application of knowledge, skills, and sense of responsibilities to new settings and complex problems.
3. Ability to see the self as bound to all humans with ties of recognition and concern. Seek help from other, Ability to help and assist others, mentoring, commitment to others' success. Sensitivity towards global, societal, environmental, moral, ethical and professional issues, and sustainability. Respect for the intellectual property of others. Work ethics.
4. Abstraction and transition between levels of abstraction, representation skills spatial and temporal modeling skills, structuring skills, and theorizing.
5. Algorithmic and structured thinking. Logic, pattern matching, logical what-if analysis, problem decomposition and synthesis, etc.
6. Analytical skills.
7. Communication skills.
8. Constructive criticism.
9. Curiosity, interest in 'how things work' and 'how to create things that work,' interest in the power of technology, humility, observation skills, ability to see things as they are, broader understanding and interests, respect for the classic authors of the great books, openness to constructive criticism, value and readiness for lifelong learning. Active listening skills. Ability to develop a very good understanding of domain specific vocabulary, its semantics, and established thinking patterns.
10. Decision making skills.
11. Design skills.
12. Domain competence.
13. Entrepreneurship, intrinsic motivation to create something, desire to improve things, initiative taking, enjoy challenges, sense of mission, perseverance, concentration, result orientation, commitment, self motivation, dedication, and hard work. Adaptability, flexibility, open-mindedness, and ability to multi-task. Sense of urgency and stress management.
14. Experimentation skills.
15. Good grasping power and attention to detail: breadth, depth, clarity, accuracy, preciseness, specificity, relevance, significance, completeness, consistency.
16. Imagination: storyboarding, extrapolation, visualization, cognitive flexibly: ability to transfer and models of solutions of one situation/field to another, multi-perspective thinking, lateral thinking, inductive thinking, out-of-box thinking, unstructured thinking, creativity and idea initiation, and innovation.
17. Knowledge of contemporary issues and business practices.
18. Knowledge of physical and natural world. Intercultural knowledge.
19. Mentoring, coaching, and training skills.
20. Organizational skills.
21. Persuasion, negotiation, consensus building, and conflict resolution skills.
22. Problem orientation, problem definition and formulation, generations of alternatives. Ability to convert ill-defined problematic situations into software solvable problem. Ability of infusing different thinking patterns developed through their experience in other domains. Inclination for reuse and synthesis by integration. Emphasis on elegant and simple solutions.
23. Problem solving skills: solution implementation and verification.

24. Project planning and management, project scoping, estimation, process planning and management,
25. Quality, cost, and security consciousness, pursuit of excellence, intellectual accountability and responsibility, intellectual integrity, intellectual courage, strength of conviction: assertive without being aggressive. Commitment to systematic documentation of the work. Recognize and act upon the need to consult other experts, especially in matters outside their area of competence and experience. Commitment to the fulfillment of needs of all users and persons who get affected by the technological solutions. Eagerness and inclination to understand the unintended consequences of creating software inappropriate or at odds to its real purposes. Commitment to health, safety, dignity, and welfare of the users and also the people who will be affected by their systems. Sensitivity towards constraints like economic disadvantage and physical disabilities that may limit software accessibility.
26. Reasoning: quantitative and verbal, and critical thinking: ability to question, validate, and correct the purpose, problem, assumptions, perspectives, methods, evidence, inference, reliability, relevance, criteria, and consequences. Numerical ability.
27. Reflection and transition between ladders of reflection. Meta-cognition.
28. Research skills: methods of mathematical research, engineering research, design research, and social science research.
29. Self-acceptance, self-regulation, self-awareness, self-improvement: strength to resist instant gratification in order to achieve better results tomorrow. Being honest and forthright about one's own limitations of competence. Tendency to avoid false, speculative, vacuous, deceptive, misleading, or doubtful claims. Faith in reason and review, inclination for verification and validation, respect for facts and data. Awareness and regulation of automatic thoughts.
30. Systems-level perspective, 'big picture' view, holistic and multi-perspective thinking, knowledge integration, consideration for multilateral viewpoint, and user-centeredness. Process and rule-oriented mindset. Tolerance to ambiguity and risk. Ability to understand and also build upon other's work. Ability to work such that others can easily understand and build upon.
31. Technical competence to solve the software solvable problems using tools and techniques, Use of open source software. Knowledge of industry's best practices and standards, appreciation of what is technically feasible. Identify the risk level of each piece of work.
32. Wealth creation skills.
33. Work load management.

Appendix A3: Revised Survey on Required Competencies, 2007

In 2007, we significantly revised and expanded the list of surveyed competencies from twenty-three (Table A1.1, Appendix A1) to thirty-five. Table A3.1 maps these two set of competencies.

Table A3.1: Comparison of competencies examined in SPINE-based and revised study

Old Competencies (SPINE based study 2004-05) (Table A1.1)		**Revised Competencies (Revised study, 2007) (S.No. as per Table A3.2)**
S.No.		
1	Problem solving	Problem solving skills (8)
2	Analysis/Methodological skills	Analytical skills (5) Attention to detail (10) Experimentation skills (25) Numerical ability (26)
3,4, 8, and 16	Basic engineering proficiency Development know-how Practical engineering experience Specialized engineering proficiency	Ability to apply knowledge (3) Technical competence (7) Design skills (15)
5	Teamwork skills	Ability to work in teams (2)
6, 7, 10, and 15	English Language skills Presentation skills Communication skills Social skills	Listening skills (9) Communication skills (16) Constructive criticism skills (27) Persuasion skills (28)
9 and 18	Leadership skills Management of business process and administration skills	Decision making skills (21) Organizational skills (23) Mentoring skills (24) Ability to assist others through mentoring and philanthropic donations (30) Entrepreneurship (35)
11	Ability to develop own engineering expertise	Readiness for lifelong learning (14)
12	Research know-how	Research skills (17)
13	Ability to develop a broad general education	Knowledge of contemporary issues (32)
14, 19, and 22	Awareness of environmental issues Sensitivity towards socio-economic aspects for sustainable technological development Law	Sensitivity towards global, societal environmental, moral, and ethical issues and sustainability (34)
17	Project management skills	Project planning and management (11)
20 and 21	Finance Marketing	Wealth creation skills (31) Cost consciousness (33)
23	Other language skills	dropped from further investigation
		Perseverance, commitment, and hard work (1) Integrity and authenticity (4) Accountability and responsibility (6) Quality consciousness and pursuit of excellence (12) Critical thinking (13) Adaptability and ability to multi-task (18) "Be the customer" mentality (19) Systems-level perspective (20) Creativity and idea initiation (22) Sense of urgency and stress management (29)

Seventy-one experts working in thirteen companies with additions like Accenture, Borland Software, SUN, and TCS responded to our new survey. The responding experts had industrial experience ranging from 1 year to 22 years, with an average experience of 5.6 years. Data was analyzed in a similar manner to our earlier SPINE-based study (Appendix A1). For classification of competencies we added another category at the top to distinguish the topmost recommendation and termed it as 'Existential.' The normalized figure of merit (NFOM_R_j) for these competencies was ten or very close to ten. Table A3.2 provides the summary of the 2007 results in descending order of the normalized figure of merit.

Table A3.2: Importance of thirty-five competencies as rated by Indian engineers and managers working in Indian and multi-national software companies (Revised Study 2007)

Category	S.No.	Competency (SNo as per Appendix A2)
Existential	1	Perseverance, commitment, and hard work (13)
	2	Ability to work in teams (1)
Pivotal	3	Ability to apply knowledge (2)
	4	Integrity and authenticity (25)
	5	Analytical skills (6)
	6	Accountability and responsibility (25)
	7	Technical competence (31)
	8	Problem solving skills (22 and 23)
Critical	9	Listening skills (1)
	10	Attention to detail (15)
	11	Project planning and management (24)
	12	Quality consciousness and pursuit of excellence (25)
	13	Critical thinking (26)
	14	Readiness for lifelong learning (9)
	15	Design skills (11)
Obligatory	16	Communication skills (7)
	17	Research skills (28)
	18	Adaptability and ability to multi-task (13)
	19	"Be the customer" mentality (1)
	20	Systems-level perspective (30)
	21	Decision making skills (10)
	22	Creativity and idea initiation (16)
Desirable	23	Organizational skills (20)
	24	Mentoring skills (19)
	25	Experimentation skills (14)
	26	Numerical ability (26)
	27	Constructive criticism skills (8)
	28	Persuasion skills (21)
	29	Sense of urgency and stress management (13)
	30	Ability to assist others through mentoring and philanthropic donations (3)
	31	Wealth creation skills (32)
	32	Knowledge of contemporary issues (17)
	33	Cost consciousness (25)
Complimentary	34	Sensitivity towards global, societal, environmental, moral, and ethical issues and sustainability (3)
	35	Entrepreneurship (13)

Appendix A4: Mapping of Thirty-five Competencies (Appendix A3) with Final Set of Twelve Core Competencies

The following tables give the mapping of thirty-five competencies of Table A3.2 (Appendix A3) with our finally reduced set of twelve competencies given in Table 3.1.

Table A4.1a: Mapping of thirty-five competencies with the Final set of twelve core competencies, part –I

S.No.	Core Competencies identified in 2007 (Table 2.6)	Subsuming Twelve Core Competencies (Table 3.1) (S.No as per Table 3.1)
1	Perseverance, commitment, and hard work	Reflective thinking (7) Decision making perspective (10) Systems-level perspective (11)
2	Ability to work in teams	Communication competence (4) Systems-level perspective (11)
3	Ability to apply knowledge	Technical competence (1)
4	Integrity and authenticity	Decision making perspective (10)
5	Analytical skills	Domain Competence (3) Computational thinking (2) Critical thinking (7) Systems-level perspective (11)
6	Accountability and responsibility	Decision making perspective (10) Systems-level perspective (11)
7	Technical competence	Technical competence (1)
8	Problem solving skills	Complex problem solving competence (5) Computational thinking (2) Domain Competence (3) Intrinsic motivation to create/improve artifacts (12) Creativity and Innovation (8) Systems-level perspective (11)
9	Listening skills	Communication competence (4) Attention to detail (6) Critical thinking (7) Systems-level perspective (11)
10	Attention to detail	Attention to details (6) Computational thinking (2)
11	Project planning and management	Domain Competence (3) Decision making perspective (10) Complex problem solving competence (5)
12	Quality consciousness and pursuit of excellence	Attention to detail (6) Critical and reflective thinking (7) Systems-level perspective (11)
13	Critical thinking	Critical thinking (7) Computational thinking (2)
14	Readiness for lifelong learning	Curiosity (9) Intrinsic motivation to create/improve artifacts (12)
15	Design skills	Technical competence (1) Domain competence (3) Computational thinking (2) Intrinsic motivation to create/improve artifacts (12) Creativity and Innovation (8) Reflective thinking (7) Systems-level perspective (11)

Table A4.1b: Mapping of thirty-five competencies with the Final set of twelve core competencies, part-II

S.No	Core Competencies identified in 2007 (Table 2.6)	Subsuming Twelve Core Competencies (Table 3.1) (S.No as per Table 3.1)
16	Communication skills	Communication competence (4)
17	Research skills	Curiosity (9) Intrinsic motivation to create/improve artifacts (12) Critical and reflective thinking (7) Creativity and Innovation (8)
18	Adaptability and ability to multi-task	Systems-level perspective (11)
19	"Be the customer" mentality	Attention to detail (6) Curiosity (7) Domain competence (3) Communication competence (4) Systems-level perspective (11)
20	Systems-level perspective	Systems-level perspective (11)
21	Decision making skills	Decision making perspective (10)
22	Creativity and idea initiation	Intrinsic motivation to create/improve artifacts (12) Creativity and Innovation (8) Computational thinking (2) Domain Competence (3)
23	Organizational skills	Systems-level perspective (11) Communication competence (4)
24	Mentoring skills	Communication competence (4) Curiosity (9) Critical and reflective thinking (7)
25	Experimentation skills	Attention to detail (6) Curiosity (9) Critical and reflective thinking (7) Intrinsic motivation to create/improve artifacts (12)
26	Numerical ability	Technical competence (1)
27	Constructive criticism skills	Critical and reflective thinking (7) Systems-level perspective (11)
28	Persuasion skills	Communication competence (4) Decision making perspective (10)
29	Sense of urgency and stress management	Complex problem solving competence (5) Decision making perspective (10) Systems-level perspective (11)
30	Ability to assist others through mentoring and philanthropic donations	Systems-level perspective (11) Reflective thinking (7)
31	Wealth creation skills	Dropped
32	Knowledge of contemporary issues	Curiosity (9)
33	Cost consciousness	Critical thinking (7) Systems-level perspective (11)
34	Sensitivity towards global, societal, environmental, moral, and ethical issues and sustainability	Systems-level perspective (11) Reflective thinking (7)
35	Entrepreneurship	Decision making perspective (10) Intrinsic motivation to create/improve artifacts (11) Creativity and Innovation (8) Reflective thinking (7)

Appendix A5: Catalogue of Technical and Technically Oriented Activities Related to Software Development

1. Overarching Activities:	2. Ubiquitous Activities:
Technology Entrepreneurship Program Management Infrastructure Management and Maintenance (Operations Management) Contract Management Partnership/Outsourcing/Vendor Development Procurement Process Quality Assurance and Control	Measurement Technical Documentation and Presentation Innovation Research Presenting Ideas and Insights Configuration Management Product Quality Assurance and Control Knowledge Management Training and Talent Development Group Work, People Management, and Leadership Idea Convergence

3. Client Interface:	4. Design:	5. Realization:
Technical Marketing Consulting Feasibility Study Work flow/Process Study and Modeling Visualization Knowledge Elicitation Requirement Engineering Migration Assessment Test assessment Product/Requirement Definition and Specification Business Technology Alignments Deployment and roll out User Acceptance and Usability Analysis User interface Design End User Documentation Customer Support Infrastructure planning	Prototyping Component and interface Design Component Selection Algorithm/Computational Procedure Design Architecting Application Design Service Design Product Design System Design Network Design Process Design Infrastructure Design Security Architecture Design Process Tailoring Test Design Content Design Standardization Restructuring Intellectual Property Management	Application Customization Application Development Component Development Product Development Service Development System Integration Infrastructure Setup Process Implementation and Change Management Code Analysis Build and Release Management Validation and Verification (Testing) Maintenance, Enhancement, Up-gradation, Porting Data Migration Technology Migration Performance Tuning System Administration Database administration Network administration Security administration Service Management Standards and regulatory Compliance Program Comprehension and re-documentation Reconstruction Code Archaeology Disaster Recovery Production Support

6. Planning:	7. Evaluation:
Time to Market Planning Estimation and Costing Resource Planning and Management Project Scheduling Risk Planning and Mitigation Staffing and Team Development Project Monitoring and Control	Application Audit Process Audit Technology Audit Tools and Technology Selection and Evaluation Architecture Evaluation Impact Analysis Value Analysis Usability Analysis

Appendix A6: Taxonomy of Common Software Bugs

Programming Fundamentals related bugs Data loss bugs, data overflow bugs, operator precedence related bugs, string handling bugs, multi-way branch related bugs, logical operators related bugs, arithmetical function or operator related bugs, function-like macro bugs, not checking return value for success or failure, etc.
Operating system related bugs b1. Memory related Bugs: b1.1. Stack Corruption: local buffer overrun, returning a pointer to a automatic variable that has gone out of scope, declaring local storage which exceeds the size of stack of the process, function arguments passed are too large to be accommodated on the stack, etc. b1.2. Heap Corruption: dynamically allocated buffer over-run, freeing already freed memory, freeing memory not allocated dynamically, etc. b1.3. Invalid Memory Access: NULL pointer access, uninitialized memory access, dereferencing pointer to freed memory, etc. b1.4. Memory Leaks: Memory allocated but not freed in all legs of error handling, allocated memory handle changed, freeing array of pointer holding dynamically allocated memory, etc. b2. Synchronization related bugs: lack of or inconsistent synchronization, lock acquired but not released in all scenarios, taking recursive lock which is not supported by OS, taking recursive lock which is supported by OS but not unlocking correspondingly, blocking call from ISRs etc. b3. Inter-process or inter-thread related bugs: shared file or socket closed by one thread and being accessed by other threads or parent process, and heap memory corrupted by one thread result in malfunctioning of some other thread, priority inversion, etc.
Compiler related bugs c1. Data Structure padding related bugs: access structure members as raw memory, and structures used for message communication on different machine, etc. c2. Source code optimization related bugs: accessing shared memory through pointer, and accessing memory mapped input/output ports of device, etc. c3. Object-oriented Language Support related bugs: default constructor construction, default copy constructor for class with pointer member, constructor for class with compiler generated internal members, erroneous reference counting due to named return value optimization, initializing a class member with another class member using member initialization list, lack of virtual destructor in base class, incorrect usage of delete for deleting array of class objects, throwing exception without proper cleanup, deleting array of derived class objects with base class pointer, lifetime of compiler generated temporary objects, and throwing an exception from destructor, etc.
Software architecture related bugs Lack of validation of input parameters, error handling, deadlock, live-lock, reentrant function, concurrency, parallelism, memory fragmentation, etc.

Appendix A7: Proposed Curriculum for Masters in Archaeo-heritage Informatics

In 2005, we also proposed the design of a two year master's program in *Archaeo-heritage Informatics* [177]. We viewed information technology as an enabler to enhance productivity, quality, efficacy, creativity, and also to facilitate integration of innovative ideas in the archaeological activities of survey, testing, research, conservation, restoration, dissemination, and management. The courses included three streams: (a) art, archaeology, anthropology, and heritage studies, (b) computing and multimedia, and (c) communication and management. In the specific context of Indian universities, we proposed the scheme given below"

Ist semester: (i) basic computing tools, (ii) digital media, (iii) oriental philosophy and linguistics, (iv) art, architectural aesthetics, and design, (iv) anthropology, and (vi) learning methods.

2nd semester: (i) web-enabled content creation, (ii) computer based visualization, (iii) computer aided qualitative and quantitative analysis, (iv) archaeo-heritage documentation methods, (v) archaeo-heritage conservation methods, (vi) group project.

3rd semester: we proposed to include (i) GIS and digital field methods for archaeology, (ii) digital library, (iii) knowledge management, (iv) entrepreneurship, (v) archaeo-heritage research methods, and (vi) group project.

4th semester: (i) quality and creativity management, (ii) design methods, and (iii) dissertation.

Appendix A8: Some Suggestions for Breadth Courses

Within the context of many knowledge disciplines in sciences, mathematics, engineering, management, social sciences, and humanities, a body of knowledge has already been created around systems and systems thinking. In this context, *operations research and mathematical theory of systems* are well developed areas in mathematics. *Control systems, system engineering, and modeling and simulation* can be engineering departments' contribution for this purpose. *Biological subjects* are traditionally organized in terms of various systems, and over the last few years, the area of computational modeling of biological systems has also been well developed. Topics on *statistical physics, complex system physics and nonlinear physics* can be considered for exposing students to think in terms of large complex systems. With reference to social sciences, humanities, and management, courses on the *history of ideas, diversity of human languages, comparative economic systems, world cultures, world epics, socio-cultural systems analysis, operations management systems, business process modeling*, etc., offer huge potential to engage students in systems thinking. Richness and diversity of such exposure will reinforce systems thinking and help in developing the ability to learn new domains.

Appendix A9: Inadequate Development of Curiosity in Software Development Education

In 2005, we carried out an empirical study through a detailed questionnaire on nature of questioning in the class. Twenty-nine undergraduate students of computer science and engineering and information technology gave their responses. A summary of their responses is given in Table A9.1.

Table A9.1: A summary of students' responses on 'questioning in the class'

1. Learning is a consequence of thinking, and knowing facts is only small part of it.
2. Most students like to depend more on self study than on lecture classes and they attend classes of most teachers mostly to meet the attendance requirements.
3. Only a few teachers ask sufficient number of questions during lecture classes. Most teachers do not normally ask more than three questions in a one hour lecture class. However, some teachers may ask even up to eight questions during the same duration.
4. Only some teachers give sufficient wait-time (at least few seconds) before calling a student to answer their questions during their lectures.
5. Only a few teachers ask questions that helped them to think and learn. Most questions asked by most teachers are related to facts, syntax, formula, procedure or recall that do not require deep thinking. They ask questions to check if students are attentive and are following them, to keep the class interactive, to revise, to create interest, to boost the morale of students. When most teachers ask questions during their lectures, they usually have a "right" answer in mind, and they do not want to hear what students think; they just want to hear that answer only. Very few actually ask questions to provoke the students' mind to think beyond the point where the teacher stops in the class.
6. Very few teacher questions enhance creative/analytical thinking, or promote teamwork.
7. Only a few teachers typically wait for at least few seconds before speaking after a student has answered their questions during their class. Further, only a few teachers typically explain and critique students' answers.
8. When students give an incorrect answer to a question, only a very few teachers try to find out why students answered as they did.
9. Very few teachers help students to expand their initial answers through more probing conversations or help them through cues and clues.
10. Only some students take initiative to ask questions when confused or curious, and very few asked questions that required thinking and contribute to classroom discussions.

Software developers' views

Responding to the state of curiosity development in undergraduate computing education, a software project manager from Romania made the observation, *"…neither undergraduate nor graduate computing education are doing enough for enhancing this curiosity. In order to find answers, it is best to always have with you a set of questions. The real problem is if he can generate an interesting set of questions…"* A senior software engineer (educated in India, working in USA) commented, *"Curiosity is what you have as your nature. …college education can absolutely not help you here on it's own. This is a human nature and college is of no help here..."*

Appendix A10: Survey: "Software developers - (How) Did your college help you in your development?"

A. Effectiveness of Teaching Methods: Survey of Software Developers (2009)

In 2009, our study on teaching methods (Appendix A1) was further extended and refined by refining and adding a few more teaching methods. Through the online global community LinkedIn.com, and online surveying tool surveymonkey.com, we conducted a survey, "Software developers - (How) Did your college help you in your development?" among working software professionals.

We asked them to rate various educational experiences of college studies with respect to their direct/indirect contribution for respondent's later technical/professional/academic activities in terms of skill, knowledge, problem solving methodology, mindset, thinking, habits, values, etc. We received 67 responses out of which 49 also revealed their identity and affiliations. These respondents are working in many companies and have varied experience levels. Some of them have more than twenty years of experience. We offered them a list of twelve types of educational experiences. We asked them to associate each of these experiences with six choices: (i) extremely useful, (ii) mostly were useful, (iii) many were useful, (iv) some were useful, (v) not useful, and (vi) rarely/never experienced during college studies. We assigned a decreasing numeric value ranging from 4-0 to first five of these options, and a zero to the last option. Table A10.1 shows the results of this survey in the descending order of average rating.

Table A10.1: Effectiveness of educational experiences for competency enhancement of software developers
"Software developers - (How) Did your college help you in your development?"

	Extremely useful **(4)**	Mostly were useful **(3)**	Many were useful **(2)**	Some were useful **(1)**	Not useful **(0)**	Rarely/never experienced during college studies **(0)**	Rating Avg **(0-4)**
1. Projects	**61.2% (41)**	23.9% (16)	9.0% (6)	6.0% (4)	0.0% (0)	0.0% (0)	3.40
2. Laboratory work	**38.8% (26)**	35.8% (24)	10.4% (7)	9.0% (6)	3.0% (2)	3.0% (2)	2.99
3. Discussions with other students	**35.8% (24)**	**35.8% (24)**	16.4% (11)	9.0% (6)	1.5% (1)	1.5% (1)	2.96
4. Teaching peers/juniors	**32.8% (22)**	31.3% (21)	16.4% (11)	7.5% (5)	3.0% (2)	9.0% (6)	2.84
5. Thinking and work oriented Lectures	**32.8% (22)**	25.4% (17)	23.9% (16)	11.9% (8)	1.5% (1)	4.5% (3)	2.76
6. Discussions with Faculty	**31.3% (21)**	28.4% (19)	14.9% (10)	11.9% (8)	4.5% (3)	9.0% (6)	2.70
7. Industrial Training.	**31.3% (21)**	23.9% (16)	14.9% (10)	11.9% (8)	7.5% (5)	10.4% (7)	2.60
8. Research Literature survey oriented assignments	20.9% (14)	**32.8% (22)**	20.9% (14)	13.4% (9)	3.0% (2)	9.0% (6)	2.55
9. Discussions with others	16.4% (11)	**31.3% (21)**	19.4% (13)	22.4% (15)	1.5% (1)	9.0% (6)	2.39
10. Homework and Tutorial	13.4% (9)	20.9% (14)	**26.9% (18)**	20.9% (14)	14.9% (10)	3.0% (2)	1.97
11. Knowledge transmission oriented Lectures (explain and follow the textbooks)	10.4% (7)	14.9% (10)	29.9% (20)	**35.8% (24)**	4.5% (3)	4.5% (3)	1.91
12. Written examinations and required preparation	10.4% (7)	14.9% (10)	28.4% (19)	**32.8% (22)**	9.0% (6)	4.5% (3)	1.85

Usually, the traditional educational systems and approach over-emphasize three educational methods: (i) knowledge transmission oriented lectures, (ii) homework and tutorials, and (iii) written examination and required preparation. Very interestingly, as shown in Table A10.1, these most valued methods were found to be the least valuable by our respondents for contributing to the development of their skill, knowledge, problem solving methodology, mindset, thinking, habits, values, etc., for their later technical/professional/academic activities. These were the only three methods that were found to have an average rating of less than 2 on a scale of 0 to 4. That means that a good number of our respondents found only some or none of these

methods to be helpful in their multidimensional development. *Project work, laboratory work, discussions with other students, thinking and work oriented lectures, and teaching peers/juniors were rated as the most valuable educational experiences.* All these experiences are learner-centric, whereas the least rated three experiences are essentially teacher-centric. These findings further validated our earlier SPINE-like study discussed above.

A1. Effectiveness of Teaching Methods-II: Effect on Desired Competencies

In this survey, we had also asked them to rate the effectiveness of these pedagogical engagements for developing specific competencies, as discussed in Chapter 3. The results of this survey have been discussed in twelve competency specific sections of Chapters 4 to 6. Table A10.2 provides the results of this part of this survey. In the first part of this table, Table A10.2 (i) part–I and part-II, we summarize the responses about the basic eleven competencies, identified at the time of this survey. These eleven competencies were later revised into five basic competencies, listed in Table 3.2. In the second part of this table, Table A10.2 (ii), we summarize the responses about the basic three competency driver-habits of mind, identified at the time of this survey. The competencies are also listed in Table 3.2. In the last part of this Table, Table A10.2 (iii), we summarize the responses about the six competency conditioning attitudes and values, identified at the time of this survey. These were later revised into four competency conditioning attitudes and perspectives, as listed in Table 3.2.

Table A10.2 (i) part-I: Perceived effectiveness of pedagogical engagements with respect to enhance of specific competencies – basic competencies: perceptions of software professionals
"Software developers - (How) Did your college help you in your development?"

Competencies – basic competencies:
1. Technical Competence (1st competency in Table 3.2)
1A. Analytical skills (included in 1st competency in Table 3.2)
1B. Design skills (included in 1st competency in Table 3.2)
1C. Implementation skills (included in 1st competency in Table 3.2)
1D. Debugging skills (included in 1st competency in Table 3.2)

Pedagogical Engagements	**1**	**1A**	**1B**	**1C**	**1D**
Total Number of Responses	**50**	**48**	**49**	**49**	**49**
Knowledge transmission oriented Lectures (explain and follow the textbooks)	**54.0% (27)**	8.3% (4)	18.4% (9)	8.2% (4)	6.1% (3)
Thinking and work oriented Lectures	40.0% (20)	**54.2% (26)**	46.9% (23)	6.1% (3)	6.1% (3)
Home work and Tutorials	48.0% (24)	41.7% (20)	20.4% (10)	24.5% (12)	20.4% (10)
Written examinations and required preparation	36.0% (18)	25.0% (12)	8.2% (4)	16.3% (8)	4.1% (2)
Research Literature survey oriented assignments	32.0% (16)	**58.3% (28)**	28.6% (14)	20.4% (10)	10.2% (5)
Laboratory work	**70.0% (35)**	**62.5% (30)**	**61.2% (30)**	**83.7% (41)**	**85.7% (42)**
Projects	**76.0% (38)**	**75.0% (36)**	**91.8% (45)**	**89.8% (44)**	**83.7% (41)**
Industrial Training	36.0% (18)	33.3% (16)	49.0% (24)	49.0% (24)	34.7% (17)
Teaching peers/juniors	32.0% (16)	20.8% (10)	22.4% (11)	20.4% (10)	30.6% (15)
Discussions with other students	38.0% (19)	37.5% (18)	26.5% (13)	24.5% (12)	24.5% (12)
Discussions with Faculty	36.0% (18)	14.6% (7)	28.6% (14)	12.2% (6)	8.2% (4)
Discussions with others	10.0% (5)	8.3% (4)	14.3% (7)	4.1% (2)	4.1% (2)

Table A10.2 (i) part-II: Perceived effectiveness of pedagogical engagements with respect to enhance of specific competencies – basic competencies: perceptions of software professionals
"Software developers - (How) Did your college help you in your development?"

Competencies – basic competencies:

2. Communication skills (4th competency in Table 3.2)
3. Domain Competence (3rd competency in Table 3.2)
4A. Abstraction and transition between levels of abstraction (included in 2nd competency, Computational thinking, in Table 3.2)
4B. Algorithmic and structured thinking (included in 2nd competency in Table 3.2)
5. Problem solving: ability to synthesize other competencies in the context of new settings and complex problems. Ability to convert ill-defined problematic situations into software solvable problem. Project scoping and estimation (included in 5th competency in Table 3.2)

Pedagogical Engagements	**2**	**3**	**4A**	**4B**	**5**
Total Number of Responses	**49**	**49**	**47**	**50**	**51**
Knowledge transmission oriented Lectures (explain and follow the textbooks)	12.2% (6)	**51.0% (25)**	12.8% (6)	36.0% (18)	17.6% (9)
Thinking and work oriented Lectures	20.4% (10)	28.6% (14)	38.3% (18)	**60.0% (30)**	**51.0% (26)**
Home work and Tutorials	8.2% (4)	34.7% (17)	21.3% (10)	36.0% (18)	37.3% (19)
Written examinations and required preparation	12.2% (6)	30.6% (15)	12.8% (6)	28.0% (14)	23.5% (12)
Research Literature survey oriented assignments	8.2% (4)	**51.0% (25)**	40.4% (19)	40.0% (20)	35.3% (18)
Laboratory work	8.2% (4)	38.8% (19)	31.9% (15)	**58.0% (29)**	**58.8% (30)**
Projects	22.4% (11)	**61.2% (30)**	**57.4% (27)**	**72.0% (36)**	**78.4% (40)**
Industrial Training	44.9% (22)	26.5% (13)	19.1% (9)	26.0% (13)	33.3% (17)
Teaching peers/juniors	**71.4% (35)**	30.6% (15)	21.3% (10)	24.0% (12)	25.5% (13)
Discussions with other students	**83.7% (41)**	26.5% (13)	19.1% (9)	22.0% (11)	49.0% (25)
Discussions with Faculty	**69.4% (34)**	28.6% (14)	21.3% (10)	22.0% (11)	31.4% (16)
Discussions with others	**51.0% (25)**	18.4% (9)	4.3% (2)	6.0% (3)	5.9% (3)

Table A10.2 (ii): Perceived effectiveness of pedagogical engagements with respect to enhance of specific competencies – habits of mind: perceptions of software professionals
"Software developers - (How) Did your college help you in your development?"

Competencies – habits of mind: 6. Attention to details (6th competency in Table 3.2) 7. Critical and reflective thinking(7th competency in Table 3.2) 8. Creativity and innovation (8th competency in Table 3.2)			
Pedagogical Engagements	**6**	**7**	**8**
Total Number of Responses	**51**	**50**	**51**
Knowledge transmission oriented Lectures (explain and follow the textbooks)	17.6% (9)	14.0% (7)	7.8% (4)
Thinking and work oriented Lectures	21.6% (11)	48.0% (24)	**52.9% (27)**
Home work and Tutorials	27.5% (14)	10.0% (5)	17.6% (9)
Written examinations and required preparation	27.5% (14)	14.0% (7)	3.9% (2)
Research Literature survey oriented assignments	37.3% (19)	42.0% (21)	45.1% (23)
Laboratory work	35.3% (18)	24.0% (12)	39.2% (20)
Projects	**70.6% (36)**	**50.0% (25)**	**82.4% (42)**
Industrial Training	33.3% (17)	18.0% (9)	29.4% (15)
Teaching peers/juniors	37.3% (19)	30.0% (15)	31.4% (16)
Discussions with other students	23.5% (12)	44.0% (22)	45.1% (23)
Discussions with Faculty	21.6% (11)	44.0% (22)	39.2% (20)
Discussions with others	5.9% (3)	22.0% (11)	21.6% (11)

Table A10.2 (iii): Perceived effectiveness of pedagogical engagements with respect to enhance of specific competencies – attitudes and values: perceptions of software professionals
"Software developers - (How) Did your college help you in your development?"

Competencies - attitudes and values:

9: Curiosity with humility: self-learning, ability to develop good understanding of domains' vocabulary, semantics, and thinking processes, faith in reason, and review. (9th competency in Table 3.2)

10: Decision making. (10th competency in Table 3.2)

10A. Project planning and management (included in 10th competency in Table 3.2)

11: Systems-level perspective: Inclination for reuse and synthesis by integration, to build upon others' work. (11th competency in Table 3.2)

11A: Ability to accommodate himself to others. Ability to work such that others can easily understand and build upon. (included in 11th competency in Table 3.2)

11B: Accountability and responsibility: Ability to see the self as bound to other (all) humans with ties of concern, sensitivity towards global, societal, environmental, moral, ethical, professional issues, and sustainability. Strength of conviction & self-regulation. (included in 11th competency in Table 3.2)

12: Urge to create/ improve things and open-mindedness. (12th competency in Table 3.2)

Pedagogical Engagements	**9**	**10**	**10A**	**11**	**11A**	**11B**	**12**
Total Number of Responses	**50**	**48**	**48**	**50**	**50**	**49**	**50**
Knowledge transmission oriented Lectures (explain and follow the textbooks)	26.0% (13)	6.3% (3)	12.5% (6)	10.0% (5)	6.0% (3)	14.3% (7)	14.0% (7)
Thinking and work oriented Lectures	42.0% (21)	31.3% (15)	14.6% (7)	24.0% (12)	8.0% (4)	26.5% (13)	**54.0% (27)**
Home work and Tutorials	26.0% (13)	20.8% (10)	12.5% (6)	22.0% (11)	12.0% (6)	20.4% (10)	16.0% (8)
Written examinations and required preparation	12.0% (6)	8.3% (4)	6.3% (3)	4.0% (2)	4.0% (2)	16.3% (8)	6.0% (3)
Research Literature survey oriented assignments	**62.0% (31)**	29.2% (14)	16.7% (8)	46.0% (23)	14.0% (7)	24.5% (12)	**58.0% (29)**
Laboratory work	38.0% (19)	37.5% (18)	27.1% (13)	34.0% (17)	24.0% (12)	30.6% (15)	42.0% (21)
Projects	**66.0% (33)**	**77.1% (37)**	**89.6% (43)**	**68.0% (34)**	**64.0% (32)**	**51.0% (25)**	**74.0% (37)**
Industrial Training	36.0% (18)	35.4% (17)	**70.8% (34)**	32.0% (16)	38.0% (19)	40.8% (20)	30.0% (15)
Teaching peers/juniors	32.0% (16)	35.4% (17)	29.2% (14)	26.0% (13)	**56.0% (28)**	44.9% (22)	44.0% (22)
Discussions with other students	36.0% (18)	31.3% (15)	16.7% (8)	28.0% (14)	46.0% (23)	30.6% (15)	**50.0% (25)**
Discussions with Faculty	36.0% (18)	29.2% (14)	12.5% (6)	18.0% (9)	22.0% (11)	32.7% (16)	**50.0% (25)**
Discussions with others	14.0% (7)	18.8% (9)	12.5% (6)	8.0% (4)	20.0% (10)	28.6% (14)	24.0% (12)

B. Effectiveness of Teaching Methods: Survey of Students (2009)

The finding of Table A10.1 were also further validated through an almost similar survey among the final year (seventh semester) computing students at Jaypee Institute of Information Technology. Both SPINE-like studies showed that projects were the most valuable educational experience with reference to later professional activities. Hence, we asked the students to rate the effectiveness of their earlier educational experiences with respect to its contribution on their final year project. We asked them to rate the following educational experiences of the last 3+ years with respect to their direct/indirect contribution for this project in terms of skill, knowledge, problem solving methodology, mindset, thinking, habits, etc. There was a slight modification in the list of the educational experiences. Since, as a department, we have been using all the methods listed in Table A10.3, we dropped the last option of 'rarely/never experienced during college studies' in this survey. The respondents, who did not respond to some option, were treated as 'no comments' for that educational experience with a numeric value of zero. The first five options were used for this survey. We received a total of 210 responses. Table A10.3 shows the results of this survey.

Table A10.3: Effectiveness of educational experiences for competency enhancement of computing students

	Extremely useful (4)	Mostly were useful (3)	Many were useful (2)	Some were useful (1)	Not useful (0)	No comments (0)	Rating Average (0-4)
13. Minor project-I/Minor project-II of 3rd year	**39% (80)**	31% (65)	13% (27)	13% (27)	4% (8)	3	2.8
14. Mini projects as part of specific courses	31% (65)	**37% (77)**	18% (38)	11% (22)	3% (7)	1	2.8
15. Laboratory work (during laboratory classes)	28% (58)	**32% (67)**	25% (53)	11% (24)	4% (8)	-	2.7
16. Industrial Training	**33% (68)**	25% (52)	14% (30)	15% (32)	13% (27)	1	2.5
17. Developmental work (for laboratory classes)	24% (49)	**29% (49)**	**25% (52)**	19% (40)	3% (6)	4	2.5
18. Discussions with faculty	30% (53)	**35% (62)**	22% (40)	12% (21)	1% (2)	32	2.4
19. Literature survey oriented assignments	15% (31)	21% (43)	**40% (84)**	20% (41)	5% (10)	1	2.2
20. Discussions with peers/seniors	23% (41)	28% (50)	**25% (45)**	18% (33)	6% (10)	31	2.1
21. Lectures	7% (14)	21% (44)	31% (54)	**35% (74)**	6% (13)	1	1.9
22. Tutorial	10% (21)	21% (43)	24% (50)	**28% (58)**	18% (37)	1	1.8
23. Written examination and required preparation	8% (17)	17% (36)	24% (49)	**33% (68)**	18% (37)	3	1.6
24. Mentoring juniors	10% (17)	16% (29)	**32% (57)**	29% (51)	13% (24)	32	1.5

Broadly speaking, the result of this survey also reconfirms the supremacy of projects and laboratory work as the best educational experiences with reference to their contribution for final year project in terms of skill, knowledge, problem solving methodology, mindset, thinking, habits, etc. In the same context, it also reconfirms the inadequacy of lecture, tutorial (homework), and written examination and required preparation. All these are teacher-centric activities. It is very interesting to note that the students find discussions with faculty as very useful for their project, where their response for lecture is very poor. This result in Table A10.3 has one significant variation with respect to the result of Table A10.1. The lowest rating of mentoring juniors is attributed to the fact that a good number of the respondents gave no

comments for this experience. Mentoring juniors is a student-centric activity for the senior students. The details of this scheme are discussed in Section 9.2.3.2. A large fraction of 58% of the students found that their experiences in mentoring of juniors were either extremely useful, mostly useful, or many were useful with reference to their project work. The effect of 'mentoring the juniors/peers' experiences on enhancement of specific competencies as perceived by working professionals has been discussed in the fourth, fifth, and sixth chapters. As per the report of the faculty, nearly 50% of the final year students very seriously participate in the mentoring program. We can interpret that most of those who had enthusiastically participated in the mentoring program, found that experience useful even for their final year project.

Appendix A11: Empirical Examination of Software Development Education Through Bloom's Taxonomy

The main aim of this study conducted by us in 2003 [11-12], was to empirically understand the degree to which the formal components of the traditional teaching-learning-evaluation process in engineering education succeed in creating opportunities for enhancing various competencies in terms of Bloom's taxonomy.

Activity Verbs for Bloom's Cognitive Levels

Several authors, e.g., Bloom [133], Krumme [305], and TALS [306] have reported mappings of activity verbs to different Bloom levels. Existing Bloom-level-to-activity-verb-lists mappings were extended to include the verbs that were not found in the current literature. Table A11.1 gives the list of verbs used for this study.

Table A11.1: List of verbs used for assessing engineering education wrt Bloom's taxonomy

Level 1 - Remember: acquire, cite, define (studied definitions), derive, fill in the blanks, identify, label, list, name, obtain, prove (studied theorem, studied method), recall, recite, recognize, reproduce, show (studied fact, studied method), and state. **Level 2 - Understand:** arrange, associate, categorize, change, clarify, classify, compare, convert, describe, discuss, distinguish, draw, exemplify, explain, illustrate, interpret, match, outline, rephrase, represent, restructure, rewrite, sort, summarize, tell, and translate. **Level 3 - Apply:** apply, calculate, compute, demonstrate, determine, estimate, evaluate (computation), experiment, find, practice, show (understanding fact in the direct context of studied material), solve, and transform. **Level 4 - Analyze:** analyze, conclude, contrast, debug, deduce, detect, differentiate, discriminate, examine, extend, extrapolate, generalize, infer, justify, point out, predict, rearrange, select, specify, test, and verify. **Level 5 - Create:** build, combine, comment, compose, constitute, construct, correlate, create, define (new things), design, develop, devise, document, formulate, implement, integrate, modify, organize, plan, prepare, present, produce, propose, prove (unstudied things), reorganize, report, revise, schedule, sketch, and synthesize. **Level 6 - Evaluate:** appraise, argue, assess, decide, evaluate (the options), judge, question, review, revisit, standardize, validate, value, and weigh.

A survey was conducted among two groups of engineering students and professional engineers. These three groups were requested to respond to three different but complimentary questions around a unified and alphabetically sorted list of activity verbs. The first group of about fifty 2nd year Computer Science and Information Technology students was asked to select and individually rank the identified verbs based on the frequency of their use in the teaching-learning-evaluation process. A second group of sixteen students was asked to rank the verbs

according the learning effectiveness of the verbs. Thirteen professional engineers were requested to select and rank 10-15 verbs, that if used more often by the faculty, will help in better preparing the students for professional life.

Their responses were collated into three different groups, and a group rating was calculated for every verb. A combined rating of group perception about a verb was statistically extracted from individual ranks: where a large numerical value of the combined rating by the first group of students would imply a perception of high usage of that verb, and a smaller numerical value would imply infrequent or zero usage. A high numerical value for the combined rating assigned by the second group of students would imply that most of them learn more when that verb is used to communicate the activity for evaluative or non-evaluative tasks, and a small numerical value would imply that few or none of them experience effective learning when that verb is used. Similarly, a high numerical value for the combined rating assigned by professional engineers' would imply that most of them want the verb to be used often, and a small numerical value would imply that few or none of them recommend it to become or continue as a commonly used verb in administering evaluative or non-evaluative tasks.

Respondents assigned contiguous natural numbers starting from 1 without any specified upper limit as ranks to the verbs of their choice. Some chose to give a unique rank to every verb thereby assigning ranks in the range of 1 to around 50. Many chose to give a common rank to many verbs in the range of 1 to around 10. They had the freedom of not assigning any rank to some verbs. A lower numerical value implies higher ranking, 1 being the highest rank.

Verb-specific group ratings, $V_{Rj\text{-student-I}}$, $V_{Rj\text{-student-II}}$, and $V_{Rj\text{-professional}}$, are defined as follows:
$V_{Rj\text{-student-I}}$ is the sum of the multiplicative inverse of valid ranks for the j^{th} verb by the first group of students, i.e.:

$V_{Rj\text{-student-I}} = \sum_{k=1 \text{ to } 50} (1/Rank_{kj})$ Where $Rank_{kj} \neq 0$, and represents the perceived usage rank given by the k^{th} student to the j^{th} verb. There were 50 student respondents.

$V_{Rj\text{-}student\text{-}II}$ is the sum of the multiplicative inverse of valid ranks for the j^{th} verb by second group of students, i.e.:

$V_{Rj\text{-}student\text{-}II} = \sum_{k=1 \text{ to } 16} (1/Rank'_{kj})$ Where $Rank'_{kj} \neq 0$, and represents the perceived learning effectiveness rank given by the k^{th} student to the j^{th} verb. There were 16 student respondents.

$V_{Rj\text{-}professional}$ is the sum of the multiplicative inverse of valid ranks for the j^{th} verb by professional engineers, i.e.:

$V_{Rj\text{-}Professional} = \sum_{k=1 \text{ to } 13} (1/Rank''_{kj})$ Where $Rank''_{kj} \neq 0$, and represents the recommended usage rank given by the k^{th} professional engineer to the j^{th} verb. There were 13 professional respondents.

Verb-specific group ratings, $V_{Rj\text{-}student\text{-}I}$, $V_{Rj\text{-}student\text{-}II}$, and $V_{Rj\text{-}professional}$ were then normalized with respect to the maximum values of VRj-student-I, VRj-student-II, and VRj-professional respectively, to calculate activity verb-specific normalised group ratings as follows:

$$V'_{Rj\text{-}student\text{-}I} = V_{Rj\text{-}student\text{-}I} / max_j \{V_{Rj\text{-}student\text{-}I}\}$$
$$V'_{Rj\text{-}student\text{-}II} = V_{Rj\text{-}student\text{-}II} / max_j \{V_{Rj\text{-}student\text{-}II}\}$$
$$V'_{Rj\text{-}professional} = V_{Rj\text{-}professional} / max_j \{V_{Rj\text{-}professional}\}$$

Hence, $V'_{Rj\text{-}student\text{-}I}$, $V'_{Rj\text{-}student\text{-}II}$, and $V'_{Rj\text{-}professional}$ all have a value between 0 to 1. Values close to 1 indicate that most respondents from the specific category have assigned a high rank to the jth verb, whereas low values indicate low ranks by most of the respondents. Table A11.2 gives ordered lists of activity verbs as per their usage rating by students, learning effectiveness by students, and professional engineers' recommendations. These lists are in descending order $V'_{Rj\text{-}student\text{-}I}$, $V'_{Rj\text{-}student\text{-}II}$, and $V'_{Rj\text{-}professional}$ respectively. The first list in Table A11.2 indicates that most faculty members assigned activities directly asking students to calculate, explain, prove (studied theorem, studied method), define (studied definitions), write, solve, compute, show (studied fact, studied method), evaluate (computation), or derive. The second list in Table A11.2 indicates that most students experience maximum learning when asked to design, analyse, understand, build, apply, adapt, implement, create, develop, or demonstrate. The third list in Table A11.2 indicates that professional engineers recommended that the faculty should repeatedly direct or ask students to analyse, design, develop, implement, evaluate (the options),

integrate, build, conclude, define (new things) or acquire (knowledge). There is a significant similarity between the second and the third list. This demonstrates that the preferred learning style of most of the students is in alignment with the demands of the post university professional life. However, there is a very serious difference between the first and other two lists, so much so that none of the top ten verbs of the first list also appears in one of top ten slots of either of the other two lists. While universities focus on regularly updating their curriculum, the differences in these lists demonstrate the need for transforming the teaching-learning-evaluation processes from a content-based curriculum to a process-based curriculum.

Table A11.2: Ordered lists of activity verbs

Ordered list of activity verbs as per their usage rating: calculate, explain, prove (studied theorem, studied method), define (studied definitions), write, solve, compute, show (studied fact, studied method), evaluate (computation), derive, state, describe, determine, find, analyse, justify, comment, distinguish, consider, illustrate, compare, apply, classify, identify, fill in the blanks, differentiate, conclude, examine, discuss, develop, implement, name, create, deduce, obtain, exemplify, construct, specify, design, categorize, estimate, propose, draw, generalize, demonstrate, recall, cite, summarize, convert, predict, formulate, argue, prepare, list, tell, point out, combine, sort, modify, represent, rearrange, devise, clarify, transform, compose, change, present, outline, rewrite, match, show (unstudied fact in the direct context of studied material), contrast, evaluate (the options), interpret, validate, organize, translate, label, build, decide, discriminate, produce, relate, recognise, synthesize, standardise, integrate, extend, plan, assess, recite, associate, document, reproduce, select, detect, arrange, infer, and judge.
Ordered list of activity verbs as per their learning effectiveness: design, analyse, understand, build, apply, adapt, implement, create, develop, demonstrate, validate, define (new things), show (unstudied fact in the direct context of studied material), illustrate, compare, enjoy, correlate, argue, research, evaluate (the options), compile, propose, derive, summarize, evaluate (computation), find, discover, explain, suggest, submit (deadline), show (studied fact, studied method), question, present, modify, devise, compute, construct, debate, solve, incorporate, focus, critique, improve, justify, examine, differentiate, prove (unstudied theorem), change, contrast, organize, associate, experiment, utilise, study, integrate, express, challenge, act, survey, transform, establish, interpret, grade, collaborate, administer, describe, progress, produce, duplicate, discuss, decide, contribute, conclude, teach, support, determine, prove (studied theorem, studied method), calculate, perform, accept, use, quote, negotiate, deduce, formulate, consider, categorize, simulate, relate, expand, chart, view, test, standardise, judge, document, combine, clarify, assemble, arrange, trace, rewrite, generalize, experiment, sketch, plan, perceive, exemplify, define (studied definitions), write, structure, restructure, memorise, convince, classify, anticipate, state, revise, reconstruct, restate, invent, simplify, convert, communicate, and reason.
Ordered list of activity verbs as per professional engineers' recommendations: analyse, design, develop, implement, evaluate (the options), integrate, build, conclude, define (new things), acquire, demonstrate, justify, assess, organize, formulate, estimate, summarize, categorize, validate, document, standardise, identify, appraise, calculate, manage, represent, review, reproduce, devise, apply, comment, generalize, specify, explain, extend, state, schedule, compare, present, classify, compute, consider, constitute, debug, decide, define (studied definitions), distinguish, examine, extrapolate, interpret, modify, name, point out, prove (unstudied theorem), recognise, reorganise, rephrase, report, revise, revisit, solve, synthesize, test, transform, transmit, weigh, create, prove (studied theorem, studied method), show (unstudied fact in the direct context of studied material), change, illustrate, practice, verify, question, clarify, discuss, propose, restructure, compose, recall, differentiate, and find.

These three lists were further distilled using Bloom's level to verb list mapping. All the verbs belonging to one Bloom level were grouped into one unit and Bloom level-specific consolidated ratings $L_{R\text{-student-I}}$, $L_{R\text{-student-II}}$ and $L_{R\text{-professional}}$ were computed as follows:

$L_{R\text{-student-I}}$ and $L_{R\text{-student-II}}$ are the sum of $V_{Rj\text{-student-I}}$ and $V_{Rj\text{-student-II}}$ respectively, for all the verbs belonging to the i^{th} Bloom level, i.e.:

$$L_{Ri\text{-student-I}} = \sum_j V_{Rj\text{-student-I}} \qquad \text{Where the } j^{th} \text{ verb belongs to the } i^{th} \text{ Bloom Level}$$

$$L_{Ri\text{-student-II}} = \sum_j V_{Rj\text{-student-II}} \qquad \text{Where the } j^{th} \text{ verb belongs to the } i^{th} \text{ Bloom Level}$$

$L_{Ri\text{-professional}}$ is the sum of $V_{R\text{-professional}}$ for all the verbs belonging to the i^{th} Bloom level**,** i.e.:

$$L_{Ri\text{-Professional}} = \sum_j V_{Rj\text{-Professioanl}} \qquad \text{Where the } j^{th} \text{ verb belongs to the } i^{th} \text{ Bloom Level}$$

Bloom level-specific consolidated ratings $L_{R\text{-student-I}}$, $L_{R\text{-student-II}}$, and $L_{Ri\text{-professional}}$ are then normalized as follows:

$$S_{R\text{-student-I}} = \sum_{i = 1 \text{ to } 6} L_{Ri\text{-student-I}}$$

$$S_{R\text{-student-II}} = \sum_{i = 1 \text{ to } 6} L_{Ri\text{-student-II}}$$

$$S_{R\text{-professional}} = \sum_{i = 1 \text{ to } 6} L_{Ri\text{-professional}}$$

$$L'_{Ri\text{-student-I}} = L_{Ri\text{-student-I}} / S_{R\text{-student-I}}$$

$$L'_{Ri\text{-student-II}} = L_{Ri\text{-student-II}} / S_{R\text{-student-II}}$$

$$L'_{Ri\text{-professional}} = L_{Ri\text{-professional}} / S_{R\text{-professional}}$$

The next stage of this research investigated verb usage in question papers. The sample comprised fifteen question papers of different subjects, given to around 1200 engineering students of the 1^{st}, 2^{nd} and 3^{rd} year Electronics, Computer Science (CS), and Information Technology (IT) and Biotechnology disciplines. Bloom level-specific consolidated ratings, $L_{Ri\text{-Examination}}$ were computed from this data as follows:

$L'_{Ri\text{-Examination}}$ is the fraction of the i^{th} Bloom level questions across all question papers, where:

$L_{Ri\text{-}Examination}$ = Number of questions belonging to the i^{th} Bloom level

$S_{R\text{-}Examination} = \sum_{i = 1 \text{ to } 6} L_{Ri\text{-}Exam}$

$L'_{Ri\text{-}Examination} = L_{Ri\text{-}Examination} / S_{R\text{-}Examination}$

Table A11.3 tabulates $L'_{Ri\text{-}student\text{-}I}$, $L'_{Ri\text{-}student\text{-}II}$, $L'_{Ri\text{-}professional}$, and $L'_{Ri\text{-}Examination}$ where large values indicate high ranks by most of the respondents.

Table A11.3: Comparison of Bloom level-specific normalized consolidated ratings

Bloom's Cognitive levels (i)	What students think they get $L'_{Ri\text{-}student\text{-}I}$	What students get in examinations $L'_{Ri\text{-}Exam}$	What students think works well for them $L'_{Ri\text{-}student\text{-}II}$	What professional engineers recommend $L'_{Ri\text{-}professional}$
Remember	0.24	0.36	0.04	0.09
Understand	0.24	0.16	0.11	0.10
Apply	0.22	0.40	0.13	0.10
Analyze	0.14	0.04	0.15	0.19
Create	0.14	0.05	0.46	0.38
Evaluate	0.02	0.00	0.11	0.15

Table A11.4 gives the correlation coefficients between these three ratings (each can be viewed as an arrays of 6 elements).

Table A11.4: Correlation between different consolidated ratings

	What students think they get $L'_{Ri\text{-}student\text{-}I}$	What students get in examinations $L'_{Ri\text{-}Exam}$	What students think works well for them $L'_{Ri\text{-}student\text{-}II}$	What professional engineers recommend $L'_{Ri\text{-}professional}$
What students get in examinations $L'_{Ri\text{-}Exam}$	0.77		-0.25	-0.57
What students think works well for them $L'_{Ri\text{-}student\text{-}II}$	-0.22	-0.25		0.96
What professional engineers recommend $L'_{Ri\text{-}professional}$	-0.38	-0.57	0.96	

Appendix A12: Anecdotes of Most Effective Learning Experiences/Lectures

Table A12.1: Anecdotes about the best lectures offering most effective learning experience, as recalled by senior computing students

S.No.	Anecdote
1	Good presentation using nice illustrations for algorithm visualization, followed by ***good student responses*** to teacher's questions.
…	……
3	***One-sided,*** the teacher did all the talking saying only that which was important. The flow was linear and progressive and no sudden jumps from one topic to another.
…	……
5	30 min almost non-interactive talk, followed by 20 min of discussion in the form of teacher using real-life examples, taking views of students, 6-7 students responded. He analyzed all the views. He again ***asked for responses*** while consolidating and converging the response to draw conclusions.
…	…..
11	In one or two lectures classes (2.5-3 hrs.), we had large groups working as teams for a long duration (1-1.5 hrs.)--- a great way to learn things real fast. In fact any large class where you have a large number of students involved in an ***open discussion*** even for 15 min. is great from a learning perspective.
…	…..
13	15-20 min. of one-sided presentation. The teacher asked "Is it clear?" "No," replied students. "Where is the problem?" asked the teacher. No response from the students. The teacher started asking questions related to the topic and continued to ask 5-6 questions. ***Critiqued studens' responses*** and finally told the correct answer. This took around 20 min. This question-answer session clarified the concept. The teacher continued with his presentation, while I continued to reflect upon the prior presentation and subsequent conversation. This class was great because of the central 20 min.
…	……
19	The teacher talked non-interactively (20 min). The teacher ***gave a problem***. We worked on it. Discussed it with the neighbor and then with the teacher (10+ min). The teacher re-explained the same concept (10 min). Gave another problem and asked us to work (5+ min). Introduced another concept. Asked for doubts and clarified (15 min).
…	…
22	Teacher gave us a design problem and asked us to ***work on it in a group***. (15-20 min). Many groups were asked to explain their solutions (10+ min). Teacher consolidated the solutions. Restated the problem in a more comprehensive and formal manner and gave us home assignments (20+ min).
…	…..
26	Teacher's presentation (15 min), students were asking questions, and teacher was answering. Teacher gave a problem, we worked on it (5 min). Teacher showed the correct solution, and asked us to verify our solution. This ***presentation-problem-solution*** cycle happened 3-4 times.
….	….
28	Students made presentation on their projects on the same topic. The ***presentations were critiqued by all***.

Table A12.2: Anecdotes about the best lectures offering most effective learning experience, as recalled by sophomore computing students at the beginning of their 3rd semester

S.No.	Anecdote
1	We were given just 5 min. to think about a particular topic and then we had to give a presentation on that topic. It was like a ***brainstorming session*** in which we could show our creative analytical and linguistic skills.
…	…..
4	The teacher made the topic so easy that there was no need to go through the book. I made notes and understood concepts.
…	…..
7	Teacher used to ask us everyday what we learnt on the previous day. And used to explain every concept very clearly. It was an ***interactive class***. We used to listen to her very attentively.
…	…..
14	He used real-life examples to describe. I attentively attended the class.
…	…..
16	I learnt to apply the concept in the real world. Also, I cleared my previous doubts. I attended and listened to it very carefully.
17	Teacher discussed problems at the end of each and every topic he discussed. I was able to ***ask my doubts*** simultaneously as the lecture proceeded.
…	…..
24	It went very slowly. I recalled my earlier exposure to the subject material.
…	…..
30	All the unclear concepts were clarified on the spot and mapped to real-life examples. I carefully listened to the teacher's presentation.
…	…..
41	It was pin-drop silence, and the subject was very clear to us. I tried to understand what was being taught.
…	…..
49	I concentrated, listened, and wrote notes properly.
…	…..
57	Teacher presented it very well. I tried to listen.
…	…..
68	Teacher did not merely explain the topic. He gave real-life examples that enabled the matter to be firmly stored in mind. No notes were taken. There was no necessity. We just sat and listened.
…	…..
76	***Very interactive class***. Involved myself in interaction.
…	…..
82	Topic was practically relevant. Teacher asked us to think and ponder about different theoretical concepts and helped us visualize them in reality and ***asked us to think about them***.

Table A12.3: Anecdotes about the best lectures offering most effective learning experience, as recalled by faculty members of engineering institutes from their student life

S.No.	Anecdote
1	No especially memorable very effective class. However some were interesting when the teachers brought in real-life applications.
…	…..
6	Teacher started with a basic point. And everything was built from that. Mostly one way communication with very few queries.
…	…..
8	20 students, 3 hr class with a break. We were given a reading assignment on the previous day (7-8 pages on the concept and a case). He asked "What have you gathered about the concept?" Everybody responded, and the key points were all put up on the board without any value addition from him (20 min). He asked, "What have you gathered about the case?" Everybody responded, and the key points were all put up on the board without any value addition from him (20 min). He related the two, added value, went into a conversation mode and summarised.
…	…..
11	When it was not bound within the subject and the topic. The analogies came from a very wide spectrum of domains. Asked questions, and sometimes students also asked questions.
….	……
18	Teacher used to give good homework, and then related the results of the problem given to the everyday incidence. He taught electrical machines. He forced us to explain, analyse, and relate our classroom learning with real-life occurrences.
….	……
24	…started most of the lectures with questions and answers. Explained basic concepts and asked questions. Gave case studies in group and we had discussion. Explained concepts using diagrams and real-life examples. Gave problems to solve in the class so as to make the concept clearer.
….	……
28	Very energetic. Raise a very simple topic. Asked students to respond. Almost everybody answered. Wrote all responses on board and discussed with students. Last 30 min, Introduce the topic in relation to the responses. Humour element in the discussion. No presentation, no slide, No book.
….	……
33	Teacher started with an example and started asking us questions about our interpretation. It forced us to think and apply more brain. … I was satisfied because I had contributed something.
….	……
44	20 students, 6 hrs. Brief explanation of subject matter. Problem definition. We were distributed in groups. We developed the solutions, and in due course learnt how the solutions exist.
….	……
65	We were given an exercise that was discussed in pairs, then the related theory was revealed by the teacher along with the current example.
….	……
72	Problem introduction (15 min). Basic concept (15 min). Problem solving (20 min). Future aspect (10 min).
…	…..
84	Delivered lecture in systematic manner, and explained each and every thing very clearly. Very organized, systematic and prepared for the topic.
….	……
95	20 min of interactive lecture, highly informative, interesting 10 m in of discussion, 15 min of introduction a new concept. Discussion and problem solving for 15 min.
….	……
99	Presentation. Asked a number of questions from the students. The teacher evaluated various answers. Each student equally participated, more interactive, develop designing ability; I developed the analysis process to find efficient solutions.

Table A12.4: Anecdotes about the best lectures delivered by the faculty members of engineering institutes, as recalled by them

S.No.	Anecdote
1	Throughout the course, a ***variety of queries*** came during presentations, I reformatted and redirected them back to students and tried to find out the solution themselves and ***responded*** well.
….	……
5	An extra class, attendance not compulsory. 110-120 students were present. Took 2 hr. I was not teaching the topic, I was discussing the topic. Gave 20 min of basic concept. Opened up a problem for their discussion. 20 min discussion. 25-30 students stood and said something in this conversation. I summarized and evaluated different solutions. This way we covered 3-4 topic in the same style of (lets call it ***presentation-conversation-summation***).
….	……
11	Introduced the topic for 20 min and asked them to relate the new topic with their existing knowledge by comparing the syntax with earlier syntax. Asked them to apply the topic in their way. Walked around for 10 min seeing them ***work out their solutions***. Selected 3 solutions for 5 min each. Introduced next topic with the help of a case study.
….	……
13	I asked a series of questions that ***led them to the rediscovery*** of tunnel diode and its characteristics.
….	……
18	30 min presentation. No response. Switched to daily life, asked them to form **groups**. Gave them an opportunity to design. The groups discussed with me. No consolidation.
….	……
25	Checked student's background knowledge by asking questions for 5 min. Explained the concept with the help of an analogy from a daily life experience (20 min). This was followed by ***20-25 min of questions*** from students and my answers.
….	……
28	Explained the concept (15-20 min). Gave an example (5 min). Came ***questions*** from students. (5 min). Another example (5 min). Came more Questions. (5 min). Wound it up all. Asked them a few Questions.
….	……
30	…started with a Q, and a ***lot of responses*** came. No value addition from my side (30 min). I introduced the concept with my slides and related it to earlier responses. This was also interactive (by invitation and interruption).
….	……
32	Part I do part they do, I check.
….	……
41	Started with ***asking questions*** on the topic and subtopic. Gave my non-interactive presentation for 10 min. ask them if they understood or not. ***Gave them a problem*** to solve. Walked through the class and checked randomly. Solved their problem. Gave another problem. Analyzed the concept and went to another subtopic.
….	……
43	I was able to link some of theoretical aspects with real-life examples. Students were satisfied and happy with the approach

Appendix A13: Quantitative Study of Computing Students' Perspective of Effective Lectures

Based 252 anecdotes from students and faculty (Appendix A12), a list of fourteen non-exclusive lecture properties was prepared as possible attributes of different lecture formats. These are listed in Table A13.1.

Table A13.1: Attributes to characterize variety of lecture format in engineering/software development education

a.	Lecture classroom is primarily a place for **careful listening** to teacher's presentation and **prepare class notes**.
b.	During the lecture classroom, the main purpose of this presentation is to **explain a textbook**.
c.	Lecture format encourages and allows many students to **on-the-spot seek clarifications** about unclear concepts in a teacher's presentation.
d.	Lecture format encourages and allows you to **seek clarifications** about home and laboratory work.
e.	Lecture format encourages and demands you to get on-the-spot practice of **problem solving** as an individual.
f.	Lecture format encourages and demands you to do on-the-spot **creative thinking**.
g.	Lecture format encourages and demands you to do **in-class-group-work**.
h.	Lecture format encourages and demands you to in-class **create conceptual designs**.
i.	Lecture format encourages and demands you to in-class **analyze presented information**.
j.	Lecture format encourages and demands you to on-the-spot **communicate your creations to neighbor students.**
k.	Lecture format encourages and demands you to on-the-spot **communicate your creations to the entire class.**
l.	Lecture format encourages and demands you to on-the-spot **critique** other student's work.
m.	Lecture format encourages and demands you to on-the-spot evaluate several creations and options.
n.	Lecture format encourages and demands you to on-the-spot **discover** conceptual knowledge through thinking and work rather than mere listening to teacher's presentation.

With respect to a given student's activities in the lecture classrooms, **we classify these attributes into the following four categories:**

5. Passively engaged student: The student only listens and does not add any content to the discourse (attributes a and b).
6. Reactively engaged student: The student reacts and asks for some clarifications without adding any other type of content to the discourse (attribute c).
7. Actively engaged student: The student gets individually engaged in some kind of problem solving activity, and adds some content to the discourse (attributes d, e, f, h, i, l, m, and n).
8. Collaboratively engaged student: The student proactively collaborates with others to solve problems and adds content to the discourse in the lecture classroom (attributes f, g, h, i, j, k, l, m, and n).

These categories together form a **typology of learning environments based on learner's conditioning**. It may be noted that some of the attributes (f, h, i, j, k, l, m, and n) belong to the 3rd as well as 4th group. Different subsets of these attributes can characterize different lecture styles being used by engineering faculty. This work attempts to separately identify the perceived effectiveness of the listed attributes. In order to have the perceptions of more experienced learners, who might have experienced a larger range of the abovementioned lecture attributes as part their formal education, a survey was conducted among senior undergraduate and postgraduate engineering students.

Respondents were requested to identify different 'lecture format categories' on the basis of distinguishable attribute combinations. They were then asked to assign 'usage rank' and 'learning effectiveness rank' to lecture format categories, identified by them. In all, 36 responses were received. Their responses for 'most effective lecture format,' 'least effective lecture format,' 'often used lecture format,' and 'least often used lecture format' are collated in Table A13.2. The first column of Table A13.2 represents the lecture attributes as per Table A13.1. *Column A* gives the attribute-wise fraction of respondents who felt that the lecture category with a given attribute is most effective for their learning. *Column B* shows the fraction of the respondents who felt that the lecture category with a given attribute is least effective for their learning. *Column C* gives the fraction of the respondents who felt that the lecture category having given attribute is most often used by teachers. *Column D* shows the fraction of the respondents who felt that the lecture category with a given attribute is least often used by teachers. For example, 75% respondents felt that lectures that encourage and demand them to do on-the-spot creative thinking (attribute f), are the most effective for them, whereas only 5% found such lectures to be least effective. Only 9% students thought that this is one of the attribute of the most often used format, whereas 45% thought that this is one of the attributes of the least used lecture format.

Table A13.2: Comparison of computing students' perception of effectiveness and usage rate of lecture format attributes

Lecture Format property	Most Effective for learning (A)	Least Effective for learning (B)	Most Often used (C)	Least Often used (D)
a. careful listening and preparing notes	36.36%	70.45%	79.55%	27.27%
b. explain textbook	11.36%	90.91%	88.64%	15.91%
c. on-the-spot seek clarifications	47.73%	38.64%	47.73%	29.55%
d. seek clarifications	34.09%	27.27%	25.00%	18.18%
e. problem solving	56.82%	15.91%	18.18%	31.82%
f. creative thinking	75.00%	4.55%	9.09%	45.45%
g. in-class-group-work	63.64%	4.55%	2.27%	47.73%
h. create conceptual designs	59.09%	2.27%	2.27%	45.45%
i. analyze presented information	59.09%	11.36%	6.82%	43.18%
j. communicate your creations to neighbor students	38.64%	11.36%	2.27%	63.64%
k. communicate your creations to the entire class	50.00%	6.82%	0.00%	63.64%
l. critique	43.18%	9.09%	2.27%	47.73%
m. evaluate	47.73%	4.55%	2.27%	61.36%
n. discover	63.64%	2.27%	0.00%	63.64%

This data in Table A13.2 was consolidated for the four categories of the lecture format attributes. The consolidated fractional ratings under each category of attributes were summed up. Table A13.3 shows the result of this summing up.

Table A13.3: Attribute category-wise consolidated ratings by computing students

Lecture format attribute category (engagements no in Table 7.13)	Most effective for learning (A)	Least effective for learning (B)	Most often used (C)	Least often used (D)
6 Passively engaged student (a and b)	0.48	**1.61**	**1.68**	0.43
7 Reactively engaged student (c)	0.48	0.39	0.48	0.30
8 Actively engaged student (d, e, f, h, i, l, m, and n)	4.39	0.77	0.66	3.57
9 Collaboratively engaged student (f, g, h, i, j, k, l, m, and n)	**5.00**	0.57	0.27	**4.82**

Appendix A14: Summary of SERO Style Lectures in Two Courses

Summary of SERO lecture in a Computer Graphics course (2004)

1. Seed 1.1.1: CG has picture description as input and picture as output.
2. Seed 1.1.2: Required inputs = fn (desired output).
3. Evolution 1.1: Output picture taxonomy for CG
   - static vs dynamic picture (degree of dynamism)
   - colour vs B&W (colouredness in the whole spectrum from binary to true color)
   - interactive vs non-interactive (degree of interactivity)
   - realistic vs symbolic (degree of realism)
   - objects vs abstract (degree of abstraction)
   - geometric objects vs natural objects
4. Reseed 1.1 (**Homework**): Refine the taxonomy.
5. Seed 1.2: Demonstration of a simple working graphics program and its code, with a focus on initialisation and closing of graphics mode, and some introduction to other functions.
6. Reseed 1.2(**Homework**): Practice using the graphics library.
7. Reseed 1.3: Identify some static and b&w picture and describe it in a machine readable format.
8. Evolution 1.2: Get your description critiqued by your partner, and rewrite your description.
9. Reseed 1.4: Develop a description scheme for encoding a description of a tree in machine readable format in a text file.
10. Evolution 1.3: Three solutions proposed by students:
    - (i) Row major 1/0 (ii) List of points for which colour is 1. (assumption: all others are 0)
    - (ii) Vectorised information
11. Reseed 1.5 (HW): Develop a description scheme for encoding a tree description in machine readable format in a text file. Create this file. Write a program to read this file, and create a tree on the screen.
12. Evolution 1.4 (HW): Design and programming work over the week involving 2 hrs. of batch-wise practical session with laboratory instructors in batches of 30 students, and group-wise discussions with the teacher with some groups on their initiative.

Learning Outcome #1: Students got an insight into the working of a simple graphics program already created by one of their peer student. They also succeeded in conceiving and evolving the taxonomy of graphics and data structures for static graphics.

Summary of SERO lecture in a Data Structures course (2005)

1. Vivek's non-recursive solution for list traversal without using a large number of temporary variables, or changing the structure of the list from singly-linked to doubl- linked. Estimated number of nodes to be traversed is $O(n^2)$.
2. Tanu's **recursive solutions** for linked list traversal: forward traversal, backward traversal.
3. Count the number of nodes traversed in the recursive solutions.
4. **Tabular Analysis** of control flow, lifetime of variables, and visibility of variable in recursive algorithms.
   i. Number all executable statements in the source code (including the last '},' indicating the return or end of function, of the functions and also of the main function).
   ii. Each call of the recursive function as expected to be made at run-time, is numbered as i, ii, iii, and so on.
   iii. Hence, each run time statement is numbered **as i.1, i.2, …, ii.1, ii.2 , ...,** and so on.
   iv. Key variables (parameters and local variables declared within recursive function) such as **varname1** are also labeled **as i.varname1, ii.varname1**, and so on.
   v. Create a **control flow analysis table**:
      v.i. The first column has the estimated run-time statement number of the current statement, e.g., i.1,i.2, simulating the logic of control flow.
      v.ii. The second column has list of live **variables**, e.g., i.varname1, ii.varname1, and their respective expected values, after executing the current statement. Underline the variables visible in this activation of the recursive function. On return from the jth call of the the function, all variables labeled as j.varnamex go out of scope, and are no more available in the memory.
      v.iii. Third column has the run time **estimated statement number of next statement** to be executed after execution of the current statement.
5. Assignment: Analyze all recursive programs so far written by you with the help of this **tabular analysis** technique as discussed in the class.

Appendix A15: Evolutionary Stages of Student Projects

Object-oriented Programming: (developed with S.K Singh, T.K. Tewari, M. K Thakur, and Manisha Rathi) 1. Single application: single class, single function 2. Single application: single class, multiple functions 3. Single application: multiple classes, multiple functions, simple relation 4. Single application: multiple classes, multiple functions, complex relation (association) 5. Single application: multiple classes, multiple functions, complex relation (aggregation, composition) 6. Single application: multiple classes, multiple functions, complex relation (inheritance) 7. Single application: multiple classes, multiple functions, complex relation (polymorphism) 8. Secure single application: multiple classes, Multiple functions, complex relations 9. Robust (exception handling) and secure single application: multiple class, multiple function, complex relation 10. Robust and secure multiple applications: multiple classes, multiple functions, complex relations
Database Management Systems: (developed with Indu Chawla) 1. Simple database applications : Single-user, Multi-user, Multiple type multi-user 2. Web-enabled database applications 3. Robust Web-enabled database applications 4. Robust Web-enabled database applications with a large no. of concurrent users 5. Secure Robust Web-enabled database applications with a large no. of concurrent users 6. Secured Robust Web-enabled database applications with multimedia user interface and database 7. Mobile accessible Secure Robust Web-enabled database applications
Web Application Engineering: (developed with Jolly Shah) 1. Single Thin Client Web Application 2. Single Thick Client Web Application 3. Multiple Thick Client Web Application 4. Multiple Rich Client Web Application 5. Multiple Rich Client Web Application with automated database population 6. Secure Multiple Rich Client Web Application with automated database population 7. Mobile enabled Secure Multiple Rich Client Web Application with automated database population
Enterprise Application Development: (developed with Ritu Arora) [367] 1. Single Thin Client Web Application 2. Multiple Thin Client Web Application 3. Multiple Thick Client Web Application 4. Multiple Rich Client Web Application 5. Modular Multiple Rich Client Web Application 6. Modular Multiple Rich Client Web Application with multiple GUI support 7. Secure Modular Multiple Rich Client Web Application with multiple GUI support 8. Secure Modular Multiple Rich Client Web Application with distributed access 9. Mobile enabled Secure Modular Multiple Rich Client Web Application with distributed access
Software Engineering: (developed with Manisha Rathi) 1. Initial, direct, independent and well-defined requirements 2. Initial, direct, independent and ill-defined requirements 3. Initial, direct, inter-dependent and ill-defined requirements 4. Initial, derived, inter-dependent and ill-defined requirements 5. Evolutionary, derived, inter-dependent and ill-defined requirements
Information Systems (IS): (developed with Jolly Shah) 1. Building IS using Single Thin Client 2. Building IS using Multiple Thin Client 3. Building IS using Multiple Rich Client 4. Building IS using Modular Multiple Rich Client 5. Building IS using Modular Multiple Rich Client with Multiple GUI Support 6. Building IS using Secure Modular Multiple Rich client with Multiple GUI Support 7. Building IS using Secure Modular Multiple Rich Client Web Application With Distributed Access 8. Building mobile based IS

Appendix A16: Reflective Engagements

Table A16.1: Format for reflective report on final year project

Reflecting upon your final year project, answer the following questions:
1. What is the problem you have tried to solve? Why is this work important? 2. Does your project open new ways of thinking? How? 3. What was the division of the task among group members? 4. What were the main challenges? How did you address these challenges? 5. What is your approach or solution? 6. Why is it better/different than other existing approaches or solutions? 7. What personal, technical, and other professional competencies have you been able to strengthen due to your engagement in this project? How? 5. What new things did you learn? 9. What mistakes did you make with respect to your project? 10. If you were to start again, how would you approach the project? 11. Can your project act like a seed for some future projects? If yes, how? List the problem statements for future projects as a natural extension of your project. 12. Has your project been used by persons outside your group? Have you done anything to solicit (potential) user's feedback? 13. Did you collaborate with any other project group? How? 14. What kind of new inter-project collaboration possibilities can you now propose with any of the other ongoing projects?

Table A16.2: Reflective assignments in three final year elective course

Software Documentation (designed in collaboration with Bharat Gupta, Parmeet Kaur, and Hema N.)	**Software Risk Engineering** (designed in collaboration with Sangeeta Mittal, Vivek Mishra, and Anuja Arora)	**Software Construction** (designed in collaboration with Shikha Mehta, Sandeep K. Singh, Maneesha Srivastava, and Alok Agarwal)
Using the documentation templates, from the study material of the software documentation course, re-document your 7th semester project work paying special attention to the following: 1. Coding guidelines, best practices, checklist, user/client documentation 2. Requirement engineering documentation (as per IEEE 830 standard) 3. SEI Architecture Design Documentation 4. Software and System Test Documentation (as per IEEE 829 standard) 5. Details of the working of the Documentation Tool that can be used for the project	1. Individually reflect upon your 7th semester project experience, and retrospectively identify the main risks. Use the SEI software risk taxonomy and checklist for identifying software project risks about requirement, specifications, design, coding, testing, integration, product, system, maintainability, and intra-team communication and compatibility. 2. Based on individual activity, each group of two will identify the top six risks. 3. Each member will develop a detailed plan for managing three of these top risks for the 8th semester project work. Submit your report as per the prescribed format and templates.	Write a report about your 7th semester project addressing the following aspects: 1. Assertions 2. Exceptional Handling 3. Error Handling 4. Generics/Templates 5. Code Optimization 6. Debugging 7. Source Code Organization

Table A16.3: Two sample assignments in 'software arteology,' emphasizing on reflection

Assignment #3: **Review any ten published research based papers already studied by you in any project/assignment so far, and submit your report in the following format.** 2. Name of peer-reviewer of your work 3. Classify each of your chosen ten papers as per the nature of the goals of reported inquiry. (Exploratory Informative/Descriptive Informative/Explanatory Informative/Normative.) 4. For each paper, identify the central research question. 5. For each paper, identify the theoretical constructs used in the research. 6. For each paper, identify the theoretical constructs modified/created by the research. 7. For each paper, identify the empirical constructs used in the research. 8. For each paper, identify the empirical constructs modified/created by the research. 9. What is the big picture? What do these specific cases mean/signify on the whole? Generic discussions/common pattern/differences etc. What are the possible reasons? 10. ***Now what***? What does it imply for the future? What are your own learning outcomes from this assignment? What kind of personal practices do you intend to change, if any?
Assignment #5: Interact with some creative professionals, in any profession, about their creative experiences during problem definition, alternative generation of solution approaches, and evaluation criteria design. Submit your report in the following format. 1. Name of peer-reviewer of your work 2. Give a brief description of the professional's profile from whom you got inputs for this assignment 3. Description of specific cases of creativity in 'problem definition.' Separately narrate each case. How many iterations, stimulants, process, duration? 4. Description of specific cases of creativity in 'generating alternatives' before deciding the final solution approach. Separately narrate each case. How many alternatives, iterations, stimulants, process, duration? 5. Description of specific cases of creativity in defining the 'criteria for selection' out of alternate solutions approaches. Separately narrate each case. How many options, iterations, stimulants, process, duration? 6. So what is the big picture? What do these specific cases mean/signify on the whole? Generic discussions/common pattern/differences etc. What are the possible reasons? 7. ***Now what***? What does it imply for the future? What are your own learning outcomes from this assignment? What kind of personal practices do you intend to change, if any?

Table A16.4: Some sample responses to last sub-question (now what?) of some assignments (Table A16.5)

• My personal learning is that if you want to do a task well, you have to ask the right questions, and for that you need to get into the shoes of the customer and think like him, i.e., have background knowledge of the domain. • This assignment has brought to light the importance of questions. I realize that question formation is more important and difficult than answers, because it involves creativity. I will give more stress to bombarding my mind with questions. • I have always thought that we do a lot of theories and less of implementation. That perspective has changed to a greater extent. • Two problems have plagued me throughout most of the activities undertaken. Firstly, I was unable to differentiate problems from symptoms, and secondly, even on being able to filter out the problem, I was unable to answer it in an appropriate manner. I now seem to have understood what the appropriate answer for a question ought to be. • It implies that every project or research needs both empirical and theoretical constructs. In the future, I would concentrate on theory, and then will try to bring that theory in a modified form into the emperia. • … it had brought changes in me. The first thing is that we should not rush into seeking a solution without knowing the purpose. We should first do iterations in our minds and play with the problem statement again and again, and then come to a definite track. • In the future instead of jumping onto a problem for solution, I would rather define the problem with a better understanding and look, for all possible existing solutions, and very clearly list down the criteria to choose the best possible solution. I will keep in mind the need to carry out many iterations to improve upon the problem statement, possible solutions, and the criteria for selection, and reflect after an iteration in one of these areas over the other areas too.

Appendix A17: Feedback from the Cross-level Mentors on Infusion of Some Pervasive Topics in Foundation Courses

Table A17.1: Mentor feedback on infusion of web technology

Host course	Sem	Mentors' perspective on course specific infusion of web technology (number of mentors in each view category)				Avg rating
		Extremely valuable for immediate foundation and long term benefits, worth the extra work (3)	Valuable for their foundation/long term benefits, worth the extra work (2)	Valuable for their foundation/long term benefits, but not worth so much extra work at this stage (1)	Only marginally useful, not worth the extra work at all now (0)	(0-3)
Introduction to Computers and Programming	1	12	28	25	10	1.56
Object-oriented Programming	3	16	7	0	0	2.7
Database Systems	3	12	8	1	0	2.52

Table A17.2: Mentor feedback on infusion of multimedia technology

Host course	Sem	Mentors' perspective on course specific infusion of multimedia technology (number of mentors in each view category)				Avg. rating
		Extremely valuable for immediate foundation and long term benefits, worth the extra work (3)	Valuable for their foundation/long term benefits, worth the extra work (2)	Valuable for their foundation/long term benefits, but not worth so much extra work at this stage (1)	Only marginally useful, not worth the extra work at all now (0)	(0-3)
Introduction to Computers and Programming	1	25	30	13	6	2
Object-oriented Programming	3	9	11	2	1	2.22
Database Systems	3	4	9	7	1	1.76
Web Application Engineering	5	5	7	6	0	1.94

Table A17.3: Mentor feedback on infusion of mobile technology

Host course	Sem	Mentors' perspective on course specific infusion of mobile technology (number of mentors in each view category)				Avg. rating
		Extremely valuable for immediate foundation and long term benefits, worth the extra work (3)	Valuable for their foundation/long term benefits, worth the extra work (2)	Valuable for their foundation/long term benefits, but not worth so much extra work at this stage (1)	Only marginally useful, not worth the extra work at all now (0)	(0-3)
Database Systems	3	5	6	9	1	1.71
Web Application Engineering	5	5	4	6	2	1.71

Table A17.4: Mentor feedback on infusion of security aspects

Host course	Sem	Mentors' perspective on course specific infusion of security aspects (number of mentors in each view category)				Avg. rating
		Extremely valuable for immediate foundation and long term benefits, worth the extra work (3)	Valuable for their foundation/long term benefits, worth the extra work (2)	Valuable for their foundation/long term benefits, but not worth so much extra work at this stage (1)	Only marginally useful, not worth the extra work at all now (0)	(0-3)
Object-oriented Programming	3	9	8	6	0	2.13
Database Systems	3	9	8	4	0	2.24
Software Engineering	5	11	8	0	0	2.58
Web Application Engineering – Instantaneous client and server-side data validation	5	7	11	0	0	2.39
Web Application Engineering – other security aspects	5	6	8	3	1	2.06

Table A17.5: Mentor feedback on infusion of systems design aspects

Host course	Sem	Mentors' perspective on course specific infusion of design diagramming (number of mentors in each view category)				Avg. rating
		Extremely valuable for immediate foundation and long term benefits, worth the extra work (3)	Valuable for their foundation/long term benefits, worth the extra work (2)	Valuable for their foundation/long term benefits, but not worth so much extra work at this stage (1)	Only marginally useful, not worth the extra work at all now (0)	(0-3)
Introduction to Computers and Programming – Necessity of flowcharting	1	46	20	3	4	2.48
Object-oriented Programming – Basic UML	3	12	7	4	0	2.35
Object-oriented Programming – Concept map	3	5	11	5	1	1.91
Object-oriented Programming – Evolutionary project scoping	3	3	15	4	0	1.95
Database Systems – ER and EER	3	13	7	0	0	2.65
Software Engineering – Concept map	5	6	8	4	1	2
Software Engineering – Evolutionary project scoping	5	3	9	5	1	1.78
Web Application Engineering– Evolutionary project scoping	5	1	12	5	0	1.78

Table A17.6: Mentor feedback on infusion of PSP (time logs)

Host course	Sem	Mentors' perspective on course specific infusion of estimation tools (number of mentors in each view category)				Avg rating
		Extremely valuable for immediate foundation and long term benefits, worth the extra work (3)	Valuable for their foundation/long term benefits, worth the extra work (2)	Valuable for their foundation/long term benefits, but not worth so much extra work at this stage (1)	Only marginally useful, not worth the extra work at all now (0)	(0-3)
Introduction to Computers and Programming – PSP (time log)	1	14	30	15	16	1.56
Object-oriented Programming– PSP (time log)	3	1	5	11	6	1.04
Database Systems– PSP (time log)	3	0	4	9	8	0.81
Software Engineering– PSP (time log)	5	4	5	7	3	1.53
Software Engineering– Estimation and other metrics	5	3	11	5	0	1.89
Web Application Engineering– PSP (time log)	5	3	6	6	3	1.5

Table A17.7: Mentor feedback on infusion of open source

Host course	Sem	Mentors' perspective on course specific infusion of open source (number of mentors in each view category)				Avg rating
		Extremely valuable for immediate foundation and long term benefits, worth the extra work (3)	Valuable for their foundation/long term benefits, worth the extra work (2)	Valuable for their foundation/long term benefits, but not worth so much extra work at this stage (1)	Only marginally useful, not worth the extra work at all now (0)	(0-3)
Introduction to Programming – Initial laboratory work in Python	1	13	27	27	7	1.62
Object-oriented Programming – Mini project	3	8	9	5	1	2.04
Database Systems – Mini project	3	9	5	7	0	2.1
Software Engineering – Mini project	5	10	8	0	1	2.42
Software Engineering – Program Comprehension and reverse engineering	5	5	9	4	1	1.95
Web Application Engineering	5	8	7	3	0	2.28

Table A17.8: Mentor feedback on infusion of PSP (Bug log)

Host course	Sem	Mentors' perspective on course specific infusion of PSP (Bug log) (number of mentors in each view category)				Avg. rating
		Extremely valuable for immediate foundation and long term benefits, worth the extra work (3)	Valuable for their foundation/long term benefits, worth the extra work (2)	Valuable for their foundation/long term benefits, but not worth so much extra work at this stage (1)	Only marginally useful, not worth the extra work at all now (0)	(0-3)
Introduction to Computers and Programming	1	18	27	13	17	1.61
Object-oriented Programming	3	0	8	10	5	1.13
Database Systems	3	1	4	10	5	1.05
Software Engineering	5	5	4	8	1	1.72
Web Application Engineering	5	3	5	7	3	1.44

Appendix A18: Multi-level Infusion of Security Related Aspects

(Developed in Collaboration with Jolly Shah)

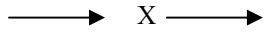
Sem.	Course	Security Aspect
I	Introduction to Computers and Programming	Signed/unsigned problems, poor standard C library functions, overflow (buffer, heap stack, format string errors, pointer issues, file system security issues
II	Data Structure	Overflow/underflow issues in stack, queue, and array data structure, hash-table security issues (like DOS), linked list security issues, heap exploitation techniques and solutions, invalid B-tree node size can lead to data loss
III	Object-oriented Programming (C++/Java)	Limiting the accessibility of classes, methods, interfaces, fields, preventing the un-authorized construction of sensitive class, preventing constructors from calling methods that can be overridden, duplicating the security manager checks enforced in a class during serialization and de-serialization, Guarding sensitive data during serialization safely invoking standard APIs, defining wrapper methods around modifiable internal state, defining wrappers around native methods, Purging sensitive information from exceptions, defending against partially initialized instances of non-final classes
	Database System	DBMS buffer overflow, confidentiality, integrity and accuracy of data, secure sharing of databases, database threats (private threats, privilege activity threat, malicious software threats, remote access threats, distributed database configuration threat, inference, aggregation), SQL injection
IV	Fundamentals of Algorithms	Comparison analysis of symmetric and asymmetric cryptography algorithm, message digest algorithms, e.g., SHA, MD5, digital signature algorithms: DSA, fault tolerance algorithm, forward and backward recovery algorithm
V	Operating System	logon security, digital certificate security, file and folder security, shared resource security, security policies, remote access security, disaster recovery
	Software Engineering	Secure software development lifecycle, threat model, threat tree pattern, secure UML
	Web Application Engineering	Security of web application, attacks on web application- phishing, cross-site scripting vulnerabilities, SQL injection, denial-of-service attack on web server, unvalidated parameters, broken access control, broken account and session management, cross-site scripting flow, buffer overflow, common injection flaw, error handling problems, insecure use of API, remote administration flaw, web and application server mis-configuration, guidelines for securing web application, seven habits for writing secure PHP applications
VI	Computer Network	LAN Security, firewall, VPN, internet security protocol, network vulnerabilities, wireless Security, internet vulnerabilities (phishing, farming, DOS, cross-site Scripting)
	Compiler Design	Compiler based Protection (securing stack data, potentially vulnerable heap data, adding run time checks, adding protection mode), countering code-injection attacks with instruction-set randomization, enforcing compiler security checks

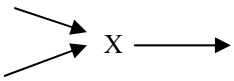
Appendix A19: Description of the Notation for Concept Mapping

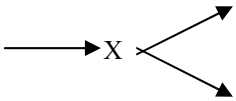
This concept map provides a bird's-eye view of a collection of interacting and collaborating data tanks and data items. Do not mix or confuse this concept mapping technique with any other diagramming technique like DFD or ER diagram and so on. There may be some similarities with some, but it is different.

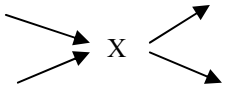
1. Look at each collection of homogeneous (similarly structured) data items as a Data tank. These individual data items could be atomic or compound.
2. Identify the nouns and verbs of the systems description. Some nouns will become data tanks in Concept Map (CM). The nouns could be singular as well as plural. Verbs will represent the processing box in CM.
3. This Concept map is a diagram of **inter-connected data tanks via processing units** with boxes and arrows, marked labeled. It gives an indication of **what, when, and how some data moves or changes** in any data tank.
4. Write the properties and functional behavior for each data tank by giving a clear description of the content, and also permitted legal operations on each data tank and data item. This collection may or may not require some inter-data item organizational constraints.
5. Use **double line boxes** for data tanks containing several homogeneous data items, and **single line boxes** for single data item/packets, if any.
    a. Input data: Add incoming arrow head on left side of data box.
    b. Output data: Add outgoing arrow on right side of data box.
    c. I/O data: Add arrowheads on both the left and right side.
    d. Processing data: no arrowhead.
6. **Name your data tanks** as a **set of …** (e.g. set of trains, set of passengers, set of books, set of users, and so on) that contains many homogeneous items only.
7. Put the name of the data that flows in/out of data tanks.
8. Data copy transfer: directed links between data boxes.
9. Use **elliptical boxes** to show processing of chosen data items. **Name your processing units** as verbs, only representing the process.
10. Put a **small circle on the top right corner of data tank boxes**, if it represents **dynamic data,** i.e., the data can change as a result of valid operations.
11. Put **a circle on the top left corner**, if the **data population size can change during processing** because of insertions and deletions. This dynamic data (at 16 and 17) is not to be confused with dynamic data structure, as this higher-level of dynamism can be implemented with dynamic or static data structures at lower layer.
12. Draw four dotted horizontal lines and divide data tank into **five sub-boxes**.
13. Put the name of data tanks in the **top (first) sub-box** and give some **examples** of representative data items in the same sub-box.
14. Write the **attributes** (fields) in the **second sub-box**. First put, and also underline, the attribute(s) that are required to have **unique values,** e.g., ID No., etc.
15. Identify all the **operations** that are required to be performed on this data tank during the lifetime of a given application. Write these operations in the **third sub-box**.
16. If the data tank has limited life during processing, write the **scope** of the data tank in the **fourth sub-box**. Mention the event(s) that brings this tank to existence, or remove it from the systems using **created on** and **destroyed on** clauses.
17. If your data tank is **compound,** i.e., you need some **additional ancillary and smaller data tanks** (e.g., indices, and so on) to support efficient searching of appropriate data items in the principal data tank, include the names of these ancillary data tanks in the **first sub-box of the principal data tank** itself by dividing the first box into two units by a vertical line, and list the **names of ancillary tanks in the right half of first sub-box.**

18. Later on, expand this compound data tank into a principal data tank inter-connected with ancillary data tanks in a different diagram.
19. **Name your ancillary data tanks** using the name of the principal data tank, followed by an underscore, and then by another plural noun, e.g., the ancillary data tanks for data tank 'trains' could be named as 'set of trains_train-nos,' 'set of trains_destinations,' and so on.
20. Vertically divide the **fifth box** into two parts.
21. See if the data tank is required to maintain some order on the elements or not. If no order is required, leave the fifth left and fifth right sub-boxes empty.
22. Check the type of order - Is it **ordered chronologically,** i.e., based on the time of insertion of records or **ordered on attribute(s)/value(s).** Also check if you need ascending or descending order. If ordered on value, identify the attribute(s) that control the order of records.
23. If ordered on time, then put 'T' in the top left corner of the **fifth left sub-box,** if the tank is ordered on time. Put an upward arrow for ascending order; put a downward arrow for descending order. If ordered on value, put 'V' and the attribute followed by appropriate arrow.
24. If the data tank is an ordered collection of X, see how the relative position of a specific data item is defined with respect to other similar data items. Indicate it in the **fifth left sub-box**. Some possible arrangements are as follows.

a. 

b. 

c. 

d. 

25. Relative position could be in terms of order of insertion or relative value of some data item.
26. Examine relative positional eligibility for accessing: All/only some strategic relative positions.
27. Examine relative positional eligibility for updation: All/None/only some strategic relative positions.
28. Examine relative positional eligibility for insertion: Any empty slot/only some strategic relative positions.
29. Examine relative positional eligibility for deletion: All/None/only some strategic positions.
30. If access, updation, insertion, and deletion are dependent on some well-defined strategic relative position within the data tank, observe, identify and define these positions. Indicate these positions in the **fifth right sub-box**. Some examples of strategic relative positions can be as follows:
    a. Based on order of insertion: earliest, latest, after the latest insertion, before the earliest insertion, 3rd earliest, 4th latest, next relative to the current position as per insertion order, previous relative to the current position as per insertion order, and so on.
    b. Based on the value: minimum, maximum, 3rd minimum in between a given range of values, in between an appropriate range of values, and so on
31. Your concept map should be hierarchical, i.e., it should gradually show more details in different diagrams rather than showing all the details in one diagram. Initially focus on the most critical aspects.
32. All the data tanks that have same abstractions of operations, ordering, strategic positions, pre-conditions, and post-conditions belong to same **Abstract Data Type**, e.g., stacks, queue, all binary tree, graph, table, data cube, octree, etc. (can be extended to the notion of 'Class').

Appendix A20: Some Proposed Instructional Interventions for Infusing Debugging in Computing Laboratories

Bug generation

Students can be given assignment to produce bugs of various types relevant to their course. The activity of intentionally writing code to generate a bug will entrench the bug, its cause, and symptoms in student's mind. Hence, a student will be better equipped to relate any subsequent encounter of a previously experienced bug to its actual cause, thereby facilitating quicker and efficient debugging.

Comparative study of debugging tools

Assignments to do comparative study of debugging tools will help in developing facility with debugging tools. Also, cost benefit ratio of using the various tools can be understood by doing a comparative study of debugging tools for a particular class of bugs. This assignment will help student in identifying the correct tool for debugging a particular scenario. Students can be asked to prepare a tool evaluation matrix (Figure A20.1) to show the strength and weaknesses of different tools with respect to their support for different types of bugs.

Bugs / Tools	Bug 1	Bug 2	Bug 3	…
Tool 1				
Tool 2				
Tool 3				
…				

Figure A20.1: Debugging tool evaluation matrix

Program Comprehension of existing debugging tools

Program comprehension of the source code of existing debugging tools will enable students to have an in-depth knowledge of implementation of these tools. Through the study of implementation of tools, student will understand the underlying rules being used by tools for bug

detection. Consequently, learner will identify the premise on which the tool works, thereby deciphering the basis of bug detection. As part of the deliverables, a student can be asked to prepare reports about the working and internals of a chosen debugging tool. In this report, the working principle of a tool under study can be clearly indentified. Further, major data structures and algorithms used in the implementation of tools, supporting the identified working principle can be incorporated in the report. Open source debugging tools such as Valgrind, Dmalloc, Splint, ftnchek, GDB, etc., can be used for this assignment.

Creation of simple debugging tools and enhancement of existing debugging tools

Assignments for creating simple debugging tool can be given to students. These tools should be capable of detecting simple specific bugs, and need not be generic for a class of bugs. Students should be encouraged to take up an activity of enhancing existing debugging tools as a capstone project. This will benefit students in gaining a deeper insight about debugging as well as prepare them with the ability to do enhancement to existing tools to meet their own unique requirements, or come up with an optimal tool for their development environment

Appendix A21: Collaborative Pair Programming

There were a total of one hundred and seventy-eight students enrolled in this course, and they were divided into three batches for laboratory classes. The students were divided into two categories. The first category consisted of 66 students having a prior programming experience during their K-12 education. This set of students was asked to work independently all throughout the semester. The other group consisted of 112 students who had no experience in programming. These students were asked to work in pairs.

Experienced programmers were required to solo-program the combined task directly. Inexperienced students working in pairs had to solve the two problems in the first part individually, and separately. Each student had to provide the solution for his or her sub-problem. Thereafter, they had to collaboratively solve the combined task in the second part by combining the concepts they applied in their independent work.

Table A21.1: Sample laboratory assignment for introduction to programming

Individual task 1	Individual task 2	Combined task
Write a program to make an n*n square using the "$" symbol. Get the value of n from the user. Use an incrementing loop (i=0, i<n; i++).	Write a program to make an n*n square using the "$" symbol. Get the value of n from the user. Use a decrementing loop (i=n;i>0;i--).	Combine the programs in such a way that exactly half the design is made by incrementing the counter value 'i,' while the other half design is made by decrementing the counter 'i'.
Write a program to enter ten names and roll numbers and print them.	Write a program to enter ten names and ages and print them.	Write a program combining the two codes such that the output shows the name, roll number, and age for all the common names.
Write a program to input two words and print the number of occurrences of each letter in each word.	Write a program to input two words and print all the common letters in both. Note that a letter will be considered common in both words if number of occurrences of that letter is the same in both words.	Write a program that inputs two strings of any length and checks whether they are anagrams or not. Anagram is a word or phrase made from another by rearranging its letters (Ex.: now ➔won, dread ➔ adder, riot ➔ trio). Also, if they are not anagrams, report the number of letters that are not same.
Write a function (char_count) to count the number of characters in a given string. The string will be the input argument for the function. Answer has to be printed in the main function.	Write a function (word_count) to count the number of words in a given string. The string will be the input argument for the function. Answer has to be printed in the main function.	Write a program such that for a given input string the main function calls the word_count function first and as soon as a word is identified it calls the char_count function with this word as input. In this way calculate the number of words and total number of characters in the given sentence.

Table A21.2: Comments of students on their experience with collaborative peer programming

1. "I agree that when two people with no programming background work together they learning more easily."
2. "Working with a partner helped me. We could identify different ways of solving the same problem when we combined individual tasks."
3. "This new method made us think deeply, and shaped our views towards a good approach to problem solving. Both partners had a feeling that they had the support of each other, and this added to the motivation level."
4. "Working with a partner really helped me. I had no programming background, but I could ask my partner all doubts without any hesitation. Combining individual programs made us come across more mistakes."
5. "The new method was a life saver for students with no programming background. They were able to grasp much more. Two of my friends secured an 'A' even though they had no programming background."

Appendix A22: Sample Collaborative Quadruple Programming Assignments For J2EE

Laboratory assignment on servlets (without session-tracking) (Designed with Ritu Arora)

Generic Problem: To create a login page that may be used with any application. Choose an appropriate data structure to store the login names and password as key-value pairs in the program.

Specific Problem: To create a login page, in which the user would be authenticated based on the username and password entered. The username and password would be validated against the data stored in a hash-table. If the user is a valid user, he/she would be displayed a welcome page, or else an error is generated, and the user is asked to re-enter the information.

Background Studies and Practice:

1. Learn to configure and start-up the Tomcat Server. 2. Practice the "Hello World" example servlet.

Solo Task	**Generic Description**	**Specific Example**
Task A	Create a HTML page consisting of input text box and submit button. On submission, servlet should be invoked.	Learn to create an HTML page that would display to the user an input text box. The page should also consist of a submit button, clicking on which would invoke the desired servlet.
Task B	Create a servlet to display the names and values of incoming request headers.	Learn to create a servlet that would display the names and values of all the field of the request header.
Task C	Create a servlet to facilitate navigation through various HTML pages/servlets.	Learn to create a servlet that would navigate to different HTML pages/servlets, depending on the value of a particular variable.
Task D	Create a servlet that would create a data structure to store key-value pairs, and iterate through it.	Learn to create a servlet that creates a hash-table. Iterate through the hash-table and display its contents.
Pair Task		
Task AB	Create a HTML page to take user input and, on submission, display the values entered by the user on the console.	Create a HTML page that displays two input text boxes, one for entering username and other for password (with "*" being displayed for each character). On clicking the submit button, the input parameter values should be read by the servlet, and displayed on the console.
Task CD	Create a servlet that would perform searching through the chosen data structure, and navigate to the HTML page/servlet depending on the results of search operation.	Create a servlet that would search through a hash-table for the existence of a value, given the key. If the key itself does not exist, it should navigate to a welcome/error page.
Quadruple Task		
Task ABCD	To create a login page that may be used with any application. Choose an appropriate data structure to store the login names and password as key-value pairs in the program.	Create a Login Page, in which the user would be authenticated based on the username and the password entered.. The user name and password would be validated against the data stored in a hash-table. If the user is a valid user, he/she would be displayed a welcome page, or else an error is generated and the user is asked to re-enter the information.

Appendix A23: Alumni's Feedback on Learning Gains through Cross-level Mentoring

In 2009, we conducted a survey amongst our alumni members. All these respondents had mentored their juniors during their final year. These alumni members were requested to rate the effect of mentoring experience on sixteen competencies by comparing the same with the effect of several other academic experiences at undergraduate level. They were given following five options for each of these sixteen competencies:

1. Least effective as compared to all other academic experiences (-2)
2. Less effective as compared to several other academic experiences (-1)
3. Comparable to several other academic experiences (0)
4. More effective as compared to several other academic experiences (1)
5. Most effective as compared to all other academic experiences (2)

Thirty-six alumni members of all batches graduating from 2005 to 2009 have rated sixteen competencies on these five levels. Many of these students were among the toppers and the best programmers during their college days. Several of them have completed (or are pursuing) higher studies at top ranking universities like Stanford, Cornell, Columbia, Utah, and University of Southern California, etc. Some of them are working with top ranking organizations like Microsoft, Intel, Adobe, etc., while many others are working in well known software consulting companies like Accenture, Wipro, Infosys, etc. The composite rating of the effectiveness of the mentoring experience in term of a *comprehensive effect on all sixteen competencies is 0.8 on a scale of -2 to 2*. The distribution of all their votes casted for these five levels is as follows:

1. Least effective as compared to all other academic experiences: 2 votes (0%)
2. Less effective as compared to several other academic experiences: 44 votes (8%)
3. Comparable to several other academic experiences: 164 votes (28%)
4. More effective as compared to several other academic experiences: 239 votes (41%)
5. Most effective as compared to all other academic experiences: 130 votes (22%)

Table A23.1 gives the details of the perceived effect of the mentoring experience on different competencies.

Table A23.1: Alumni reflections on the effect of mentoring on mentors' competencies

S.No	Competency	Votes comparing the effectiveness of mentoring with other academic experiences A. Least effective (-2) B. Less effective (-1) C. Comparable (0) **D. More effective (1)** E. **Most effective (2)**					**Avg. rating** (-2 to 2)
		A (-2)	B (-1)	C (0)	D (1)	E (2)	
1	Intrinsic motivation to create, improve things and open-mindedness.	1	5	8	**14**	9	0.7
2	Systems-level perspective, inclination for reuse and synthesis by integration, ability to understand and also build upon other's work.	0	6	9	**18**	4	0.5
3	**Accountability and responsibility, strength of conviction, and self-regulation, ability to see the self as bound to all humans with ties of recognition and concern, sensitivity towards global, societal, environmental, moral, ethical and professional issues, and sustainability**	0	1	9	**15**	**12**	**1**
4	**Curiosity with humility, self-learning, ability to develop good understanding of domains' vocabulary, semantics, and thinking processes, faith in reason, and review.**	0	4	6	**19**	**8**	**0.8**
5	**Ability to accommodate self to others, ability to work such that others can easily understand and build upon.**	0	3	8	**15**	**11**	**0.9**
6	**Problem solving, ability to convert ill-defined problematic situations into software solvable problem, project scoping , estimation**	0	3	**13**	**8**	**13**	**0.8**
7	**Attention to details.**	0	1	11	**18**	**7**	**0.8**
8	Abstraction, transition between levels of abstraction.	0	5	**14**	12	5	0.5
9	Algorithmic and structured thinking,	0	2	**18**	12	5	0.5
10	**Critical and reflective thinking,**	0	3	**11**	**11**	**12**	**0.9**
11	**Creativity and innovation,**	0	3	**12**	**12**	**10**	**0.8**
12	Technical, domain competence.	0	3	9	**19**	5	0.7
13	**Communication skills.**	0	1	7	**21**	**7**	**1**
14	**Analytical, design, debugging skills.**	0	3	**11**	**12**	**11**	**0.8**
15	**Decision making skills.**	0	0	**10**	**23**	**4**	**0.8**
16	**Project planning, management.**	1	1	**8**	**20**	**7**	**0.8**
	Total	2	44	164	239	130	
	Average	0.1	2.8	**10.3**	**14.9**	**8.1**	**0.8**

In terms of recalling the significant advantages of mentoring, in the context of their later academic/professional activities, some comments of these 37, and other 9 alumni respondents, are given in Table A23.2.

Table A23.2: Advantages of mentoring as identified by alumni

1.	Properly defining problem
2.	… best thing I learnt was to look at the other side of the coin …
3.	…ability to move from macro to micro details and vice versa, patience and openness to critically analyze alternative approaches…
4.	… working with unknown person or a team …
5.	… Working in such large team and coordinating with multiple project …
6.	Things which I thought I understood were actually understood while I was making someone else understand. … It helps the mentor grow in almost every dimension … subject matter is strengthened and he gains clarity … one gets to hear his own thoughts … It's one of the best ways to discover ourselves and our creativity.
7.	Makes you feel like a bigger person. Makes you believe in yourself more when others believe in you …
8.	… instilled a sense of an extra added responsibility…
9.	… I had to explain them in a simple manner…
10.	… improved my ability to present the same topic from different angles …
11.	… my confidence increased as I matured with classes, my tolerating power increased … my ability to think out of the box and also trying to think more than students and also commenting on their performance increased my critical analysis ability...
12.	Having to explain one's thinking to someone else seems to help get it straight in one's own mind …
13.	… one is able to find out gaps in knowledge and determine understanding of the subject …
14.	… I realized that every problem could be solved through different techniques … Mentoring helps thinking out of the box … the joy you get when they come out with flying colors is incomparable
15.	You tend to bring out the best in you
16.	Questions thrown up by the mentee sometimes made me look deeper for some concepts to which I had never paid much attention earlier …
17.	I was able to better revise my subjects …
18.	… communicate effectively, use and upgrade his own skills
19.	… I also noticed a change in the way I started explaining things to other people …
20.	… It gave me the chance to continuously improve myself …
21.	Mentoring provide inner satisfaction. … makes you a better person …you have to critically analyze the drawbacks and tradeoffs and justify your advisee, which makes things clearer to you … you learn how to read and understand someone else's code … more responsible, more disciplined … It motivates you to become better at your own work
22.	It helped me shape my personality and enhanced my leadership and interpersonal skills. … my tolerating power and patience had surely increased … I was able to communicate much better to different people and could express my ideas in a more effective manner.
23.	…Self-confidence level increased … got to know varied and completely out of the box concepts … patience level increased. I had to give a logical explanation as to why this idea will/will not work… understood that teaching is not an easy job…
24.	The decision making and project management skills that got polished during the mentoring really helped me in long term

Appendix A24: Advantages of Mentoring as Identified by Final Year Students Involved in Cross-level Mentoring of Juniors, 2009

1.	good revision of all fundamentals and some good genuine doubts solutions
2.	… deal with my subordinates
3.	… unique addition to my ability
4.	give my hundred percent knowledge and also act like a team leader
5.	… into every problems in different ways and helps us to find various solutions.
6.	Patience and listening
7.	Enhancing my teaching skills
8.	… think more and think in line with the people working with me and in my surroundings
9.	helpful to me for some higher examinations
10.	deeper understanding
11.	I am gaining on mentoring skills and ways to communicate a problem to different people. Also it is helping me understand the mind of different coders.
12.	… now that we are going to sit for placements, it's very important
13.	Communication skill in explaining ourselves to others
14.	I am much more expressive now and can explain and present things better
15.	boosts my confidence and helps me in the process of self-learning
16.	understand the responsibilities and duties of being a supervisor
17.	Building rapport with different kinds of students, understanding others; code, Taking responsibility
18.	I have found a teacher inside me.
19.	would definitely aid me in applying for Teaching Assistantship
20.	built my leadership quality a lot
21.	I want to become a lecturer so it's helping me understand the student mind
22.	Improved leadership skills, multiple perspectives
23.	I can now understand the problems which a new comer faces
24.	Keeps me update
25.	I am strengthening my concepts of programming
26.	be more receptive to the problems of others
27.	quality of working as a team leader and resolving the problems faced by the people
28.	inculcating qualities of a project manager
29.	will definitely help me in campus selection
30.	I have clarified my concepts on requirement engineering which has helped me in my final year project report
31.	software quality and testing concepts along with designing
32.	how to approach towards a given problem
33.	learning some new technologies
34.	It helps me to understand how a problem is perceived differently by different people and hence helps me to understand the common error which a coder can do and in future I'll try to remove those technical snags which usually don't come to mind
35.	Broadened our mental skills

Annexure AN1: Important Theories about Human Learning, Intelligence, and Thinking

During the course of this study, we have studied a large number of theories of education, 'learning', intelligence, human development, curriculum design, and thinking. Tables A'1.1a and A'1.1b list some of these important theories and modes.

Table AN1.1a: A chronological list of some important theories about human learning, intelligence, and thinking (pre 1990)

1. Connectionism (Thorndike, 1913)	36. Approaches to learning (Marton and Saljo, 1976)
2. Genetic epistemology (Piaget, 1915)	37. Social learning theory (Bandura, 1977)
3. Theory of Curriculum (Bobbit, 1918)	38. Theory of tri-archic intelligence (Sternberg, 1977)
4. Social development theory (Vygotsky, 1920s)	39. Script theory (Schank, 1970s and 80s)
5. Gestalt theory (Wertheimer, 1924).	40. Modes of learning (Norman and Rumelhart, 1978)
6. Theory of cognitive development (Piaget, 1930s onwards)	41. Logical categories of learning (Bateson, 1979)
7. Contiguity theory (Guthrie, 1938)	42. Flow theory of motivation (Csikszentmihalyi 1979)
8. Fluid and crystallized intelligence (Cattell, 1941)	43. Four quadrant model of the brain (Herrmann's 1979)
9. A theory of human motivation (Maslow, 1943)	44. Repair theory (Brown and VanLehn, 1980)
10. Theory of inventive problem solving (TRIZ/TIPS) (Altshuller, 1946)	45. Self determination theory (Deci and Ryan, 1980 onwards)
11. Phenomenology (Rogers, 1951),	46. Adult learning theory (Cross, 1981)
12. Information processing theory (Miller, 1956)	47. Structure of the Observed Learning Outcomes (SOLO) Taxonomy (Biggs and Collis, 1982)
13. Taxonomy of educational objectives (Bloom, 1956)	48. Multiple intelligence theory (Gardner, 1983)
14. Cognitive dissonance theory (Festinger, 1957)	49. Component display theory (Merrill, 1983)
15. Motivation to work (Herzber, 1959)	50. Tri-archaic theory of intelligence (Sternberg, 1970s and 80s)
16. Two cultures (Snow, 1959)	51. Learning style and experiential learning theory (Kolb, 1984)
17. Originality (Maltzman, 1960)	52. Concept mapping and Vee mapping (Novak and Gowin, 1984)
18. Conditions of learning (Gagne, 1962)	53. Nature of moral stages (Kohlberg, 1984)
19. Systems thinking (Emery and Trist, 1965)	54. Mathematical problem solving (Schoenfeld, 1985)
20. Constructivist theory (Bruner, 1966)	55. Intellectual functioning in three levels (Costa, 1985)
21. Structure of intellect (Guilford, 1967)	56. Levels of professional expertise (Dreyfus brothers, 1985)
22. Lateral thinking (Edward de Bono, 1967)	57. Women's 5 ways of knowing (Belenky et al, 1986)
23. Experiential learning (Rogers, 1960s)	58. Cognitive load theory (Sweller, 1988)
24. Sub-sumption theory (Ausubel, 1960s)	59. Cognitive apprenticeship (Collins et al, 1987)
25. The stage theory (Atkinson and Shiffrin 1968)	60. Four perspectives on professional expertise (Kennedy, 1987)
26. ERG theory (Alderfer, 1969)	61. Knowing in action (Schön, 1987)
27. Intellectual and ethical development (Perry, 1970)	62. 3P model (Biggs, 1987-99)
28. Androgogy (Knowles, 1970)	63. Dimensions of learning (Marzano, 1988)
29. Levels of processing (Craik and Lockart, 1970s)	64. Mental self-government learning theory (Sternberg, 1988)
30. Framework of reflective activities (Borton, 1970)	65. Style of learning and teaching (Entwistle, 1988)
31. Conscious competence theory (Gordon Institute, early 1970s)	66. Framework for reflection (Gibbs, 1988)
32. Classification of disciplines (Biglan, 1973)	67. Cognitive load theory (J. Sweller, 1988)
33. Attribution theory (Weiner, 1974)	68. Framework for reflection on action (Smyth, 1989)
34. Conversation theory (Pask, 1976)	
35. Double loop learning (Chris Argyris, 1976)	

Table AN1.1b: A chronological list of some important theories about human learning, intelligence, and thinking (1990 onwards)

69. Minimalism (Carrol, 1990)	90. Constructivist alignment (Biggs, 1999)
70. Situated learning (Lave and Wenger, 1991)	91. Phases in critical reflective inquiry (Kim,1999)
71. Investment theory of creativity (Sternberg, 1991)	92. Collaborative learning (Dillenbourg, 1999)
72. Curriculum integration (Fogarty, 1991)	93. Heutagogy (Hase and Kenyon, 2000)
73. Cognitive flexibility theory (Spiro et al, 1992)	94. Taxonomy of learning (Marzano, 2000)
74. Capability (Stephenson, 1992)	95. Framework of critical thinking (Minger, 2000)
75. Model of critical thinking (APA, 1992-2006)	96. Taxonomy of Curriculum Integration (Harden 2000)
76. Epistemological reflection model (Baxter-Magolda, 1992)	97. Learning Style (Entwistle, 2001)
77. Value inventory (Schwartz, 1992)	98. Bloom's revised taxonomy (Anderson & Krathwohl, 2001)
78. Learner managed learning (Graves, 1993)	99. Story centered curriculum (Schank, 2002)
79. Reflective judgment model (King and Kitchener, 1994)	100. Models of interplay between emotions and learning (Kort, 2001)
80. Learning by design (Kolodner et al, 1995-2004)	101. Balance theory of wisdom (Sternberg, 2003)
81. Model of critical thinking (Paul, 1996)	102. Community of practice ellipse (Medeni, 2004)
82. Work-based learning (Gattegno, 1996; Hase, 1998).	103. Spiral of experience based action learning (SEAL) (Medeni, 2004)
83. CHC theory (McGrew 1997, Flanagan 1998)	104. Taxonomy of knowledge Types (Carson, 2004)
84. Intelligence as developing expertise (Sternberg, 1997)	105. Theory of successful intelligence, (Sternberg, 2005)
85. Framework of learning style (Vermunt, 1998)	106. Framework for information and information processing of learning systems (Rauterberg, 2005)
86. Socialisation, Externalisation, Combination, and Internatisation (SECI) (Noanaka &Takeuchi, 1998)	107. Six factors of psychological well-being (Ryff & Singer, 2006)
87. Action learning (Kemmis & McTaggart, 1998)	108. Teaching for wisdom, intelligence, creativity, and success (Sternberg et al, 2009)
88. Propulsion theory of creativity (Sternberg, 1999)	
89. Ergonagy (Tanaka and Evers, 1999)	

Annexure AN2: Competency Recommendations by Accreditation Boards of Some Countries

The EC2000 criteria defined by Engineering Accreditation Commission (EAC) of Accreditation Board for Engineering and Technology (ABET), United States [90], recommends that engineering graduates must attain:

a. An ability to apply knowledge of math, science, and engineering,
b. An ability to design and conduct experiments, as well as analyze and interpret data,
c. An ability to design a system, component or process to meet desired needs,
d. An ability to function in multi-disciplinary team,
e. An ability to identify, formulate and solve engineering problems,
f. An understanding professional and ethical responsibilities,
g. An ability to communicate effectively,
h. An understanding the impact of engineering solutions in a global and societal context,
i. A recognition of need and ability to engage in life-long learning,
j. A knowledge of contemporary issues, and
k. An ability to use the techniques, skills and modern engineering tools necessary for engineering practice.

The **Technology Accreditation Commission (TAC) of ABET** prescribes the following abilities for the graduates of an engineering technology program [91]:

a. An appropriate mastery of the knowledge, techniques, skills and modern tools of their disciplines,
b. An ability to apply current knowledge and adapt to emerging applications of mathematics, science, engineering and technology,
c. An ability to conduct, analyze and interpret experiments and apply experimental results to improve processes,
d. An ability to apply creativity in the design of systems, components or processes appropriate to program objectives,
e. An ability to function effectively on teams,
f. An ability to identify, analyze and solve technical problems,
g. An ability to communicate effectively,
h. A recognition of the need for, and an ability to engage in lifelong learning,
i. An ability to understand professional, ethical, and social responsibilities,
j. A respect for diversity and a knowledge of contemporary professional, societal and global issues, and
k. A commitment to quality, timeliness, and continuous improvement.

The **Computing Accreditation Commission (CAC) of ABET** [92] has proposed that the program outcomes for information technology and similarly named computing programs should minimally include the following abilities:

a. Use and apply current technical concepts and practices in the core information technologies;
b. The ability to analyze, identify and define the requirements that must be satisfied to address problems or opportunities faced by organizations or individuals,
c. Design effective and usable it-based solutions and integrate them into the user environment,
d. Assist in the creation of an effective project plan,

e. Identify and evaluate current and emerging technologies and assess their applicability to address the users' needs,
f. Analyze the impact of technology on individuals, organizations and society, including ethical, legal, security, and global policy issues,
g. Demonstrate an understanding of best practices and standards and their application,
h. Demonstrate independent critical thinking and problem solving skills,
i. Collaborate in teams to accomplish a common goal by integrating personal initiative and group cooperation,
j. Communicate effectively and efficiently with clients, users and peers, both verbally and in writing, using appropriate terminology, and
k. Recognize the need for continued learning throughout their career.

The **United Kingdom Standards for Professional Engineering Competence (UK-SPEC)** [93] has prescribed that an Incorporated Engineer must be able to:
a. Use a combination of general and specialist engineering knowledge and understanding to apply existing and emerging technology,
b. Apply appropriate theoretical and practical methods to design, develop, manufacture, construct, commission, operate and maintain engineering products, processes, systems, and services,
c. Provide technical and commercial management,
d. Demonstrate effective interpersonal skills, and
e. Demonstrate a personal commitment to professional standards, recognizing obligations to society, the profession and the environment.

The **UK-SPEC further refines the first two of these competencies for Chartered Engineers**. A Chartered Engineer must be able to:
a. Use a combination of general and specialist engineering knowledge and understanding to optimize the application of existing and emerging technology, and
b. Apply appropriate theoretical and practical methods to the analysis and solution of engineering problems.

The **Institution of Engineers, Singapore (IES)** [94] defines the following competencies as part of its accreditation criteria of engineering programs:
a. Apply knowledge of mathematics, science and engineering,
b. Design and conduct experiments, analyze, interpret data and synthesize valid conclusions,
c. Design a system, component, or process, and synthesize solutions to achieve desired needs,
d. Identify, formulate, research through relevant literature review, and solve engineering problems reaching substantiated conclusions,
e. Use the techniques, skills, and modern engineering tools necessary for engineering practice, with appropriate considerations for public health and safety, cultural, societal, and environmental constraints,
f. Communicate effectively,
g. Recognize the need for, and have the ability to engage in life-long learning,
h. Understand the impact of engineering solutions in a societal context and to be able to respond effectively to the needs for sustainable development,

i. Function effectively within multi-disciplinary teams and understand the fundamental precepts of effective project management, and
j. Understand professional, ethical and moral responsibility.

The **Engineers Australia Accreditation Board** [95] has identified similar generic attributes that are as follows:
a. Ability to apply knowledge of basic science and engineering fundamentals,
b. Ability to communicate effectively, not only with engineers but also with the community at large,
c. In depth technical competence in at least one engineering discipline,
d. Ability to undertake problem identification, formulation, and solution,
e. Ability to utilize a systems approach to design and operational performance,
f. Ability to function effectively as an individual and in multi-disciplinary and multi-cultural teams, with the capacity to be a leader or manager as well as an effective team member,
g. Understanding of social, cultural, global and environmental responsibilities of the professional engineers and the need of sustainable development,
h. Understanding of the principles of sustainable design and development,
i. Understanding of professional and ethical responsibilities and commitment to them, and
j. Expectation of the need to undertake lifelong learning, and capacity to do so.

The **Japan Accreditation Board for Engineering Education (JABEE)** [96] emphasizes the following competency set:
a. The ability and intellectual foundation for considering issues from a global and multi-lateral viewpoint,
b. Understanding of the effects and impact of technology on society and nature, and of engineers' social responsibilities (engineering ethics),
c. Knowledge of mathematics, natural sciences and information technology, and the ability to apply such knowledge,
d. Specialized engineering knowledge in each applicable field, and the ability to apply such knowledge to provide solutions to actual problems,
e. Design abilities to organize comprehensive solutions to societal needs by exploiting various disciplines of science, engineering and information,
f. Japanese-language communications skills including methodical writing, verbal presentation and debate abilities, as well as basic skills for international communications,
g. The ability to carry on learning on an independent and sustainable basis, and
h. The ability to implement and organize works systematically under given constraints.

Accreditation criteria defined by NBA, India

Table AN1.1: Accreditation Criteria and Weights defined by NBA, India for Diploma (Dip.), Undergraduate (UG), and Postgraduate (PG) Engineering Programs

No	Parameters	Max. Marks		
		Dip.	UG	PG
1	Organization and governance Planning and Monitoring, Recruitment Procedure & its Effectiveness, Promotional Policies/Procedure, Leadership, Motivational Initiatives, Transparency, Decentralization and Delegation & participation of faculty, and Constitution of General council and bodies.	30	80	50
2	Financial resources, allocation, and utilization Budget allocated to the Institution and Utilization. Budget allocated to the Department and Utilization.	70	70	50
3	Physical resources (central facilities) Students' Hostel, Power back up, Reprographic facilities, Bank, Post Office, Counseling and Guidance, Language Lab., Medical Facility, Internet Facility, Canteen, and Transport.	50	50	50
4	Human resources: faculty and staff Faculty Numbers, Student Faculty Ratio, Cadre ratio, Average experience, faculty retention, Turnover, Qualifications, Participation of faculty in Institutional development/Departmental development/Academic matters/Students, Development/Self growth, Implementation and Impact of Faculty Development initiatives, Analysis and Follow-up of Performance appraisal, Service rules, pay package, and incentives. Support Staff (Technical/Administrative) Numbers, Qualification/skills, and Skill up-gradation.	200	200	200
5	Human resources: students Student admissions, Academic results, Performance in competitive examinations, and Placement.	100	100	100
6	Teaching-learning processes Delivery of syllabus, contents, Contents beyond the syllabus, Academic calendar, Continuous evaluation procedure, Utilization of Laboratories, Information access facilities, Student-centric learning initiatives, Students feedback.	450	350	250
7	Supplementary processes Extra & co-curricular activities, Personality Development initiatives, Professional society activities, Entrepreneurship Development, Alumni Interaction, Ethics, and Students Publications/Awards.	50	50	50
8	Research & development and interaction effort Budget for in-house R&D activities and its utilization, Academic/Sponsored/Industrial research and development, Publications and Patents, Industry participation in developmental and student related activities, Continuing Education, Consultancy and Testing, Students' Project Work.	50	100	250

Annexure AN3: Some Models for Classification of Competencies

Bloom

In 1956, Benjamin Bloom [133] arranged the educational objectives into six major levels in a hierarchical order. Beginning with the simplest level and increasing in complexity, these levels are: *Knowledge, Comprehension, Application, Analysis, Synthesis and Evaluation.*

Anderson and Krathwohl

Anderson and Krathwohl modified Bloom's taxonomy by *adding another dimension of knowledge types: Factual, Conceptual, Procedural, and Meta-cognitive*. They renamed the levels of earlier dimension from nouns to verbs [134]. They also interchanged the positions of the uppermost two levels.

Costa's model of intellectual functioning

In 1985, based on this taxonomy, Costa [135] proposed his *model of intellectual functioning* with the following *three levels*:

ii. Recalling information
   b. Remember: repeat, match, …
   c. Show understanding: paraphrase, give example, locate, summarize, …

iii. Making sense of gathered information
   d. Use understanding: operate, apply, demonstrate, infer, relate, …
   e. Examine: compare, diagram, categorize, point out, question, outline, …
   f. Create: compose, design, prepare, modify, formulate, plan, compile, …

iv. Applying or evaluating information
   g. Decide: judge, predict, estimate, select, conclude, rate, evaluate, …
   h. Supportive evidence: prove/support your answer, why or why not? ...

Kennedy's perspectives on professional expertise

Kennedy [136] proposed four alternative perspectives on professional expertise: technical skills, application of theory or general principles, critical analysis, and deliberate action. If we view these perspectives as manifestations of different types of emphasized competencies, these can

also be used as a classification of competencies. Kennedy observed that different perspectives were dominant in different professions and engineering education shifted its emphasis from the first to the second perspective after the 1950s. Passow [78] has called for an appropriate balance of all these four perspectives for designing engineering educational programs.

Categories of competencies expected of college graduates (Stark et al)

Stark et al [137] advocated to *blend the professional and liberal education*, and also classified the competencies expected of college graduates into three broad categories: *traditional professional competencies, liberal professional competencies, and attitude*. As per their classification, *traditional professional competencies* comprised of conceptual, technical, and integrative competencies as well as career marketability. The second category of *liberal professional competencies* included interpersonal (communication), contextual, and adaptive competencies as well as critical thinking and leadership capacity. The third category of *attitudes* integrated professional identity, professional ethics, scholarly concern for improvement, motivation for continued learning, and aesthetic sensibility.

Marzano's Revised Taxonomy

In 2000, Marzano [140] proposed his modifications as a *two dimensional taxonomy*: (i) knowledge domain comprising of information, mental procedures, and psychomotor procedures, and (ii) processing in cognitive, meta-cognitive and self-system providing the following hierarchical levels of processing:

1. Cognitive system: processes all the necessary information, and
    a. Retrieval
    b. Comprehension
    c. Analysis
    d. Knowledge utilization
2. Meta-cognitive systems: sets goals and keeps track of how well they are being achieved
3. Self-system: decides whether to continue the current behavior or engage in the new activity

Various Competency Classification Schemes Cited by García-Aracil and Van der Velden

García-Aracil and Van der Velden [132] have studied the required competencies of graduates with reference to the requirements of the new situation in the European labor market. They have cited the following earlier *competency classification schemes* proposed in the last twenty years:

1 Becker: *general and firm specific*,
2 Nordhaug: *firm specific, task specific, and industry specific*,
3 Heijke: *acquired at school and of direct use in later work, acquired at school which facilitate acquisition of new competencies after school, and those that are acquired mainly in work context*,
4 Bunk: *specialized, methodological, participative, and socio-individual*, and
5 Kellerman: *general academic, scientific operative, personal professional, socio-reflexive, physiological handicraft*.

Coate's schema for curriculum design

Kelly Coate [141] developed a *schema for curriculum design*. It included three overlapping domains of *'knowing,' 'acting,' and 'being.'* She suggested that the crucial aspect of this schema is the domain of 'being.' Though all these models have been used by several education researchers, they have not yet attracted any noticeable attention of computer sciences education researchers.

Annexure AN4: Metzger's Observations about Debugging

Design errors

Metzger observes that *design errors* may occur because of *errors in data-structure, algorithm,* or *interface specifications related to user-interface, software-interface, or hardware-interface* [157].

He has also enumerated some common *conception stage errors* in software development. The *data structure related errors* are: missing/incorrect/unclear/contradictory/out-of-order data definition, missing/incorrect/out-of-order shared-data access control, capacity limitation, inappropriate representation resulting in data loss, ignored input or intermediate data storage requirement, and slow access to data. *Algorithm related errors* include: invalid assumptions about input/program state, omission of logical possibilities, high time-complexity, and missing/superfluous/incorrect/out-of-order logic-sequence/input-check/output-definition/special condition handler. Conception *errors about interface* include: invalid assumptions about users, collateral software, or hardware, missing/superfluous/incorrect/unclear/out-of-order specification item with reference to user interface, software-interface, or hardware-interface.

Coding errors

Metzger posits that *coding errors* include initialization errors, finalization errors, binding errors, reference errors, static/dynamic data structure errors, memory problems, missing operations, extra operations, control flow problems, value precision errors, invalid expressions, incorrect usage of or defect in compiler/tools/system library/third-party library/operating system.

Errors because of rule-based reasoning

Metzger catalogues the software errors because of rule-based reasoning into two broad categories: (i) misapplication of good rules occur when a time-tested rule is applied by overlooking the additional conditions that warrant another rule, (ii) application of a bad rule occurs when conditions are wrongly represented, or ineffective/inefficient action is chosen. The first category includes errors in the sub-categories of 'general rule, exception condition,' 'conflicting signals,' and 'excess information, multiple rule match.' Humans make rigidity errors

because they have a strong bias to apply techniques that worked in the past, even though the current circumstances are no longer the same. In the second category of applying bad rules, Metzger includes the subcategories of 'incorrect rule conditions,' 'incorrect rule actions,' 'frequently ineffective rules,' 'formerly effective rules,' and 'occasional effective rule.'

Debugging as a search problem

Metzger has viewed debugging as a search problem like mathematical problem solving that is solved using a variety of search strategies like binary search, greedy search, depth-first search, and breadth-first search.

Heuristics for solving debugging problems

(i) stabilize the problem, (ii) create a standalone test case, (iii) categorize the problem with reverence to correctness, completion, robustness, and efficiency, (iv) describe the problem according to a standard methodology, (v) explain the problem to someone else, (vi) recalling a similar problem, (vii) drawing diagrams like control flow graph, data flow graph, and complex data structures with pointers, and (viii) choosing a hypothesis from historical data.

He has also suggested some strategies like program slice strategy, deductive reasoning strategy, and inductive reasoning strategy for debugging.

Annexure AN5: Lethbridge's Study on Most Important and Influential Topics in Software Development Education

Lethbridge [46-48] surveyed approximately 200 practicing software engineers and managers. The respondents had degrees in computer science, computer engineering, electrical engineering, information systems, software engineering, and other engineering disciplines. They represented a broad cross-section of the industry, and developed software for management information systems, data processing, consumer or mass market software, real-time systems, and other application software. They were asked to rate educational topics on the basis of four criteria: (Q1) how much they had learned about it in their formal education, (Q2) how much they know now about it, (Q3) how important the topic has been in their career, and (Q4) how much influence the topic had on their overall thinking.

Lethbridge included a total of seventy-five topics from thirteen subject categories in the survey. The ten topics identified by them as most important in terms of career related utility of details of topic and also overall influence on thinking were: specific programming languages, data structures, software design and patterns, software architecture, requirements gathering and analysis, HCI/user interfaces, object-oriented concepts and technology, ethics and professionalism, and analysis and design methods.

In terms of the perceived gap between Q3 and Q4 compared to Q1, out of the thirteen subject categories, the respondents felt most serious deficiencies in the three categories of software engineering process, humanities and skills, and software design core.

Their report shows that *five out of the thirteen subject categories did not contribute even a single topic to the list of twenty-five most important and influential topics*, while these categories were felt by the respondents to be over emphasized in the curriculum. *These subject categories are theoretical computer science, mathematical topics in computer science, other hardware topics, general mathematics, and basic science.*

Annexure AN6: Some Important Models on Problem Solving

Jonassen's Taxonomy of Problems

Jonassen [193] has proposed a taxonomy of problems based on variations in problem types and representations. The problem types vary in a three dimensional continuous space of three factors: *structured-ness*, *complexity,* and *degree of domain specificity*. The first among these is *structured-ness*, varying from extremely well-structured to absolutely ill-structured in a continuum, as discussed above. The second factor is *complexity* that depends upon a number of issues, functions, variables, and also interactions and degree of uncertainty of behavior of these. The third factor is *degree of domain specificity*.

Based on the cognitive task analysis of various kinds of problems, Jonassen has identified eleven different kinds of problems – (i) logical problems, (ii) algorithmic problems, (iii) story problems, (iv) rule using problems, (v) decision making problems, (vi) troubleshooting problems, (vii) diagnostic-solution problems, (viii) strategic-tactical performance, (ix) situated case problems, (x) design problems, and (xi) dilemmas. It may be noted that as per his classification, algorithmic problems deal with direct application of known algorithms.

Polya's Model on Mathematical Problem Solving

Polya [195] listed four phases of problem solving: (i) understand the problem, (ii) plan the solution, (iii) execute the plan, and (iv) review the results. Table AN6.1 gives further details for each of these phases.

Table AN6.1: Polya's recommended cognitive engagement of mathematical problem solving

1. Understand the mathematical problems: (i) what is unknown, (ii) what is data, (iii) what is the condition, (iv) is the condition sufficient/insufficient/redundant/contradictory to determining unknown, (v) draw a figure, introduce suitable notation, and (vi) separate the various parts of the condition.
2. Plan for solution finding: (i) is the problem familiar, (ii) identify related problems, (ii) identify related theorem(s), (iii) identify a similar problem that has been solved before for similar unknown, (iv) is the solution plan of such problem reusable in terms of results and/or method (with some auxiliary elements, if needed), (vii) restate the problem in different manners, (viii) go back to the definitions, (ix) if the problem can't be solved, solve some related problem that may be more general, more specific, more special, or analogous, solve some part of the problem, (x) how far can the unknown be determined by dropping or varying part of the condition, and can something useful be derived from this data? (xi) think of other data appropriate to determine the unknown, (xii) can data and unknown be changed, and/or brought nearer to each other? and (xiii) have all the data, condition, essential notions being considered?
3. Plan execution: Polya recommended engagements like –(i) carry out the plan and check each step, (ii) can you see clearly that the step is correct? and (iii) can you prove that it is correct?
4. Review stage: (i) check the result, (ii) check the argument, (iii) can you derive the result differently? (iv) can you see it at a glance? and (v) can the result or the method be used for solving some other problem?

Galotti's Collation of Some Techniques for Solving Puzzle-like Problems

Galotti collates some general domain independent techniques for puzzle-like problems [196].

1. Generate and Test involves generating possible solutions and then testing them. It is useful when there are only a few possibilities to track, and loses its effectiveness rapidly when there are many possibilities, and when there is no particular guidance for the generation process.
2. Means-Ends Analysis consists of comparing the goals with the starting point, thinking of possible ways of overcoming the gap, and choosing the best. If required, the sub-goals are created to break down the task into manageable steps. It does not necessarily ensure the best solution.
3. Working backward also reduces the gap between current state and the goal state by determining the last step need to achieve the goal, then for next to last step, and so on. It is very effective when the backward path is unique.
4. Backtracking involves making provisional choices, and unmaking the wrong choices if they turn out to be wrong so that one can back up to a certain point of choice and start over again by making newer choices.
5. Reasoning by analogy works when the problem solver is able to form an abstract schema of the presented stories, and apply the same to new analogous problem. Research has shown that not many persons are able to form such schema and see the analogy unless told to do so.

Nickols' typology of problem solving approaches

Nickols proposed a typology of problem solving approaches [198]. A repair approach is required to put things back the way they were, improvement approach is required to improve upon existing arrangements, and engineering approach is suitable for creating new, far superior arrangements. The repair approach starts from symptoms and focuses on causes/corrective measures through fault isolation. The improvement approach starts from existing systems/arrangements and focuses on constraints/modifications through structural analysis. The engineering approach starts from the required results and focuses on required design through structural design.

16 Habits of Mind, Costa and Kallick

Costa and Kallick [203] have identified the following sixteen characteristics of what intelligent people do when they are confronted with problems, the resolution to which is not immediately apparent - (i) persisting, (ii) managing impulsivity, (iii) listening to others with understanding and empathy, (iv) thinking flexibly, (v) thinking about our thinking (meta-cognition), (vi) striving for accuracy and precision, (vii) questioning and posing problems, (viii) applying past knowledge to new situations (ix) thinking and communicating with clarity and precision, (x) gathering data through all senses, (xi) creating, imagining, and innovating, (xii) responding with wonderment and awe, (xiii) taking responsible risks, (xiv) finding humor, (xv) thinking interdependently, and (xvi) learning continuously.

Annexure AN7: Some Theories on Attention

Galotti gives an excellent account of their findings on this aspect [196]. We give a brief summary in Annexure AN6. The term 'selective attention' means that we usually focus our attention on one or a few tasks or events rather than on many. In 1958, Broadbent proposed his 'filter theory' which specified that we could only attend to one stimulus at a time. In the 1960's, Anne Treisman proposed her 'attenuation theory' as a modification to the filter theory. She suggested that rather than being fully blocked and discarded, unattended signals are weakened and some information is retained for future use.

In the 1960's, Deutsche and Deutsch, and also Norman, proposed their 'late selection theory,' taking a position that all messages are routinely processed for at least some aspects of meaning – the selection of message for response happens later. At low level of alertness, only very important messages captured attention, whereas at higher level of alertness, less important messages can be processed. In 1978, Johnston and Heinz proposed a broader model in the form of 'multimode theory,' which viewed attention as a flexible system that allows selection of a message over others at several different points. Later selection requires more processing, capacity, and effort.

In 1973, Kahneman presented his model of attention viewing that the availability of mental resources is affected by overall level of arousal, or state of alertness. In the 1980's, Anne Treisman showed that *perceiving individual features takes little effort or attention, whereas gluing features together into a coherent object requires more.* As per the '*capacity theory of comprehension*' proposed in 1991, differences in working memory capacity of individuals can account for qualitative and quantitative differences in comprehension. In 2001, Conway et al showed that lower capacity of working memory results in lesser ability to focus. Research has shown that *practice plays an enormous role in performance on simultaneous dual tasks* but there are serious limitations on the number of things we can do simultaneously. Complex individual tasks make it even more difficult.

Annexure AN8: Some Important Perspectives on Curiosity

Arnone [245] cites Daniel Berlyne, who in 1960's had identified two form of curiosity - diversive (e.g., novelty seeking) and specific (e.g., uncertainty, conceptual conflict, information seeking). Arnone also refers to Loewenstein's information gap theory of specific epistemic curiosity, according to which a feeling of deprivation occurs when an individual becomes aware of a difference between "what one knows and what one wants to know."

Peterson et al [246] view curiosity as one of the core cognitive virtues for all humans. According to their meta-analysis of various philosophical perspectives and research findings *curiosity includes interest, novelty seeking, and openness to experience. It implies taking an interest in ongoing experience for its own sake, finding topics and subjects fascinating, as well as tendencies for exploring and discovering.*

Peterson et al have given an excellent account of research on curiosity [246]. Cognitive process theory of curiosity results from two traits of openness to novel stimuli and a concern for orderliness. According to this theory curiosity is a function of assimilating and accommodating novel stimulus into one's cognitive map. Personal growth facilitation model of curiosity suggests a four step process – (i) allocation of attention and energy for recognizing and pursuing cues of novelty and challenge, (ii) cognitive evaluation and behavioral exploration of challenging activities, (iii) deep absorption of these activities, and (iv) integration of curiosity experience through assimilation and accommodation. In seemingly boring situations, highly curious people are more oriented towards finding novelty and also sensitive to cues that can increase interest in meaningful and unavoidable activities. Peterson et al cite research that has shown that in college, students with a high curiosity trait asked five times more questions than students with a low curiosity trait.

Annexure AN9: Some Important Perspectives on System Thinking

Senge's Laws for Systems Thinking

Senge emphasizes on *purpose, observing repeated events, and patterns of change* [278]. He developed eleven laws of systems thinking as detailed in Table AN9.1. Solovey has found these laws to be applicable to software development [279].

Table AN9.1: Senge's laws of systems thinking

1	Today's problems come from yesterday's solutions,
2	The harder you push, the harder the system pushes back,
3	Behavior grows better before it grows worse,
4	The easy way out usually leads back in,
5	The cure can be worse than the disease,
6	Faster is slower,
7	Cause and effect are not closely related in time and space,
8	Small changes can produce big results, but the areas of highest leverage are often the least obvious,
9	You can have your cake and eat it too, but not at once,
10	Dividing an elephant in half does not produce two small elephants, and
11	There is no blame.

Characteristics of Systems Thinkers

The highest rated characteristics of engineering systems thinkers found from their empirical study as well as proposed by Sweeney and Meadows [283] are given in Table AN9.2.

Table AN9.2: Characteristics of systems thinkers

Characteristics of engineering systems thinkers (Frank and Waks, 2001)	Characteristics of systems thinkers (Sweeney and Meadows, 2008)
1. Ability to solve system failures 2. Ability to understand complex systems 3. Ability to anticipate the implications of a system 4. Ability to understand the synergy of a given system 5. Ability to see the whole or to perceive how the component functions as a part 6. Multi-disciplinary knowledge in addition to specialization in one 7. Ability to understand generally a new system on his first encounter with it	1 See the whole picture, 2 Change perspectives to see new leverage points in complex systems, 3 Look for interdependencies, 4 Consider how mental models create our futures, 5 pay attention and gives voice to the long-term, 6 'go wide' (uses peripheral vision) to see complex cause and effect relationships, 7 Find where unanticipated consequences emerge, 8 Lower the 'water line' to focus on structure, not blame, and 9 Hold the tension of paradox and controversy without trying to resolve it quickly.

Meadows' Perspectives on Systems Thinking

Meadows [275] posits that systems thinking allows us to (i) reclaim our intuition about whole systems, (ii) hone our ability to understand the parts, (iii) see interconnections, (iv) ask "what if" questions about possible future behaviors, (v) be creative and courageous about system redesign.

Meadows made the following important observations about systems' structure and behavior. A system's behavior may be adaptive, dynamic, goal seeking, self-preserving, and even evolutionary. Among the elements, interconnections, and purpose of a system, its purpose is the most crucial determinant of a system's behavior followed by the interconnections. The elements are usually easily replaceable without changing a system's behavior.

In order to understand the complex behavior of systems, one needs to concentrate on dynamics (behavior over time) of stock and flows in the systems. Stocks generally change slowly and also act as buffers, delays, or shock absorbers, or source of momentum in a system. They also allow the inflows and outflows to be independent and also temporarily out of balance. System thinkers see the world as a collection of "feedback processes" for regulating the levels in the stocks by manipulating flows. Balancing feedback loops are equilibrating or goal seeking structures in systems and are both sources of stability and sources of resistance to change. Reinforcement feedback loops exist in situations where the stocks have the capacity to reinforce or reproduce itself. Such loops may lead to exponential growth or to runaway collapse over time.

Levels of Systems Thinking

Dennis Meadows proposed seven levels of systems thinking expertise as given in Table AN9.3.

Table AN9.3: Levels of systems thinking expertise (Dennis Meadows)

1	Understand the system,
2	Carry out specific decisions,
3	Implement a recommended policy,
4	Modify a mature model,
5	Construct a new model,
6	Teach others to build new models, and
7	Guide organizational change.

Boulding's nine-level hierarchy of real world complexity

Boulding [276] proposed the following *nine-level hierarchy of real world complexity,* as shown in Table AN9.4. Software developers need to deal with complexity levels upto third level in this hierarchy. Many applications require them to understand and analyze socio-cultural systems. Hence, depending upon their application domain, they may also be required to understand and analyze complexity levels upto 8^{th} level in this hierarchy.

Table AN9.4: Boulding's hierarchy of real world complexity

1	Structures and frameworks exhibit static behavior,
2	Clockworks exhibit predetermined motion,
3	Control mechanisms exhibit closed-loop control,
4	Open systems exhibit structural self-maintenance,
5	Lower organisms which have functional parts, exhibit blue-printed growth and reproduction,
6	Animals which have a brain to guide behavior, are capable of learning,
7	People who possess self-consciousness, know that they know, employ symbolic language,
8	Socio-cultural systems which are typified by the existence of roles, communications and the transmission of values, and
9	Transcendental systems, the home of 'inescapable un-knowables,' and which no scientific discipline can capture.

Schwartz Value Categories

Schwartz identified ten distinguishable values. Table AN9.5 gives a summary of these value categories.

Table AN9.5: Schwartz Value Categories

1. **Self-Direction.** Independent thought and action; choosing, creating, exploring;
2. **Stimulation.** Excitement, novelty, and challenge in life;
3. **Hedonism.** Pleasure and sensuous gratification for oneself;
4. **Achievement.** Personal success through demonstrating competence according to social standards;
5. **Power.** Social status and prestige, control or dominance over people and resources;
6. **Security.** Safety, harmony, and stability of society, of relationships, and of self;
7. **Conformity.** Restraint of actions, inclinations, and impulses likely to upset or harm others and violate social expectations or norms;
8. **Tradition.** Respect, commitment, and acceptance of the customs and ideas that traditional culture or religion provide the self;
9. **Benevolence.** Preserving and enhancing the welfare of those with whom one is in frequent personal contact (the 'in-group');
10. **Universalism.** Understanding, appreciation, tolerance, and protection for the welfare of all people and for nature.

Annexure AN10: Some Important Perspectives on Intrinsic Motivation

Aristotle identified twelve end motives: confidence, pleasure, saving, magnificence, honor, ambition, patience, sincerity, conversation, social contact, modesty, and righteousness.

Descartes listed six intrinsic motives: wonder, love, hatred, desire, joy, and sadness.

James and McDougall, famous psychologists had identified thirteen basic desires: saving, construction, curiosity, exhibition, family, hunting, order, play, sex, (avoid) shame, (avoid) pain, herd, and vengeance. Construction is a related desire to build and achieve.

In 1938, **Murray** suggested 27 psychogenic need: abasement (to surrender and accept punishment), achievement, acquisition, affiliation, aggression, autonomy, blame avoidance, construction, contrariance (to be unique), counteraction (defend honor), defendance (justify actions), deference (to follow/serve), dominance, exhibition, exposition (provide information/teach), harm avoidance, infavoidance (to avoid failure/shame), nurturance (protect the helpless), order, play, recognition, rejection (to exclude another), sentience (enjoy sensuous impressions), sex, similance (to empathise), succorance (seek sympathy), and understanding.

In 1959, **Herzberg** modified Maslow's model and suggested that man has two sets of needs: hygiene (or maintenance) and motivator. The satisfaction of hygiene factors does not motivate, but absence of these results in dissatisfaction. The motivator factors include achievement, recognition, responsibility, personal growth, advancement, and work itself.

In 1964, **Vroom** proposed his expectancy theory. As per this theory, strength of tendency to act in a certain way depends on the strength of the valence (attractiveness of the outcome to the individual), and strength of expectation that the act will be followed by the given outcome.

In 1969, **Alderfer's** proposed his ERG Theory. This theory viewed needs as a three level hierarchy: existence, relatedness, and growth. The growth needs are satisfied by an individual by making creative or productive contributions. He also postulated that if a person is continually

frustrated in attempts to satisfy growth needs, relatedness needs reemerge as a major motivating force.

Reis [294] identified 16 basic needs - power, curiosity, independence, status, social contact, vengeance, honor, idealism, physical exercise, romance, family, order, eating, acceptance, tranquility, and saving.

Annexure AN11: Successful Practices in International Engineering Education (SPINE) Study

Successful Practices in International Engineering Education (SPINE) is a benchmark study [78a] focusing on the analysis of successful practices in engineering education in ten leading European and U.S. universities including MIT, CMU, and ETH, Zurich. In the SPINE project, *543 professors of these universities, 1372 engineers and 145 managers* of European and US companies were questioned. The study attempted to measure the perceived importance and assessment of fifty-one parameters on *quality of education, teaching methods, engineering competencies, general professional skills,* and *aspects of reputation of institute* through a quantitative analysis.

Engineering and General Professional Competencies

The SPINE study identified and assessed the importance of twenty-one engineering and general professional competencies. The subset of engineering competencies included *ability to develop own engineering expertise, analysis/methodological skills, basic engineering proficiency, development know-how, practical engineering experience, problem solving, research know-how,* and *specialized engineering proficiency.* The general professional competencies included *ability to develop a broad general education, communication skills, English language skills, finance, law, leadership skills, management of business process and administration skills, marketing, other language skills, presentation skills, project management skills, social skills,* and *teamwork skills.*

In the SPINE report [78a], the following observations have been made about respondents' perception of various engineering and general professional competencies:

i. The *highest rated engineering competencies*, both by professors and engineers were *analysis/methodological skills, basic engineering proficiency,* and *problem solving skills*. Engineers and Professors also agree on the lowest rated competencies: *development know-how* and *practical engineering experience*.
ii. Engineers rated *specialized engineering proficiency* and *research know-how* as *lesser important* engineering competencies.

iii. General professional competencies that were considered *very important* are: *communication skills, English language skills, teamwork abilities, presentation skills, and leadership skills*.

iv. General professional competencies that were assigned medium importance are: *social skills, ability to maintain and develop a broad general education, and management of business processes and administration.*

v. General professional competencies of *marketing, finance,* and *other language skills* were rated as lesser important.

vi. All three groups regarded *law* as least important general professional competency.

Teaching Methods

The respondents of SPINE study also rated the effectiveness of *nine teaching methods* of *group projects, homework/out-of-class assignment, industrial training/internship, lecture, projects, practical training, seminars, computer based training,* and *written projects/studies*. In the SPINE report, the following observations have been made about respondents' perception about teaching methods and learning environment:

1 The *best teaching method in the opinion of professors* is diploma/final *projects and lectures.*

2 The engineers gave highest rating to diploma/final projects, but assessed lectures as inferior to written project/studies and practical training.

3 Engineers assessed practical experience in industry internship, seminars, group projects and homework/out-of-class assignments at the same level as lectures, whereas professors had rated the lectures as a superior teaching method (page 77-78 of [78a]).

4 Responding engineers regarded support and counseling for students, and pedagogical and didactic skills of teaching staff, as inadequate and provided the lowest rating to these two parameters out of eight parameters of learning environment. On the other hand, the professors gave a much higher rating to both these parameters (page 80-81 of [78a]).

5 The quality of professors/teaching staff is not high enough in view of the importance of this item (page 74 of [78a]).

Annexure AN12: Some Theoretical Perspectives about Learning and Teaching

Ostrow [307] refers to Dewey' observation "*when knowledge is framed as something one receives, holds, and then releases, the message to students is that all knowledge is preexisting. The world needing to be known is as it is, and no more. We thereby train a populace that could not be more ill-equipped for an active responsiveness to a fluid, constantly changing world.*" Winston Churchill opined *"I am always ready to learn although I do not always like to be taught."* Honan [308] narrated a speech by Hamilton Holt, president of Rollins College, *"… Holt declared that Yale and Columbia, which he had attended in his youth, taught him virtually nothing. … learning takes place most profoundly when students are led to make personal discoveries, often with other students, rather than when inundated with facts and called upon to remember them in examinations."*

While traditional lectures may be an effective pedagogy for some students in some classes, it is probably not the most effective way to teach most classes [309]. Northwood et al [310] have paraphrased Woods' [311] comparison of traditional teaching with ***Problem Based Learning***: "*In a traditional program, students embark on learning by being told what they need to know, learning it, and then being given a problem to illustrate how to use what they have learned. This is a linear, teacher-centered process. Conversely, in PBL, the learning begins with a problem, students identify what they need to know, they learn it, they apply it to the solution of the problem and, most likely, they generate more problems and more learning needs in this cyclical process."*

Merrill [312] has suggested that '*Information is not Instruction*.' A major characteristic of learning is that it is active and interactive [313]. In their work, the authors stress that learning is a social activity, and have suggested that teachers must assume new roles of facilitating and mediating learning rather than merely imparting information, as is done in orthodox classrooms. The authors further stress students' interacting and learning with and from others. Each student can make unique contributions to his/her own learning, and the learning of others because of his/her experiences, knowledge, and cultural background. *Engagement theory*, proposed by Shneiderman et al [314], is based upon the idea of creating successful collaborative teams that

work on ambitious projects, and all student activities involve active cognitive processes such as creating, problem-solving, reasoning, decision-making, and evaluation. Incorporating social learning theory into their experiment, authors report that peer coaching provided the learners the benefits of enhanced knowledge, cognition, and meta-cognition [315].

Schank [316] opines that much of human reasoning is case-based rather than rule-based, and to be really valuable, generalization has to be constructed by learners themselves. Having a broad, well-indexed set of cases is what differentiates the expert from the textbook-trained novices. According to him, generalizations that are told, have no place to sit in memory, and no cases to tie together, are quickly forgotten from lack of use. Arias et al [317] and Fischer [318] have studied various design domains, and have concluded that the knowledge to understand, frame, and solve problems evolves during the problem solving process.

Instructional design theory database project [319] and the 'Theory Into Practice Database' (TIP) [320] provide excellent explanation of several theories related to instructional design theories and learning. *Instructional Transaction Theory* [321], *Open Learning Environments* [321], *Constructivist Learning Environment* [322], *Anchored Instruction* [323], *Case Study Method of Instruction* [324], *First Principles of Instruction* [325], *Collaborative Problem solving* [321], and *Problem-based Instruction* encourage an open-ended learning environment with emphasis on self-learning to promote critical thinking, and heuristic based learning in ill-defined domains. *Cognitive Apprenticeship* and *Learning by doing* [321] recommend that in order to develop real-life problem solving ability, classroom content should have a real-life context and learners should be engaged in performing and reviewing the tasks. *Elaboration Theory* recommends teaching of broader concepts before narrower concepts, and teaching of procedural and heuristic tasks should follow an expert's way of thinking. *Four Component Instruction Design Model* recommends that complex cognitive skills can be developed by engaging the learners in concrete whole process tasks.

Cognitive Flexibility Theory [206] posits that the traditional linear teaching may be ineffective for ill-structured knowledge domains. *Aptitude-Treatment Interaction* posits that highly structured treatment is good for low-ability students, but hinders high-ability students. *Random*

Access Instruction also stresses upon the need to spontaneously restructure one's knowledge in order to respond to a varied and changing situational demands. These theories recommend that for developing this flexibility, especially for ill-structured domains, rather than using over-simplification, compartmentalisation, and transmission of knowledge, instruction should support its context dependence, multiple representations, construction, and interconnectedness.

Many of the attitudes, perceptions, and values considered to be important for software development activities cannot be assumed to naturally develop as a natural result of traditional computing education. Change of attitude, perceptions, and values necessarily requires deconstruction of some of the existing beliefs. Kort and Reilly [326] view learning as a spiral process of construction and de-construction (of misconceptions) phases through positive as well as negative emotions. As per ***Cognitive Dissonance Theory*** [327] and also the *Structured Design for Attitudinal Instructions* instruction can be designed to create short-term dissonance such that it facilitates the learners to first recognize the need to change attitude. *Collaborative Problem Solving*, *Situated Learning* and *Caring Community of Learners* leverage cooperative learning for fostering social and ethical knowledge along with conceptual and procedural skills.

Levels of Processing recommends deep processing for facilitating durable learning of the material. *Landamatics* [321] recommends the usage of guided discovery and expository teaching to develop higher-order thinking skills. Bateson proposed four *categories of learning* [328] that result into change of action, underlying assumptions, or the motivating factors depending upon the levels of reflection. Deepening levels of reflections open up newer solution spaces for problem solving. Biggs and Collis [329] proposed five-level taxonomy, *Structure of the Observed Learning Outcome* (SOLO) in terms of increasing structural complexity and abstraction.

Andragogy, initially defined as "the art and science of helping adults learn," has taken on a broader meaning since Knowles' first edition [330]. The term currently defines an alternative to pedagogy, and refers to learner-focused education for people of all ages [314]. It postulated that as learners mature, their motivation as well as perspective shift from external to internal and from postponed application of knowledge to immediacy of application respectively. The

andragogic model asserts that five issues be considered and addressed in formal learning. They include:

4. Learners need to know why they need to learn something.
5. Adults need to learn experientially.
6. Adults approach learning as problem-solving.
7. Adults learn best when the topic is of immediate value.

In the teacher's guide [331], the authors discourage teachers to occupy the role of expert and suggest not to impose solutions and their view. As an alternative, they recommend posing questions, raising contradictions and co-learning as effective approaches to use in classroom. A proper balance among self-directed learning, peer mentoring, group work, and direct instruction, helps learner along many pathways. UNICEF has identified planning, gathering resources, connecting learners to activities, connecting learners to each other, guiding and observing and a focus on equitable participation as key items for increasing the level of active learning in classrooms [332].

List of Author's Publications

(Extracted from the list of references. [Ref No] indicates the S.No in the reference list)

1. [11] Sanjay Goel, What is high about higher education: Examining engineering education through Bloom's taxonomy, The National Teaching & Learning Forum, Vol. 13, pp 1-5, Number 4, 2004.
2. [12] Sanjay Goel and Nalin Sharda, What do engineers want? Examining engineering education through Bloom's taxonomy, Proceedings of 15th Annual AAEE Conference, pp173-185, 2004.
3. [84] Sanjay Goel, Investigations on required core competencies for engineering graduates with reference to Indian IT industry, European Journal of Engineering Education, Taylor & Francis, UK, pp 607-617, October, 2006.
4. [139] Sanjay Goel, Competency Focused Engineering Education with Reference to IT Related Disciplines: Is Indian System Ready for Transformation? Journal of Information Technology Education, Vol. 5, Informing Science Institute, USA, pp 27-52, 2006.
5. [177] Sanjay Goel, Om Vikas, Mukul Sinha, Guidelines for Masters in Archaeo-heritage Informatics, Indo US S&T Workshop on Digital Archeology, Musoorie, India, Invited paper, Nov 11-13, 2005.
6. [348] Sanjay Goel, Do Engineering Faculty Know What's Broken?, The National Teaching & Learning Forum, James Rhem & Associates, USA, Vol. 15, pp 1-10, Number 2, 2006.
7. [358] Sanjay Goel, Activity based flexible credit definition, Tomorrow's Professor, Stanford University, 2003, http://ctl.stanford.edu/Tomprof/postings/513.HTML.
8. [367] Ritu Arora, Sanjay Goel, "Software Engineering Approach for Teaching Development of Scalable Enterprise Applications,", 22nd IEEE-CS Conference on Software Engineering Education and Training CSEET, , pp.105-112, February 2009.
9. [389] Sanjay Goel and Vanshi Kathuria A Novel approach for pair programming, Journal of Information Technology Education, USA, Accepted with revision, Revised copy submitted, 2009.
10. [402] Sanjay Goel, A proposal for a tutorial on enriching the culture of software engineering education through theories of knowledge and learning, Proceedings, 22nd IEEE-CS Conference on Software Engineering Education and Training, CSEET, , pp.279-282, February 2009.
11. [403] Sanjay Goel, Multimedia for Cultural learning, International workshop on Computer Applications in Archaeology, H.B. Bahuguna University, Sri Nagar, India, Invited paper, 2002.
12. [404] Siddharth Batra and Sanjay Goel, Digislim: A learning tool for logic level digital electronics, Computers in Education Journal, Vol XVIIII No 3, American Society of Engineering Education, USA, pp 17-27, July, 2009,
13. [405] Sanjay Goel and Mukul K. Sinha, Virtual Archaeolo-Heritage Exploratorium: A model design for School students, Indo-US S&T Forum Workshop on Digital Arcahaeology: A New Paradigm for Visualizing Past through Computing and Information Technology, India, Invited paper, Nov. 2005.
14. [406] Sanjay Goel, Anshul Jain, Priyank Singh, Saaransh Bagga, and Siddhartha Batra, Computer Vision aided Classification and Reconstruction of Indian Potteries, Indo-US S&T Forum Workshop on Digital Archaeology: A New Paradigm for Visualizing Past through Computing and Information Technology, India, Invited paper, Nov. 2005.
15. [407] Sanjay Goel, A Model Design for Computer based Cognition Support Systems, International Conference on Multimedia in Humanities, IGNCA, 1998.
16. [408] Sanjay Goel, Design of Interactive Systems: Looking Beyond Cognitive domain, INCITE'07, EU-India co-operation in IT research Workshop, New Delhi, Invited talk, 2007.