Methodology in Software Development Capstone Projects

Diane E Strode
d.strode@whitireia.ac.nz

Jill Clark
j.clark@whitireia.ac.nz

Faculty of Business and Information Technology
Whitireia Community Polytechnic
Porirua, NZ

Abstract

Capstone projects which provide the opportunity for student teams to experience ‘real-world’ software development form part of the final semester of study in many computing degrees. This paper describes a number of development methodologies that are currently used both in industry and software development capstone projects. Such projects are carried out under a unique set of constraints due to their nature as instances of experiential learning in an educational setting. These constraints are discussed and then a number of methodologies are described along with a discussion of the suitability of the methodology for capstone projects. Issues that must be addressed by instructors are considered. Finally recommendations are made and a plan for a study into capstone development methodologies is described. The goals of this paper are to provide an overview of current methodologies available for software development capstone projects, to clarify the benefits and problems encountered when using these methodologies in capstone projects, and to indicate suitable resources for those involved in these projects.

Keywords: Capstone projects, systems development methodology, software development method, development process.

1 Introduction

In the mid 1990’s Jayaratna reported that there were over 1000 system development methodologies (1994). Since then many more have been developed, for the object-oriented paradigm, web development, and agile development. The range is extensive and it is difficult for developers who want a well-organised development process to decide among the many possibilities. In addition, many of the undergraduate computing degrees offered in the tertiary sector require that students complete capstone projects during their studies (Clear, Goldweber, Young, Leidig, & Scott, 2001) and students are typically required to specify and use an appropriate development methodology for their capstone project. But which methodologies are really suitable for such small, short-term projects? Is any methodology necessary at all?

This quality assured paper appeared at the 20th *Annual Conference of the National Advisory Committee on Computing Qualifications* (NACCQ 2007), Nelson, New Zealand. Samuel Mann and Noel Bridgeman (Eds). Reproduction for academic, not-for profit purposes permitted provided this text is included. www.naccq.ac.nz

This paper provides an initial framework for selecting an appropriate methodology for a software development capstone project. A number of methodologies are reviewed and discussed with respect to their suitability for small software development teams in the educational capstone environment.

This paper is organised in the following way. First the constraints on capstone projects are considered. Then a number of well known types of methodology are reviewed with particular attention paid to how each one caters for the constrained environment of capstone projects. Then recommendations are made based on this comparison. Following this a design for a study of the methodologies currently in use for software development capstone projects is presented.

Terminology in this discussion paper is that defined by Clear, Goldweber, Young, Leidig, and Scott (2001) who provide a useful discussion regarding many aspects of computing capstone projects. We have made no distinction between methodology, method, and process, with respect to software development (see Crinnion (1992), Graham, Henderson-Sellers, & Younessi (1997a), and Avison and Fitzgerald (2006b) for discussions on interpretation of these terms). Different fields of computing (particularly information systems and computer science) and different continents (particularly Europe and USA) tend to favour one or the other term. In general they all describe development as a series of phases leading from one to another, with distinct activities carried out during phases, leading to progressive elaboration and production of artefacts culminating in a software product which forms part of a system.

2 Constraints on Capstone Projects

Software development capstone projects are common in all of the fields of computing; informatics[1], computer science, software engineering, information systems, information technology, computer engineering, and related fields (Adams, Goold, Lynch, Daniels, Hazzan, & Newman, 2003). They are designed to give the student an opportunity to draw on the full range of skills they have acquired during their degree, both hard skills such as the application of analysis, design, and programming skills to a realistic problem, and soft skills such as teamwork, conflict resolution, and dealing appropriately

[1] The study of the application of information technology (e.g. in business, music, medicine).

with project sponsors. Projects require more commitment and usually have more meaning to students than typical assessments as they go beyond meeting assessment criteria by involving industry sponsors who have some real need for the software under development. So the capstone experience gives the student real-world experience before they move into the workplace. However the capstone project cannot fully replicate workplace experience. There are a number of constraints on projects due to the educational environment in which the project is carried out. These restrictions are primarily those of time and commitment, experience level, scope and complexity, technology, and the need to meet assessment criteria. Each of these issues is now discussed.

Projects are restricted in time; typically the whole project must be completed to some acceptable level within one semester of about 12 to 16 weeks. Deadlines in capstone projects are firm, unlike a project in industry where a few extra weeks may be allowable to ensure completion. In addition the commitment of students is seldom 100% as many will be managing other courses and assessments alongside a larger body of time spent on the project itself.

Experience levels are low in many areas for the students. Knowledge and experience in an unstructured environment, solving a problem that involves a variety of different programming languages and technologies, may be limited or absent. In addition most students are inexperienced in project management, software development, working in teams in a development environment, and in dealing effectively with sponsors and instructors. Experience levels can also be low for the instructors who may not be familiar with all of the various technologies that are needed to carry out the project.

Scope constraints are usually defined in the set up phase of the project by the project coordinator and instructors in consultation with sponsors. This usually occurs before students become involved. Instructors must select projects that are both complex enough to be satisfying for a small team (often 2 to 5 students) and the sponsor, and not so complex that they are overwhelming and can not be completed in whole or part within the available time. The difficulties of assessing the scope and complexity of capstone projects is acknowledged by Mann and Smith (2005). Scope and assessment are related. Scope must be managed carefully as the project progresses and it is not advisable to change scope much during the project. This is because large changes in scope could invalidate the project as an assessment tool, particularly if students reduce the scope too much. Increases in scope must be managed by instructors, in consultation with sponsors and students, as this can also impact on assessment as individual projects must be fairly similar in size across all student groups. When scope is set other considerations come into play such as the technology used.

The technology used may be restricted in some way due to Faculty budgetary constraints or the experience levels of instructors and students. Budgetary restrictions may limit the provision of, and technical support for, hardware and software. The programming languages used are usually restricted to those that the student cohort and the instructor, or some available mentor, has previously experienced. Standard in most software capstones are object-oriented concepts and languages, Internet development, and use of relational databases. Other technologies appear on the horizon constantly such as web services, distributed systems, mobile devices, wiki's, and an infinite number of new technologies and language variations. The ability to undertake projects where instructors are not familiar with the technology is constrained to some extent by the flexibility and willingness of individual instructors to provide support for technologies which may not be covered in the curriculum.

Assessment imposes particular restrictions on capstone projects. The student project generally must satisfy not only the sponsor's requirements, but also a set of assessment criteria, in order for the student team to gain credit for the capstone course. These assessments normally impose additional deadlines and tasks on the project well beyond that required by the sponsor. Items such as specification documents, presentations, work logs, and critical reflection reports, are typical (Chard, Lloyd, Strode, & Wempe, 2004; Clear et al., 2001). These add overhead beyond that encountered in industry practice.

So, the following constraints typically act on software development capstone projects:

1. Time and commitment
2. Experience levels
3. Scope and complexity
4. Technology
5. Assessment criteria

3 Common Methodologies for Capstone Projects

This section of the paper describes and discusses a number of common systems and software development methodologies used in industry and also reported to be in use in software development capstone projects around the world (Adams, Goold, Lynch, Daniels, Hazzan, & Newman, 2003). Their appropriateness for student capstone projects and how the methodology functions given the constraints identified in capstone projects is then discussed.

3.1 The Traditional Systems Development Lifecycle

The traditional systems development lifecycle methodology (SDLC), defined in 1970, was one of the first developed to give structure to the process of systems development for large systems (Royce, 1987). It provides for a full set of specification documents to be produced at different phases and covers the full development lifecycle, although exactly what activities should be carried out at each phase were not specified in the original publication, but were added in elaborations of the methodology in methods such as Information Engineering (Martin & Finkelstein, 1981) and SSADM (Structured Systems Analysis and Design Method), (Eva, 1994). The traditional SDLC provided for throw-away prototyping, limited iteration, and full testing. It specified exactly how much user interaction should be scheduled into development and the exact role of the user. It has various problems in the present development environment. For example when it was first introduced it was common to have existing manual systems that could be studied in an analysis phase in order to reproduce these systems in automated form. Requirements were assumed to be discoverable at the beginning of the lifecycle. The situation in 2007 is that some systems have no precursors, either manual or automated, for example certain web development problems are unique or as yet unimagined. Analysis then becomes a brainstorming activity, or the whole analysis phase becomes distributed across the lifecycle as requirements arise. Typical SDLC artefacts (e.g. Data Flow Models) are unsuited to this type of activity. In addition, typical SDLC methods (e.g. top down functional decomposition) were designed for programming languages that were procedural and evolved before the advent of object-oriented concepts and languages.

The SDLC is reported as effective for use in software development capstone projects (Beasley, 2003; Groth & Hottell, 2006), although Catanio (2006) tailored the method to allow for incremental prototypes. However problems with the SDLC for capstones are apparent in a number of areas (Mann & Smith, 2006). Avison and Fitzgerald (2006a) discuss general problems with the method. Those that are relevant to capstone projects are the dropping of final phase tasks when schedule slippage occurs (typically system completion, system testing, and final documentation). This type of slippage is even more likely in a project where the team is inexperienced. Another drawback of the SDLC is the late delivery of the product in the lifecycle. Students can spend inappropriate lengths of time producing analysis and design documents before beginning development. Only once development is underway can the sponsor begin to see actual progress. It is at this stage that late requirements are likely to appear as the sponsor, especially if this is a technically naïve person, sees the possibilities that the technology can provide. This type of development also does not allow for scheduled revision or improvements to the code base.

3.2 Object-oriented Methodologies

There are many object-oriented methodologies; Graham, Henderson-Sellers, and Younessi (1997b) list many of them (see table 1). One of the most well known, probably due to the availability of commercial tool support by Rational Corporation and later IBM, is the Rational Unified Process (RUP), (Kruchten, 2000). The object-oriented methodologies have a close fit with the object-oriented programming languages C++ and Java that came into common use in the 1990s. These methodologies provide both analysis and design techniques for object based systems development and generally advocate an iterative and incremental development process.

Roggio (2006) reports successful use of the heavyweight RUP in software development capstone projects. He used RUP in conjunction with Rational Rose which is a tool for development and storage of Unified Modeling Language (UML) analysis and design artefacts, and for limited code generation. RUP is an iterative and incremental development process covering the full lifecycle from inception to product release. RUP iterations are based around delivery of prioritised requirements allocated on a highest-risk-first basis. RUP specifies roles, activities, and phases and is tailorable for both large and small projects (Collaris, Dekker, & Warmer, 2006). However this tailoring requires some previous experience in the application of RUP, otherwise the tailor may not know which parts to trim and which to leave. The main problem with RUP is in the tailoring. Instructors would need to tailor the RUP either generically for all projects, or for each individual project before it is used, as this is not a task that students could achieve alone.

UPEDO is a pre-tailored version of RUP developed for use in student projects ("Yoopeedoo", 2004). But independent reports on the successful use of this method in capstone projects are not available.

3.3 Traditional Project Management

Traditional project management is taught in many computing courses. Projects using traditional project management practices are expected to progress though four phases of development; concept, development, implementation, and close-out (Schwalbe, 2006). Schwalbe’s text is based around the PMBOK Guide of the Project Management Institute (2004) which documents best practice for all types of project. Traditional project management is primarily concerned with management of the projects scope, schedule, cost, quality, risk, and people (as resources, and with communication between the people). However Schwalbe differentiates between project and product lifecycles, and suggests that for software development projects the traditional project management lifecycle is modified to accommodate the product lifecycle. Product lifecycles can be any of; the traditional SDLC (or waterfall), spiral, incremental, prototyping, Rapid Application Development (RAD), or agile methods.

Examples of software capstones where the traditional project management process is followed are few and details of the actual development process are not discussed (Goold, 2003).

Traditional project management does not focus on project artefacts, such as documents, apart from specific concept phase documents such as a business case, objectives definition, and detailed schedule plans. It provides no guidance on appropriate techniques for software development. Project management principles, although useful knowledge in any project, are not directly applicable in very small capstone teams where cost is not normally an issue. Its main benefits appear to be in emphasis on communication and feedback from teams on progress. Detailed schedule plans based on work breakdown structures developed by inexperienced teams of capstone project students are unlikely to be realistic or

Table 1: Object-oriented methods published in the 1990s. An expansion of the list published by Graham, Henderson-Sellers and Younessi (1997b).

Year published	Object-oriented method	Author
1988/1991	Shlaer and Mellor	Shlaer and Mellor (1988, 1991)
1990	OOA/OOD Object-oriented Analysis/Object-oriented Design	Coad and Yourdan (1990)
1990	RDD (Responsibility Driven Development)	Wirfs-Brock, Wilkerson and Wiener (1990)
1991	Booch's method	Booch (1991)
1991	SOMA (Semantic Object Modeling Approach)	Graham (1991)
1991	Synthesis	Page-Jones (1991)
1991	OMT (Object Modeling Technique)	Rumbaugh, Blaha, Premerlani, Eddy, & Lorenson (1991)
1992	OSA (Object-oriented Systems Analysis): a model driven approach	Embley, Kurtz, & Woodfield (1992)
1992	Martin/Odell	Martin and Odell (1992)
1992	Objectory/OOSE (Object Oriented Software Engineering): a use-case driven approach	Jacobson, Christerson, Jonsson, & Overgaard (1992)
1992	OBA (Object Behaviour Analysis)	Rubin and Goldberg (1992)
1993	Firesmith's method	Firesmith (1993)
1993	MeNtOr	Object-Oriented Pty Ltd ("Documentation for MeNtOR", 1993)
1994	Fusion	Coleman, Arnold, & Bodoff (1994)
1994	Syntropy	Cook and Daniels (1994)
1994	MOSES	Henderson-Sellers and Edwards (1994)
1994	ROOM (Real-time Object Oriented Modeling)	Selic, Gullekson, and Ward (1994)
1995	BON (Business Object Notation)	Walden and Nerson (1995)
1996	OOram	Reenskaug, Wold, and Lehne (1996)
1997	OPEN (Object-oriented Process, Environment, and Notation)	Graham et al. (1997b)
1999	UP (Unified Process)	Jacobson et al. (1999)
1999	Catalysis (for frameworks)	D'Souza & Wills (1999)

manageable. Traditional project management tends to assume that scope is set at project initiation and deviations are very bad. This idea (closely aligned with that of the traditional SDLC) has been overturned during the 1990s with iterative and incremental development ideas.

This brief discussion points to a lack of specific guidance within traditional project management for software development. Project management must be used in conjunction with some other software development methodology to be effective. Thus courses that cover only traditional project management will require instructors to teach additional skills to give students the ability to progress effectively through a software development project. (Palmer & Felsing, 2002)

3.4 The Team Software Process

The Team Software Process (TSP), developed in the late 1990s, is designed for software engineering and focuses strongly on metrics for product quality and progress in a team development environment (Humphrey, 2000b). Team members are first trained in the Personal Software Process (Humphrey, 2000a), and the team is assumed to be working within a Capability Maturity Model environment. In the TSP each team member has a defined role and makes detailed plans to enable personal progress tracking, project tracking and quality assessment. A team relaunch process which occurs every 2-3 months is followed and at each relaunch plans are updated to reflect the knowledge gained (about the product, progress and process) in the previous relaunch. Detailed templates are provided for teams to follow during the process and TSP coaching is recommended for new teams. An earned value method is used to track progress on the project. To calculate earned value each task is given a value based on its estimated percentage of the total project time estimate. When a task is complete it has earned that value. Thus total earned values for a week give an estimate of the projects percentage completion. TSP specifies weekly task plans, quality plans which estimate defect injection rates, and progress reports.

TSP is reported as the process of choice by Conn (2004) in his Information Systems capstone course, although there is no sponsor on the course, so all projects may be producing the same product. He notes that TSP can be used in any type of project and is not designed for any one type of technology or project type (it can be used for software development projects and other engineering projects). The way that Conn has tailored the TSP is to have two development cycles (iterations) which involve a complete pass at creating the product. Students are required to generate a large number of documents during the project alongside the product itself. He also reports that students seemed to successfully learn that team projects are different from individual efforts.

TSP is a document heavy process and does not include mechanisms for analysis, design, coding or how to develop in an iterative manner. Therefore it may need to be enhanced by the instructor with guidance on suitable content for each iteration (i.e. what parts of my software product should/must be developed first, which second and so on for each iteration) or used in conjunction with some other development methodology which offers this guidance. TSP in its published form assumes projects are of a duration long enough to accommodate a number of relaunches. Each relaunch takes the length of a typical capstone project, so relaunches would need to be dropped or shortened considerably.

3.5 Agile Methods

There are a small number of recognised agile methods (Strode, 2006) which generally includes Extreme Programming (XP) (Beck, 2000), Scrum (Schwaber & Beedle, 2002), Lean Development (Poppendiek & Poppendiek, 2003), Adaptive Software Development (ASD) (Highsmith, 2000), Crystal methods (Cockburn, 2002), Feature Driven Development (FDD) and Dynamic Systems Development Method (DSDM) (Stapleton, 1997). Experience reports on the use of XP in industry are common (Marchesi, Succi, Wells, & Williams, 2003). DSDM experience reports are less available as this is a commercial product controlled by a consortium. Experience reports on Scrum are growing but empirical evidence for the other methods are not commonly reported in research literature. The use of individual XP techniques in classroom settings is now a well established research field in computer science and resources are available (McDowell, Werner, Bullock, & Fernald, 2006; Williams, Smith, & Rappa, 2005).

There is a tendency in non-academic literature to lump all of the agile methods together and discuss ‘agile method’ as though it was a single methodology. This is not the case; each agile method stands alone in its own right, each method is different from the others and each serves a different purpose (Strode, 2006), but all are assumed to conform to the principles and practices of the Agile Alliance (AgileAlliance, 2001) manifesto. Some comply with these practices and principles more than others. Table 2 shows the properties common to XP, DSDM, Scrum, ASD, and Crystal methods. Note that all agile methods are iterative and incremental and most assume that object oriented concepts and languages are familiar to the team (Strode, 2005).

DSDM and XP are the most likely candidates for a capstone project. DSDM is a framework to provide guidance to those using RAD techniques. Because it is a framework this means it can be used with appropriate techniques from other methodologies. So its use in a capstone environment would require that the instructor or student team select techniques to fulfil DSDM goals (e.g. Test first development from XP, sprint backlogs from Scrum). We could locate no reports of its use in capstone projects.

Reports of XP use in capstone projects are available. LeJeune’s (2006) empirical study reported mixed results for the various techniques of XP. Umphress, Hendrix and Cross (2002) compared a number of methodologies (XP, TSP, IEEE107, mil-std-498, ad hoc development) and concluded that some of the XP techniques are “deceptively difficult” (ibid, p.84) although they thought that “XP was a suitable process because it was malleable enough to fit the diversity of our development efforts and the variety of student skills” (ibid.).

Table 2: Common properties of agile methods from Strode (2006)

- Published between 1995 – 2002 in the USA and UK
- Objectivist methods which provide technical solutions
- Address business problems
- Practitioner based
- Project manager and developer perspective
- Incremental development
- Iterative development with 1 month iterations optimal
- Projects undergoing constant change
- Active user involvement
- Feedback and learning
- Teamwork
- Empowered teams
- Communication between all stakeholders is critical
- Small teams of 3-10 programmers is optimal
- Frequent meetings, daily is optimal
- Working software is the main product of development
- Modelling techniques are not mandated
- Minimise documentation

Keefe and Dick (2004) found that it was important to teach the XP techniques and introduce the necessary tools early before the project begins and to continue coaching during the project. Dubinsky and Hazzan (2005) reported increased awareness of customer needs and testing issues when using XP.

We believe that there are obvious benefits and drawbacks of XP in capstone projects. There are a number of practical and useful techniques in XP that make it suitable for capstone projects. XP is designed for small teams, it is designed to heighten communication within the team using pair programming and collective ownership, planning game, and collocated teams. It has explicitly defined techniques for producing quality software using coding standards, test-driven development, refactoring and continuous integration. The method has clearly defined roles and responsibilities for each team member and Beck (2000) states that the method can be adapted to local conditions. However some techniques and practices are drawbacks in capstone projects. Documentation is not considered to be an artefact of development and most of it is imbedded in the code or in a form not suitable for assessment (this could be solved however with an electronic whiteboard or a digital camera to submit temporary documents for assessment). Another solution to this problem is to treat documentation as a requirement of the system as has been done when using XP in ISO compliant organisations (Nawrocki, Jasinski, Walter, & Wojciechowski, 2002). Documentation guidelines for agile methods are provided by Ambler (2002). Customer-on-site is clearly not feasible in most cases; capstone sponsors are unlikely to devote so much time to a capstone project. The concept of a 40 hour week is also not useful in capstone projects. There is also a problem in architecture design with XP (Keefe & Dick, 2004). System metaphor is the recommended technique but it is poorly explained by Beck (2000) and alternative techniques are needed. Coaches are recommended to train the team and keep the process in place during the project. Instructors would need to be experienced and have enough time available to fulfil this role. Alternatively a course covering the XP techniques and philosophies and offering experiences in its use would be required. Finally in order to successfully use XP, faculties need both tools and expertise in test-first development and continuous integration. In general there seems to be a consensus about the use of XP in senior student groups that the whole method is too large and complex to adopt at once (Astrachan, Duvall, & Wallingford, 2003; Johnson & Caristi, 2003; Mugridge, MacDonald, Roop, & Tempero, 2003). Some instructors disagree with the use of agile methods in the classroom at all, usually because of the lack of emphasis on documentation and a perception that agile methods are ad hoc development in disguise (Bunse, Feldmann, & Dorr, 2004; Sanders, 2003; Schneider & Johnston, 2003). Others have mixed reactions (Melnik & Maurer, 2005; Noble, Marshell, Marshall, & Biddle, 2004).

In conclusion adoption of XP in an education setting depends on the goals of the course offered. If your goal is to teach students how to deal with large systems while meeting the criteria of a body of software engineering knowledge then XP is not a good choice as it cannot satisfy many of the criteria (Schneider & Johnston, 2003). However if your goal is to teach students how to work well in small teams, to improve testing capabilities, and deliver functioning software in a short time frame, then the studies of XP in the classroom indicate that XP is a useful choice.

3.6 Ad hoc development

Ad hoc development is the absence of any repeatable development process or methodology. Although ad hoc development is not considered appropriate in an educational course (Clear et al., 2001), it is common in industry settings (Fitzgerald, 2000; Groves, Nickson, Reeve, Reeves, & Utting, 2000). In fact some methodologists consider methodology to be unnecessary in a team of only two (Beck, 2000) while others argue for methodology for individual developers. Examples include the Personal Software Process (Humphrey, 2000a), a process for individuals based on RUP (Kruchten, 2002), and guidelines for best practice as described in Pragmatic Programming (Hunt & Thomas, 2000). Given that methodology has been specified for single developers, and examples are available, as educationalists, we should be taxing our students with following a methodology even when working alone on a project.

4 Recommendations

This discussion has shown that there is no one methodology that is a perfect fit for use in software capstone projects. The key issue is to have some methodology in order to provide structure and guidance to the development. Based on our experiences as capstone coordinator and instructors of more than 60 projects we recommend an iterative development method rather than a waterfall as iteration provides natural checkpoints on product development and elegantly accommodates changes in requirements. Most process models recommend this. It is also necessary to provide formal lessons on the philosophy and techniques of any methodology used, alongside some experience in its use, before the project begins. Another important aspect is to create buy-in with the capstone stakeholders. Instructors need to be convinced of the need for methodology with training or discussion before the project begins. Students buy-in can be encouraged with appropriate lessons in methodology including the reasons why it is important. Sponsors must also have buy-in because they may be asked for more, or less, input into the development process depending on the methodology selected. Finally it is necessary to make methodology use an assessable component of the course. This helps to keep students and instructors aware of their chosen methodology, and hopefully allows for reflection on the benefits and drawbacks of methodology in practice.

5 Further Study

In order to increase knowledge in this area we envisage a research study to identify methodologies, methods, and processes used in software development capstone projects. Information on issues, problems encountered, and successful adaptations of methodologies should be gathered from coordinators, instructors and students. This would be a valuable addition to knowledge of best practice in supervision and execution of software development capstone projects and would contribute to successful outcomes for students, instructors, and sponsors.

6 Conclusion

Methodology and process is an important aspect of software development projects. Table 3 summaries the points made in this article. We have shown that there are numerous methodologies and processes to choose from and there are methodologies available for individuals as well as small teams. Each methodology has benefits and drawbacks when applied in the capstone environment. Instructors must pay attention to this area as there are a number of factors to consider which will impact on the smooth running of capstone projects. This is an area that is important to educational practice and to industry because it is the student experience which influences industry practice over time.

7 Acknowledgements

Thanks to Brenda Lloyd for her assistance in accessing appropriate resources.

Table 3: Summary of key points for each methodology type

Type	**Common examples**	**Models**	**Distinguishing characteristics**	**Issues for capstone projects**
Traditional SDLC	SDLC (Royce, 1987) SSADM (Eva, 1994) IE (Martin & Finkelstein, 1981)	None specified DFD ERD	▪ Best suited to large projects ▪ Numerous documents produced ▪ Functional decomposition used ▪ Model intensive (SSADM and IE) ▪ Limited prototyping	▪ Assumes well defined manual system exists ▪ Assumes procedural programming language ▪ Severe problems occur on schedule slippage ▪ Product delivered late in lifecycle
Object-oriented	RUP (Kruchten, 2000) OPEN (Graham et al., 1997b)	UML UML early variant	▪ Provides extensive support for object-oriented analysis and design ▪ Maps readily to object-oriented programming languages ▪ Iterative and incremental development	▪ Considered to be heavyweight (model intensive) ▪ Can be tailored down – but experience needed
Traditional Project Management	Project Management Institute (2004)	None specified	▪ Advocates a staged process ▪ Must be used with existing development methodologies ▪ Bases development around a business case and firm schedule	▪ Must be matched with a development methodology ▪ Work breakdown structures may be unrealistic ▪ Best suited to large projects ▪ Not easily reconciled with iterative and incremental development styles
Team Software Process	TSP (Humphrey, 2000b)	None specified	▪ Strong metrics for software quality and progress tracking ▪ Highly proscribed ▪ Numerous planning/recording produced ▪ Templates available	▪ Considered to be heavyweight ▪ Numerous documents produced ▪ No guidance on analysis, design, coding or iterative development
Agile Methods	XP (Beck, 2000) Scrum (Schwaber & Beedle, 2002)	UML knowledge assumed UML knowledge assumed	▪ Many proscribed techniques (XP) ▪ Early production of code ▪ Iterative and incremental development ▪ Best suited to small projects ▪ No formal documents ▪ Strong emphasis on quality and testing ▪ Sponsor involvement important ▪ Copes with late changes to requirements	▪ Assumes high sponsor involvement ▪ Assumes knowledge of UML and object-oriented programming ▪ Documentation deemphasised ▪ Some techniques require training and practice before use (e.g. pair programming/test-first development/refactoring) ▪ Provides good team support mechanisms

8 References

Adams, L., Goold, A., Lynch, K., Daniels, M., Hazzan, O., and Newman, I. (2003): Challenges in teaching capstone courses. *ITiCSE'03*. Thessaloniki, Greece: ACM Press.

AgileAlliance. (2001): Manifesto for agile software development. Accessed February 17, 2003. http://www.agilemanifesto.org

Ambler, S. (2002): *Agile modeling: Effective practices for Extreme Programming and the Unified Process*. New York, John Wiley & Sons, Inc.

Astrachan, O. L., Duvall, R. C., and Wallingford, E. (2003): Bringing extreme programming to the classroom. In *Extreme Programming Perspective.s* 237-250. MARCHESI, M., SUCCI, G., WELLS, D., and WILLIAMS, L. (eds). Boston, Addison-Wesley.

Avison, D., and Fitzgerald, G. (2006a): Chapter 3 The life cycle approach. In *Information systems development methodologies, tools and techniques*,4th edn. 31-49. London, McGraw-Hill Education.

Avison, D., and Fitzgerald, G. (2006b). Chapter 27 Issues. In *Information systems development methodologies, tools and techniques* 4th edn. 567-590. London, McGraw-Hill Education.

Beasley, R. E. (2003). Conducting a successful senior capstone course in computing. *Journal of Computing Sciences in Colleges*, **19**(1): 122-131.

Beck, K. (2000): *Extreme programming explained: Embrace change*. Boston, Addison-Wesley.

Booch, G. (1991): *Object oriented design with applications*. Menlo Park, CA, Benjamin/Cummings.

Bunse, C., Feldmann, R. L., and Dorr, J. (2004): Agile methods in software engineering education. In *5th International Conference on Extreme Programming and Agile Processes in Software Engineering, XP2004* (Vol. 3092 Lecture Notes in Computer Science) 284-293. ECKSTEIN, J. and BAUMEISTER, H. (eds). Berlin, Springer-Verlag.

Catanio, J. T. (2006): An interdisciplinary practical approach to teaching the software development life-cycle. *Proc. 7th Conference on Information Technology Education*, Minneapolis, Minnesota, USA, ACM Press.

Chard, S., Lloyd, B., Strode, D. E. and Wempe, N. (2004): Student industry projects: Streamlining the process for a win-win. *Proc. Eighth Annual New Zealand Association for Co-operative Education NZACE*. Christchurch, New Zealand. Accessed March 15, 2006. http://www.nzace.ac.nz/past_conf_proc.htm

Clear, T., Goldweber, M., Young, F. H., Leidig, P. M. and Scott, K. (2001): Resources for instructors of capstone courses in computing. *ACM SIGSE Bulletin*, **33**(4): 93-113.

Coad, P. and Yourdan, E. (1990): *Object-oriented analysis*. Englewood Cliffs, N.J, Prentice-Hall.

Cockburn, A. (2002): *Agile software development*. Boston, Addison-Wesley.

Coleman, D., Arnold, P. and Bodoff, S. (1994): *Object-oriented development: The Fusion method*. Englewood Cliffs, NJ, Prentice-Hall.

Collaris, R., Dekker, E. and Warmer, J. (2006, 17 March 2006). Tailoring RUP made easy: Introducing the responsibility matrix and artifact flow. *The Rational Edge, Sept 2006.* Accessed March 15, 2006. http://download.boulder.ibm.com/ibmdl/pub/software/dw/rationaledge/sep06/TheRationalEdge_September2006.pdf

Conn. (2004): A reusable, academic-strength, metrics-based software engineering process for capstone courses and projects. *Proc*. *35th SIGCSE Technical Symposium on Computer Science Education*. Norfolk, Virginia, ACM Press.

Cook, S. and Daniels, J. (1994): *Designing object systems*. UK, Prentice-Hall.

Crinnion, J. (1992). The evolutionary development of business systems. *IEE Colloquium on Software Prototyping and Evolutionary Development.* Accessed June 1, 2005. IEEE Xplore database.

D'Souza, D. F. and Wills, A. C. (1999): *Objects, components and frameworks with UML the Catalysis approach*. Reading, M.A, Addison Wesley.

Documentation for MeNtOR. (1993): Sydney, NSW, Object-oriented Pty Ltd.

Dubinsky, Y. and Hazzan, O. (2005): The role of a project-based capstone course. *Proc. 27th International Conference on Software Engineering*. St. Louis, MO, USA, ACM Press.

Embley, D. W., Kurtz, B. D. and Woodfield, S. N. (1992): Object-oriented systems analysis: A model-driven approach. Englewood Cliffs, NJ, Yourdan Press/Prentice-Hall.

Eva, M. (1994): *SSADM Version 4: A user's guide*. 2nd edn. London, McGraw-Hill Book Company.

Firesmith, D. G. (1993): *Object-oriented requirements analysis and logical design: A software engineering apporach*. New York, Wiley.

Fitzgerald, B. (2000): Systems development methodologies: The problem of tenses. *Information Technology and People* **13**(3): 174-185.

Goold, A. (2003): Providing process for projects in capstone courses. *Proc. 8th Annual Conference on Innovation and Technology in Computer Science,* Thessaloniki, Greece, 26-29, ACM Press.

Graham, I. (1991): *Object oriented methods*. Harlow, UK, Addison-Wesley.

Graham, I., Henderson-Sellers, B. and Younessi, H. (1997a): Chapter 2: Process as the keystone. In *The OPEN process specification.* Harlow, England: Addison-Wesley.

Graham, I., Henderson-Sellers, B. and Younessi, H. (1997b): *The OPEN process specification*. Harlow, England, Addison-Wesley.

Groth, D. P. and Hottell, M. P. (2006): Designing and developing an informatics capstone project course. *Proc. 19th Conference on Software Engineering Education and Training CSEET'06*, 61 - 68. Accessed March 3, 2006. IEEE Xplore database.

Groves, L., Nickson, R., Reeve, G., Reeves, S. and Utting, M. (2000): A survey of software development practices in the New Zealand software industry. *Proc. Australian Software Engineering Conference*. Accessed June 1, 2005. IEEE Xplore database.

Henderson-Sellers, B. and Edwards, J. M. (1994): *BOOKTWO of Object-oriented knowledge: The working object*. Sydney, Prentice-Hall.

Highsmith, J. A. (2000): *Adaptive software development: A collaborative approach to managing complex systems*. New York, NY, Dorset House Publishing.

Humphrey, W. S. (2000a): Guest editors introduction: the Personal Software Process - status and trends. *IEEE Software*, **17**(6): 71-75.

Humphrey, W. S. (2000b): The Team Software Process (TSP), CMU/SEI-20000-TR-023. Accessed March 13, 2006. www.sei.cmu.edu/pub/documents/00.reports/pdf/00tr023.pdf

Hunt, A. and Thomas, D. (2000): *The pragmatic programmer*. Boston, Addison Wesley.

Jacobson, I., Booch, G. and Rumbaugh, J. (1999): *The unified software development process*. Reading, Massachusetts, Addison-Wesley.

Jacobson, I., Christerson, M., Jonsson, P. and Overgaard, G. (1992). *Object-oriented software engineering: A use case driven approach*. Harlow, UK, Addison-Wesley.

Jayaratna, N. (1994). *Understanding and evaluating methodologies NIMSAD: A systematic framework*. London, McGraw-Hill.

Johnson, D. H. and Caristi, J. (2003). Extreme programming and the software design course. In *Extreme programming perspectives*. 273-285. MARCHESI, M., SUCCI, G., WELLS, D., and WILLIAMS, L. (eds). Boston: Addison-Wesley.

Keefe, K. and Dick, M. (2004). Using Extreme Programming in a capstone project, *Proc. 6th Conference on Australasian Computing Education - Volume 30*. Dunedin, New Zealand, Australian Computing Society.

Kruchten, P. (2000). *The Rational Unified Process: An introduction* 2nd edn. Boston, Addison-Wesley Longman.

Kruchten, P. (2002, 15 March 2006). The software development process for a team of one. *The Rational Edge (2).* Accessed June 2, 2006. http://download.boulder.ibm.com/ibmdl/pub/software/dw/rationaledge/feb02/TheRationalEdge200202Issue.pdf

LeJeune, N. F. (2006). Teaching software engineering practices with Extreme Programming. *Journal of Computing Sciences in Colleges,* **21**(3): 107-117.

Marchesi, M., Succi, G., Wells, D. and Williams, L. (eds.). (2003): *Extreme programming perspectives*. Boston, Addison-Wesley.

Martin, J. and Finkelstein, C. (1981): *Information Engineering. Vol 1 and 2*. Englewood Cliffs, New Jersey, Prentice Hall.

Martin, J. and Odell, J. J. (1992). *Object-oriented analysis and design*. Englewood Cliffs, NJ, Prentice-Hall.

McDowell, C., Werner, L., Bullock, H. E. and Fernald, J. (2006). Pair programming improves student retention, confidence, and program quality. *Communications of the ACM,* **49**(8): 90-95.

Melnik, G. and Maurer, F. (2005): A cross-program investigation of student's perceptions of agile methods. *Proc. 27th International Conference on Software Engineering* New York, 481-488, ACM Press.

Mugridge, R., MacDonald, B., Roop, P. and Tempero, E. (2003): Five challenges in teaching XP. In *4th International Conference on Extreme Programming and Agile Processes, XP 2003* (Vol. 2675 Lecture Notes in Computer Science) 406-409. MARCHESI, M., SUCCI, G., WELLS, D., and WILLIAMS, L. (eds). Berlin, Springer-Verlag.

Nawrocki, J., Jasinski, M., Walter, B. and Wojciechowski, A. (2002): Combining extreme programming with ISO 9000. In *Proceedings of the First EurAsian Conference on Information and Communication Technology, EurAsia-ICT 2002* (Vol. 2510 Lecture Notes in Computer Science, pp. 786-794). SHAFAZAND, M.H. and TJOA, A.M. (eds). Berlin, Springer-Verlag.

Noble, J., Marshell, S., Marshall, S. and Biddle, R. (2004): Less Extreme Programming. *Proc. Sixth Conference on Australasian Computing Education - ACE'04*, Darlinghurst, Australia, **30**: 217-226, Australian Computer Society.

Page-Jones. (1991): Relationship between the structure and object-oriented worlds. *TOOLS '91 tutorial notes, Paris, March 4-8, 1991*.

Palmer, S. R. and Felsing, J. M. (2002): *A practical guide to Feature-Driven Development*. Upper Saddle River, Prentice Hall.

PMI. (2004): A guide to the project management body of knowledge. In P. M. Institute (ed) 3rd edn.Vol 2006. USA, Project Management Institute.

Poppendiek, M. and Poppendiek, T. (2003): *Lean software development an agile toolkit*. Boston, Addison-Wesley.

Reenskaug, T., Wold, P. and Lehne, O. A. (1996): *Working with objects. The OOram software engineering manual.* Greenwich, CT:,Manning.

Roggio, R. F. (2006): A model for the software engineering capstone sequence using the Rational Unified Process,.*Proc. The 44th Annual ACM Southeast Regional Conference*. Melbourne, Florida, ACM Press.

Royce, W. W. (1987): Managing the development of large software systems, *Proc. The 9th international conference on Software Engineering*. Los Alamitos, CA, IEEE Computer Society Press (Reprinted from Proceedings, IEEE WESCON, August 1970 p. 1-9. Originally published by TRW).

Rubin, K. S. and Goldberg, A. (1992): Object behaviour analysis. *Communications of the ACM,* **35**(9): 48-62.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorenson, W. (1991): *Object-oriented modeling and design*. Englewood Cliffs, NJ, Prentice-Hall.

Sanders, D. (2003): Student perceptions of the suitability of extreme and pair programming. In *Extreme programming perspectives*. 261-272. MARCHESI, M., SUCCI, G., WELLS, D., and WILLIAMS, L. (eds).Boston, Addison-Wesley.

Schneider, J. and Johnston, L. (2003): Extreme programming in universities: An educational perspective. *Proc. The 25th International Conference on Software Engineering*, Washington, DC, USA, 594-599, IEEE Computer Society.

Schwaber, K. and Beedle, M. (2002): *Agile software development with Scrum*. Upper Saddle River, New Jersey, Prentice Hall.

Schwalbe, K. (2006): *Information technology project management* (4 ed.). Australia, Thomson Course Technology.

Selic, B., Gullekson, G. and Ward, P. T. (1994): *Real-time object-oriented modelling*. New York, Wiley.

Shlaer, S. and Mellor, S. (1988): *Object-oriented systems analysis: Modeling the world in data.* Englewood Cliffs, N. J, Yourdan Press Computing Series.

Shlaer, S. and Mellor, S. (1991): *Object lifecycles. Modeling the world in states*. Englewood Cliffs, N. J, Yourdan Press/Prentice Hall.

Stapleton, J. (1997): *DSDM Dynamic Systems Development Method*. Harlow, England, Addison-Wesley.

Strode, D. E. (2005): The agile methods: an analytical comparison five agile methods and an investigation of their target environment. Unpublished Master of Information Science thesis. Massey University, Palmerston North, New Zealand.

Strode, D. E. (2006): Agile methods: a comparative analysis. *Proc. 19th Annual Conference of the National Advisory Committee on Computing Qualifications (NACCQ 2006), Wellington, New Zealand.* Wellington, New Zealand.

Umphress, D. A. Hendrix, T. D. and Cross, J. H. (2002): Software process in the classroom: the capstone project experience. *IEEE Software,* **19**(5): 78-81.

Walden, K. and Nerson, J. M. (1995): *Seamless object-oriented architecture*. Englewood Cliffs, N. J., Prentice-Hall.

Williams, L., Smith, S. E. and Rappa, M. (2005): Resources for Agile Software Development in the Software Engineering Course. *Proc. 18th Conference on Software Engineering Education and Training, CSEEandT 2005.* 236-238.

Wirfs-Brock, R. J., Wilkerson, B. and Wiener, L. (1990): *Designing object-oriented software*. Englewood Cliffs, NJ, Prentice-Hall.

Yoopeedoo. (2004): Accessed March 15, 2006. http://www.yoopeedoo.org/index.asp