Software Engineering Curriculum Design

Thesis submitted to the University of Surrey

for

the degree of Doctor of Philosophy

Martin James Loomes BSc., PGCE., FIMA

April, 1991

Department of Electrical and Electronic Engineering,

University of Surrey

Abstract

Software engineering design is a vital component of modern industry, unfortunately, the processes involved are still poorly understood. This poses a major problem for teachers of the subject, who are under constant pressure to improve the quality of education, but are unsure how to bring this about, or even how to detect such improvement. This thesis attempts to start the process of clarifying what we mean by "software engineering design", and apply the insights gained to the activity of curriculum design.

First, we establish a method for the research, by constructing a framework to constrain and guide the process of seeking new insights. This leads to a decidedly eclectic approach to the problem, as software engineering design is viewed, and reviewed, from a number of different perspectives. Next, these views are synthesised into a model of the software engineering design process, and new insights are sought to refine the model. The central theme of this model is the idea that the design process can be considered as a one of theory building. Finally, we bring this model into direct contact with the task of curriculum design, both in a general sense, and also by providing illustrations of some of the consequences of its use.

Acknowledgements

I would like to thank my supervisor, Professor Bernie Cohen, for his belief in the value of this research programme, and his faith in my ability to carry it out. Without his vision and support I would never have had the courage to embark on such an enterprise.

Thanks is also due to my employer, Hatfield Polytechnic, for financial support, and for making resources available to support this research.

My colleagues, staff and students, at Hatfield Polytechnic, have played an essential part in this research programme. It is impossible to acknowledge all the corridor encounters that have led to valuable insights, but Richard Mitchell and Bob Dickerson deserve special recognition for their rôle in orienting the research during its early stages. I would also like to thank Juliet Brown, for helping me to remember that teaching is fun.

To my wife, Maureen, and children, Julia and Monica, whose love and support has been utilised in the full during this research, I can only say "thank you", and may the phrase "quiet children, daddy's working" resound less frequently in future years.

Finally, I would like to thank you, the reader, for being prepared to embark on this voyage of exploration: you are part of this book, as I hope you will realise by the time you have read it.

Contents

Chapter 1

The Gift of Wonder

"In philosophy methods are unimportant; any method is legitimate if it leads to results capable of being rationally discussed. What matters is not methods or techniques but a sensitivity to problems, and a consuming passion for them: or, as the Greeks said, the gift of wonder"

Karl Popper

The research documented in this thesis has a number of unusual characteristics. These have arisen because the author, rather than seeking out a suitable, if arbitrary, problem for a doctoral research programme, has set out to tackle a problem which he was actually experiencing at a personal level. This problem was how to improve his teaching of software engineering design, and how to help others to improve their teaching of the subject too. Magee sums up the ensuing situation very neatly, when he writes

> "A consequence of always proceeding from problems which really are problems—problems which one actually *has*, and has grappled with—is, for oneself, that one will be committed to one's work; and for the work itself, that it will have what Existentialists call "authenticity". It will not only be an intellectual interest but an emotional involvement, the meeting of a felt human need. Another consequence will be an unconcern for conventional distinctions between subjects; all that matters is that one should have an interesting problem and be trying to solve it." [Mag73, page 68]

The most significant consequence of deciding to tackle such a problem has been the decision to embrace eclecticism, reacting against the current trend in western culture of partitioning academic disciplines into smaller and smaller units. This decision was not made lightly, for it has far reaching consequences for the research programme, not least of which has been to deprive the author of the benefit of precedent in selecting both the research method and also the style of thesis presentation. Another implication of the choice of problem is that it has led to the researcher "being forced into philosophy by the pressure of non-philosophical problems" [Pop63, page 73].

The title of the thesis also warrants some clarification, for the reader may be under the misapprehension that the dénouement of this work will be the presentation of a curriculum, neatly packaged and ready to teach. This, as we will argue in Chapter Seven, is not a sensible view of curriculum, and would not solve the problem we are setting out to tackle. The end product of this research is a resource that curriculum designers and teachers can use in carrying out their duties. This resource comprises the documented exploration of the nature of software engineering design, from a teacher's perspective, together with the production of a model around which future discussions, or the teaching process itself, can take place. It is the task of presenting this exploration and model, and linking them into existing bodies of knowledge, that constitutes the doctoral aspects of the research programme. No apology is made for the fact that very few answers are presented during the course of this discussion. We would argue that teachers must arrive at their own answers if their lessons are to carry authority and commitment. The rôle of the discussions presented here is to analyse questions, distilling them into more precise forms that can be fruitfully discussed. This, as Dewey has said, is the true rôle of educational philosophy [Dew69].

It might be argued that an analysis of software engineering design should not be carried out by those currently working within the discipline of Computing, but by philosophers, psychologists and sociologists. It has indeed been argued, for example, that scientists should not attempt to discuss the processes of science, but should leave such discussions to philosophers of science [Fin86]. Feuer presents a discussion of the cases for and against this point of view [Feu69]. Suffice it to say, however, that at present there is no other discipline within which suitable discussion of the processes of software engineering is taking place. Moreover, due to the intimate connections between the methods of the discipline and the tools, techniques and languages being developed within it, methodology is currently seen as a mainstream activity within software engineering: our methods are part of our technology. We would also argue that if Software Engineering aspires to becoming a genuine engineering profession, it cannot avoid the responsibility for self-analysis, as this is the route to improvement in quality.

One result of the eclectic approach has been the emergence of striking similarities between the processes of software design and curriculum design. In retrospect, this is not surprising as both activities share a number of common characteristics:

1. They are both concerned with the design of complex artifacts which are required to be adaptive in an environment of complex values.

2. The processes involved are usually carried out by teams of professionals without particularly autocratic management structures.

3. There is at present little scientific support for many of the decisions that need to be made.

4. Entities which can be classified as "information" and "knowledge" are manipulated and transmitted within the final products, so both Education and Software Engineering use complex linguistic systems.

5. External pressures exist for change (such as the call for more effective methods

of design or of teaching reading), but little time is made available for internal reflection, theory formation and experiment.

It is interesting to note that the production of software systems has been dubbed "science", "engineering", "architecture" and "craft", but never education. Similarly, education is rarely viewed as an engineering discipline, in spite of the considerable amount of design that takes place.

This insight has led to a number of dual applications of ideas, where a topic originally researched for its relevance to the software design process has subsequently been applied to curriculum design, and vice versa. The emergence of such meta-level discussions was predicted by Popper, whose philosophy suggests that a general investigation of scientific matters, without imposition of the particular research methods associated with individual disciplines, is a possible approach to the whole area of epistemology. The issues being investigated here are merely vehicles which serve as unusually well-structured, small-scale, models of more general problems involving knowledge. As the questions we are asking are at an unusually high level of abstraction, we should not be surprised if the answers display an unusually high degree of applicability. These ideas are not developed explicitly in this thesis, as they add a meta-level to the discussion which is unnecessary for main the thrust of the arguments. We will be unable to ignore some of the implications of this relationship, however, when we come to apply our results to curriculum design later in the thesis.

In the rest of this chapter we will set the scene for the discussions to follow. We will start by discussing the background to the identified problem, highlighting some of the features that add to the complexity of the task. We will then make the aims of the research more explicit. The research method adopted to meet these aims will then be presented and discussed. Finally, a brief overview of the structure of the rest of the document will be given.

1.1 Background to the Problem

It has been suggested that the software industry worldwide is being hit by a "software crisis". The manifestations of this crisis include the number of bugs found in systems after delivery, the missing of deadlines, the delivery of software that simply does not work, and the problems of software maintenance. A hypothesis central to this thesis is that it is counter-productive to consider such a state of affairs as a "crisis". A crisis is supposed to be a decisive moment, a turning point, not an ongoing state of affairs. Moreover, a crisis will usually be resolved by a rapid action, or sequence of actions. The adoption of the term "crisis" by the software industry has lead to a period of belief that all we need to do to resolve the crisis is to find the appropriate actions. Many such actions have been suggested in the past, but the plethora is not always helpful. As Riddle has observed:

> "So many people seem so certain about how better to achieve the full potential of Software Engineering, that I feel more crippled than the blind man who couldn't identify the elephant." [Rid85, page 1]

Amongst the panaceas that have been identified in recent years, are

- Automatic programming, whereby a machine takes over the task of generating programs.
- Prescriptive design methods, whereby the designer can be made to function as a machine, following sets of rules which will lead to working systems.
- Formal methods, which would make the design of software more "scientific" and hence less fallible.
- Tools, or factories, which would reduce the complexities of the task by orders of magnitude.

One of the major assumptions of this thesis is that no such panacea can exist. Software Engineering is an essentially difficult task. The "crisis" is a reflection of the fact that the demands made upon the discipline have consistently exceeded the discipline's ability to cope. This observation is not particularly helpful, however, unless some suggestions are made for improving the situation. The central tenet of this thesis is that proper education of software engineers would be of major benefit, and that current educational practice in the area falls short of that which is possible and desirable.

The education of software engineers in the United Kingdom is currently hindered by a number of factors, in addition to those hindering education as a whole.

Pace of Development

Computing is a relatively new academic discipline. Computers, however, have found their way into every possible walk of life. It has proved impossible for the academic discipline to keep abreast of technological change, carry out the task of consolidation, and spend time reflecting on the nature of the subject itself. Most educators in the subject would readily admit that they have to run just to keep up with changing technology and external demands. Moreover, many educators have not been formally educated in the discipline themselves, and so they have no fall-back position of preserving the traditional approach. As Popper has noted,

> "...the long term 'proper' functioning of institutions depends mainly on such traditions. It is tradition which gives the persons (who come and go) that background and that certainty of purpose which resists corruption." [Pop63, page 134]

In the absence of these traditions, Computer Science has also failed to adopt the established norms of science. Dijkstra identifies two causes of this phenomenon. The pioneers of the subject generally came from scientific backgrounds, but had never been trained to carry scientific thinking across subject boundaries, and "many of them must have felt that scientific thought was a luxury that one could afford in the more established disciplines, but not in the intellectual wilderness they now found themselves in." The second cause is the large number of people who have entered the discipline from non-scientific backgrounds. "By their sheer numbers they form by themselves already an explanation for the phenomenon." [Dij82, page 61]

If education is to play a major rôle in overcoming the problems facing the software industry then it must be given time to reflect and determine a strategy. This is not a requirement for software engineering education specifically, but for education as a whole. Brameld noted, in 1969, that

> "education has suffered because it has not maintained adequate perspective. It has not viewed itself from a distance, as it were, so that when a crisis occurs, ..., it is unprepared to do much more than go on the defensive with loud and unintelligible noises." [Bra69, page 218]

Since this observation was made the situation appears to have deteriorated. Even less time and resources are now available for "non-productive" activities such as discussing the curriculum. This criticism has been levelled against institutional education on many occasions. Cyert, for example, says

> "Perhaps the most difficult organization to change in society is the university. Scratch a Professor from any discipline and you will receive a lecture on how business organizations, churches, governments etc., should reform. Yet universities ignore the problems of education in their own institutions." [Cye80, page 7]

Discussion of software engineering education certainly does take place, but this concentrates on features that can be considered fairly superficial. Considerable attention is paid to the transient details, such as which language should be used as a vehicle for teaching programming, or which microprocessor is suitable for teaching students about computer architecture. The literature is primarily concerned with matters of content (that is *what* should we teach), but without any real appeal to the deeper issues such as the value systems that should underpin the selection of content (that is *why* we should teach it). There is also very little informed debate as to *how* we should teach the subject. There are, of course, exceptions such as Cohen's paper on curriculum inversion [Coh86], and the publication of the debate on teaching Computing sparked off by Dijkstra [Dij89], both of which do raise fundamental issues.

Demands of Industry

Industry has two types of expectation when recruiting graduates into Software Engineering. First, it expects the recruits to be immediately useful, knowledgeable in current technologies, and to be able to work as part of a team. Second, it expects recruits to have a reasonable "useful life expectancy", being able to play increasingly important rôles in design teams for the duration of their careers. These expectations are certainly not mutually exclusive, and are no different from the expectations of other engineering disciplines. A difference arises, however, in the balance of these requirements. A study of the appointments advertisements for software engineers reveals that employers are requiring knowledge of several current, but usually short-lived, methods, languages, platforms and tools. Similar advertisements for civil engineers, for example, are rarely so specific.

In attempting to make their students generally employable, educators are faced with the task of covering several such technological ephemera. Unless a way can be found to turn experience with these ephemera into transferable skills, then this requirement militates against the long term effectiveness of the recruit. Very often, educators fall into the trap of teaching a particular technique simply because industry currently uses it, ignoring the fact that industry is also saying that the technique is not very good. Often the approaches adopted are driven primarily by this aim, and are not always very constructive. To quote Mills,

> "There is a real danger in over using soft topics and survey courses loaded with buzz words to provide near-term job salability. But without adequate technical foundations people will become dead-ended in mid-career, just when they are expected to solve harder problems as individuals, as members or as managers, of teams." [Mil80a, page 1161]

The problem may be ascribed in part to a tactical error on the part of educators. When institutions attempt to adapt to changes in demand as quickly as possible, they lose the stability usually associated with the educational system. As Kozmetsky has observed, however, this may not be the best tactic. What is actually required is that the *student* can adapt, not the institution. If we can find ways of educating students so that they can adapt quickly in a world of constant change then the institutions can adapt more slowly, thus maintaining their stability [Koz80, page 152].

Political

Education has always been subjected to various political forces, both external and internal. In general, there has been a large amount of inertia within the system to provide stability in the face of these pressures. Computing, however, has been subjected to great pressures but has never been given the time to stabilise. These pressures are often not well thought out: they manifest themselves in slogans and buzz words which Computing departments are expected to utilise: Information Technology, IKBS, AI, databases, Software Engineering, Information Systems, design methodologies, formal methods, real time systems, safety critical systems, ... But exactly what do these terms mean? What is a "non-expert system"? What system is not a "database"? Is a "formal method" a method? Is an editor "safety critical", given that it may be used to write the code for nuclear reactor control systems? There is no time for such debates. Personal experience shows that raising such questions is likely to lead to rebukes such as "This is all very interesting, but it is not getting us anywhere", the implication being that adopting the terminology is in some sense progressive, but to what end? The professional educator has an obligation to "submit slogans and rules of thumb to critical analysis" [Tay69, page 28].

This problem is compounded by the fact that research funding is often available for investigating these "subjects", so there is little incentive amongst academics for showing that the subjects may not be very appropriate ways of partitioning the discipline.

Another problem that has arisen for education generally is the imposition of a producer-consumer model. Educational establishments are increasingly being seen as the providers of a service which must meet the needs of consumers. Unfortunately, the manner of funding obscures the issue of who the consumer actually is. For schools, the parents are often referred to as consumers, but for higher education that rôle is variously adopted by the government, the funding agencies, the students, their parents, society at large, potential employers, or professional bodies. For a stable discipline, the effects of this attitude are reduced by inertia. Everyone "knows" what Physics is all about, so consumer pressure may produce changes over time, but not dramatic ones. For Computing, however, the pressures can be catastrophic, leading to gross inconsistencies. Rather than the discipline converging to a stable state as time goes on, it could diverge and become inherently unstable. Software engineers know the dangers of applying patches to systems they do not fully understand in order to meet changes in requirements; unfortunately they persist in patching the curriculum in just this fashion. Blind acceptance of a producer-consumer model, coupled with the cultural acceptance of the slogan "the customer is always right", could destroy the discipline. The educator must accept responsibility as a professional, and be prepared to take informed decisions based on wider issues than this. Failure to do so will make the teacher little more than an educational technician, and remove all claim to professional status.

1.2 The Research Aims

The primary aim of this research is to improve the quality of software engineering education and hence to improve the quality of software systems being produced in industry. An assumption that is made in meeting this aim is that a deeper understanding of the software design process will place the educator in a better position to bring this about, by facilitating curriculum design in the widest sense. Thus our primary aim can be met by helping the educator to reach such a deeper understanding.

We would argue that this cannot be achieved simply by presenting a neatly packaged description of the system design process, supported by elegant arguments, for this will not necessarily increase understanding. Neatly packaged ideas tend to be accepted on authority, or ignored in their totallity, they do not provoke the reader sufficiently to promote deeper understanding, for they do not involve the reader sufficiently in the process. Our aim, therefore, must be to provoke the readers into challenging their existing constructs by raising questions and viewing issues from unusual angles. We will offer suggestions for ways the system design process can be construed, but this will be done in the full knowledge and expectation that they will be rejected.

To assist the reader, however, we must provide the necessary framework for exploration to take place, so that challenges are not presented as a number of *ad hoc* questions, but through a structured analysis of the problem. This gives rise to three subsidiary aims of the research:

1. To develop a framework for the discussion of the software engineering design process.

2. To use this framework for the development of a model of the design process that can be used to discuss curriculum issues.

3. To illustrate how the increased understanding reached through these discussions can be translated into action through curriculum design.

It is important to realise that we are not proposing the framework, or the resulting model, as "right" or "true". We are adopting Bacon's maxim that "Truth emerges more readily from errors than from chaos", so we are prepared to make mistakes in imposing structure, for at least then they can be recognised as such. Indeed, we anticipate that it is primarily through disagreeing with the ideas put forward in this thesis, including objections to the choice of framework, that the aim of increasing the reader's understanding will be met. We should also stress that there is no intention of arriving at a set of axioms from which to deduce a curriculum. As Hirst has observed, discussion can only serve to inform and guide the curriculum design process, not to define it [Hir69, pages 178-183].

Typical of the sort of issues that our analysis will cause us to consider is what we really mean by a number of terms and slogans. What exactly do we mean when we say that "functional specifications should state what a system is to do, not how it is to do it"? Is there any substance to the debate that takes place regarding the teaching of "theoretical" versus "practical" skills? We will also consider the rôle of issues such as "methods" and "formalisation" in the system design process, and the implications of our discussions for how such topics should be presented in the curriculum.

Although it is not an aim of this research, the discussion carried out may also prove useful to those charged with the task of developing design methods and CASE tools, for research in these areas has identified a set of questions that need to be addressed which are similar to those arising from our problem [Tul87].

1.3 Research Method

The problem being addressed is highly complex, and it was clear from the outset that no existing research programme would be found upon which to build. In particular, no established research method would suffice to tackle the problem, for the issues are largely technological, and, as Rapp has said,

> "...technology has by no means yet received the attention in philosophical literature which is commensurate with its actual significance. ...As a consequence, there exists no generally accepted theoretical frame of reference or inventory of methodological tools to which one can resort in any particular investigation." [Rap81, pages 19-20].

The fact that modern technological problems are giving rise to philosophical issues that require a new approach for their exploration has also been noted by Lenk and Ropohl, who write

> "Philosophy has to accept the challenge of interdisciplinary efforts ...It has to step out of the ivory tower of restricted and strictly academic philosophy." [LR79, page 47].

This research is in no way "scientific", in the traditional sense of the term. There is no attempt to adopt a process of conjecture followed by empirical refutation, for example. Such an approach necessarily invokes simplifications of the problem domain, which are never arbitrary, but arise naturally out of the problems being addressed. It is the analysis necessary to identify these sorts of simplifications that is being carried out in this research programme.

There have been attempts to study very simple problems of software engineering education scientifically, notably the ways in which people learn to program effectively [SI86], and some tentative results from these studies have been incorporated into this research. In general, however, the number of assumptions that are made in such studies (such as equating "programming" with "procedural programming in the small", and "effective programming" with "running programs") limits the applicability of these results. Similarly, there have been some investigations carried out into how students learn. Most research into these areas has been limited to young children, however, and little seems to have been done in relation to students in higher education, or adults. This is changing, and cognitive science is starting to turn its attention to areas more pertinent to this research programme. If cognitive science develops into a body of knowledge that contains really useful results, then clearly we must be prepared to utilise them. We cannot assume that education will always remain "safe from scientific solution" [Bro69, page 117].

The approach adopted here is more like the philosophical discussion that takes place in the pre-scientific phase of any discipline. The aim is to clarify the problems to be discussed scientifically, both those of Software Engineering, and also those of Software Engineering education, and to seek some unification of ideas upon which a curriculum can be constructed. Modern views of science frequently accept that this is a valid stage of science, and not "pre-scientific". Dye, for example, says that

> "The most creative part of enquiry seems to reside largely in generating the right questions. The question is the elemental, indispensable, scientific instrument." [Dye86, Page 103]

This corresponds to the analytical and speculative modes of philosophy identified by Kneller [Kne71], and is metaphysical in the sense of establishing a framework of models within which analytical theories are expected to develop. The need for such pre-scientific stages is discussed in [Hir69, page 185] and [Asi74]. As Ducasse has observed, however, this philosophical reflection is not being carried out for its own sake, nor as a matter of personal choice; it is something that is forced upon any person facing "practical problems of a certain type" [Duc69, page 169].

The realisation that pre-scientific analysis leads us into philosophical investigations is of little help in identifying a method, however, for as Popper has observed,

> "Philosophers are as free as others to use any method in research. *There is no method peculiar to philosophy.*" [Pop59, page 15]

The adopted method is essentially that recommended by Feuer for educational matters:

> "Are there any methods of philosophy other than being most honest with oneself concerning one's spontaneous, uncoerced beliefs? And don't we reject doctrines because we feel behind the facade of pedantic profession a certain dishonesty?" [Feu69, page 40]

Care must be taken not to over-react, however. It is neither practical or sensible to question *everything* we do, for this too would allow us to escape from tackling the really hard issues of Software Engineering education. Poincaré makes this point most forcibly:

> "To doubt everything and to believe everything are two equally convenient solutions; each saves us from thinking." [Poi13, page 37]

It is not the questioning that is important *per se*, but the rational processes that lead to the questions and the answers. The questions must be clearly posed, and the answers critically analysed. Even some seemingly pointless questions can serve to increase our understanding of the world. A careful study of Zeno's paradoxes, for example, leads to some significant insights into the nature of space, time and motion, even though the paradoxes may seem esoteric and artificial.

A rationalisation of the method actually adopted, and it can be no more than that for no clear methodology existed at the outset of the project, is as follows. First, an investigation into the nature of software engineering design was carried out in the light of a number of existing bodies of knowledge. The original intention had been to start with the philosophy of engineering, but a survey of the literature showed that no such thing existed, so the philosophy of science was chosen instead. In retrospect, this proved to be a very fortunate accident. The design process was also considered in the light of a number of other bodies of knowledge, including the (embryonic) philosophy of technology, the psychology of problem solving, and the theory of discourse. As a result of these initial explorations, a model of the software design process was developed. The model gave rise to a number of issues that required clarification, which in turn led to a refinement of the model. Bringing this model into contact with the task of curriculum design requires the adoption of a philosophy of education, and also discussion of some aspects of learning theory. It was obvious by this stage in the proceedings that learning theory also has a rôle to play in our discussion of the system design process, so one particular theory was explored to broaden our discussion, and also to provide a bridge to examples of curriculum design.

Alongside the development of the model, use was made of the understanding being gained by the analysis to carry out a number of curriculum design activities. These were not "scientific", controlled, experiments, but they allow discussion of some possible interpretations of our model in terms of practical teaching activities.

1.4 Presentation of the Research Results

The nature of this research programme poses some interesting problems in finding a suitable way to present the results. Most academic disciplines have developed specific styles for the publication of its literature, and the wise doctoral student conforms to these norms. Moreover, the end product of most research programmes is a set of conclusions, and the logic that is used to support and justify these conclusions can also be used to provide a structuring mechanism for the presentation of the results. This research, however, has been eclectic, and its end point is not a set of conclusions in the traditional sense, but the presentation of an exploration.

The main thrust of the research has been philosophical, but the styles that are often adopted for presenting the results of philosophical enquiry are not necessarily appropriate for our purposes[1]. The motivation for this research dictates that the thesis must be accessible to the software engineering education community. Educational practitioners find little of value in an academic philosophical approach consisting of "stories about philosophers, and ghosts from a philosophical cemetery" [New69, pp 165–167], and neither do engineers [Asi74, page 152]. The style we have adopted for the presentation of

[1]Indeed, many philosophers question the suitability of these presentation styles for the purposes of communication with other philosophers.

results can best be described as a practical philosophical one.

We start off in Chapter Two by considering a fairly specific, if complex, question: "in what sense is software engineering a scientific enterprise"? This question is focussed by Popper's philosophy of science, and then broadened by a discussion of three pieces from the Computing literature which have a direct contribution to make. This starting point is too restricted, however, to support the development of our model, so in Chapter Three we widen this base by considering software engineering as a technological activity, then as a problem solving activity, and finally as a process of discourse.

Chapters Two and Three are primarily resources that are drawn upon in the development of our model of software engineering design, providing insights and terminology that prove useful in this process. The construction of the model begins in Chapter Four, where the view of software design as a theory building process is presented and developed. This model gives rise to the need to refine two key concepts: theories and methods. This refinement is undertaken in Chapters Five and Six.

One aspect of the model, how the individual builds theories initially, shares a common theme with the issue of how students learn, for both can be considered to raise the question of how people construe themselves, and the world about them. This is clearly a complicated question, and it is far beyond the scope of this research programme to provide an answer, but in Chapter Seven Kelly's Personal Construct Theory is presented and discussed as a candidate framework within which the question can be addressed. This allows us to build links between our model and the teaching process, so that we have sufficient context to discuss some examples of the application of our model to practical curriculum design. Chapter Eight briefly discusses the application of this exploration, and describes some of the curriculum developments that the author has undertaken during the latter stages of this research.

Chapter Nine comprises concluding comments on the research programme, including a discussion of how well the aims have been met and observations on the research method. It also highlights future research that arises naturally out of this exploration.

A number of issues raised by this research have led to conference papers and publications available elsewhere. Appendix A contains a list of these, together with a brief description of the rôle they have played in meeting the aims of the research programme.

Chapter 2

Computing as a Science

"Science is nothing more but trained and organised common sense, differing from the latter only as a veteran may differ from a raw recruit: and its methods differ from those of common sense only as far as a guardsman's cut and thrust differ from the manner in which a savage wields his club"
Thomas Huxley

In this chapter a discussion of the question "In what senses is Software Engineering a Scientific Discipline" is set out. This question is important not because it helps us to classify academic disciplines but because it forces us to evaluate aspects of the discipline within a readily available body of knowledge, the Philosophy of Science.

This Philosophy of Science, of course, does not *describe* science but discusses it, and proposes a number of conflicting views on different aspects of the subject. It is not feasible or sensible to utilise all of these views in our discussion, so one particular view has been selected as a starting point.

Two principal criteria were used for the selection of a suitable view:-

1. The view selected must be mature enough to form a well-documented, stable, basis for the intended purpose. This is easy to meet, as a large number of philosophies of science have been proposed, and criticised, and a wealth of excellent, readily available, literature exists.

2. The view selected must still be progressive, in the sense that philosophers are still bothering to debate and criticise it as a candidate explanation. This excludes philosophies that have been found to be uninteresting because they can be refuted too easily, and also the classic philosophies, which, although still widely discussed for historical interest and because of the foundation they provide, are no longer generally held as explanations of scientific progress. This criterion is rather harder to satisfy, and perhaps needs some justification. Strictly, there is no reason why a classic view of science, such as Plato's or Aristotle's, should not be adopted. The danger in this, however, is that the discussion of Software Engineering turns into a thinly disguised evaluation and refutation of the philosophy itself, rather than a constructive analysis of the particular discipline within the given framework.

There is also the worry that a discussion framed in such terms could lack credibility among practitioners of software engineering education: such a reaction would be ill-founded, of course, but likely none the less.

The philosophy chosen as a starting point for the discussion is that of Karl Popper. In subsequent chapters, discussions will be broadened to include additional views as the need arises. An uncritical exposition of Popper's philosophy is presented in this chapter, for it is not important that it should be "true", only that it provides a good starting point for the discussions to follow.

This chapter could be viewed as setting the scene for a demarcation between Software Engineering and Computer Science. This is certainly not its intention, however, and two reasons can be given why any such demarcation, drawn as a side effect, may be detrimental.

1. The discipline of Computing has been saddled with the terms "software engineering" and "computer science". Any attempt to "define" these terms is a nominalist exercise that could have far-reaching consequences but little obvious merit. Such attempts that have been made in the past often seem politically motivated, rather than constructive (for example, departmental empire building, the partitioning of research funds, marketing advantages, and so on). The view adopted here is that if, as a result of a deeper understanding of the discipline of Computing, two disciplines emerge called "Software Engineering" and" Computer Science" then this is natural evolution, and a discussion of the results will be warranted : consideration of the *terms* simply because they currently exist is fruitless, and potentially dangerous. As Popper says,

   > "Disciplines are distinguished partly for historical reasons and reasons of administrative convenience (such as the organisation of teaching and of appointment), and partly because the theories which we construct to solve our problems have a tendency to grow into unified systems. But all of this classification is a comparatively unimportant and superficial affair." [Pop63, page 67].

   We shall follow this lead, "adopt the current lack of respect for etymology and go on to more serious things" [Bun74, page 19].

2. The traditional distinction between science and engineering is one of purpose [Rap74, page 94]. Science is motivated by a goal of increasing understanding, engineering by a goal of production [Sko72, page 43]. The link between the two is that the theories produced by scientists are utilised by engineers [Fei72, page 33]. If one adopts the view that the products of Software Engineering (i.e. software systems) are actually theories, in some sense, then this distinction is clearly not satisfactory. It may well be the case that an investigation of software engineering design will deepen our understanding of the relationship between the activities of science and engineering, which is more fruitful than classification for its own sake.

Support for the view that Computer Science and Software Engineering are too close to separate usefully for the purpose of curriculum design is provided by Denning, who writes that

> "In the core material, there is no fundamental difference between the two fields." [Den88, page 41]

2.1 Popper's Philosophy of Science

It is important to note that Popper is not playing nominalist games in his philosophy, that is, he is not attempting to define terms or to answer the question "what is science?", but to explore the ways in which science is actually carried out. Lakatos sums up Popper's view of science very succinctly when he states that it "can best be put in terms of 'conventions' or 'rules' governing the 'game of science' " [Lak74, page 243]. This investigation naturally leads to several subsidiary issues, some of which do require us to ask what we mean by certain terms, but these issues arise from the consideration of a particular problem. The idea that all philosophical investigation should be motivated by practical problems, and that "philosophy" carried out as purely linguistic analysis, contrary to Wittgenstein, is both pointless and meaningless, is vital to understanding all Popper's philosophy, including his philosophy of science. Indeed, much of Popper's philosophy of science can be fully appreciated only in the context of his philosophy as a whole. Unfortunately, this holistic view cannot adequately be reflected in such a short summary of the principal features of his philosophy.

Although Popper makes no attempt to *define* science, he does stress the importance of drawing a line of demarcation between scientific and metaphysical theories. He is at pains to point out, however, that he is not following Kant in equating the metaphysical with the meaningless, or suggesting that metaphysics is in some way inferior to science. This is an important point to note, because as we shall see this line of demarcation will partition a scientific *discipline* itself into scientific and metaphysical *pursuits.*

The classical (non-Popperian) distinction between science and metaphysics hinges on the idea that science proceeds by observation of the real world and the discovery (by induction) of the laws, embodied in theories, governing nature, whereas metaphysics proceeds by pure thought. Conventionalists, such as Poincaré, developed the view that we impose these theories on nature, rather than engaging in a search for *the* inherent, God-given, rules. Under this interpretation, theories are not true or false, but useful or not useful in particular situations.

Popper seeks to maintain the notion of truth in science, but to refute that of inductive proof. This he does by introducing the idea that theories are entirely man-made, in the sense that they are the products of intuition, possibly guided by history and observation, but that they can be shown to be false if predictions based on them can be shown not to correspond with the facts. In this way he also removes the idea that there is one "true" theory (that is, the true essence of the subject) and allows for a multiplicity of theories, all of which ultimately may be shown to be false. He replaces the idea of a hypothesis that science seeks to prove true, by that of a hypothesis which science seeks to prove false. The "truth" of a theory can be nothing more a measure of the extent to which science has attempted, but failed, to falsify the statements that follow deductively from it.

This approach is based on the idea that every theory contains universal statements, but that no number of individual observations

$$P(x_1) \wedge P(x_2) \wedge P(x_3) \ldots P(x_n)$$

can ever be sufficient to permit the deduction of

$$\forall x : T \bullet P(x)$$

unless the domain of interest, T, is completely covered by $x_1 \mathinner{..} x_n$. Moreover, if this ever happens, then the theory is of little interest, since it just records a number of observations. Popper claims that statements over such a restricted domain are not truly universal.

Although we are unable to induce the truth of a theory from any number of observations, we are able to infer its falsity from a single one. Popper assumes that a first order logic is sufficient for such deductions. From a number of true statements we can deduce, using a suitable logic, another true statement. This property is usually called the *transmission of truth*. Of course, we cannot conclude that a theory is true simply because we can deduce true things from it. However, if we can deduce things from a theory which we can show to be false, then we can assert the theory to be false. This property is usually called the *retransmission of falsity*.

Scientific activities comprise, according to Popper, just those activities for which such an approach is possible, and where a willingness to practice these activities is demonstrated. It thereby excludes all forms of activity where statements are made which are not refutable, such as

- would-be sciences like astrology, where the predictions made are usually so vague that it is hard to decide if they have come true or not.
- pure mathematics, where "truth" is just a measure of internal consistency[1], or provability in the case of constructive mathematics.
- activities based on "theories" which are held dogmatically, where observations which appear to refute the theory are simply re-interpreted so they support it, (such as Marxism's blind acceptance of Hegel's philosophy), situations where terms are

[1]Although Popper expresses this view quite strongly in his earlier work, he later complicates matters when he introduces his notion of the third world. This is discussed below.

redefined in an ad hoc way to exclude all refuting cases from the theory, or where metaphysics is prior to experience.

It must be stressed that Popper in no way denigrates such activities: it merely observes that they should not be considered scientific.

One very important corollary of this line of demarcation is that the *methods* of science, and any theories we may have regarding them, are not themselves generally amenable to the activity of science. This separation of the objects from the methods shows clearly that we must take care when using the term "scientific" to establish which of its many senses we are invoking. For example, the term "scientific method" does not mean a method that can be studied scientifically, but a method that is used in a discipline where scientific theories are the norm.

Popper identifies several main thrusts to scientific activities based on this notion of refutation. First, various properties of the theory can be checked. In particular, we can ask whether a theory is (internally) consistent. If a theory allows the derivation of both P and $\neg P$, then it must be rejected, for $P \ , \ \neg P \vdash Q$, regardless of the Q we choose. Clearly a Q that does not correspond with the facts is trivially found, and hence the theory is easily refuted. Similarly, we must reject as useless any theories which are tautologous, or self-fulfilling. Such a theory will merely tell us things about all possible worlds (in which our underlying logic holds), such as $P \vee \neg P$ or $P \Longrightarrow P$, and will add nothing to our understanding of the world: they are information free.

Second, we can compare each theory with other similar theories that exist. Not all pairs of theories will be directly comparable, as they may make statements about different facets of the real world. For those that are comparable, Popper provides an opinion on possible metrics that can be used for selecting the better theory. Of particular interest is Popper's claim that his view of science turns upside down the intuition that we should look for the most likely explanation of things. For him, the bolder a theory, the less likely it is, but also the more refutable it will be. Refutable theories are more credit-worthy than theories which are hard to refute, therefore we should be seeking theories which are less probable, rather than more probable. This seems to be using the term "probable" in a rather unconventional way [Put74, page 224]. An important point to note here, is that we must remain in the domain of observability: producing a theory for the orbits of planets based on the existence of some extra-terrestrial, undetectable, being might be an improbable theory, but it is not refutable (at this moment in time) and so is not an acceptable candidate as a scientific theory.

Following on from this idea, we can see that more general theories are to be considered more useful (in science) than less general. We can consider as a normal form for a theory T

$$\forall x \bullet P_T(x) \Longrightarrow Q_T(x)$$

where $P_T(x)$ defines the domain of application of the theory and $Q_T(x)$ defines the conclusions that can be drawn. The theory T', expressed in normal form as

$$\forall x \bullet P_{T'}(x) \Longrightarrow Q_{T'}(x)$$

will be more general than T if

$$\forall x \bullet P_{T}(x) \Longrightarrow P_{T'}(x)$$

For example, if a theory allows us to deduce that some property follows from the assertion that x is a mammal, then the theory is clearly more general than a similar theory that allows us to deduce the property only if x is a dog, for if x is a dog then x is also a mammal, That is,

$$\forall x \bullet dog(x) \Longrightarrow mammal(x)$$

As well as generality, precision can be used for the comparison of similar theories. The theory that says more precisely what will happen is more useful than the one that is less precise. The theory T'', expressed in normal form as

$$\forall x \bullet .P_{T''}(x) \Longrightarrow Q_{T''}(x)$$

will be more precise than T if

$$\forall x \bullet Q_{T''}(x) \Longrightarrow Q_{T}(x)$$

For example, *eats*(*pork chops*) is more precise than *eats*(*meat*), since

$$eats(pork\ chops) \Longrightarrow eats(meat)$$

We can thus form a lattice, using generality and precision as orderings on our theories. Figure 2.1 shows the lattice for the examples introduced.

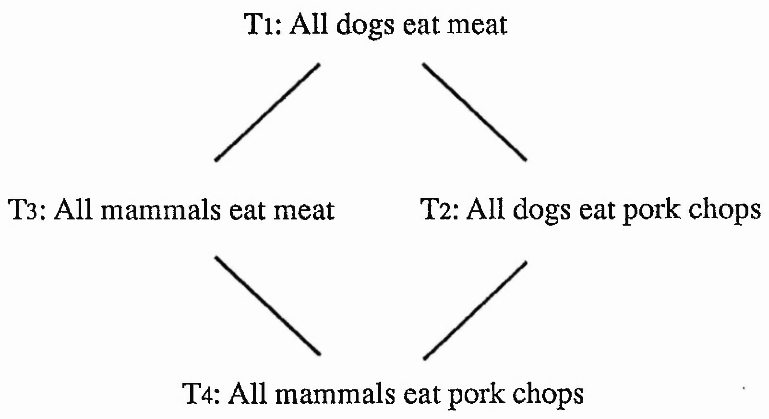

Figure 2.1: Lattice of Generalisation and Precision

T_1 is the least bold theory, and is hardest to refute. We need to find at least one dog that does not eat meat. T_2 is more precise, and hence easier to refute than T_1: any pork-hating dog will do. T_3 is more general than T_1, any meat-hating mammal will serve to refute it. T_4 is the most general and the most precise. To refute it, any mammal that does not eat pork chops will do.

The third major scientific activity we can engage in is to check our theories in some way with the "real" world. This is the central issue of debate for most of the philosophies of science, and there have been many suggestions as to what it means for a theory to be "true". The classical view was that a true theory was a perfect discovery, from the world of forms, of the essence that God gave to some object. A more modern view, that of the pragmatists, is that a theory is neither true of false, merely useful or not useful. Theories are not uncoverings of God's laws, but man's successful attempts to impose order on the real world. Another proposed view is that theories are man made, but that they are true if they are consistent with the totality of existing knowledge, that is, with all other theories. Such a coherence theory of truth necessarily means that the "truth" must change as theories change. Popper prefers to dissolve this issue, by declaring that the truth of theories is not the real question, rather we should be concerned with the truth of the statements that they entail. He adopts a correspondence theory of truth, that is, statements are true if they correspond with observable facts. This is not a new stance, but one that was largely discredited by the number of paradoxes that could be constructed by its adoption. Tarski, however, with his observation that we cannot discuss the meaning of terms in a language without recourse to some meta-language, effectively rescues the correspondence theory from many of these criticisms [Tar56]. Thus we can say that

The statement P is true if and only if P corresponds to the facts.

where P is the name of some statement in another language.

The acceptance of a correspondence theory of truth raises another important point, namely the objectivity of science. That is to say, we want to distinguish between

P corresponds with the facts.

and

I believe (or I know) that P corresponds with the facts.

Popper notes that objectivity is a methodological issue, and hence not itself a scientific concern. Methodology cannot be considered as a scientific concern until we know how to make refutable statements about it. The current state of disciplines such as psychology, sociology and management is such that refutable statements about how individuals act are hard to find[2]. It cannot become a scientific concern until psychology, for example,

[2]Popper originally stated this several decades ago, but there seems to be no reason to believe that the situation has changed much

gives us the necessary framework to analyse scientifically statements about how people think and formulate theories.

In common with many other philosophers, Popper places a layer of indirection between the scientific theories and reality. This layer comprises statements about the real world that we will call observation reports. Popper says that objective statements about the real world must be *"inter-subjectively testable"*. That is to say, they must be reflected by observation reports that are amenable to inter-subjective criticism. In fact, Popper does not insist that these tests take place, just that they must be possible. This is the reason for requiring the property of repeatability for scientific experiments. We must be able to re-observe experimental results if we wish. Thus the decision to accept an observation statement as "true" is another methodological concern: it is the decision not to keep attempting to refute the observation by further investigation. Popper sums this up very elegantly when he says

> "The empirical basis of objective science thus has nothing "absolute" about it. Science does not rest on solid bedrock. The bold structure of its theories rises, as it were, above a swamp. It is like a building erected on piles. The piles are driven down from above into the swamp, but not down to any natural or "given" base; and if we stop driving the piles deeper, it is not because we have reached firm ground. We simply stop when we are satisfied that the piles are firm enough to carry the structure, at least for the time being." [Pop59, page 111]

Popper's philosophy is clearly methodological in the sense that it contains severe constraints on the behaviour that he is prepared to admit as scientific. The ideal of refutation, for example, provides a meta-methodological rule: *No rule may protect statements from being refuted.* His views on method seem to be summed up by his statement

> "pronouncements of this theory [of method] are ...for the most part conventions of a fairly obvious kind. Profound truths are not to be expected of methodology." [Pop59, page 54]

It has been largely left to those following Popper, such as Kuhn, Lakatos and Feyerabend, to discuss the implications of his philosophy for method, in the sense of guidelines for scientific progress. He does, however, observe that

> "If the method of trial and error is developed more and more consciously, then it begins to take on the characteristic features of a "scientific method". This "method"[3] can briefly be described as follows. Faced with a certain problem, the scientist offers, tentatively, some sort of solution—a theory. This theory

[3] *Popper's Footnote:* It is not a method in the sense that, if you practice it, you will succeed; or if you don't succeed you can't have practiced it; that is to say, it is not a definite way to results: a method in this sense does not exist.

science accepts only provisionally, if at all; and it is most characteristic of the scientific method that scientists will spare no pains to criticise and test the theory in question." [Pop63, page 313]

Popper sees his method of refutation as akin to the process of natural selection: the "fittest" theories survive.

One of the few concessions to method that Popper does appear to make is the observation that, when a theory appears to be refuted, we need ways of deciding how much of the theory is to be rejected, or, indeed, if the theory can be protected in some way. Consider a set of assumptions a and some theoretical system T. If we consider the consequence closure of these as τ, then our hypothesis H for refutation can be expressed in the form

$$\tau \Longrightarrow H$$

If we observe $\neg H$ then clearly our laws of logic allow us to infer $\neg\tau$. As τ was a closure, however, we cannot infer what part of it is to be rejected. It would be methodologically unsound to reject all of τ, and start again from scratch every time. It would, of course, be logically sound.

One approach might be to change our set of assumptions: this is often used as a way of propping up theories that are held dear, and Popper identifies this as, in general, undesirable. He does acknowledge that on occasions such a commitment to a theory has proved well-founded, such as when Newtonian mechanics was defended in the light of the (apparent) refutation arising from the failure of Uranus to follow its predicted path. The introduction of an auxiliary hypothesis asserting the existence of an additional planet, Neptune, effectively rescued the theory. This defence is only acceptable as science because the existence of Neptune could itself give rise to refutable predictions. The position of Neptune could be predicted to a sufficiently small region of the space-time continuum that it was deemed possible to search it exhaustively. Had the defence been the existence of some heavenly body whose location could not be predicted, then this would not have constituted scientific practice.

The alternative is to refute T, but do we need to refute all of T? Popper suggests, pre-empting Kuhn and Lakatos, that if we construct T as an enrichment of some other theory, T', which has stood the test of time (that is, has been subjected to severe attempts at refutation, but survived) then an obvious place to start is by assuming that the extensions have been refuted rather than the whole of T. Care needs to be taken here with implicit assumptions, such as those embedded in the theories we are using for criticising our observation statements or those embedded in the notation we are using. In the physical sciences, for example, all measurements are taken indirectly, often using a complex theory of measurement.

In choosing between theories, Popper suggests a number of factors that should be taken into account, including precision, generality, degree of detail, survival of tests, number of tests suggested, and degree of unification. Further discussion of these issues will be delayed until the chapters on methods and theories, where more concrete examples relevant to computing can be given.

A particularly important question is the source of theories. For the inductivist, theories can be found lurking among the collection of observations that have been made. Popper refers to this as the bucket theory of knowledge [Pop72, pages 341-361]. The observations are made, stored, and then a theory emerges. He dismisses this out of hand, saying that it is nonsense to suggest that observations can precede theory in this way. We cannot just "observe" but we have to "observe something".

He replaces the bucket theory with the searchlight theory [Pop72, pages 341-361]. We search out observations to refute theories, or possibly to provide psychological support that our theory can at least pass some tests. The searchlight is provided by the current theory. This view requires motivation for the search, of course, and this is provided by *problem situations*. In Popper's philosophy, problems are the root not only for all philosophical enquiry but also for all scientific enquiry. His method can be expressed rather simplistically as

$$P_1 \rightarrow TT \rightarrow EE \rightarrow P_2$$

The initial problem, P_1 is investigated and gives rise to a tentative solution, TT, which will be a theory within which the problem can be solved. The scientific method is then applied to this solution, resulting in error elimination, EE, which in turn leads to new problems, P_2. The problems themselves, however, are not theory free. They will be phrased in terms of existing theories. This raises the question of which comes first, the problem or the theory, but as Popper implies, this all happened long ago, so does it really matter? All problems that are expressed in current civilisations inherit a vast array of theories.

As to the initial source of individual theories, Popper is unconcerned. These sources are in the second world (see below) and hence are the problem of psychologists. He does observe that intuitions undoubtedly have a rôle to play, but that we must not forget that these intuitions are likely to develop as a product of theories, so are not static during the solution of a particular problem.

The notion of refutation that Popper introduces for science potentially leaves mathematics on the non-scientific side of the line. In his earlier works, Popper says this is correct; mathematics is a metaphysical concern, but it does have a major rôle to play in science. Mathematical theories can be made (empirically) scientific by bringing them into contact with the real world, that is, by providing interpretations for their terms. At this point, says Popper, they cease to be mathematical theories and become descriptive theories of the real world. It is here that Popper differs from the conventionalist: he asserts that such descriptive theories can now be refuted, as their logical status no longer protects them. Strictly, of course, it is the interpretation of the theory that is being refuted: a theory can only be refuted as a theory *of* something. All non-scientific theories have this potential (as science develops, statements often become refutable that could only be treated metaphysically before), but mathematics is especially useful. The nature of mathematics itself is such that it provides rich deductive systems that can aid in the process of refutation. The more that can be deduced within the theory, the more refutable it is, and hence the better the theory.

In his later work, however, Popper appears to adopt a different attitude to reality, and hence to mathematics, that raises several questions regarding his earlier philosophy. Unfortunately, he does not review all his published works on other issues to accommodate this dramatic change, and so we are left guessing as to how he intends to answer these questions. The change involves the introduction of three worlds: the (first) natural world, the (second) world of ideas and thoughts private to individuals, and a third world of man-made structures. In this third world reside, for example, mathematics, theories, books of knowledge, and man-made physical objects. Such third world objects may have embodiments in both the first and second world, as well as being in the third world. Books, for example, have a physical form, and may also be conceptualised, as holistic entities, by individuals. Popper states that the true domain of interest of epistemology is this third world, and we should not get bogged down in trying to understand the ways in which knowledge is stored, manipulated and communicated by using the other worlds.

This third world, according to Popper, is where a proper study of any discipline should occur. We should study the structures (such as theories) that are produced before we start to ask how they might be arrived at by using the second world processes. Furthermore, this third world has assumed an autonomy of its own. We can now treat third world objects in isolation from the processes that brought them about. In particular, we can *discover* properties of these objects that we never knew they had: that is to say, these objects can themselves be considered as amenable to scientific investigation. This idea is not new, and the potential dilemma it poses has been recognised by Euler, for example, when he writes

> "As we must refer the numbers to the pure intellect alone, we can hardly understand how observations and quasi-experiments can be of use in investigating the nature of the numbers. Yet, in fact, as I shall show there will be good reasons, the properties of the numbers known today have been mostly discovered by observation, and discovered long before their truth has been confirmed by rigid demonstration." cited in [Pol68, page 3]

Given, as a third world object, a formally expressed theory, we can ask whether term x is a deductive consequence of terms y and z. It may seem that Popper's third world is similar to Plato's world of forms, with a separate existence that can be discovered. There is a major difference, however: Popper's world is not Divine but man-made. For this reason we must accept that it can be changed by men, and so we must reject all notions that the third world captures any eternal "truth".

Our acceptance of the third world allows us to modify somewhat Popper's original statement that mathematics is a metaphysical activity unless we are using a mathematical theory to describe the real world. We must now admit the possibility that a mathematical theory can be subjected to scientific analysis, thus giving rise to more mathematical meta-theories. These meta-theories are scientific, but the "reality" that they describe is not the natural world but other mathematical objects in the third world. It is this that allows a scientific treatment of conjectures regarding, for example, the theory of prime numbers.

We must now reconsider the whole idea of refutation, however, for it seems that such a "scientific" exploration might well yield not only a refutation of Euler's conjecture, but also a "proof". The solution to this dilemma, which Popper does not appear to have discussed anywhere, is to be found in the bedrock of mathematical culture. To use Popper's earlier analogy, as we seek to drive the piles deeper and deeper we reach a layer of universal agreement upon which our deductive proof sits. This layer includes concepts such as orderings, which allow us to construct proofs using mathematical induction. These are, of course deductive in nature. Such proofs rest on the acceptance of a number of implicit theories. In proving

$$\sum_{1}^{n} i = \frac{n(n+1)}{2}$$

for example, we assume the existence of integers with particular properties. Properties like these have become such a central part of our third world that we cannot imagine them to be false, and so we proceed as if they are undoubtedly "true". We should remember, of course, that this has not always been the case. The existence of the number 0, for example, was disputed for a long time before it found its way into the collection of generally accepted third world objects, and "proofs" based on naive set theory were also accepted largely without question until quite recently (and in some quarters continue to be so).

The ultimate example of this, of course, is the very logic itself that Popper is quite content to use for the purpose of deduction leading to refutation. This is itself a third world object. Dare we allow the possibility of refuting it? If so, what possible logic do we use to do so? This is a very real problem which Popper has allowed to slip in, but we can actually survive without an answer. We will just accept that certain mathematical and logical foundations of our third world are to be treated as not refutable, and revel in the fact that most of the more advanced mathematical concepts we require can be constructed from these simpler foundations. This effectively overcomes the problems of language shift that accompany theory development, where terms take on new meanings as theories are developed. Terms such as "matter" and "energy", for example, have quite different meanings before and after Einstein. In our Third World, we can usually give a formal semantics to our language, that is, we can define our terms using terms drawn from other established Third World objects. We can define relations, for example, in terms of sets.

2.2 Computing as Science: a Review

It is currently considered good advertising copy to apply the term "science", or "scientific", to products that you are trying to sell, thus we have scientifically designed washing powders, toothpaste, motor cars and electric razors. It is not surprising, therefore, that there has been a move in recent years to portray the production of software systems as scientific. There is a grave danger, however, that the term is really being used to mean "high quality". If this is the case then asking the software industry to become more scientific is actually just a thinly disguised request for it to become "better". Such a request is completely lacking in guidance, both as to how such improvements are to be brought about and also how to ascertain whether any improvement has actually taken place.

This section is an attempt to view Computing as a scientific activity, as described by Popper. There is no attempt, of course, to answer the question "is Computing a science", only to discuss it. If we manage to bring Computing into reasonable contact with Popper's philosophy then we will have established a useful starting point for subsequent discussions. Areas of difficulty will have been identified for further analysis, which could lead to a refinement of our views of Computing or science. A few of these areas will be discussed in this thesis, as they are pertinent to the central pedagogical aim, but many more will be left as open research topics. If the contact can be made only by a series of totally unreasonable interpretations (such as might be found in asking the question "is an apple scientific"), then we must either look for an alternative, and more accommodating, philosophy of science, or accept that it is difficult to view Computing as a science.

The approach adopted is to review three major contributions to the literature of Computing that seem to have particular relevance to the question. These three works are very different in nature, allowing us to explore different facets of the question.

1. Hoare's "Programming: Sorcery or Science?" [Hoa84]: this paper addresses programming in a wide sense, and holds out the promise of a frontal attack on the problem being considered.

2. Gries's "The Science of Programming" [Gri81]: this book treats programming as the task of moving from a formal specification to code. It presents a number of principles that can be used and also discusses the technical details of proving programs correct.

3. Naur's "Programming as Theory Building" [Nau85]: this paper also treats programming in a very wide sense, but uses a more psychological view of theories.

"Programming: Sorcery or Science?" by Hoare

This paper [Hoa84] raises one important question regarding the literature of a scientific discipline: should the papers and books published themselves be scientific in the sense that they can be refuted? The answer assumed here is "no", for otherwise all discussion *of* science must either be detached from science or carried out in a scientific manner. Great care must be taken if we accept this answer, however, in case we unconsciously open the floodgates to such metaphysical writings at the expense of truly scientific literature. Care must also be taken to ensure that the reader is aware of the ontological status of published papers, for otherwise results put forward may be given a status as part of the discipline's scientific body of knowledge. This is unlikely to present a problem in established sciences, where there is little danger of confusing the discussion of how an experiment should be conducted with the report of its results. In computing, however, there are complications, such as the production of CASE tools, where there is a risk of elevating discussion of how computing should be carried out into a theoretical basis for such tools, and treating such a basis as scientific. Hoare's paper is clearly metaphysical, in the sense that it is talking about the processes involved in programming, rather than talking about programs (or even programmers) in a scientific way.

Once the status of this paper has been noted, we can accept it as part of the scientific literature. It is a piece of persuasive discourse, the purpose of which is to convert those who see programming as a black art to the point of view of those who believe that software production should be "based on underlying theories and follow the traditions of better-established engineering disciplines"(page 5). One remarkable fact about this paper is that Hoare manages to carry out the whole discussion without any real reference to "science". He seems to equate professional engineering practice (which he assumes as a goal) with scientific practice, and in so doing answers the question in his title simply by definition. The rest of the paper is largely divorced from the title, being a discussion of the ways in which software development should proceed. We can, however, analyse some of the more relevant statements and see if they offer any insights into Computing as science.

Hoare says, for example,

> "The chief programmer, like the architect, will start by discussing requirements with his client. From education and experience, the programmer will be able to guide his client to an understanding of his true needs." (page 8)

This may well be an accurate reflection of what should happen, but it does little to convince the reader that programming is scientific. Architecture, particularly the modern variety, is notorious for its highly subjective nature, and by introducing the client's understanding into the picture, Hoare admits blatant psychologism, which Popper continually sought to remove from the realms of science. Similarly, the term "true needs" requires careful consideration. If we accept Popper's view that observations are theory laden, then the discovery of needs can only be as "true" as the theories we used to search them out. Hoare seems to be setting the scene for absolute certitude in science, and hence in programming, by assuming that initial observations can be taken as "true". In so doing, he is promising more than either programming or science can deliver.

Hoare goes on to say that

> "This activity will culminate in a complete, unambiguous, and provably consistent specification for the entire end product. It will serve the same rôle as blueprints in engineering or scaled plans and elevations in architecture."(page 8)

Not content with absolute truth, Hoare now admits the whole truth! As Tarski has noted, however, relative completeness is a more practical tool than total completeness. The latter, in logical terms, means that the maximal set of consistent conclusions can be drawn. Clearly it is relative completeness that Hoare intends: he wants to be able to draw all consistent conclusions relevant to the problem. Made explicit, the problem with this statement is manifest. What procedure can the programmer adopt that will guarantee discovery of a complete specification? In science it has long been accepted that such a procedure is impossible.

In similar vein we must ask what Hoare means by an "unambiguous specification". He could be suggesting the use of a notation with an unambiguously parsable grammar, a fairly simple thing to ensure. He could also be requiring a language with some universally acceptable semantics that can be interpreted in only one way (and he must accept the problem of subjective understanding, because of the psychologism he admits elsewhere); a rather harder thing to achieve, and we clearly do not want such an unambiguous specification. An elevation in architecture, for example, does not uniquely define a wall, but leaves open various properties such as choice of construction, facing and geographical location. If we assume that Hoare requires some form of unambiguity in the second sense above (and not just an unambiguous grammar) then we may observe a possible cause of the problem. Hoare wants to use the specifications in two ways; as artifacts to establish contractual boundaries, but also as blueprints for construction, although he does later observe that alternative forms (such as prototypes and models) may be more suitable for the first purpose. This dual purpose is problematic because we want the ambiguity to be in the details of construction and not in the properties of the artifact that the customer is interested in. This can be achieved by noting that observations are theory laden, so we want to observe the specification in different lights: the theory that has been developed in conjunction with the customer should yield unambiguous observations; that used by the construction engineer should be unambiguous in certain important aspects but leave freedom of interpretation in other respects. It might be said that implementation details should not appear in a specification at this level, so that all aspects should be unambiguous, or that any "ambiguity" should be removed by the introduction of non-determinism. The fact remains, however hard we wriggle, that every observation must be interpreted in the light of some theory, and unless we can ensure that a common theory is being used to illuminate the observation, we cannot achieve unambiguity. One way around this is to use not second world semantics (what we read into a specification) but a formal semantics. This does not help us to "convince" customers, or to ensure common "understanding", of course, but it does support formal analysis, and may expose areas of misunderstanding or strategies that can be used to provide convincing arguments.

Once the problems of requirements capture and analysis have been overcome, Hoare is on safer ground. He then displays a true Popperian spirit by adopting refutationalism:

> "...it will be possible to devise a series of rigorous and searching acceptance tests ...If the product fails the test, and the implementors claim that the test is unfair, any competent logician or mathematician will be able to decide who is right" (page 9)

He is at odds with Popper, however, when he says that

> "...the chief programmer will convince himself and his colleagues by mathematical proof that if each of the components meets its specification, then when all the components are assembled, the overall product will meet the overall specification agreed by the client." (page 9)

for such "conviction" has no place in Popper's science. Rather we should view proof as an aid to refutation, by exposing yet more statements that could possibly be tested.

The problem here is that we are starting to make the transition from scientist, working with the real problems of a client, to mathematician, working with conventionally defined third world objects, in addition to our agreed specification. These objects, according to Popper, are still open to refutation, but in practice we accept them as irrefutable. Once this transition has been made, we can use deductive reasoning to establish irrefutable chains of reasoning, that is, formal proofs. We should still avoid confusing such proofs with "convincing arguments", however, for there are too many convincing arguments that are logically flawed, or even sound logical arguments that are not at all convincing. Rather we should separate the two concerns, allowing conviction to act as a reason for questioning some of our assumed theories, when a deduction is counter-intuitive for example, but using formal proofs to provide the chain of reasoning without the distraction of intuition.

In observing that constructing a large and complex program, then attempting to debug it, is a flawed approach, Hoare is essentially observing that it is not sensible to construct a large theory and continue to defend it against refutation on the way (by refusing to apply deductive reasoning), and then submit it to a massive dose of refutation after we have constructed an empirical (first world) object. Rather we should stay in control of our third world theories and get these consistent with the requirements, and all other accepted theories, as far as possible before unification and implementation.

He does seem to be overstating the case, however, when he says

> "Because of the clarity of program structures and the completeness of design documentation, it will be quite easy to determine which parts of the design and coding need to be changed in order to meet a new requirement." (page 10)

It may well be the case that we can identify areas of change if the new requirements can still be viewed in the light of the existing theory, for then we have an appropriate searchlight. If the new requirements effectively refute the existing theory then there is no way of knowing how radical, or localised, the changes need to be. It may well be argued that such a scenario is a complete redesign, and not a modification, for as Turski has observed,

> "The crux of the matter is that an ineptly chosen term masks the real issue. (Once again, sloppy linguistic habits and a childish enthusiasm for new games that can be played without rules have lead us astray.) Maintenance, as defined by dictionaries, is the act of maintaining, i.e. of keeping in an existing state, of sustaining against opposition or danger, etc. Yet, to quote a friend of mine, software engineering is the only discipline where adding a new wing to a building would be considered as a maintenance activity." [Tur81, page 107]

In this case Hoare must provide a mechanism, or meta-theory, that can be used to decide when to modify and when to start again.

Hoare's optimism also betrays itself when he talks about improved estimates of project cost, delivery time and size of the final program. The existence of good substantive theories does not ensure the existence of effective meta-theoretical procedures. If it did, then we should expect scientists to be able to predict how long it will take to find a cure for cancer, or a mathematician to predict how long it will take to find a particular proof . In these circumstances it is only experience that can help. Indeed, it may even be easier to make accurate estimates if we stick to a non-scientific approach, for rules-of-thumb are stable and there are necessarily case histories to study. Of course, quality cannot be ensured. Just as the motor mechanic who must service a car in a fixed time may have to ignore any faults that have not been allowed for, and may adopt a work routine so that he does not even notice them, so may the software engineer ignore additional "features" that might creep unintentionally into the system, and avoid being too analytical in case such problems are noticed. Neither should we ignore the fact that many estimates are currently made by those with vested interests in keeping estimated costs as low as possible, and not by those with the technical expertise and experience to make sensible judgements. Being scientific will no more overcome these problems than it overcame the funding problems of many great scientists who had to work in under-resourced laboratories even when they were about to make great discoveries.

Hoare goes on to mount a fairly explicit attack on inductionism in Computing when he notes that

> " ...interpolation and extrapolation are wholly invalid [in Computing]. The fact that a program works for values zero and 65535 gives no confidence that it will work for any of the values in between, unless this fact is proved by logical reasoning based on the very text of the program itself. But if this

logical reasoning is correct, then there was no need for the test in the first place." (page 12)

Unfortunately he has again used a psychological term in talking of "confidence". The fact is that for most people such testing *does* build confidence. The real question is whether such confidence is well-founded, and, as Popper has noted, no number of observations of results we think are probable should serve to increase our confidence. That should happen only when something that is *unlikely* but predicted turns out to be the case. This is is partially reflected in the testing strategy that favours special cases over random instances. Of course, a search for such cases requires a theoretical understanding of the program.

Hoare gives the impression that this approach will do away with the whole of testing, replacing it by logical reasoning. This cannot be the case, of course, if our programs are to be considered in any sense as first world empirical objects. If we are prepared to treat them as purely formal objects, forever located in the third world, then we can escape the need to test, but then our programs will be of little practical use. What Hoare actually achieves is a relocation of the onus of testing; by establishing, once and for all, the customer's requirements before the program design is started, he removes the need to test the program as a suitable candidate for solving the customer's problem. This may be theoretically valid, and could even be legally enforcible, but in practice, the *professional* software engineer will still want to test such areas as the user interface design, and possible many others, during the design stage of a project to give the clients a chance to change their minds. As is well known, user's "requirements" change as they see the artifact being developed, that is, they alter their own theories as the empirical evidence of observation becomes available for their refutation. It may be argued that our new software engineers will be so proficient at requirements analysis that their clients will not need this second chance; in practice, just as the building of a new road causes extra traffic flow, so the client, seeing what great improvements were effected by the engineer before the design started, will be even more keen to monitor the progress of the design in case the possibility of more improvements presents itself. Provided that the software engineer is able to manage the pricing of such changes, there seems no reason to throw out such testing just because it is not necessary to achieve conformance with the original specification. We should observe, however, that such "testing" might be better called "test-driving", because although the client may view it as a test, the engineer should not. We shall reserve the term "animating" for testing of this kind [Coh82].

The second relocation of testing is caused by the use of formal proofs of correctness as evidence that software satisfies its specification. Figure 2.2 shows a proof that a given program meets a given specification. This proof is carried out in a simple Hoare logic[4].

[4]We will not quibble over the fact that there should clearly be an upper bound on the data: the data set is practically infinite, in the sense that for most real problems it is infeasible to test all possible cases. This is analogous to the acceptance of Newton's theory for certain practical tasks, the onus being on the engineer, of course, to ensure that the theory is not used outside its domain of application. In this case, we assume that the programmer intends to prepend a sufficiently strong initial condition guard.

Show that

$\{(y > 1\}$
$n := 0;\ k := 1;\ \mathit{WHILE}\ y > k\ \mathit{DO}\ \mathit{BEGIN}\ k := k * 2;\ n := n + 1;\ \mathit{END}$
$\{2^{n-1} < y \leq 2^n\}$

Proof

1	$\{(y > 2^n) \wedge (k = 2^{n+1})\}n := n + 1\{(y > 2^{n-1}) \wedge (k = 2^n)\}$	Axiom
2	$\{(y > 2^n) \wedge (k * 2 = 2^{n+1})\}k := k * 2\{(y > 2^n) \wedge (k = 2^{n+1})\}$	Axiom
3	$\{(y > 2^n) \wedge (k * 2 = 2^{n+1})\}$	
	$k := k * 2;\ n := n + 1$	
	$\{(y > 2^{n-1}) \wedge (k = 2^n)\}$	1, 2 IR Comp.
4	$(y > 2^{n-1}) \wedge (k = 2^n) \wedge (y > k) \Longrightarrow (y > 2^n) \wedge (k * 2 = 2^{n+1})$	Lemma
5	$\{(y > 2^{n-1}) \wedge (k = 2^n) \wedge (y > k)\}$	
	$k := k * 2;\ n := n + 1$	
	$\{(y > 2^{n-1}) \wedge (k = 2^n)\}$	3, 4 IR Conseq.
6	$\{(y > 2^{n-1}) \wedge (k = 2^n)\}$	
	$\mathit{WHILE}\ y > k\ \mathit{DO}\ \mathit{BEGIN}\ k := k * 2;\ n := n + 1;\ \mathit{END}$	
	$\{(y > 2^{n-1}) \wedge (k = 2^n) \wedge \neg(y > k)\}$	5 IR Iteration
7	$(y > 2^{n-1}) \wedge (k = 2^n) \wedge \neg(y > k) \Longrightarrow 2^{n-1} < y \leq 2^n$	Lemma
8	$\{(y > 2^{n-1}) \wedge (k = 2^n)\}$	
	$\mathit{WHILE}\ y > k\ \mathit{DO}\ \mathit{BEGIN}\ k := k * 2;\ n := n + 1;\ \mathit{END}$	
	$\{2^{n-1} < y \leq 2^n\}$	6, 7 IR Conseq.
9	$\{(y > 2^{n-1}) \wedge (1 = 2^n)\}k := 1\{(y > 2^{n-1}) \wedge (k = 2^n)\}$	Axiom
10	$\{(y > 2^{-1}) \wedge (k = 2^0)\}n := 0\{(y > 2^{n-1}) \wedge (1 = 2^n)\}$	Axiom
11	$y > 1 \Longrightarrow y > 2^{-1}$	Lemma
12	$\{(y > 1\}n := 0\{(y > 2^{n-1}) \wedge (1 = 2^n)\}$	10, 11 IR Conseq
13	$\{(y > 1\}n := 0;\ k := 1\{(y > 2^{n-1}) \wedge (k = 2^n)\}$	9, 12 IR Comp.
14	$\{(y > 1\}$	
	$n := 0;\ k := 1;$	
	$\mathit{WHILE}\ y > k\ \mathit{DO}\ \mathit{BEGIN}\ k := k * 2;\ n := n + 1;\ \mathit{END}$	
	$\{2^{n-1} < y \leq 2^n\}$	8, 13 IR Comp.

Figure 2.2: A Proof of Program Correctness

For the initial problem, we were faced with an infinite set of test cases. Now we are faced with no tests at all *unless* we attempt to refute the theory of the programming language upon which the proof rests (or, perhaps less likely, the theory of inequalities, or even predicate logic itself). We might argue, of course, that these theories are to be treated as conventionally true. They are beyond question. They *define* properties rather than describing them. This might well be the case for the lemma on line 11, for example. We could treat the axioms and rules of the Hoare logic in the same way, but this will only work if we are prepared to accept that our programming language is a pure third world object. Once we accept that our language is implemented in some real way, then we have simply shifted the onus of testing onto the implementor of the compiler, who must now show that each of the instructions used gives rise to empirical observations that will not refute the theory, and also that the implementation of natural numbers will satisfy the lemmas.

This task is no simpler than the testing of the original program, for each assignment statement (in fact, each possible call of an assignment statement), for example, must be shown to satisfy its axiom. It does, however, allow the reuse of test results. The compiler writer can adopt a similar strategy, of course, by formally proving his compiler, contingent upon the code of the microcode writer, who can pass the task down to the hardware designer. Ultimately, the responsibility can be laid at the door of the engineer responsible for the fabrication system used in the production of the devices used. It is debatable whether the engineer has any right to pass responsibility on to the scientist responsible for the theories of solid state physics that underpins his design.

Alternatively, we could think of our Hoare logic not as a convention, but as an empirical theory produced by analysis of an existing implementation of a language. Testing in this case is an attempt to refute the theory in the usual way. This view seems less strange if we remember that languages such as Ada were implemented before being given a formal semantics.

If we accept that the relocation of testing can be thought of as the avoidance of testing for a particular software engineer, then Hoare's case can be supported. Formal deduction allows this relocation, and effectively permits software engineers to build upon the theories of others, in just the same way as scientists do. In conclusion, Hoare's paper seems to be recommending a scientific attitude, together with the use of the mathematico-scientific method, as the way ahead.

"The Science of Programming" by Gries

This book [Gri81] is one of the most influential items of Computing literature. Gries, like Hoare, restricts his attention to "programming", but it is clear that he is using the term to denote the production of code from a formal specification. He does not, for example, introduce issues such as requirements capture or management of the development process into the discussion.

Gries is clearly aware of the confusion that may arise from his use of the term "science", and so takes the trouble to explain which of the Oxford Dictionary definitions he had in mind when adopting the term:

> "Sometimes, however, the term *science* is extended to denote a department of practical work which depends on the knowledge and conscious application of principles; an art, on the other hand, being understood to require merely knowledge of traditional rules and skill acquired by habit."

He goes on to say that, although programming started as an art, the science is just emerging. It is unfortunate that science and art should be seen as mutually exclusive in this way, and even more unfortunate that a value judgement is implied that places science above art. There is no evidence in the book that Gries agrees with such a value judgement, but neither does he show us that he disagrees. A discussion of programming as an art is contained in Knuth's Turing Award Lecture [Knu74].

Gries does not make explicit the fact that the book is addressing topics on two distinct levels:

1. The activities of programmers, and the principles that should guide their actions.
2. The programming languages that programmers use and theories that govern their use.

On the first level, Gries is trying to establish a number of principles that govern the action of writing good programs. In a sense this is truly an attempt at a science of *programming* rather than a science of programs. Adopting Popper's argument, however, we can see that this is not a science in our strict sense of the term (as Gries clearly realised when he took the trouble to provide his working definition of science). Gries's principles are not refutable laws, but guidelines for action, albeit guidelines that have arisen from theoretical considerations. In fact, we can view these guidelines as a very simplified reflection of Gries philosophy of programming. If we take this liberty, then we can compare this philosophy with Popper's comments on methods, and see if any similarities or conflicts can be found. This, of course, presupposes that we are prepared to accept some correspondence between programs and scientific theories. For the moment, we ask the reader to take this on trust, or, at least, to suspend disbelief.

We will only consider some of the more general principles raised in the book, for these are sufficient for the task in hand. Many of the other principles are specific to a particular programming paradigm and therefore sensibly comparison with theory constructs would only be possible by the introduction of particular theory presentation paradigms.

> " • **Principle:** A program and its proof should be developed hand-in-hand, with the proof usually leading the way." (page 164)

To help us understand what he means here, Gries provides us with another definition, this time of a proof, from Webster's Third New International Dictionary

> " the cogency of evidence that compels belief by the mind of a truth or fact."

Thus Gries is admitting psychologism as a basis, a position that can easily be defended by noting that his principles are guides to action, and action is governed by second world entities such as belief or emotion. It is evident from the context of the rest of the book, however, that this "cogency of evidence" is to be provided through rigorous deductive reasoning, just as for Popper, and that all the assertions turn out to be inter-subjectively testable observation reports. This means that we can stop short of using the evidence for the purpose of inducing belief, and consider Gries' deductions as third world objects just like Popper's. The motivation for these deductions is clearly different, however, for Popper would have us deduce things that can be observed, and hence act as refutations of the theory, whereas Gries deduces results which he hopes are not refutable, and which he can take as "true". This comes about partly because Gries accepts certain base theories and initial conditions as effectively irrefutable.

The notion that a proof and a program (or theory) should be developed hand-in-hand is held by both Popper and Gries. For Gries, the program is made more precise by adding the details of chunks of code in a top-down fashion, whereas Popper would add bolder conjectures. The idea that the proof leads the way is harder to reconcile. This appears directly contrary to Popper's view that evidence in support of a theory should be sought only after a theory has been proposed, and will take the form of severe tests of the theory. We cannot know what tests will be needed until we have analysed the bold conjecture. Indeed, Gries himself seems to support this view with another principle:

> "• **Principle:** Use theory to provide insight; use common sense and intuition where it is suitable, but fall back on the formal theory for support when difficulty and complexities arise." (page 165)

This gives us the hint we require to reconcile the two views, for it is the proof of program version 1 that gives rise to program version 2; for Popper, refutation will require us to build new theories, and failure to refute should lead to construct bolder theories, whereas for Gries, attempting deductive proofs of programs will lead either to faults that must be corrected, or to lemmas that must be satisfied by pieces of code yet to be written.

For both Popper and Gries the notion of problem solving is important.

> "• **Principle:** Programming is a *goal-oriented* activity." (page 173)

This ties in neatly with Popper's simplified diagram

$$P_1 \rightarrow TT \rightarrow EE \rightarrow P_2$$

For Popper, problems give rise to tentative theories, from which we attempt to eliminate errors, and this in turn leads to new problems to solve. For Gries, problems give rise to outline programs, which leave gaps in the deductive chain, that require additional problems to be solved in generating the required code to fill them. Unfortunately, this model is so general that it can also be used to explain undesirable practice: take a problem, guess a solution, discover the bugs and try to fix all the problems.

The crucial differences between such hacking and the methods of Gries and Popper are

1. The tentative theories (programs) are developed with a view to refuting (proving) them, and not in an *ad hoc* way.

2. The step from TT to EE involves deductive reasoning, not seemingly random observation of behaviour. The hacker will observe the symptoms of any error and induce a patch to reduce the severity of the problem. The scientist will use logical tools to attempt to identify possible sources of the problem and propose changes to the theory (program) accordingly.

The deductive system available to the scientist is critical to this process. Both Popper and Gries assume a base line of a first order logic. For Popper, however, this is the only irrefutable base; everything else is held only tentatively, awaiting refutation. In practice, he does concede that the scientist will have some theories that are elevated, for particular purposes, to such a level that he or she will proceed as if they are "true", and call them into question only as a last resort. It is at this point that the second stage of Gries's book is called into play, for here he discusses in detail a suitable theory of program statements, in a particular language, that can be taken as part of this irrefutable deductive system. This effectively localises the domain of application of his approach to situations for which the deductive system has not been refuted.

Gries does not refer to his formal system, which is actually Dijkstra's calculus of weakest preconditions [Dij76], as a theory, but as a semantics. He is thus treating the theory in a conventionalist way, effectively saying that this theory *cannot* be refuted as it *defines* the programming language comprising the program statements in question. This, as we saw in Hoare's paper, does not remove the possibility of refutation; it just pushes the problem down to the individual trying to implement the compiler for the language, or to the person trying to select a compiler that can be shown to implement the language in question.

The choice of Dijkstra's weakest precondition semantics does raise one other important issue, namely that we usually think of this as a calculus rather than a deductive system. That is, we are trying to calculate the weakest precondition rather than trying to prove some theorem. This is actually a trivial distinction, for any theory that uses equality will contain theorems of the form $x = y$ which can be interpreted as either true propositions, or as rules for calculating the values of one term from another. This point counters the assertion that programs cannot be viewed as theories because they yield values not statements[Joh88]. This is ill-founded, as we can interpret the values that are yielded as pure values, or as statements with an implied left hand side ("$answer = x$"). Such discussion, however, must be carried on outside of the theory, and cannot be a property of the theory itself.

In conclusion, Gries's book serves to provide both a methodological and a theoretical basis for programming. It is perhaps unfortunate that this dual aspect is not made more explicit in the text. The book does show very clearly, however, that suitable deductive systems can be found for reasoning about programs (as third world objects, rather than psychological objects or running machines).

"Programming as Theory Building" by Naur

In this paper [Nau85] Naur, like Hoare, treats programming "in a wide sense ... to denote the whole activity of design and implementation of programmed solutions", but he makes more of the fact that it should be viewed "as a human activity". Unlike Hoare, however, Naur takes a holistic view of the activity, identifying only two major sub-tasks, namely the initial creation of a program and its subsequent modification. Naur's central tenet is that

> "programming properly should be regarded as an activity by which the programmers form or achieve a certain kind of insight, a theory, of the matters in hand. This suggestion is in contrast to what appears to be regarded as a production of a program and certain other texts." (page 253)

This statement both serves to bring his philosophy into contact with that of Popper, and also to create a significant gap. Clearly, the stated aim of program creation as the construction of a theory is very similar to Popper's stated aim of science. Unfortunately, the linking of theory with "insight", and Naur's subsequent refusal to decouple the two, leads us down the route of psychologism, which Popper was at great pains to avoid. Naur seems to encourage us down this route, and actually seems to be saying that theories *must be* psychological, and cannot be shared.

> "A main claim of the Theory Building View of programming is that an essential[5] part of any program, the theory of it, is something that could not conceivably be expressed, but is inextricably bound to human beings." (page 258)

As we shall see, this leads him into a paradox which he seems not to notice, or at least, fails to acknowledge.

The suggestion that there is more to programming than producing programs and related documents is a complex one. It is unclear whether Naur is making a judgement of the form "there are more important things that come out of programming than programs" or a methodological observation that "in order to achieve the aim of writing a program, there are several tasks that need to be carried out in addition to that of writing down the code" or any one of many other possible interpretations. We will interpret the statement as meaning that the traditional models used for discussing programming are not sufficiently rich to cover the real issues, as they concentrate too much on the process of code production and other documentation. In particular, they do not pay sufficient attention to the people involved in the process. This is a recurring theme in modern philosophies of science, where it is often stated that we cannot understand the history of science simply from the scientific literature, as this ignores the contexts, or problem situations, within which discoveries are made. Consequently any philosophy that seeks to rationalise science only as reflected in the scientific literature will not be a philosophy of the actions of real scientists.

Naur makes no claim that the theories he is discussing are in any way scientific, but his paper does amount to a claim that science is not possible. His notion of "theory" is so general that it must include scientific theories. It appears, for example, as if Naur

[5]We will assumed here that Naur is not adopting an essentialist view of theories, but is using the term "essential" in its modern conventional sense of necessary. To discuss this further would be pointless, as there are no further hints in the paper as to the validity of this interpretation.

has ignored the fact that one of the aims of modern science has been to find ways of expressing theories so that they are as independent as possible from the context in which they arose. In Popper's terms, we seek to move the theories from the Second World to the Third, thus making them part of science rather than part of an individual's thoughts. Scientists are positively discouraged from including in their scientific writings details of the thought processes that led to their hypotheses, rather they are trained to present a coherent, and logical, reconstruction of their activities utilising only the presented facts and established theories. As Medawar has noted

> "The conception underlying this style of scientific writing is that scientific discovery is an inductive process." [Med64, page 8]

and this leads scientists to ignore the actual sources of their ideas, especially if they cannot rationally reconstruct them, and to proceed as if the ideas arose only out of the scientific data under consideration. We must not infer from this, however, that a logical discussion of the theory, and its truth, should depend on the sources of the theory. It is central to Popper's philosophy that we must decouple the theory from its source, and this in turn means that we must find a way of expressing theories so that they can be understood without understanding the workings of the brain. Consideration of the processes involved in the conception of a theory is a psychological problem, but by treating the theory only as embedded in the Third World we can allow science to meet its objective ideals. In suggesting that such a separation is impossible Naur strikes at the very heart of modern science.

Naur raises the interesting question as to whether it is ever possible to complete a programming task, just as Popper states that we can never find *the* theory of something. If we express the requirements of a system in terms of some other external agent(s), then we must recognise that these agents are liable to change over time. Consequently, we must make it clear what it means to "meet the requirements". We must also be aware that by expressing the system in terms of these requirements, we are functionally binding the system to the external agent. That is, the requirements are now a function of the agent (at a given time), but also it is possible to interpret this binding in the other direction, and thus make the agent a function of the system. Many companies have discovered this to their cost, when they realise that a computerised set of procedures is quite capable of determining, usually by constraint, the changes possible for the company. An extreme case of this is the major impact of computerised systems on company mergers: if the computer systems are not compatible, then the mergers often cannot take place, regardless of the commercial desirability.

The implications of this for Software Engineering are obvious. We must consider the extent to which such changes are to be accommodated, how to achieve stability, how to cost and manage change, and how to provide legally binding contractual boundaries. This is considered further in subsequent chapters.

Naur's evidence in support of the theory building view of programming is largely anecdotal, which is not necessarily a bad thing for it is the way much of the philosophy of science itself is constructed and justified, but sadly he seems to fall into the trap of using these anecdotes to support just his preferred conclusions. The central tenet of his argument is that programmers who design a system are better able to modify it than are newcomers. This he attributes to the fact that the designers have the theory of the system, and that the newcomers will never be able to acquire it, in spite of the documentation. His evidence, however, could equally well support the view that the current state of system documentation is woefully inadequate for programmers who have to maintain systems. This is a conclusion that many will find much more acceptable, as typical documentation seems to address the issues of specification (the "what") and implementation (the "how") but rarely explication (the "why"), thus leaving the layers of documentation unlinked by any logical structure. If he had adopted this route, then his subsequent attack on design methods would also be more coherent, for these are predominantly attempts to transform the "what" documents into the "how" documents via a prescribed set of rules. He could also have mounted an attack on the life-cycle model for system development, which suffers from the same lack of logical coherence. We will take up this challenge in subsequent chapters.

A second problem with Naur's conclusions is that he seems to be assuming a unique theory for the program (he constantly refers to "the theory"), and thus overlooks the possibility that the newcomers might construct a better theory. This new theory, however, would not be a true reflection of why the programmers did what they did, but a post-rationalisation of events. There is no evidence that this would prevent the newcomers from modifying the program. Research into reverse engineering, for example, is evidence that some people believe quite the opposite. It may well be the case that a team of newcomers, by bringing new theories and perspectives, may identify a more radical and effective set of modifications.

This also gives rise to a major paradox within Naur's view, for he merrily talks of a team of programmers and their theory, thereby suggesting that several programmers can share a theory, but then wants us to accept that this theory can exist only inside their heads. He seems to want the original team of programmers to have a capability for theory transmission that the newcomers cannot have. He obviously wants theories to be held corporately, but is not prepared to allow for the corpus to grow after the theory has been constructed. In this case, how does the first theory get communicated? The first programmer to have a theory forms a corpus of one, and the other members of the team are newcomers: the team cannot bootstrap itself. This aspect of Naur's view seems untenable.

The problem seems to be that Naur has failed to distinguish between having a theory inside your head, and the existence of a suitable theory in the Third World. This is strange, because, although he uses Ryle's notion of theory[Ryl49], he refers to Popper's Third World explicitly when he observes that Ryle's notion of theory "appears as an example of what Popper calls unembodied World 3 objects and thus has a defensible philosophical standing. (page 255)

Perhaps this is an example of the misuse of philosophy, using it to provide answers and justifications instead of expecting it to raise questions. The reason that Popper introduced the Third World was precisely to overcome the problems that Naur is creating for himself. Theories in the Third World can be treated objectively and can be communicated. Newton's theory, for example, as a Second World object died with Newton, but as a Third World object it is alive and well. Naur could have developed a far more persuasive argument had he noted that programmers are reluctant to allow their theories into the Third World, where they can be objectively tested and possibly refuted, hence their reluctance to provide answers to the "why" questions in the documentation. This reluctance can be seen as a major barrier preventing Computing from becoming more scientific. It provides a very clear link between the papers of Hoare, Gries and Naur, and the philosophy of Popper, and seems to be at the heart of our question. Hoare is asking that we use theories explicitly to improve the engineering of software; Gries is making available some guidelines for the use of theory, and also contributing a solid body of theory for our use; Naur is observing not only that we must use theories, but that we must construct them; and Popper is providing a framework within which we can view theories themselves.

Naur and Popper agree that there can be no method, in the sense of a prescribed sequence of actions that we know in advance will lead to the discovery of a correct theory. This does not mean, however, that there cannot be guidelines and hints that may be useful in prompting the mental processes that are likely to lead to the discovery of new ideas. Naur uses this idea to criticise those methods of software development that are geared towards the production of specific documents, for these cannot be guaranteed to work, and may even inhibit the programmer who is forced to follow them from ever building a correct theory. This point is further developed in Chapter Six.

A key consequence of Naur's view is the need to elevate the status of the programmer. A programmer who is just following the rules, and creating the prescribed documents, is effectively de-skilled. Naur's view requires programmers to make all the decisions, thus they can no longer be

> "regarded as a component of ...production, a component that has to be controlled by rules of procedure and which can be replaced easily." (page 260)

Unfortunately, Naur extends this attack on methods to include an attack on the view of programmers as those who

> " ...formulate certain arguments in terms of rules of formal manipulation" (page 260)

Once again he seems to be missing the point, although perhaps in this case the criticism should be levelled equally at the formalists who have failed to explain their point of view adequately. The formal arguments are not used to deduce the theories, but to argue for their correctness. Popper is not seeking to replace inductive science by deductive, but to replace induction as justification for theories by deduction as an aid to refutation.

The main thrust of Naur's argument here, however, is that we must recognise the programmer as

> "a responsible developer and manager of the activity in which the computer is a part. In order to fill this position he or she must be given a permanent position, of a status equivalent to that of other professions, such as engineers or lawyers, whose active contributions as employers of enterprise rest on their intellectual proficiency." (page 261)

It is interesting that Naur implies this status is a gift to be given, whereas Hoare suggests it is a thing we should adopt for ourselves. This is not an important point of disagreement, for undoubtedly they both intend that the status should be both earned and awarded, but it does highlight the fact that not all of the changes that may be needed to improve current practice are necessarily within the control of the discipline itself.

Naur also notes the importance of educational change:

> "The raising of the status of programmers suggested by the Theory Building View will have to be supported by a corresponding reorientation of the programmer education. While skills such as the mastery of notations, data representations, and data processes, remain important, the primary emphasis would have to turn in the direction of furthering the understanding and talent for theory formation. To what extent this can be taught at all must remain an open question." (page 261)

In conclusion, in spite of its many shortcomings, Naur's paper raises the important notion that programming can be seen as theory building, rather than merely an activity that uses existing theories. Although Naur is completely at odds with Popper in his insistence that theories can only be psychological things, which seems to prohibit all of science as demarcated by Popper, he has raised the whole question as to whether programming is science rather than merely using science. This contribution must be acknowledged as very important to all that follows. In fact, Naur provides a quotation that could be seen, suitably generalised, as the very *raison d'être* for this research.

> "A more general background of the presentation is a conviction that it is important to have an appropriate understanding of what programming is. If our understanding is inappropriate we will misunderstand the difficulties that arise in the activity and our attempts to overcome them will give rise to conflicts and frustrations." (page 252)

2.3 Summary

In this chapter we have set the scene for the discussion that follows, by starting to view the activities of the Software Engineer within the framework provided by Popper's philosophy of science. This philosophy will be refined as we progress, but it provides a useful basis for the discussions to follow. We have also set the scene for the idea that Software Design is closely allied to the process of theory building, an idea that will be developed further in Chapter Four.

Chapter 3

Computing as Technology, Problem Solving and Discourse

"If one considers at once all the ramifications and ultimate consequences of each exploratory act, he will be overwhelmed and unable to formulate any new constructs. One who has directed graduate students in their research efforts will have frequently seen this kind of intellectual drowning take place"
George Kelly

In the previous chapter we started to explore the nature of software engineering design, but from a very limited perspective. The main purpose of this chapter is to broaden the discussion by considering the process of software design from three additional perspectives. The first of these relates to the aims of the activity. It is usually assumed that science is the quest for knowledge or truth, primarily for its own sake. There may well be occasions in Computing where such an aim is acceptable. More generally, however, the production of software systems is motivated by utilitarian values. Activities with utilitarian goals are more usually termed "design", "technology" or "engineering". Unfortunately, these terms are also frequently associated with the production of "real objects", and it is far from obvious that information or knowledge based systems meet the generally accepted criteria for such real objects. In section one we will discuss Software Engineering in the light of a brief review of the philosophies of technology, design and engineering to see if it can be considered a technological discipline.

The second and third perspectives we shall take seek to broaden the discussion by raising the human dimension. The groundwork for this has already been laid by accepting Popper's philosophy as a starting point, for, as Agassi observed:

> "...when Karl Popper's *Logic der Forschung* of 1934, or its expanded translation, *The Logic of Scientific Discovery* of 1959, caught readers' attention, they found hardest to comprehend his coupling—his explicit and quite systematic coupling—of attitudes and theories. He demarcated scientific theories, not attitudes, yet he added a condition relating to attitudes. Theories are scientific, he said, if and only if they are empirically refutable; this however, is conditioned on our willingness to subject theories to empirical tests and

our readiness to jettison them once they are empirically refuted. It is a fact within living memory that people found this very puzzling." [Aga86, page 39]

The decision to expand the discussion to include the practitioner as well as the practice brings with it a number of additional questions. The most fundamental of these is our perception of society itself. A number of views have been expounded regarding the nature of society, ranging from inductivist individualism, which takes society to be just an aggregate of a number of individuals, to organicism, or holism, which takes each individual to be merely a component in society. This question is discussed further in Chapter Six, where the issue materially affects the question of methods. The two questions we will discuss in this chapter, however, are, how do individuals solve problems, and what language processes are involved in the arriving at these solutions.

The first question gives rise to a section on problem solving. This subject can be addressed from a number of perspectives giving rise to several fairly disjoint bodies of literature. Research has been carried out on the psychology of problem solving, concentrating on the production of models of the mental processes involved and of the structures built up during the process. There has also been research carried out on improving the practice of problem solving, both by individuals and groups. This work has produced a number of classifications and heuristics for problem solvers. Another body of work is that of the A.I. community, who have sought ways of automating the problem solving process. Of particular interest to us is the approach that attempts to discover and emulate human methods. In our discussions of problem solving, ideas drawn from the literature of these three perspectives will be introduced, but no attempt will be made to forge them into a unified theory.

The final section in this chapter reviews the issue of using language to solve problems. The approach used will be similar to that adopted for Chapter Two. A particular theory of discourse has been selected (that of Kinneavy [Kin84]) and an attempt will be made to discuss software development within the framework this provides. This process is potentially very complex, as there are many disparate types of discourse that take place during the development of a typical software system. These might include, for example, the persuasive discourse of sales staff, the social interactions necessary for a group of individuals to develop into a project team, as well as all the technical discussions that need to take place and the documents that need to be written. We will concentrate solely on the technical aspects of the development process, as we are concerned primarily with the technical education of software engineers.

3.1 Computing as Technology, Design or Engineering.

In setting out to ask "In what sense is Computing a technological pursuit?", we find ourselves in a very different situation to that of the previous chapter. When considering Science, the problem was a vast literature, containing many coherent philosophies, and our solution was to select one particular perspective upon which to centre discussions. In considering technology, however, no such literature exists; as Rapp says, the Philosophy of Technology "is still more of a desideratum than a concrete reallity"[Rap81, page xii]. Feibleman has noted this fact too, when he writes

> "Throughout history, technology has played a crucial part in human culture but there is no record that philosophers took any account of it. They give a lot of attention these days to the *existence* of technology but very little to the *products* of technology, but in no major writings of the classical philosophers has the existence of technology ever been mentioned. Nowhere, to my knowledge, at least, has there been a philosophy which took the full measure of human practices by assuming that whatever men did about the making and using of artifacts was filled with meanings which needed to be interpreted in philosophical terms." [Fei82, page 1]

We can only speculate on the reason for this lack of literature. Two of the reasons commonly put forward are significant to our discussion. Jarvie observes that

> "Technology seems to have been treated like Cinderella by philosophers of science. It has always been put in the second-best place, mentioned almost as an afterthought. This is perhaps understandable, since the received notion of the rôle of technology is that it is the province of engineers and other such non-gentlemen, and its philosophy thus is not a matter of great concern to the philosophical purist." [Jar74, pages 86–87].

Bunge offers an alternative, but equally relevant, view:

> These problems have been neglected by most philosophers, probably because the peculiarities of modern technology, and particularly the differences between it and pure science, are realized infrequently and cannot be realized as long as technologies are mistaken for crafts and regarded as theory-free." [Bun74, page 19]

The idea that we place values on disciplines in proportion to the degree of abstract contemplation we perceive in them has also been noted by Dehnert:

> "The degree of conceptualization required by an activity or field of study became the mark of its dignity as an art or a science." [Deh86, page 109]

To compound the problem, creativity in the arts has also largely been ignored by philosophers, who leave the matter to psychologists, preferring to concentrate on aesthetics. Thus design is impoverished by being ignored on two accounts: it is too concrete to be respectable, and it is carried out by people. This lack of philosophical study leaves it bereft of foundations upon which to flourish as an academic pursuit.

It is true that there have been philosophers who have adopted particular stances to wider issues that have significant bearing on acceptable views of technology. Aristotle, for example, adopted the position that the pursuit of knowledge for its own sake is superior to actions undertaken in the pursuit of wordly life, for the latter is merely a means towards the end of understanding the God-given cosmos by contemplation.

> "For contemplation is at once the highest form of activity (since the intellect is the highest thing in us, and the objects with which the intellect deals are the highest things that can be known)." [Ars34, Ethics X, 6, 1177, a]

We must remember that Aristotle was talking within a very different culture from that of the modern reader. Randall, a modern follower of Aristotle, claims that Aristotle would not "elevate knowing above practical action" were he writing within a modern American praxis-oriented culture [Ran60, page 248].

Aristotle's view is only one among many, of course. Karl Marx inverts this dominance of knowledge by taking the material cause as primary [Mar59]. Martin Heidegger, on the other hand, takes a radically different approach. He asserts that we must overcome this purely instrumental concept of technology if we are ever to understand it. We must come to terms with the ways in which man ontologically addresses entities.

> "...if the 'world' itself is something constitutive for Dasein, one must have an insight into Dasein's basic structures in order to treat the world-phenomenon conceptually." [Hei62, page 77]

For Heidegger we are all technicians, for creating things is the perfect expression of our rationality [Hei59, pages 16-17]. The fact that technology is such an important, and complex, part of human existence, but has largely been ignored by philosophers, leads to what Ströker calls "the paradox of its continual beginning" [Str83, page 323]. All the seminal writing on the Philosophy of Technology takes the form of sketches or initial attempts at an overview of the subject, and no-one seems prepared to build on these sketches by analytical discussion aimed at clearly defining a topic, critically assessing points of view, or seeking some global coherence. Such new writing as does take place comprises additional sketches, and hence the beginning continues. Ströker suggests that technology might present particular problems unparalleled in other subjects so far submitted to philosophical analysis, and that "a certain skepticism is aroused ...as to whether philosophical analysis in the same way as has been tried can succeed in arriving at larger contexts that exceed mere beginnings" [Str83, page 333]. This should not be taken to mean that a better philosophical systemisation is not possible, rather that the means for finding it have yet to be discovered. This endorses the view that no obvious research method exists for this programme.

It is impossible in this thesis to attempt a fundamental treatment of the Philosophy of Technology, so we must be content with a fairly superficial treatment of particular issues of relevance (superficial because any depth of treatment would require suitable foundations upon which to build). Furthermore, we must be content with the "beginnings", and accept the absence of an analytical framework within which these issues can be discussed.

Many of these issues arise under the headings of "Engineering" or "Design" as well as "Technology", but in the absence of philosophies of any of these disciplines, if they can be so called, no attempt will be made to distinguish between them. In addition, the term "applied science" is often used as a bridge between science and technology, and this term will also be taken to mean technology in the modern sense.

Science and Technology, the demarcation

One of the more widely discussed issues in the literature is the relationship, or demarcation, between science and technology[1]. Most of this can be considered "appallingly superficial" [Gas72, page 297], and seems to concentrate on idealised and outdated perceptions of both science and technology.

One of the most common dimensions of demarcation is that based on the product of the activity. The product of science is taken to be an increase in knowledge, and the products of technology are artifacts. Popper's philosophy is founded upon such an assumption: "application and predictions interest [the scientist] only for theoretical reasons–because they may be used as *tests* of theories" [Pop59, page 59]. This assumption has been sharply criticised by Putnam, amongst others [Put74, page 222], for modern science is actually concerned with the production of complex models to explain how things are. Technology is concerned with models of how things might be brought about. The distinction between that which "is" and that which "might be" is a subtle one, and is often captured in a line of demarcation that has science concerned with the natural world, and technology with the artificial [Sko72]. We must accept, however, that modern science does not just observe the natural world without intervention, but seeks to create situations in an experimental context that man has already brought about. Even observation outside of a laboratory is often concerned with issues like the environment, where it is now accepted that man has had a significant impact [Rap74, page 94].

For Computing, the situation is even less clear cut, for we become forced to consider very carefully what is "natural" about our domain of interest. Are numbers, for example, natural or artificial entities? It is here that Popper offers us assistance, by allowing such entities to have an independent third world existence. We must also remember that many of the systems we produce are actually embodiments of predictive theories, and there seems no essential difference between the environmental scientist's theory of depletion in the ozone layer and the software engineer's theory of stock control within a company.

It is often argued that it is the aims of science and technology that allow them to be distinguished rather than the products. Science aims at increasing knowledge for its own sake, technology at improving man's existence by creating useful artifacts. Bunge, for example, states that

[1]One of the side-effects of this research is the light thrown upon this demarcation by considering the location of Software Engineering. As a "marginal object", it causes us to reflect carefully on the demarcation as much as the object [Tur84, page 31].

"If the goal is is purely cognitive, pure science is obtained: if primarily practical, applied science." [Bun74, page 19]

Thus the process of technology is sometimes considered to be that of demand-pull, where the perceived needs of society or individuals, in terms of the artifacts they desire, motivates the activity. This distinction is rather naive in modern capitalist societies. It might also be argued that man's existence is not just his *being* but his *well-being*, and that the quest for knowledge is an essential part of this well-being: man only being content when he is continually increasing his understanding of the world in which he exists [Gas72, page 293]. In this case, the quest for knowledge is simply a means to the end of well-being. Even if we deny this line of argument, we must accept that modern science is rarely carried out for the pursuit of knowledge *per se*. Feyerabend, for example, suggests that

" Late 20th.-century science has given up all philosophical pretensions and has become a powerful *business* that shapes the mentality of its practitioners. Good payment, good standing with the boss and the colleagues in their 'unit' are the chief aims of these human ants who excel in the solution of tiny problems but who cannot make sense of anything transcending their domain of competence." [Fey75, page 188]

Individual scientists may derive their job satisfaction through believing they are contributing to the body of knowledge, but this should not be confused with the aims of science as a corporate endeavour within modern western society. In our society, it seems that science and technology both share a common aim, namely to meet the requirements of those funding the enterprise [Rop83, page 91]. If we assert that the nature of these requirements is fundamental to the demarcation between science and technology, then we must accept that identical actions carried out by identical individuals may be deemed science, technology or neither depending on whether the requirements of the sponsors are for improved knowledge, improved well-being of society, or an increase in wealth for the sponsor. Such a position seems unhelpful, and by adopting it we would effectively be depriving both "science" and "technology" of meaning in the modern world.

Following this line of demarcation, we can consider the codes of practice, explicit or implicit, drawn up for engineers and scientists. Typically, the scientist must value truth and honesty above all else [Sno34]. The engineer, on the other hand, "has to fulfill his professional work in the service of mankind ...to work with respect for the dignity of human life ...may not give way to those who do not fully respect the rights of a human being and who abuse the true essence of technology; he must be a faithful collaborator of human morality and culture". These were the standards set down for practicing engineers by the VDI (the German equivalent of the Engineering Council) in May 1950 [Hun83, page 51]. Such an idealised view of the engineer, desirable though it may be, does not seem an accurate reflection of the typical practitioner of Software Engineering. Dieter has noted that "it is sometimes said ...that an engineer is a person who can do for $1 what any fool can do for $2." [Die83, page 286], and the proverb "rubbish that sells isn't rubbish at all", often accredited to British business [Rop83, page 94], may well be seen as more appropriate for summing up the value systems within which the modern engineer has to work. Indeed, such value systems may even better reflect the environment of many industrial scientists.

The use to which theories are put in science and technology is another commonly adopted method of demarcation. It is often said that a technologist accepts a theory if it is useful, rather than if it is true. The scientist, on the other hand, accepts only theories that are true. Following Popper, such a position is untenable. Both the scientist and the technologist accept that a theory is neither true nor false, but rather refuted or currently unrefuted. It is the case that a technologist is generally concerned with a much more restricted domain of application than the scientist, who seeks generalisations where possible. Even the scientist, however, will continue to use the principles of Newtonian mechanics in situations where he is confident that observations cannot be made to refute them. This is often discussed using the idea of an approximate theory, but such a concept seems unhelpful in the absence of genuine metrics for the degree of applicability of a theory.

Another possible line of demarcation is the idea that science precedes technology, by providing the theoretical foundations upon which technological decisions rest, and that the production of artifacts is the result of supply-push. Martin Heidegger discusses this point, and observes that, although we may consider science *historically* prior to technology, we should accept that *ontologicaly* it follows from it [Hei77]. Ihde suggests that even the historical precedence of science is perceived rather than actual [Ihd83, page 240]. The widespread acceptance of the idea that science feeds technology, rather than the converse, is illustrated by Ryle, who states without question:

> "Engineering does not advance physics, chemistry or economics; but competence at engineering is not compatible with complete innocence of these branches of theory." [Ryl49, page 298]

In modern society the historical precedence of science is starting to wane. In 1974, Rapp wrote

> "It can be taken as a rule of thumb that sooner or later scientific findings are applied in technological practice in some way or another. As a result of the powerful pressure of competition in the economic and armament spheres the time lapse in this process between discovery of new findings and technological utilization of them is being constantly reduced." [Rap74, page 98]

Since 1973 this time lapse has reduced still further, and in many cases appears to have become negative, in the sense that technology is waiting for specific scientific results, with the application already decided upon. One side-effect of this is that the distinction between science and technology is becoming increasingly blurred. In many cases, the technologist is breaking new ground whilst solving practical problems, and thus playing the rôle of scientist by advancing the theoretical underpinnings of the discipline. This relationship between science and technology has also led to the modern technological imperative. In 1975, Teller, the so-called "father of the atomic bomb" told a journalist from "Bild der Wissenschafts" that the technologist "ought to apply everything he has understood". This creates a new moral dictum to accompany Kant's "ought to implies can", namely that "can implies ought to". Accepting both of these maxims, of course, leads to a spiral of investigation and application independent of any external axiology.

The situation is even more confused for software engineering, for it can be argued that our products have many of the characteristics of theories themselves. Moreover, our application domains are often bereft of theories to use at the outset, so the software engineer has to construct theories of what the system is to be, as well as using theories to bring this about. Civil engineers, on the other hand, do not usually need to worry to the same extent about constructing theories of the artifacts, for these are generally well-understood. They can concentrate on using the theories of components to bring about the construction. On occasion the engineers' understanding of the artifact will let them down, and the enterprise will fail. A classic example of this is the disaster of the Tacoma Narrows Bridge, where the engineers failed to understand the artifact as an aircraft wing as well as a bridge. This oversight meant that they did not foresee the effects of a high wind on the structure [Pet82, pages 164-165].

Throughout this thesis it will be assumed that the line of demarcation is very thin and fragile. Rather than attempt to delineate "science" and "technology", or "scientists" and "technologists", we will distinguish only "scientific attitudes" and "technological attitudes" to theories. The former we will take to be Popper's ideal of the attitude underpinning scientific practice, the willingness to seek refutation of theories and to propose theories with the expectation that they will be refuted. The latter we will consider to be an instrumental acceptance of theories, where refutation will be taken primarily as indicating that the theory has been inappropriately applied, although the process may well also indicate flaws in the theory itself. This very simplistic demarcation has one very important impact, however, for we must allow that "scientists" sometimes behave with technological attitudes, such as when they are designing instrumentation and experiments, and also that "technologists" sometimes behave with scientific attitudes. In particular, we will establish in the next chapter that the software engineer must adopt a scientific attitude during some stages of a typical project, and adopt a technological attitude during other stages. The link between these two stages will be provided by the theories themselves.

The Development of Modern Technology

In addition to considering the demarcation between technology and science, we can also turn to the *development* of technology for insight. Various authors have noted that technology can be considered to have progressed through a number of phases. There is, however, some confusion as to whether these phases represent the development of society as a whole, the development of individual technologies, or the development of individual technologists. If we adopt the first view, then we might expect all technologies within our culture to be in the same stage of development, whereas the second view allows for disparity between different technologies at a given time. The third view would allow individual practitioners to evolve through the various phases.

The first phase considered is usually termed the primitive phase, and is summed up by Gasset

> "How does primitive man perceive technology? The answer is easy. He is not aware of it as such; he is unconscious of the fact that among his faculties there is one which enables him to refashion nature according to his desires." [Gas72, page 307]

Some would consider that unperceived actions cannot be called technology, and so do not consider this phase at all. One of the characteristics of this phase is the limitation of technological progress inherent in the society. Sprague de Camp, in his work on Ancient Engineers, observes that

> "Primitive peoples live a hand to mouth existence ... Therefore they can less well afford to risk experiment than more advanced people...
>
> As a result, primitive societies are very conservative. Tribal customs prescribe exactly how everything shall be done, on pain of the god's displeasure. An inventor is likely to be liquidated as a dangerous deviationist." [dC77, page 6]

One possible explanation of the so-called software crisis is that the software industry is behaving like a primitive society, living a hand to mouth (or project to project) existence not through necessity but because of a desire to minimise overheads and maximise profits. At the same time, however, it is sophisticated enough to recognise the symptoms of the problem. The "solution" to the paradox this causes is to rationalise the behaviour, encapsulating the "customs" into methods and imposing these methods on the designers. In this way the engineer *cannot* accept responsibility for the design. This provides a refuge from responsibility, but locks the engineer up in it like a prisoner. This is tragic enough, but unfortunately many academic institutions have adopted these methods as cornerstones of the curriculum, thus institutionalising the students and preparing them for imprisonment.

Alongside such treatment of primitive societies, some authors consider magic as being "technology in seminal form" [Rap81, page 71], for "both activities ... attempt to achieve a given goal as reliably and simply as possible" [Ell72, page 24]. Although there are undoubtedly many interesting ideas raised by this line of enquiry, it will not be pursued further here, as it is not of direct relevance to the task in hand.

The next phase of technology is usually considered as craftsmanship. Gasset, for example, summarises this as follows:

"...in craftsmanship there is no room whatsoever for a sense of invention. The artisan must learn thoroughly in long apprenticeships—it is the time of masters and apprentices—elaborate usages handed down by long tradition. He is governed by the norm that man must bow to tradition as such." [Gas72, page 309]

Feibleman has observed that in this phase of technology there is a clear distinction between craft and science,

"In the Middle Ages, there was natural philosophy and craftsmanship. Such science as existed was in the hands of the natural philosophers, and such technology as existed was in the hands of the craftsmen." [Fei72, page 38].

This historical division is considered by some to continue:

"Many methodologists and philosophers of science insist that technology is in principle a composition of various crafts. Regardless of how sophisticated these crafts have become, they are still crafts." [Sko72, page 43]

The third phase can be considered as modern technology:

"Today the engineer embraces as one of the most normal and firmly established forms of activity the occupation of inventor. In contrast to the savage, he knows before he begins to invent that he is capable of doing so, which means that he has 'technology' before he has 'a technology' ". [Gas72, page 311]

This modern phase differs from that of the craftsman in the extent to which scientific principles are used.

"...only tentatively and rarely did technology, with its roots deep in the craftsmanship of the earliest recorded times, or perhaps in the even earlier known techniques of palaeolithic hunters and artists, reach across barriers of social class occupations to join with science, and then only in periods of particular social requirement." [Coh83, page 35].

Millendorfer adds extra resolution to the notion of modern technology, by subdividing it into three phases. First came the industrial revolution, concentrating on the material problems of man. Second came the need to solve the resulting information bottleneck. Currently we are faced with the third phase, the need to resolve the ethical questions that modern technology is posing. [Mil76, pages 408-413].

Before considering the features of modern technology in more detail, however, we will approach the question from a slightly different angle.

Unselfconscious and Selfconscious Design

One of the most significant texts written on the subject of design is *Notes on the Synthesis of Form* by Alexander [Ale64], to which page numbers in this section refer. Although this was originally written about design activities in architecture and town planning, many of the ideas are transferable to software design.

A central theme of the book is that design starts out being an unselfconscious activity, typically in more primitive societies, where it is carried out by following rules of thumb, or religious dogma. Many of these rules are distillations of good practice arrived at over centuries of repetition. If something goes wrong with the design, possibly because the rule is not quite applicable, then there may well be specific actions laid down to remedy the fault. If there are no appropriate rules laid down the designer can take immediate action, possibly even random action, observe the effect, and reverse the action if necessary. The designer in this situation does not explicitly consider his actions, and never gets a chance to compare them with those of others: actions are governed by habit or conditioned by response.

This class of design, Alexander maintains, works for certain types of problems, and is stable even when the problems change. One of the reasons for this is that the designer experiences failure first-hand, and so can react accordingly and immediately. The designer is not faced with a large number of symptoms at once, but can treat each one as it arises. Such a system could well be unstable, but the in-built traditional methods provide a damping effect, making the designer reluctant to change rules that have stood the test of time. As Alexander says,

> "Rigid tradition and immediate action may seem contradictory. But it is the very contrast between these two which makes the process self-adjusting." (page 52)

Furthermore, as the designs have evolved over a long period of time, they generally fit well with the problem because the structure of the solution has evolved alongside the recognition of the problem. Consequently, the subsystems of the solution are likely to map directly on to distinct areas of the problem, and so a number of recognisable subsystems can be isolated to work on. Thus remedies to cure a fault in one subsystem are unlikely to impact on other subsystems.

With selfconscious design, however, the situation is different. Now the designer recognises that the solution is his personal concern, rather than part of tradition. This brings with it the desire to stamp the individual's personality onto the design, but also a feeling of inadequacy, because complex problems are too difficult to get to grips with *ab initio*. To handle the complexity, the designer starts to impose conceptual order onto the problem. This order can now be communicated and taught, so we have the beginnings of an abstract discipline of design, rather than the passing on of craft techniques. Unfortunately, there is no evidence to suggest that the chosen order will map well onto the designed system, and so the selfconscious designer has sacrificed the ability to react to problems by changes to well isolated subsystems. Furthermore, now that design can be discussed, it can be carried out by groups of individuals on behalf of others. Thus the designer also loses the immediacy of feedback.

Alexander identifies a critical transition period, when a designer is changing from unselfconscious design to that of selfconscious design, for here several problems arise. In particular, the emergence is accompanied by a loss of innocence. The unselfconscious designer cannot be guilty of failures, other than by choosing to ignore the rules. The selfconscious designer, however, must make all the decisions, he has lost his claim to innocence. If the design does not work then it is apparent who is to blame. At the same time, however, the designer is trying to handle complex problems (or the move to selfconscious design would probably not have occurred). Consequently the designer feels exposed and at risk. Alexander notes two responses to this situation.

The first is for the designer to seek refuge in pretensions to genius. By claiming that the ability to design is inbuilt, he can refuse to discuss the processes of design, claiming that it just happens by inspiration. If the inspiration fails to arrive, it is clearly not his fault. Indeed, he may go further, claiming that any attempt to analyse the design process may destroy it by rendering it unavailable to inspiration. Alexander notes that

> "Enormous resistance to the idea of systematic processes of design is coming from people who recognise correctly the importance of intuition, but then make a fetish of it which excludes the possibility of asking reasonable questions." (page 9)

The second response is to seek refuge in the safety of established styles, thus returning to the way of following tradition, and alleviating the burden of decision.

It is interesting to consider Computing in this light. First, we can observe that a great deal of unselfconscious design takes place, particularly amongst those who program computers as a hobby. They do not, in general, reflect on what they are doing, but attempt to create programs to solve problems which they experience first hand. Frequently these programs are developments of existing systems, such as better games, and the starting point for the development is experience of these. The designer and user are usually one and the same person, so that the designer can try out the program, react immediately to any faults, and act accordingly. If an action does not work, it can be undone, or left *in situ* and programmed around. Some extremely complex programs can be developed in this way, and to the designer these will be "good" programs because they fulfill the perceived need. Often, of course, the need is actually forged alongside the development of the solution. That such programmers are considered inspirational, and almost divine, can be seen by the comments of many parents whose children can program in this way. They see the actions as inspirational genius, coming from within the child. The child is usually unwilling, or unable, to discuss exactly *how* the system was developed, often resorting to comments like "I just did it". This attitude is of primary concern to anyone charged with the task of educating software engineers in higher education, for it is likely to be the dominant attitude among those who have any computing experience entering higher education from schools.

The second reaction, seeking refuge in styles, can also be seen amongst computing professionals. In particular, it could be considered that the current fascination with proprietary design methods is an attempt to find escapes from the loss of innocence. If a software engineer is using IEW, for example, and the design is a failure, then it can be claimed that the method is at fault, not the designer. Again this phenomenon can be observed amongst students. Once they have been shocked out of their pretensions to genius, they often run immediately towards methods, wanting to know exactly what sequence of steps a particular method lays down for solving their current problem. One of the sad facts about this response is that the existence of a style or method as a refuge from the loss of innocence may well disguise its potential advantages. So called "formal methods" and "object oriented design", for example, are approaches to design that bring many benefits. If treated as styles for refuge, however, they are belittled and devalued. Many of the criticisms of such approaches demonstrate that they are being assessed as refuges, not tools under the control of the designer.

Another problem of selfconscious design arises because the designer, faced with complex problems, will impose form onto the problem according to historical accident. He is likely to see previous problems and solutions wherever he looks. Even the language he uses for describing problems will contain bias.

> "Caught in a net of language of our own invention, we overestimate the language's impartiality. Each concept, at the time of its invention no more than a concise way of grasping many issues, quickly becomes a precept. We take the step from description to criterion too easily, so that what is at first a useful tool becomes a bigoted preoccupation." (pages 69,70)

Alexander's solution to this is to suggest that we can abstract away from these preconceptions by using mathematics. Producing formal models of the concepts we hold, he claims, will remove their bias by retaining only their abstract structural features. This is an interesting claim, but one that is difficult to support. Alexander is in danger of comparing two dissimilar things. The question he should address is whether a description is less likely to be biased if it is expressed formally than when it uses familiar terms in English. We will return to this question when discussing theories in Chapter Five. He develops this idea by identifying two properties that his abstract models (which he calls diagrams) may have. They

> "...may have either or both of two distinct qualities, not always equally emphasised. On the one hand they may summarize aspects of physical structure ...One the other hand, the diagram may be intended to summarize a set of functional properties or constraints. ...This kind of diagram is principally a notation for the problem, rather than for the form. We shall call such a diagram a requirements diagram. ...

We shall call a diagram constructive if and only if it is both at once—if and only if it is a requirements diagram and a form diagram at the same time. " (pages 86-87)

He goes on to say

"It is the aim of science to give such a unified description for every object and phenomenon we know." (page 90)

The quest for such constructive diagrams seems to be at the heart of most structured design techniques currently in vogue. Unfortunately, these techniques do not seem to have taken on board the need for the diagrams to be sufficiently abstract, rather they attempt to be directly related to the concepts currently held by users, designers and managers alike. For Alexander, such techniques are unlikely to produce good fits, for although they may well result in constructive diagrams, these diagrams are likely to reflect the preconceived notions of the individuals involved rather than the true form of the problem. These ideas are developed further in Chapters Five and Six.

At the heart of the design process for Alexander is the idea that we are trying to produce a form that will fit perfectly with its context into a stress-free ensemble. He notes, in Popperian style, that this really means we should observe not correctness of fit, but the absence of errors, and further that there is a danger that we will concentrate on those areas for which suitable metrics exist for measuring errors.

Finally, Alexander offers a contribution to the demarcation discussion, when he says

"Scientists try to identify the components of existing structures. Designers try to shape the components of new structures. The search for the right components, and the right way to build up the form from these components is the greatest challenge faced by the designer." (page 130)

The Logic and Method of Technology

In this section we will explore the existence of a "logic" and "method" of technology analogous to that discussed by philosophers of science. It must be remembered that claims for *a* logic and *a* method of science are largely refuted now, in favour of the view that many models exist, all of which contribute to our understanding of the scientific process.

Logics and methods for technology can be discussed on at least two levels, the micro-level, concentrating on how individual decisions and actions are grounded but minimising the environment within which they take place, and the macro-level, concentrating on the organisation of projects, including their relationship with their environment, but ignoring the details of individual decisions. Discussion of the macro-level will be delayed until Chapter Six, where we can develop the subject in the light of the reviews of problem solving and discourse theory contained in the next two sections, and also centre discussion around a proposed model for the development process presented in the next chapter. We will restrict attention to the micro-level, and consider the ways in which technological actions, captured in the form of rules, are grounded in the laws we derive from scientific theories.

Bunge identifies four kinds of rules that govern technological action; rules of conduct (such as social behaviour), rules of thumb, rules of sign (such as the grammar rules governing language and the symbolic systems of mathematics) and rules of science. He asserts that

> "A rule [of science] is grounded if and only if it is based on a set of law formulas capable of accounting for its effectiveness" [Bun74, page 68].

He further asserts that the success of modern technology can be attributed to the replacement of rules of thumb by scientific rules and also to the trend to derive new grounded rules from scientific theories.

The fact that making "the step from theory to practice is to proceed from an understanding of the anticipated consequences of a particular procedure to a concrete directive for action" has been noted by Rapp [Rap81, page 59]. This step, however, involves admitting the technologist in a way that philosophers have long resisted when considering science. "Understanding" of the law is required by the technologist, and the resulting actions will be directed by this subjective understanding. In one sense this subjectivity is necessary, for the technologist must be involved in an action for it to be considered part of technology. We will refer to this as partiality [Bun74, page 72]. In another sense, however, we would like to remove individual subjectivity from the process, replacing it by inter-subjectively testable statements, for then we can provide the public with the confidence inspired by a professional community, rather than a collection of high priests. It is the reduction of this type of subjectivity that both Gries and Hoare are advocating in their publications, but which Naur seems to suggest we cannot achieve.

In the light of this discussion, is any logic of technology required other than the logic of science? The answer seems to be that the logic of science is sufficient for the laws grounding the theories, but if we require the ability to be analytical about the process of grounding itself and also the selection of rules, then we require a logic capable of encompassing both scientific laws and technological actions. Moore suggests that we also need to consider the *knowledge* of laws, for the existence of laws alone is not enough to explain actions. Furthermore, we need to be aware that knowledge is not static, and will change as we progress with any particular problem [Moo85]. The simpler logics of technology operate in a way analogous to the separation of programs and data in computer systems. The more complex logics allow programs (rules) to be altered as data [Alt89]. Since we are not seeking to be scientific about the processes of technology, only to allow technologists to be scientific in their actions, we will not consider this aspect of the problem further. We will restrict our attention to a very simple logic relating rules to the laws that ground them [Bun74].

We will centre this discussion around a scientific law of the form $A \implies B$. We will use Bunge's example to provide a particular interpretation, and call this law $L1$.

L1 If the temperature of a magnetized body exceeds its Curie point then it becomes demagnetized.

This law is part of a simplified theory, and ignores many aspects of magnetism, such as the existence of paramagnetism. We will assume that the decision to use this theory is justified by the nature of the problem being considered. This is a commonly used technological device:

> "The well established and logical action to which technology owes its efficiency is, as a rule, restricted to only the immediate *technological task*." [Rap81, page 135]

The law $L1$ provides the grounding for two practical rules

R1: In order to demagnetize a body, heat it to above its Curie point.

R2: To avoid demagnetizing a body do not heat it above its Curie point.

We can symbolise these as $\mathcal{B}$ *per* $\mathcal{A}$ and $\neg\mathcal{B}$ *per* $\neg\mathcal{A}$ respectively, where we read *per* as "by doing". The transitions from A to $\mathcal{A}$ and from B to $\mathcal{B}$ reflect the fact that A and B are assertions whereas $\mathcal{A}$ and $\mathcal{B}$ are actions. Bunge does not make this explicit in his notation, allowing context to disambiguate meaning.

Rules are not truth valued, but they may be viewed as effective or not effective[2]. This gives rise to a three valued logic, utilising values for effective, ineffective and unsure, because knowledge concerning the occurrence of $\mathcal{A}$ and $\mathcal{B}$ does not necessarily tell us anything about the effectiveness of the rule. In particular, if we do not carry out $\mathcal{A}$ then we can say nothing about the effectiveness of $R1$.

This raises an important point concerning the relationship between laws and rules. The effectiveness of a rule can tell us nothing about the "truth" of a law, although its ineffectiveness may indicate the falsity of the law. The observation that an occurrence of $\mathcal{A}$ is accompanied by an occurrence of $\mathcal{B}$, for example, does not tell us that $A \implies B$, for the law grounding this rule might be $A \wedge B$, $B \implies A$, $(A \wedge C) \implies B$ or any one of an infinite number of such constructions. This is simply a restatement of Popper's observation on refutation. It does have one significant implication for Software Engineering, however, which has never been explicitly noted. The starting point for many designs in Software Engineering are not laws but existing sets of procedures, which are to be

[2]Of course, they may trivially be encapsulated to form assertions which will be truth valued.

improved upon and automated. This means that the Software Engineer must make the transition

$$\{R_1, R_2, \ldots R_n\} \longrightarrow \{R'_1, R'_2, \ldots R'_m\}$$

not

$$\{L_1, L_2, \ldots L_n\} \longrightarrow \{R_1, R_2, \ldots R_m\}$$

as is often suggested (possibly erroneously) for traditional engineering. A major contention of this thesis is that the appropriate route for a professional engineer to take is, in simplified form,

$$\{R_1, R_2, \ldots R_n\} \longrightarrow \{L_1, L_2, \ldots L_m\} \longrightarrow \{R'_p, R'_p, \ldots R'_p\}$$

The first transition representing the scientific (theory construction) phase of the process, and the second transition the engineering (theory implementation) phase. Feibleman notes this point when he says

> "Technology is more apt to develop ... laws which are generalizations of practice rather than laws which are intuited and then applied to practice." [Fei72, page 36]

although he fails to discuss why generalisations can be distinguished from intuited laws: generalisations would seem to involve intuitions in precisely the same ways as scientific theory building. This idea forms the basis for the model of system design developed in the next chapter.

The existence of law $L1$ is not a sufficient condition for the technologist to be able to achieve goal $\mathcal{B}$, for several obstacles may present themselves. First, action $\mathcal{A}$ may not be achievable with current technology. Knowing "that" may lead to a knowledge of "how", but this should not be confused with the ability to carry the "how" out. This barrier may lead the technologist to consider problems with subgoals of $\mathcal{A}$, in order to find suitable antecedents of nomopragmatic statements, such as $\mathcal{A}$ *per* $\mathcal{C}$. This should involve the technologist in seeking corresponding laws on which to ground the rules, or possibly in commissioning a scientist to find these laws. Thus the technologist *uses* theories in the pursuit of law statements. Only in the simplest case will the transition above suffice; in general we might expect a sequence of grounded laws to be necessary to complete the transition.

$$\begin{aligned}\{R_1, R_2, \ldots R_n\} &\longrightarrow \{L_1, L_2, \ldots L_m\} \longrightarrow \{L'_1, L'_2, \ldots L'_m\} \\ &\longrightarrow \{L''_1, L''_2, \ldots L''_m\} \ldots \longrightarrow \{R'_p, R'_p, \ldots R'_p\}\end{aligned}$$

A second barrier that may arise is that laws are associated with theories, and theories explain only well-isolated subsystems of the real world. A theory may well stand the tests of refutation in a clinical laboratory, where experiments can be isolated from environmental interference, but utilising a law in a production environment requires that the environment constitutes an acceptable model of the theory. This is why the ineffectiveness of a rule does not necessarily refute a law, but may indicate that the scenario in which the law is being applied is not a satisfactory model of the theory being used. Many modern engineering problems are really concerned with finding ways of making a production environment model a theory adequately. In computing, however, we have a unusually high degree of control over our immediate environment. Hardware is susceptible to interference from physical activity, but software can usually be protected. Indeed, one of the properties usually associated with "good" software is the separation of modules into well-protected units. If Gries's laws of programming do not apply then we can usually trace the problem to errors in the implementation of another technological artifact, only rarely do problems arise due to to factors which are beyond our immediate control.

3.2 Problem Solving

Problem solving, like technology, is another area that is fundamental to many human activities, but has never been studied in a unified way. Historically there are two strands to the literature of problem solving. First there is the mathematical strand discussing how mathematicians solve problems; this is typified by the the works of Descartes, Poincaré and Polya. Second there is the psychology of problem solving, which is intimately linked to the study of thought itself. This has its beginnings in philosophy, such as Aristotle's three laws of learning and memory that gave rise to associationism, but began as an experimental science in the late nineteenth century with the work of psychologists such as James [Jam90] and Wundt [Wun73].

Recently there has been a confluence of these two strands, but this has been accompanied by the development of a number of applications leading to a divergence in the literature. The original philosophical and psychological intentions of increasing knowledge and understanding still persist, but in addition there have been developments in teaching problem solving (both domain independently and domain specifically), using the results of research to improve the heuristics on various problem domains, using problem solving as a structuring mechanism within philosophy itself, and developing computer systems capable of solving problems. Many of these issues will be dealt with in detail in later chapters, as they are directly relevant to the discussion of methods and improving pedagogical practice. In this section we will lay the foundations by introducing terminology and discussing some of the basic ideas common to all the approaches. In particular, we should note that the term "problem solving" should strictly be "problem attempting", for it refers not only to successful activity but to all activity with the goal of solution. We will restrict discussion here to individual problem solving, delaying discussion of group activities until Chapter Seven.

Any discussion of human problem solving must lead us into the realm of psychology, no matter how well disguised this excursion may be. In seeking to ask how people solve problems we are effectively asking how people think and how this thought process governs action. For the purposes of this thesis, we will adopt Clark's view that it is reasonable to partition the consideration of human thought into two parts, the "mind's eye view" and the "brain's eye view" [Cla89]. As Clark puts it, "The mind's eye view generates models based on our intuitive ideas about the kind of semantic object over which computational operations need to be defined.". The brain's eye view, on the other hand, seeks models of thought that are directly related to theories of the way the physical brain operates.

We will restrict attention to the mind's eye view, thus ignoring how people actually think, considering only models that arise out of the semantic properties of the world being considered. One side-effect of this decision is that we will not pay much attention to connectionist models of thought, which have largely arisen out of neurobiology. We will only consider the more traditional models based on an information processing view of thought. Although this restriction undoubtedly removes much of interest from the discussion, the state of the art in the philosophies of science and engineering, is still based largely on information processing models utilising von Neumann architectures, as are the current design methods in Software Engineering. It is certainly interesting to ask what a connectionist design method might look like, or whether a life cycle might be replaced by a life network, but reluctantly these questions have to be deferred until a later date.

The decision to adopt only a traditional mind's eye view does invite one major criticism, however, namely that we are embracing "folk psychology". We are, it might be said, accepting our intuitive understanding of people in terms of actions such as wanting, hoping, believing, and also accepting the concept of a state of mind. In many ways folk psychology is analogous to Hayes's Naive Physics [Hay78] [Hay85] that allows us to exist in the real world. In this case, however, we are able to rationalise people's observable behaviour in terms of convenient devices such as belief.

The adoption of folk psychology has been criticised on many occasions. Churchland [Chu81] and Stitch [Sti83], for example, raise the following objections. First, folk psychology fails to explain behaviour outside of certain norms. It is culturally specific and does not address problems such as madness, senility, or the behaviour of very young children. This objection need not concern us, for we are interested in a restricted domain of application, and our problem solvers comprise a fairly homogeneous professional body. We may well fail to address issues such as the behaviour of the psychotic software engineer who deliberately sets out to sabotage a project, but it was never our intent to cover such matters. The second objection is that folk psychology is a sterile theory. It does not lead on to bold conjectures and possible refutation. This is undoubtedly a significant criticism for psychologists, but we are using the theory as an instrumental structuring mechanism for a discussion. The third objection is that folk psychology fails to provide a theory that cuts the world at its natural joints, thus it does not integrate well with the physical sciences, and in particular with biology and the neural sciences.

Again, such integration is outside of our remit, so the criticism need not concern us here. Of more significance, however, is the observation that folk psychology does appear to cut Software Engineering at its "artificial joints". Concepts such as memory, information processing, and requirements (wants or obligations) are present in both the problems as presented and also in the solutions as implemented. This is not surprising since Software Engineering is an artifact that has arisen rapidly under the influence of humans who have accepted folk psychology unselfconsciously, and it has not yet reached the stage of maturity where such inherent assumptions have been widely criticised. The positioning of these artificial joints may not be accepted for ever, of course: questions such as how life cycle models relate to the design of neural networks, for example, may well cause radical rethinking of the way we partition the engineering process.

Clark [Cla89, pages 37-59] presents a comprehensive case for using folk psychology in circumstances where these objections can be overcome. He concludes that

> "Thus construed, folk psychology is *designed* to be insensitive to any differences in states in the head that do not issue in differences of quite coarse-grained behaviour. It papers over the differences between individuals and even over differences between species. It does so because its purpose is to provide a general framework in which gross patterns in the behaviour of many other well-adapted beings may be identified and exploited. The failure of folk psychology to fix on, say, neurophysiologically well-defined states of human beings is thus a virtue, not a vice." [page 48]

One further criticism that might be levelled against our adoption of folk psychology in this thesis is that our goal of improving educational practice means that we *should* be interested in how people actually solve problems, not merely in abstract models, because we want to change the real world. This is an interesting criticism because it raises the question as to how important an understanding of the way the brain works is to teachers. Certainly we would expect motor mechanics to have some understanding of how a car works before they attempt to improve performance, so why not expect teachers to understand how the brain works. In the past, teachers, of necessity, have treated the brain like a black box into which something is to be implanted. This implanting has taken place without consideration of the internal working of the box, mainly using tried and trusted techniques, occasionally influenced by current psychological theories. If this situation were to change radically, so that teachers become neural engineers, the consequences would be so far reaching for education that this thesis would become totally irrelevant, consequently such an eventuality will not be considered here. It is at this stage, of course, that the true relations between Software Engineering and Education would become apparent! The requirements of education could be specified by a client, designed and implemented by a "teacher", using a person as the target machine. As Huxley and Orwell, amongst others, have noted, the technical problems posed by such a scenario would be the least of our worries. We will return to this question in Chapter Seven.

What is a Problem?

Analysis of the concept of a problem is another complex task, and one that we will avoid here by accepting Mayer's use of the term. He describes three characteristics that all problems possess [May83, page 4]

1. *Givens*–the information, states of objects, etc. inherent in the problem as posed.
2. *Goals*–reflect a desired state in which the problem can be consiered solved. Conscious thought must be involved in reaching the goal state from the given situation.
3. *Obstacles*–the thinker must not already know the exact route from the givens to the goals, but must use the facilities at his or her disposal to navigate round obstacles.

An important point to note is that problems must involve conscious thought. Breathing, for example, does not pose a problem unless the obstacles present require thought to overcome them. Following on from this is the observation that, if we accept Mayer's criteria, a problem for one person may not be a problem for someone else. The task of finding a chess defence in response to a particular opening may present a problem to a novice, but be a simple memory exercise to a master. Thomas notes that we should also include the desire to overcome obstacles and hence achieve some goal in our definition of a problem. He observes that "the four-colour theorem was never "a problem"; it was a problem only for some people" [Tho89, page 318].

Within such a definition, a number of taxonomies of problems have been suggested. Reitman [Rei65], for example, classifies problems according to the degree of precision present in the stated givens and goals. The current state of Computing is such that technology can only be used to solve problems where both the givens and the goals are very specific and expressed in very simplistic terms. Most clients' problems, however, are presented without this degree of precision, and the task of the software engineer can be viewed as transforming the original problem formulation into one the technology can handle. This task will usually involve translation, interpretation, restriction and refinement, amongst other devices.

Pearl presents a classification based on the type of goal given. A goal presented as a simple set membership (such as the eight queens problem) gives rise to a satisficing problem. A goal which requires finding the supremum of some ordered set gives rise to an optimisation problem. A goal that involves finding values from a given set within some neighbourhood of the supremum is called a semi-optimisation problem. This class can be further subdivided into near-optimisation problems, where the value found must fall within the neighbourhood, and approximate-optimisation problems, where the value found must fall within the neighbourhood with some given probability. Clearly Software Engineering involves problems of all of these kinds. Correctness, for example, is generally presented as a satisficing problem, whereas efficient implementation is usually a semi-optimisation problem. One point we should note here is that optimisation is carried out over some set (which we will call the satisfaction set). If a problem involves both kinds of goals (such as finding a correct and efficient implementation) then the satisfaction problem logically takes precedence over the optimisation problem. There are a number of possible techniques we can use to combine these problem types, such as finding a characteristic correct solution and then applying transformations to find a semi-optimised solution, finding the satisficing set and deriving an appropriate selector function, or using the optimisation problem to guide the search for solutions to the satisfaction problem, thus allowing us to construct subsets of the satisfaction set. It is difficult to understand, however, what it would mean to tackle the optimisation problem first: finding the best wrong answer seems pointless.

A number of other classifications have been suggested. Bhaskar and Simon have distinguished between semantically rich and semantically impoverished problems depending on the degree of relevant knowledge the solver possesses about the problem domain [BS77]. Munson identifies a spectrum of problem types for pedagogical purposes [Mun88].

$$\text{open-ended} \Longrightarrow \text{egg race} \Longrightarrow \text{curriculum dedicated} \Longrightarrow \text{closed}$$

Open-ended problems have a variety of acceptable solutions due to loosely defined goals. Egg race problems have a variety of solutions, but quite specific goals. Curriculum dedicated problems are set with givens, goals and obstacles that are intimately tied to the intentions of the teacher. Closed problems have specific goals and also a very restricted number of acceptable solutions. It is also common to distinguish between adversary and non-adversary problems. In adversary problems there is a rational opponent trying to defeat the problem solver. We will assume that the software engineer is never faced with problems of this kind, although this is an oversimplification. Problems motivated by military applications and attempts to establish security systems may all be carried out in hostile environments. Even system design in the commercial sector may sometimes appear to the engineer to be adversary, particularly when the users or clients feel threatened by the project! We will return to the notion of "threat" in Chapter Seven.

We can also identify a problem classification determined by the software engineering solutions they give rise to. Lehman [Leh80] identified three types of program, and Pfleeger extended this classification to systems [Pfl87, pages 375–379]. We will extend the idea still further, and use the classification for the problems themselves. Like all classifications, of course, this is an abstraction, and thus simplifies the situation and creates boundaries where convenient rather than where actually present.

An *S-Problem* is one that we can solve exactly, and for all time. For example, the task of inverting a matrix is an S–Problem. The definition of matrices and their inversion is unlikely to change. The implementation of the solution may well change with technology, of course, but the *specification* of an acceptable solution to the problem is invariant.

It is often possible, however, to understand a problem well enough to know exactly what needs to be done to solve it, but to realise that implementing the solution is not feasible using current technology. Consequently we do not specify an exact solution to the problem, but we abstract from the problem a simpler model that we can solve, in the expectation that a solution to this simpler case will be an acceptable approximation to the solution of the original problem. Such a problem we will call a *P–Problem*, as its solution is based on a *practical* abstraction. The classical examples of such problems are usually found in Artificial Intelligence, such as the design of chess playing systems. The simplified model of the problem we devise can be treated as a S–Problem, so we could consider the task of model selection as giving rise to a sequence of S–Problems that develops alongside technology.

S–Problems and P–Problems are both posed in an application domain that is taken as fairly static. That is, we assume that the theory of matrices and the rules of chess are fixed. For many problems, however, this assumption of an invariant problem domain is a gross simplification. The tasks of providing an acceptable management information system or designing a real-time control system for a developing plant are examples of continuously changing problems. These involve the design of systems that to be *embedded* within an obviously changing world. We will refer to such problems as *E–Problems*. These give rise to the complexities of software maintenance, where a system has to be maintained as a solution to the current state of the problem.

A final classification is that between routine problems, requiring "reproductive thinking", and nonroutine problems, requiring "productive thinking" [Wer59]. The point is often made that teaching emphasizes reproductive thinking by setting, and assessing, routine problems [Dun45], but it is clear that many problems encountered by software engineers are nonroutine. One view of so-called design methods (or methodologies) is that they seek to provide routines for nonroutine problems. It will be argued that this approach is doomed to fail unless the method also serves to educate the problem server. It will further be argued that in general this is an ineffective and inefficient approach to education.

Models of Individual Problem Solving

One of the central questions of psychology is how people actually think when they are solving problems. This has resulted in a number of models of thinking, the major classes of which, using Mayer's headings [May83], are briefly reviewed below. This topic is clearly relevant to the task of improving pedagogical practice, but it is also relevant to the questions of methods and theories, for the view we take of thinking affects the properties we require in design methods and specification techniques.

Associationism

Associationism has its roots in Aristotle's three laws of learning and memory that seek to explain thought in terms of ideas and associations between them. The three types of associations identified were those of space and time, similarity and contrast. This idea was developed by Locke and Hobbs in the seventeenth and eighteenth centuries to include the notions of atomism (ideas are atoms linked by associations), mechanisation (movement between ideas is determined automatically by the strengths of the associations), empiricism (all ideas and associations arise from sensory perceptions), and imagery (since each idea is the result of sensory perception, movement between ideas must be understandable in these terms, such as a visual image).

Thorndike's work on cats in puzzle boxes was taken as endorsing associationism, for he found that cats respond to problems by a process of trial and error [Tho11]. With practice, certain unsuccessful responses become less common and the more successful are tried earlier in the process. This supported the model of problem solving that had the problem situation (the stimulus) associated with a number of responses (or problem solving behaviours), with the responses ordered by some notion of strength. Under this model, learning is largely a matter of adjusting strengths in the light of experience.

In its pure form, the associationist model fails to explain the ways in which insight can lead straight to a correct response without such a period of learning. To overcome this deficiency, the notion of covert responses was added, allowing undetectable responses to occur as the result of mind experiments until the correct response was identified and performed overtly. A great deal of research has been conducted to detect these supposed covert responses, and it has been suggested that they might be located in the muscles or the brain, but no conclusive evidence has been presented.

One of the important results to come out of the associationist model of problem solving is the idea of a problem solving set. It is clear that a solver's association hierarchy will be determined by past experience, but less obvious that the hierarchy will be dominated by experiences just prior to the task. This has been taken to imply that several hierarchies may exist, and recent experience may be the prime determinant in selecting the appropriate hierarchy [Mal55, page 281]. Such a set is useful in solving repetitive routine problems, but may prove a major hurdle in tackling nonroutine problems. Many of the heuristics suggested for improving problem solving performance are attempts to remove this problem solving set when tackling nonroutine problems.

Gestaltism

The Gestaltists concentrate on an organisational view of problem solving. For them, a problem is solved by achieving a structural understanding of the givens and the ways in which they may be reorganised to meet the goals. The unit of thought has thus shifted from being an association to being a structural organisation, and activity from being trials of responses to trials of reorganisation. It should be noted, however, that associationism concentrates on explaining reproductive problem solving, whereas Gestaltism is primarily concerned with productive problem solving. The symptoms of problem solving set are also recognised in Gestaltism, but here they are explained in terms of functional fixedness. If a component becomes, in the mind, bound into a particular rôle in a problem, then the solver has difficulty in seeing reorganisations in which the component adopts a radically different rôle.

Meaning Theory

In Gestaltism, the organisation of a problem situation is inherent in the components of the problem, and the solver is seeking relationships internal to the problem. In the meaning theory model, however, the solver is seeking to assimilate the presented problem to schemata already held in memory. Under this view, solving problems involves not only reorganisation to match an existing schema, but also interpretation of the present problem in terms of the components of the schema.

There is evidence that clues given regarding problem situations significantly alter the assimilation process. Bransford and Johnson [BJ72], for example, found that the comprehension of text was greatly improved by the provision of a title for the piece before reading, but providing a title after reading led to little improvement. Mayer [May75] applied a similar experiment to the learning of a programming language, the results of which are discussed in a later chapter. There is also evidence that imagery plays a significant part in the assimilation process. Translation of "better" into spatial terms of "above", for example, seems to lead to an improvement of problem solving performance [DLH65]. It is further suggested that integrated images lead to better performance than piecewise representation of problem components.

Ausubel [Aus68] and Greeno [Gre73] identify two different kinds of schema. Meaningful (or propositional) schemata contain concepts, whereas rote (or algorithmic) schemata contain rules for operating on concepts. Assimilation into rote schemata seems to lead to a better problem solving performance on reproductive problems, but to poorer performance on productive problems. Assimilation into meaningful schemata, however, leads to better performance on productive performance, but not such a good reproductive problem solving performance.

This evidence suggests that the initial perception of a problem is crucial in determining problem solving performance for different kinds of problems. This is clearly of significance to the task of teaching problem solving, but it is also important for the activity of writing specifications. It has been noted that making problems descriptions concrete in different ways leads to different methods of solution, indeed, many of the heuristics suggested for improving problem solving performance are based on the idea of finding appropriate representations and images. If we accept that specifications of requirements and designs are both statements of problems to be solved then we should expect that different ways of presenting a specification will lead to different solutions of the problem. This observation causes us to ask what it means to suggest that "specifications should specify what and not how", and whether such an objective is achievable. This question is discussed in Chapter Five, where we will treat theory presentations as the basis of problem descriptions.

One of the major applications of research on meaning theory as a model of problem solving has been in curriculum development, and has given rise to the current trend for learning by discovery, leading to assimilation into meaningful structures, rather than learning by exposition, leading to assimilation into rote structures. Such an approach, it is claimed, leads to transferable skills. Although there is evidence to support some of the principles behind this claim [EN69] [GB61] [RS68], the claim is as yet unsupported by empirical testing. We will proceed, however, on the assumption that such a claim is justified.

The current state of psychology is such that there is still no real understanding of schemata. It has been suggested that a schema can be considered as a hierarchy containing slots into which concepts can be fitted. Kintsch [Kin74] and Meyer [Mey75], for example, suggest that the understanding of English sentences is carried out by assimilation into schemata related to the grammar of the language, with the deep structure of the text dominating the hierarchy. Rumelhart [Rum75] and Thorndyke [Tho77] extend this idea to whole texts, by suggesting that the listener makes use of a story grammar comprising a setting, a theme, a plot, and a resolution.

It is interesting to speculate whether similar structures exist for understanding descriptions of information systems. For example, a fairly common notion in formal specifications of information systems is that of a signature, so it might be suggested that this is one component of such a situation grammar. An obvious candidate for a whole grammar is that of the von Neumann architecture. Many descriptions of information systems centre on concepts such as data flow, memory and processors, and it seems likely that many readers of specifications struggle to fit what they are reading into this classical situation grammar.

Information Processing Model

The final model we will consider has arisen out of the conjunction of Psychology and Computing: the information processing model. This model has been developed for two purposes. First, to facilitate the automation of the reasoning process, and second to allow psychologists access to an automated model of the human's processes for experimentation. Unlike the previous models, however, this one makes no real attempt to explain the thought processes involved in solving problems, but rather to provide a structuring mechanism. Instead of concentrating on the processes involved, the information processing model considers the path to the solution of a problem as a number of information states that must be passed through, starting with a state representing the given problem, and ending with a state representing an acceptable solution. In general, a common representation is used for all the problem states, and the totality of these states is referred to as the problem space. The problem solving task is thus reduced to a state-space traversal exercise.

Several strategies exist for traversing the problem space. The least sophisticated is one of trial and error, where legal transformations are applied to each current state to generate a chain of states in the hope that the goal state will eventually be reached. Very naive hacking may be seen as a trial and error process, where changes are made at random in the hope that the program will eventually work, although the problem space for most real designs is so complex that trial and error is really not possible. More sophisticated is a process of hill-climbing, where only transformations that lead "towards" the goal state are applied. Although this may seem intuitively more attractive than trial and error, there is the danger that hill-climbing may lead to local maxima, from which further progress is impossible. Refinement can be modelled by a form of hill-climbing, where the problem state is progressively transformed from a specification to an executable program. The goal state for the refinement process is usually taken to be correct code, that is efficiently implemented on a specified target machine, where efficiency has a variety of possible interpretations. The literature on refinement to date has concentrated on correctness and implementability on sequential machines, with some attention to efficiency of storage and algorithm performance. Although hill-climbing may lead to local maxima, most problems tackled by refinement are approximate optimisation problems, and the local maxima are taken to fall within the acceptable neighbourhood. No research seems to have been carried out into this assumption. Refinement is also limited in the problem areas it can tackle, for appropriate transformations need to be defined for each representation domain, and currently there are no suitable transformations for domains involving real-time constraints, fault-tolerance, distribution of processing and storage, human-computer interfaces, and many other features required of systems.

A more sophisticated technique is means-end analysis. Here the problem solver identifies not only the goal of the exercise but also the obstacles standing in the way of achieving them. Overcoming these obstacles is then taken as establishing a new set of goals which, if met concurrently, will lead to the solution of the original problem. Methods of functional and data decomposition can be seen as means-end analysis, effectively dividing the problem up into subproblems.

3.3 Discourse Theory

The final body of knowledge in which we will analyse Software Engineering is that of discourse theory. Like the philosophy of technology and problem solving, this body of knowledge is still very much in a preparadigmatic stage: that is, there are no generally accepted frameworks within which to evaluate new ideas. The discipline is still at the stages of proposing such frameworks, and trying to support these with the same substantive facts that the frameworks are trying to explain. Some aspects of discourse have certainly been explored and reached a level of maturity, but when this happens the topic tends to have been removed from the domain of discourse theory *per se* and subsumed under more mature disciplines. Logic, for example, has its roots in discourse, but in the early part of this century it became a focus of attention in philosophy departments, and later in mathematics departments. Similarly the literary aspects of discourse tend to have been subsumed under literature.

There is no doubt that Software Engineering can be considered as a process of discourse. From its earliest beginnings, Computing has adopted much of the culture and terminology of discourse. We speak, for example, of programming languages, specification languages, syntax, semantics, and interpretation. Early models of computer systems were based on the ideas of layers of language. More recently, the rôle of linguistic systems has been recognised as fundamental to the process of software design [TM87]. One of the problems, however, is to recognise that this process of discourse has to be viewed holistically initially, which means that we want to consider discourse theory at a stage prior to that at which significant components were transferred to other disciplines. We do not want to consider "literate programming" [Knu84], for example, by analogy to English Literature, but as an equally valid application of a fundamental theory to a particular domain. Less obviously, science too has its roots in discourse, and many of the problems we have observed in attempting to apply the Philosophy of Science to Computing might have been avoided if a general theory of scientific discourse had preceded that of the theories of the natural sciences.

In the discussion that follows, attention has been restricted to just one theory of discourse, that of Kinneavy [Kin84]. The primary reason for the selection of this particular theory is that it seems to have been developed largely by consideration of the common ground with other theories, and Kinneavy constantly reinterprets his statements in terms of other views of discourse. Thus anyone wishing to develop the discussion with alternative theories of discourse should have little trouble in so doing.

Kinneavy's Theory of Discourse

Before outlining the structure of the theory, we must delimit the scope of the concept of discourse. The terms "communication", "rhetoric", "composition", and many others, have all been used to denote what Kinneavy means by "discourse". He restricts attention, however, to full text situations. That is, he is not concerned with fragments of sentences, or single utterances. The backbone of his theory is a classification of different types of discourse into a coherent framework, together with the organisational structure, characteristics, and underlying logic for each type. We will briefly review the whole theory, and then extract just one type of discourse for a more detailed analysis.

The Underlying Model

Kinneavy adopts a fairly conventional starting place for his theory, by accepting as given the communication triangle shown in Figure 3.1.

This model has been used in one form or another since the very beginnings of studies in discourse. Aristotle adopted the triangle as a basis for the analysis of rhetoric, Carnap uses it for his work on semantics and pragmatics, Carnap and Shannon use similar models for their work on information theory, (with the addition of extra concepts such as encoders, decoders and noise). It has also been widely adopted for the teaching of communication skills and literary criticism.

Using this model, Kinneavy structures the consideration of communication into study of the signal, *syntactics*; study of the correspondence between signal and reference, *seman-*

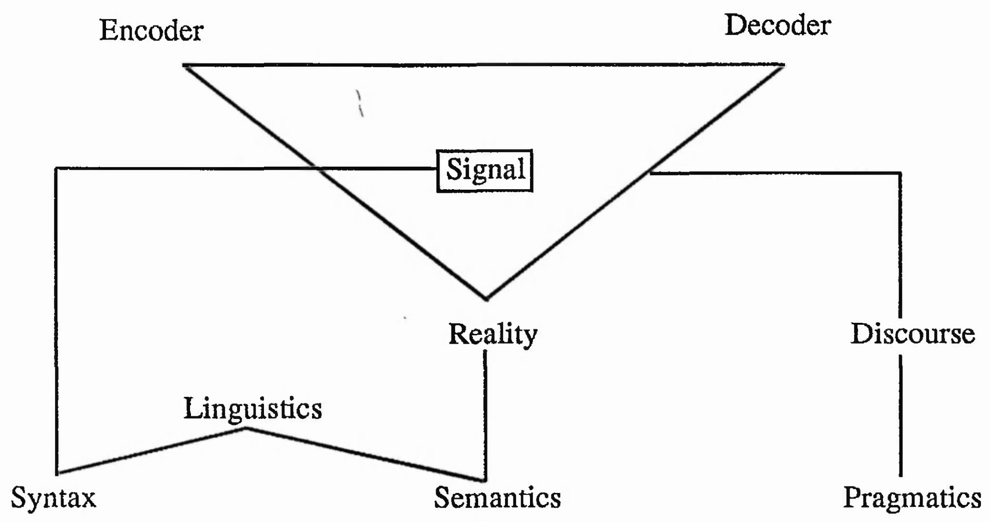

Figure 3.1: The Communication Triangle

tics; and study of the uses to which communication is put, *pragmatics*. His theory of discourse is concerned primarily with pragmatics, and includes semantics and syntactics only where they impinge upon this. Kinneavy thus forges a distinction between discourse theory and linguistics, the latter concentrating on the syntax and semantics of discourse.

The Arts and Media of Discourse

Kinneavy uses the terms "arts" and "media" to distinguish between the signal (eg. the written word or the spoken word) and the channel (eg. the newspaper or the radio programme). He excludes from his arts the act of thinking, asserting that thinking pervades the whole of communication but is not itself simply one of the skills that enables discourse. He asserts that attempts to include thinking as an art of discourse is to trivialise it, and usually reflects a fundamental flaw in the underlying model which is being used, namely the omission of thought from the model in a more significant rôle. His argument is is that the omission of a treatment of thought in language teaching, for example, is a fundamental error, caused by the separation of language teaching from discourse. This is a very subtle, but powerful, argument, and one that we should remember when we come to discuss the teaching of topics such as specification languages, where a criticism can be levied that the skills of using notations are often isolated from the thought processes that should pervade the whole activity.

One of the limitations of Kinneavy's consideration of arts and media, however, is that he restricts attention to the written and spoken word. This is understandable, since in 1971 these were taken as the primary modes of discourse, but an interesting extension to his discussion would include hypertext, where the structure of the discourse is inherent in the art to a greater extent than in conventional written texts, and also symbolic simulations, where the logic of the text is presented explicitly, for example, as a set of rewrite rules. Rather less understandable is Kinneavy's omission of drawing from the arts of discourse. We will include these, as many of the discourse processes in modern Software Engineering, rightly or wrongly, are founded upon pictorial arts, often using the media of computer screens.

Modes of Discourse

To perform adequate abstraction of texts for discussion, Kinneavy recommends that we ignore the subject matter of the text, such as "this is a physics book", and move towards a more generic classification, "this is a description of something", or "this is a story about something". He refers to these classifications as the modes of discourse, or more traditionally, the forms of discourse. Bain, in 1867, identified four such modes: narration, argumentation, exposition and description. Kinneavy adopts the terms "narration", "classification", "description" and "evaluation" for these four modes. It is important to realise that Kinneavy is separating the modes of discourse from the purpose of utilising the mode. An evaluation, for example, could be used to tease out a description, to persuade someone to buy a product, or to act as a scientific proof. Many "stories" in English literature have been written with aims far beyond the amusing of an audience with a yarn.

Descriptions in Software Engineering are used for several purposes. A system specification, for example, is usually written using a descriptive mode. It may have as its aim that of persuading a client to purchase the system. A specification may also be used, however, to inform an engineer of what is to be built, or to convince an engineer's peer group that the specified system has particular desirable properties. It is important to realise that the aims of the discourse, as well as the mode, give rise to its properties. For example, one of the major criticisms of formal notations as a basis for specifications is that they inhibit the conveying of information to a client, who is unlikely to be conversant with the notation being used. This criticism is based on the fallacy of inferring an aim of discourse simply from the mode, for the *aim* of descriptive discourse may not be information passing. To understand this more fully we must consider the possible aims of discourse.

The Aims of Discourse

Kinneavy identifies the aims of discourse by consideration of the features of the communication triangle, stressing that these aims are rarely found in isolation but that most discourse has a primary aim that can be classified into one of three kinds. Concentration on the encoder or decoder gives rise to two kinds of people discourse: if the encoder is emphasised, then the aim is said to be *expressive*, if the decoder is emphasised, then the aim is said to be *persuasive*. When the focus of attention is on the reality to which the discourse refers, however, rather than the people involved, then the aim is said to be *reference*. Moreover, each of these aims has its own norms within which discourse is evaluated. Thus advertising, whose primary aim is persuasive, should be evaluated as such, even if a secondary aim is to inform potential customers of the attributes of a product, and is therefore one of reference. Applying the norms of science, where reference is paramount, to advertising, where persuasion is the aim, is to completely miss the point of the discourse.

Discourse of all three kinds can be found in Software Engineering. Expressive discourse is not obvious in the discipline, although it seems likely that many of those who program for fun do so because they are expressing themselves through their programs, just as poets seek to do through their poetry. Certainly the impression given by many systems programmers is that they treat "their" code as extensions of themselves, remaining unworried by the existence of anyone else who can understand what they have written. Kidder quotes a professional programmer as saying

> "I loved writing programs. I could control the machine. I could make it express my own thoughts. It was an expansion of the mind to have a computer." [Kid82, page 90]

We should not pass value judgements on this activity *per se*, but we can observe that problems are likely to occur if the sender's aims are different to those of the recipient. Expressive discourse is quite respectable in computer art and music, and there is no reason to deny anyone the pleasure of expressing thoughts in any medium, but we must recognise the limitations of expressive discourse, and ensure that professional software engineers are aware of them too.

Persuasive discourse is present in most sales activities, and also in many project management techniques. It may also play a part in the writing of user manuals, where the aims of discourse can be quite complicated. Many people assume that manuals are intended to be informative. Modern manual writers, however, have also adopted the aim of persuading the users that the product is "good", and also that anyone can learn to use it. The hypercard User's Guide, for example, informs its reader that

> Any piece of information in HyperCard can connect to any other piece of information, so you can find out what you need to know in as much or as little detail as you need." [App87, page xvi]

A truly wondrous product!

Another place where persuasive discourse can often be found is in the discourse surrounding peer review of systems. Many software engineers confuse the true aims of activities such as system walk-throughs with those of convincing colleagues that a system is correct. The true aim of a walk-through should be to try to find faults with the system (a scientific aim in the tradition of Popper). Persuading colleagues that the system is fault-free achieves nothing of scientific value unless the designer has been scientifically honest, by assisting in the process of seeking out refutations. This confusion of aims is often attributable to project management methods, such as the imposition of deadlines without genuine quality milestones associated with them, and also to the loss of innocence described earlier. Engineers must now take responsibility for their actions, consequently their reputations, and possibly their career progressions, are at stake when their work is subjected to review. The temptation to protect this reputation is often too great, and the risks of being found out quite small. Most companies are not prepared to spend resources in tracking down the culprit when an error is detected: frequently even the material cause of the error is never identified, only the symptoms are addressed. It is only with the realisation that safety critical systems are being designed using these attitudes and approaches that the serious deficiences with this way of proceeding are being recognised by the software industry at large.

It is also possible that some engineers regard system specifications as persuasive discourse, in the sense that the specification should persuade the implementor to construct the system the specifier had in mind. Another view is quite different to this, namely that there should be no element of persuasion at all. The implementor should read the specification simply as a piece of reference discourse, and should be free to implement any system that meets the specification. This is an interesting issue, for most specifications actually have both aims, as we shall see below.

We should note in passing that Hoare and Naur both appear to have been engaging in persuasive discourse in the papers discussed earlier. Moreover, many of the publications in the literature of Software Engineering are of this kind. Scientists, ostensibly at least, present theories without appearing to persuade readers that the theories should be believed: it is the evidence that causes the reader to believe the theory, and this evidence should be presented honestly. Many publications in Software Engineering, however, appear to be selling an idea, notation, method, or machine. The virtues are expounded in full detail, the limitations are frequently omitted altogether. This is a cultural problem for the discipline, for even those authors who are extolling the virtues of a scientific approach add to the body of unscientific literature associated with the subject. This is often cited by opponents of scientific approaches as inconsistent, for they have failed to grasp the distinction between scientific discourse and discourse (of any type) *about* science. Although this does not give rise to an inconsistency by the proponents of scientific approaches, it does tend to dilute the scientific literature still further. This problem does not arise to the same extent in other sciences, where the unscientific discussion has been partitioned off as philosophy.

Most of the discourse used in the technical aspects of Software Engineering, however, has reference as its primary aim. For this reason, we will focus attention in the next section on a more detailed discussion of reference discourse.

Reference Discourse

Reference discourse, although subservient to the central aim of referring to reality, can be considered as being of three kinds, depending on the perception of reality being used.

Informative Discourse: If reality is perceived as known, and the discourse is conveying facts about this reality, then the aim of the discourse is informative.

Scientific Discourse: If this information is forged into a coherent system, and presented in such a way that a demonstration of its validity is possible, or even accompanies the information, then the aim of the discourse is scientific.

Exploratory Discourse: If the reality is not known, but is being looked for, then the discourse is exploratory.

This classification appears to be unique to Kinneavy's theory of discourse, but turns out to be extremely useful in discussing Software Engineering, where all three kinds of reference discourse are readily discernible.

Scientific Discourse

We have already discussed many of the attributes of scientific discourse during our treatment of the Philosophy of Science. We should note, however, that Kinneavy assumes that there is such a thing as "science", rather than various sciences each giving rise to different types of discourse. In treating the same reality, the various scientific disciplines will certainly bring different paradigms to bear, and interpret results accordingly, but the resulting discourse processes, typically the published papers, will conform to certain accepted norms, such as the style of presentation and the reduction of the rôle of the scientist. Frawley goes so far as to say that "science is discourse" [Fra86, page 68], a view that Feyerabend disputes, observing that this is a limited view of science as involving only third world entities and ignoring the importance of the second world [Fey70a, page 28].

If we accept scientific discourse, in this sense, into Software Engineering then we are accepting a set of norms for this discourse process. Acceptance of these norms is by no means universal even amongst scientists. Medawar, for example, questions the value of the accepted styles in scientific discourse [Med64]. Feyerabend goes further, and calls into question many of the norms of science itself [Fey87].

One important question concerning scientific discourse is whether it should seek to explain reality, or simply to describe it. The view that scientific discourse should only describe was prevalent until the turn of the century, and can be found, for example, in the writings of Pearson [Pea11]. Current thinking, however, is that scientific discourse must seek to explain as well as to describe, it must therefore seek to present *theories*, not just facts, or, at least, to present facts within the framework of some already presented theory. Exactly what constitutes an appropriate theory for this purpose we will discuss later. Kinneavy also notes that there is nothing in the nature of theories that rules out any of the *modes* of discourse. Scientific discourse can be narration, description, evaluation or classification, each of which can be based on underlying theories. This endorses the view of many proponents of formal methods in Software Engineering, who claim that formal specifications, although descriptive, should be scientific as well as informative. We shall add to this the claim that they should be exploratory as well.

The logics suggested as underpinning scientific discourse are well documented elsewhere. They have generally been deductive logics, which operate at the syntactic level of discourse, and inductive logics, which are semantic. Kinneavy raises the question of a pragmatic level of logic, which seeks to provide foundations for the uses of scientific discourse. This would seek to capture aspects of the proof such as how it changes the decoder's beliefs. Logics with dialogue interpretations, such as the infinite value logic $\mathbf{L}_\alpha$ of Łukasiewicz, may provide a basis for further research in the area [Gil81].

The inclusion of pragmatics into proof allows us to elevate the discussion of proof above the "how" (the syntactic details of the formal system) and the "what" (the semantic details of the interpretation) to consider the "why" (the reason for taking particular interpretations, and why a proof is of any value in a particular context, for example). In traditional scientific discourse, such discussion is rarely carried out. Most scientific papers, for example, set out what is to be proved without justification, and discuss the proof as if the method were taken for granted. Software Engineering, however, has not evolved through such a scientific culture, consequently many Software Engineers frequently do raise such issues. Faced with the existence of systematic frameworks for the discussion of syntactics and semantics, but no suitable framework for discussing pragmatics, these software engineers understandably often slip back into a rigorous discussion of easier aspects of the problem, or content themselves with a non-scientific treatment of the harder ones. There have recently been attempts to change this state of affairs. Work by Cohen and Pitt, for example, seeks to discuss proof obligations in Software Engineering, and to relate proof mechanisms and interpretations to the use of formalisms in Software Engineering [CP90b][CP90c][CP90a]. This work is still in its infancy, but some very interesting ideas are put forward that are considered later in this thesis.

The traditional view of logics is that their use is culturally independent. This view is no longer given the same credence, however, and it is now accepted that although the unity of scientific logic may well be an ideal, we must accept that scientists may have to work within an existing culture, which embodies its own logic, whilst moving towards this ideal.

This presents us with another problem, for, whilst physicists have well established sets of norms, developed and passed on primarily through a mature academic discipline, Software Engineering has a variety of "logics" in use. Many Software Engineers learnt their trade through apprenticeships, and hence picked up a craft culture, some are mathematicians, some are electrical engineers, some are classics scholars: the list is endless. Furthermore, many of the environments within which Software Engineering is practised provide their own logics: the logics of company economics and of research council funding being two examples that have significant impact on the ways in which Software Engineers work. This diversity of backgrounds ensures that Software Engineers do not blindly accept particular logics as underpinning their scientific discourse. As the discipline develops, however, we are increasingly being expected to develop cultural norms. Unfortunately this development is often interpreted as the need for a seemingly *ad hoc* selection from the norms currently represented.

Informative Discourse

The aim of informative discourse is that of conveying information to the recipient, who is thus allowed to intrude into the process. This information may be conveyed by at least three aspects of the discourse,

- The structure of the discourse may convey information: the ordering of concepts, for example, may be taken as providing a structure within which they should be interpreted.
- The components of the discourse will themselves convey information.
- The subjective background of the recipient will provide information that may be invoked by the discourse.

The syntax and semantics of informative discourse will not be discussed here, as they are well covered in the literature of linguistics, but the pragmatics of informative discourse does require a brief discussion. The pragmatics of informative discourse has to consider information transfer to a real, rather than idealised, receiver. Semantically, for example, a tautological expression carries no information. Pragmatically, however, it may convey information if the recipient was unaware of the tautology at the time of receipt. The pragmatics of informative discourse, therefore, needs to adopt a different method of measuring information content from that of information theory. Kinneavy suggests that surprise value should be used. The tautology $\vdash P \iff P$ is of little surprise to most people who understand the notation, whereas more complex tautologies may well be surprising, and hence more informative.

This intrusion of the recipient into the discourse process is in stark contrast to the situation with scientific discourse, where scientists intentionally try to minimise the intrusion of people. In writing a specification as a piece of informative discourse, therefore, the writer should be cognizant of the potential readership; in writing a specification to act as a scientific theory presentation, however, scientific style dictates that the readership should not be unduly considered. This raises the question as to whether one specification can have both aims. We will sidestep this question by suggesting that the concept of a specification is not as central to system design as is currently implied by most models of the design process. In fact, it will be suggested that specifications should be dependent upon theories, and that these theories can have many presentations, some of which will be aimed at scientific discourse, some at informative discourse, and some at exploratory discourse. This shift from specification to explication is essential if Software Engineering is to adopt the rationalist approach of modern science.

The structure of informative discourse cannot be given by the facts to be presented alone, which are isolated, but only from some extra-factual source. In scientific discourse, the structure is often dictated by the logic being used to support the arguments. In informative discourse there are several possible sources for the discourse structure. It could be supplied by some theoretical understanding of the facts, either held by the writer or assumed by the reader. It could be provided by the adoption of large scale metaphor in the discourse, where the information to be conveyed is presented in the order dictated by the metaphor. Both of the above structuring techniques imply a structure into which the reader can assimilate information as it is received. Another common structuring mechanism is that of presenting information according to some concept of importance. Again this involves the adoption of a theory, in this case an axiological theory. This approach is often used to motivate the reader. In terms of assimilation structure, however, the reader is left to devise this as information arrives, and one hopes that the importance hierarchy devised by the writer will map in some sensible way to a possible assimilation structure. An area where this frequently fails to occur is when cultural boundaries are being crossed. Software Engineers carrying out requirements analysis, for example, are often presented with information seen as important by the users, but need to seek out the information they need to start the process of building a suitable assimilation structure. One often used measure of importance is that of surprise value, presenting the most surprising facts first (a common journalistic device). Where completeness and ease of referral is important, some form of mechanical ordering may be used, such as alphabetical or dateline orderings.

Exploratory Discourse

Kinneavy's decision to identify exploratory discourse as distinct from scientific discourse is unusual. Most views of science include both discovery and verification, therefore it might seem that scientific discourse should include discourse with exploratory aims. In practice, however, the views of the exploratory process are usually restricted to the results of the process, the conjectures, and because the philosophy of science has traditionally been phrased in terms of individual scientists, rather than teams of scientists, the discourse processes leading up to these results has been marginalised. Popper, for example, has his Logic of Scientific Discovery, which covers the presentation of conjectures, the selection between rival theories, and the attempted refutation of theories. His account does not extend back to the jottings of scientists where half-formed conjectures are being assessed, where terms are being refined in meaning, or where the scientist is playing with ideas in very "unscientific" ways, waiting for inspiration. It is considered useful for the purpose of this thesis to have a category of pre-scientific (in the Popperian sense) discourse that can be considered part of the scientific process, and Kinneavy's exploratory discourse fits the bill well. This type of discourse actually fits more naturally into Lakatos's view of Science, where anomalies and contradictions form an accepted part of science within research programmes.

The logic of exploratory discourse is hard to identify. We can use deduction, for example, to explore the consequences of adopting certain views, thus embedding scientific discourse into the exploration process, but the motivation that leads us to select what consequences to explore, and the justification of this choice, is founded in the logic of exploration. Keltner suggests that the logic of discussion is one of problem solving [Kel57, pages 32-34], but there is more to exploration than just discussion. One of the major differences between exploratory and scientific discourse is that when exploring we may proceed in the full knowledge that our theories are contradictory, suspending disbelief in order to gain further insights into the problem being approached. In science, however, deduction based on inconsistent premises is rarely intentionally carried out. Another difference is that in exploratory discourse it is quite common to accept tentative, or even ambiguous, semantics, relying upon the discourse process to identify the areas that need refinement. We may also use metaphors, analogies and models widely to provide shared frameworks for discussion of novel ideas. Possible candidates for logics to underpin exploratory discourse include those based on statistics or fuzzy mathematics [Sch81].

The structure of exploratory discourse is hard to generalise. It is likely to be more fragmentary than informative or scientific discourse, and also to have a large number of "if ... then ..." type structures, but basically there are as many ways of structuring explorations as there are ways of thinking. We can, however, observe that there is likely to be a fundamental difference between exploratory and scientific discourse structures. The starting point for scientific discourse is usually a theory. This theory is then used to derive refutable statements. In exploratory discourse, however, we are presented with an *ad hoc* collection of facts to be understood and explained. One way of doing this is to construct a deductive system, a theory, that is based upon a few established facts, and from which the other facts can be derived. This is a creative process: we must design a theory, but like all design "the creative process is sometimes immeasurably facilitated by borrowing suggestions from another deductive or inductive system which seems to have similarity to the one under construction. Such a borrowed system is a 'model'. " (page 144). For it to be useful, we must already be familiar with the model we are borrowing, for we want to transfer details from one domain to the other.

> "This transfer can be in several directions. The model can help to secure a relatively unstructured domain, or simplify a domain, or complete a domain, or explain a domain, or concretize a too abstract domain, or abstract a too concrete domain, or enable a domain to get a complete picture of its own framework, or allow experimentation where the domain does not permit it." (page 144)

The use of models in exploratory discourse is clearly analogous to the structuring mechanisms suggested in the previous section for problem solving. It also turns out to be analogous to the use of theory building, for theory presentations are just models. We will return to this issue in Chapter Five, where we will seek to reconcile notions such as theory, model and analogy.

In practice, exploratory, scientific and informative discourse are not usually separately observable in the scientific process. They merge as explorations lead to a tentative hypothesis, which is gradually firmed up, communicated as information, and submitted to public attempts at refutation. A useful insight may be gained here by appeal to the literature of neural networks, for the process of training a network can be facilitated by the use of simulated annealing. We start the network running in its training mode with a high temperature (that is, a large amount of noise), and gradually cool it down as learning takes place. This attempts to overcome the problems associated with the network stabilising in a local minimum state, rather than continuing to seek better solutions [AM90, pages 112-130]. This can be illustrated graphically as in Figure 3.2.

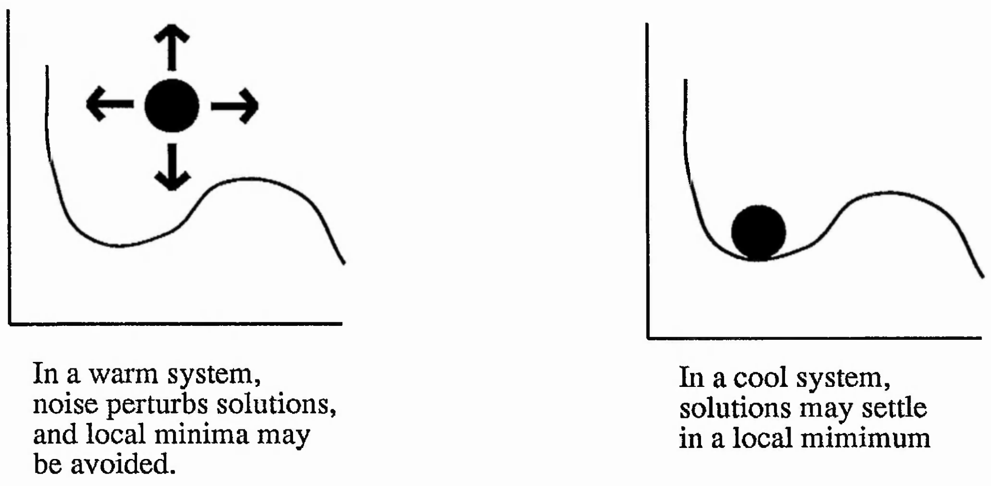

Figure 3.2: Warm and Cool Learning

Similarly we can see exploratory discourse as the activity of the warm learning phase, where noise is deliberately allowed into the system, not only because we do not know how to exclude it, but also because it helps us to find better solutions. Once we believe we have trained ourselves (that is, we have found our theory), we seek to exclude noise by adopting informative or scientific discourse, content to sit in a stable, if local, minimum. This notion of simulated annealing will recur in various guises throughout the thesis.

3.4 Summary

This chapter has, of necessity, been rather fragmentary, for it has surveyed a number of disparate bodies of knowledge, and no suitable structure exists for reconciliation of the issues raised. The next chapter seeks to provide such a structure suitable for the limited aims of this research. The material covered in this chapter forms a resource to be drawn upon by subsequent chapters, and, in particular, it has established some useful terminology.

A recurrent theme throughout this chapter has been the relationship between the person and human activity, whether it be designing, solving problems, or engaging in discourse.

This is undoubtedly one of the most complicated issues being addressed in this thesis, and is the focus of Chapter Seven. In 1981, Tweney *et al* made the point that "the psychology of scientific thinking is coming into existence" [TDM81, page 1]: this chapter may be viewed as providing a number of research programmes that it can pursue.

Chapter 4

A Model of System Design

"Discovery consists of seeing what everybody has seen and thinking what nobody has thought"

Albert Szent-Györgyi

The previous two chapters have contained wide-ranging discussions drawn from several disciplines. Although these discussions have all been motivated by the task of achieving a better understanding of Software Engineering, and have been carried out using well-established frameworks from these disciplines where possible, they undoubtedly appear to lack coherence and structure. In this chapter the intention is to start pulling these strands together. It should be stressed that no attempt will be made to do this at a deep theoretical or philosophical level, a task far beyond the scope of this thesis, but rather a model of system development will be proposed within which the rôle of the various topics discussed can be identified. Broadly speaking, the proposed model should

- admit the discussion of a scientific basis for Computing in the sense of Popper.
- establish where the more traditional engineering aspects of Software Design can be located, and how the technological basis for a design fits in.
- identify the principal areas of reference discourse in the process.
- recognise the central rôle of problem solving in the whole subject of software system design.

In addition to facilitating the structuring of the thesis itself, the model should also be useful for teachers of Computing as a structuring mechanisms for presenting material, and consequently for learners as a provider of schemata into which material may be assimilated. It is not intended, however, that it should also be useful for those charged with the task of managing or accounting large development projects, or those trying to design CASE tools or software factories. The relationships between between this model and other kinds of model, including planning models such as life cycles, will be discussed in Chapter Six.

It is intentional that this model is developed around very simple ideas, and not used to explain all the details of "real" projects. There are several rationalisations for this:

- The ideas will scale up, for large problems are just an aggregate of lots of small problems.
- In real problems, management plays a dominant rôle, often obscuring the technical processes that are taking place.
- Real projects are part of the software crisis. What is needed is a simpler model from which improved ways of proceeding can be found.

These are just rationalisations, however, and the honest reason for this decision is a belief that we should teach through simplified models. All other disciplines accept such models without question. Physics has its frictionless planes, point masses, and infinite wires, for example, and chemistry has ideal gases. Most engineering disciplines sit upon such sciences, and hence build upon these simplified models. Computing seems to be unique in its bizarre pursuit of "reality" from day one. This may be due in part to an irrational response to the political pressure to provide vocational training, but the consequences are far reaching. For example, we have put computer systems into schools that demonstrate all the intricate features of real machines[1], we have used badly designed, if popular, programming language to introduce students to programming, and we have used design methods that are ill-founded, unstable and described in vast arrays of manuals to introduce the concepts of design. These decisions seem on a par with filling chemistry laboratories with chemicals that are full of impurities, because that is how they are usually found in most industrial processes.

The suggestion is also made that teaching via simplified models, far from making the students less effective in handling real problems, offers the only hope we have of allowing inexperienced engineers to approach real problems in sensible ways. This will be discussed later in the thesis, but little scientific support will be given, for Cognitive Science is not yet developed sufficiently to provide firm empirical evidence. This suggestion is a praxiological re-statement of the principle of curriculum inversion [Coh86].

4.1 An Overview of the Model

The proposed model arose from Naur's suggestion that we should treat programming as a process of theory building [Nau85], together with Burstall and Goguen's discussion of putting theories together to make specifications [BG77]. We will extend these ideas to suggest that Software Engineering can be considered to be based on a process of theory (presentation) engineering, where the target theory presentations have certain specific properties relating to executability. The idea that we consider the artifacts of all stages of design as theories allows us to admit scientific ideals at all stages, although we must

[1]We should distinguish between the computer as a vehicle for delivering pedagogical material, like television, and the computer as an artifact of study. It is not being suggest that televisions should not be used because they are complicated, but that they should not be used to introduce electricity to primary school pupils!

acknowledge that, during the construction and communication of these theories, the modes of discourse being used may not be scientific, but exploratory or informative.

It is useful to consider the theories involved in the process of software system design as being of a number of kinds, determined by the primary source of the theory. This categorisation is, of course, only approximate, and no claim is made that it is exhaustive.

Phenomenological theories: these arise from the domain of the problem being solved. In a stock control system, for example, the laws of supply and demand might be considered part of the phenomenological theory, as might the laws underlying the company's policy on buying ahead. A theory that is primarily phenomenological will serve as a statement of requirements for our system design activity.

Intuitive theories: these are considered as arising from common sense. They are the sort of theories discussed in naive physics and folk psychology, and can be seen as underpinning the actions of many users and purchasers of software systems, although they are likely to remain unstated.

Mathematical theories: these are theories that effectively define mathematical terms, such as group theory or set theory. In solving mathematical problems, these will also be phenomenological theories.

Computer Science theories: these are the theories governing the behaviour of artifacts studied, or designed, in Computer Science. The theory of programming languages, sorting, data types and logic design might all be considered to fall into this category. These can be considered definitional or instrumental on occasions. A theory of Pascal, for example, might be used to define the language in a particular context, or to reason about Pascal programs, hence it may sometimes be considered mathematical or phenomenological.

The essence of the model is that we take a problem situation and construct a (primarily) phenomenological theory of a system,giving rise to schemas that will solve the problem [Put74]. This theory may be presented in terms of other phenomenological, intuitive, mathematical, or Computer Science theories. We then transform our theory presentation into one having behaviourally equivalent schemas (where this equivalence has to be defined), expressed in terms of the theory presentations of existing computational artifacts. The argument that programs can be considered as theory presentations will not be expanded here, but an excellent rehearsal of the argument can be found in Hoare's presentation to the Royal Society [Hoa85].

The notion of a program as a theory is also easier to accept if we adopt Dijkstra's view that the subject of interest to computer scientists is computation, not the computer [Dij89], for then we can see programs as theories governing particular computations. Additional support for the idea can be found in the seminal work of Miller, Galanter and Pribram, who write

"A plan is, for an organism, essentially the same as program for a computer. ...we regard a computer's program that simulates certain features of an organism's behaviour as a theory about the organism's plan that generated the behaviour." [MGP60, page 17]

Evidence to suggest that expert programmers tend to view programs in terms of plans has also been offered by Rist [Ris86].

This transformation process may make use of techniques for gradual translation (refinement), or involve leaps of the imagination. In either case, however, the onus is on the designer to attempt refutation of the theory at all stages where it is presented in a scientific (rather than informative or exploratory) form. Such refutations may manifest themselves as inconsistencies in the theory, or as a failure to correspond with the facts, and it is up to the designer to decide how to construct a new theory to cope with the problem. In practice, of course, many of the theories used will be treated as instrumental, and hence refutable only in extreme cases.

This makes software development a process of self-conscious design. An endorsement of this view is expressed by Neumann when he writes "Software Engineering is a *state of mind* attainable by thoughtful and farseeing people" [Neu85]. At each stage the designer must be prepared to present the current theory in such a form that it is amenable to refutation. Attempts to escape from the loss of innocence by pretensions to genius are non-scientific, as the genius will not contemplate actively seeking out refutations. Attempts to escape by adoption of classical styles or methods are also non-scientific, in Popper's sense, because they take as axiomatic a large body of assumptions and protect them from refutation. It should be noted, however, that the whole of modern science can be deemed a refuge in style: Feyerabend, for example, notes that the adoption of a western rationalism brings with it a number of hidden assumptions that are not open to debate within science itself [Fey87]. The theory presentations play the rôle of diagrams in Alexander's philosophy. Our move towards computationally biased theory presentations can be re-expressed as the need to move towards constructive diagrams. This means that we move from being able to discuss only the function of a system to being able to discuss both its function and its form, and in the case of software this form corresponds to the configuration of computational objects.

The model also allows us to abstract away from certain aspects of the desired behaviour of the system at particular stages of the design. The process of theory construction is inherently one of abstraction, and we can decide to delay the introduction of certain facets of the design until later stages in the transformation process. This delaying of detail is widely accepted in the pedagogical practices of science teaching, as demonstrated by the adoption of "school science" with its simple, idealised, models, as a precursor to "real" science, with its more complex, but still idealised, models. The simplified theory presentations are still viewed as presentations of the same theory, however, for we do not think of school science as proposing different theories, or of proposing wrong theories. This discussion will be put on a firmer footing in the next chapter, when we will discuss a suitable refinement of the notion of "theory".

In our system design task, memory constraints, for example, although known at the outset, can be ignored in our theory presentations until a suitable opportunity is arrived at for the concept to be presented. Although we we may choose to ignore some information in our current theory presentation, it should be remembered that this information potentially still influences the transformations we make. Thus some information starts off at a meta-level governing the design and selection of our theories, but may migrate down into the theory itself as the design proceeds. Some information may never be expressed formally in our presentation, but will remain reflected in the choices of presentation we make. This close relationship that exists between method and theory will be developed in several ways in the rest of the thesis, and is a recurrent theme at all level of the discussion.

In this mode of working, the software engineer behaves in many ways like a pure scientist, putting forward theories with a view to having them refuted. There is, of course, a complication: science is always assumed to be founded on honesty. Popper admits that science, as he views it, is based on the premise that no scientist would ever falsify (in the fraudulent sense) results, or intentionally express a theory ambiguously so that there are escapes from refutations. This is a *scientific attitude.* Central to Hoare's approach is the notion that Software Engineers must build such an attitude. Throughout this thesis it will be treated as axiomatic that such an attitude is both desirable and attainable within the profession, but no justification will be given. Attempting such a justification leads us into the realms of ethics, and questions the very fabric of society[2]. The reader should be aware, however, that this assumption is crucial to all that follows, and that the pedagogical discussions are implicitly founded on the additional assumption that students accept this code of behaviour or that it can be developed in them. All education has such hidden agendas, and teachers at all levels have implicitly accepted responsibility for developing suitable ethical values in their pupils. Unfortunately the producer-consumer model of education is sometimes interpreted as an excuse for abandoning the teaching of these values in favour of directly applicable skills. It would be ironic if this shift were to cause the abandonment of the very values necessary for significant improvements in software production.

4.2 The Source of Phenomenological Theories

We need to consider very carefully what objects are being discussed by phenomenological theories. The lay person's view of natural science is that it is concerned with objects that can be considered to appear naturally in the world. Modern science, however, is very hard to explain with such intuitions, and recent philosophies of science have tended to

[2]This should not be taken as an endorsement of the view that students should not spend time discussing such issues. The author was shocked to discover recently that, on surveying 45 final year Computer Science Degree students, 39 of them had never thought there were any question of morality in releasing software knowing it to have bugs in it.

move away from the pre-occupation with observation of the natural world, and attempted to provide notions of "theory" that permit a separation of theory from nature. Before considering this subject further in the next chapter, we will discuss how theories might arise in Software Design.

One view of Software Engineering, sometimes cited as a reason for not adopting scientific practices, is that it is concerned not with the natural world, but with man-made information structures. There are a number of ways in which these information structures may be manifest. They may be embedded in existing systems, either human or realised by some other artifact, or they may be contained implicitly in some problem that we are setting out to solve *ab initio*.

The task of designing a computer-based system to replicate the actions of an existing human system is quite easy to express in terms of theory building. Here we are reverse-engineering a solution to a problem that has already been solved once, in order to re-implement it in a different technology. Thus we are provided with a model upon which to experiment and from which to generalise a theory. In the majority of cases the theory underpinning the original system is not available to us explicitly, however, for usually the human system has evolved over a period of time, often through a process of unselfconscious design, and the implementors have never really needed to express the solution in a rational way. Moreover, the people who implement the existing system frequently also act as users of the system, solving difficult problems as they arise. This provides a flexible and expert system, but unselfconsciously. It also confuses the system boundaries. Such an approach usually means that the system documentation and the designer's perceptions, do not reflect the system as it actually exists to solve the problem. That is, the "designers" have no real theory as to why the system operates as it does. Stewart and Stewart illustrate this point with their delightful account of the cuddly toy quality inspector, whose description of his job was phrased in terms of rejecting the "mardy bugger". The inspector was unable to articulate exactly what constituted a 'mardy bugger", but after careful analysis using a Kellian repertory grid technique [Kel55], it was established that to avoid being so classified a toy had to have the spacing between eyes and nose the same as that between nose and mouth, and also the pupils of the eyes had to be centrally located [SS82], cited in [Jan87, page 485].

The theory of an existing system may be hard to extract, but we should beware of the naive assumption that because action is taking place there must already be a theory, or at least a plan, to extract at all. Suchman suggests that it is only the western desire for rationalisation that leads us to assume action should be underpinned by well thought-out plans. She develops the notion of situated actions as governing method, and relegates plans to post-rationalisations, or resources to be drawn upon as orientation aids when problems arise [Suc87]. Parnas and Clements express a similar view in the context of system design, defending this post-rationalisation as a worthwhile activity [PC85]. At this point it is important to understand the status of the theory our engineer is trying to construct. We are not suggesting that this phenomenological theory is governing the actions of people in the existing human system, and the task is one of discovery.

Rather we are asserting that we need to construct a rationalisation as the basis of a plan governing the actions of a machine. Current technology requires such a plan, for computer systems are still programmed, and are not capable of situated actions to the same extent as humans. It might be suggested that this is not because of a limitation of processing power, but of interface: the machine does not have access to the same wealth of detail about each situation that is available to the human problem solver. Furthermore, we are not endorsing the cognitive science view that human action may be *studied* by building computable theories assumed to lie beneath actions, rather we are asserting that *one way* to simulate these actions is to construct a computable theory. Thus our theory building is genuinely a process of design and not discovery.

This also raises the question as to whether our theory, once designed, should be exported to the application domain through the user interface. Suchman makes the pertinent observation that once designed, computer-based systems could actually instruct the users in the theory underpinning their design [Suc87, page 17]. This concords with the idea that software engineering and education have many similarities, for once educated we expect our pupils to be able to instruct others. We will not pursue this question further in this thesis, but it should be noted how naturally questions like this arise in the context of the proposed model.

On occasion, the method that the human is using for solving a problem may be difficult to devise by discussion with the individual alone. Flying helicopters, for example, is an activity where many of the pilot's actions are carried out by "feel", in reaction to the behaviour of the machine. Autopilot design to date has not attempted to exploit this "feel", but has sought decisions based upon the theory of flight, the current mission environment, and various measurable parameters. Recent research suggests that an alternative strategy might be to attempt simulation of human action by treating the pilots as empirical objects, and devising a theory that explains their actions in terms of the sense data available to them. This can be achieved by a process of rule induction. This theory can then be transformed, typically by a process of "cleaning up", where the superior speed of a computer is exploited to make the same decisions, but faster. Evidence suggests that this technique can keep the controlled device within tighter operating envelopes [Mic90].

Designing a computer-based solution to a problem currently solved by another artifact, rather than a person, such as designing a microprocessor system to replace the mechanical control unit in a washing machine, is usually a rather simpler task. Here the reverse engineering is more traditional, and in addition we may be allowed access to the specification of the original unit. The usefulness of this specification will largely be determined by the extent to which it captures the theory of the artifact in isolation from the theory of the technology in which the artifact is currently implemented. If the specification is expressed solely in terms of drawings of cogs and motors, for example, it may be of little value for this purpose: if it is expressed in more abstract concepts it is likely to be of more immediate use. In effect, we do not want a constructive diagram but a purely functional one.

In both of the above cases we have an existing object that can be empirically observed. Frequently, however, we are called upon to solve problems that have hitherto been left unsolved. In this case we have no system to observe empirically. Loosely we can imagine that we are trying to construct a theory of "what the customer wants". In order to discuss this we need to have a model of the design process that admits theories of artifacts to be constructed as well as of those that already exist. We will do this by adopting a strategy suggested by Simon in his work on the Sciences of the Artificial [Sim69] and also Alexander in his work on architecture [Ale64]. Both authors suggest that we can view an artifact in two ways: as an object or as an interface. If we have access to the "hole" into which the system is to fit, then we can experiment empirically with the interface surrounding this hole.

This is also the kernel of the slogan that we should start off by specifying the "what" not the "how". This slogan is not very helpful, however, unless we are prepared to discuss our interpretation of the "whatness" and "howness" of a system. We will interpret the slogan as meaning that we should seek to express what we want the system to be in terms of function observed across some interface before expressing what we want the system to be in terms of its form. This does raise the question as to how easily function can be expressed without implications for form: a question that we will return to in the next chapter.

Typical of the confusion that this slogan causes is the problem faced by a designer who is told at the outset that a particular programming language must be used. This may occur, for example, where a new program is to be added to an existing suite, and the customer wants to use existing staff to maintain the new program without any additional staff development. This requirement to use a particular language is then part of the what, not the how, because it is visible, to maintenance staff, across the interface. If the customers make no such stipulation then they have no right to delve into the system to observe the language after it has been constructed, and object to the designer's choice. The point to note here is that if an implementation language or target machine is specified, implicitly or explicitly, by the customer then it will influence the design in the same way as the more conventional functional attributes. The designer may, of course, choose to delay inclusion of such details in the theory until a later stage in the design process, thus allowing this influence to be shown by theory presentation selection, rather than within the presentation itself. The slogan, in this case, carries an implied methodological message regarding the ordering of functional attributes. As we shall see, however, this message is inconsistent.

We can view this process of understanding the hole in at least four ways. First we could consider that the customers and users are the true objects of our empirical study, and that we are trying to construct a theory of their beliefs about the object that will fill the hole. In general, however, no single individual will suffice for this study, and so we must study a network of interacting individuals. Each individual will have a different set of beliefs. There is no problem with beliefs being logically inconsistent provided they are kept within different systems, but before the engineer can form a unified theory, within a system simple enough to compute with on current technology, these inconsistencies will have to be resolved. This can be achieved either by genuinely persuading individuals to change there beliefs, or by persuading them to act against their beliefs. In practice, many of the inconsistencies observed are the result of the interpretation of terms, rather than inconsistent views of the world.

Traditional views of Science cannot easily cope with such objects of study. It is usually assumed that experiments are repeatable, for example, but if we are dealing with belief systems we must accept that such repetition cannot be relied upon, for there is no obligation for beliefs to remain constant in a changing world. It is also assumed that the scientist attempts to minimise interference with the world being observed. Resolving inconsistencies in the natural world by "persuading" objects to behave differently has a distinctly unscientific feel! Psychologists are well aware of the problems this assumption of non-interference brings, but where the engineer has the additional motive of wanting to seek consistency, this non-interference is itself paradoxical. This view is now considered rather naive, and it is generally accepted that at the very least "needs analysis" should replace "wants analysis"[3] in the hope that needs will be less inconsistent than wants.

The second way of viewing the process allows us to consider an object of study that does not change, and is hence more amenable to scientific approaches. We can consider the users and customer as instruments for observation of an unchanging hole. Inconsistencies and changes over time can now be seen as calibration problems. We can attempt to calibrate these instruments by bringing them into contact with concrete examples, introducing common terminology and expecting common answers. This is the approach used for many years at the Royal Observatory in Greenwich, where observers were all tested, giving rise to the "personal equations" that were used to counter individual differences when using the telescope [Gre84, pages 210-216]. These equations were based on statistical techniques, which could attempt to correct for relative, but not absolute, error. The assumption was made in this approach that perception mechanisms are fairly simplistic, and hence simple statistics would serve to compensate for differences. We must accept, however, that observation is theory laden and seek to reconcile the conflicting theories that give rise to the inconsistencies. This would appear to lead to a circular problem, unless we are prepared to break the circle by the dictatorial imposition of existing theories for the observation. Such an approach was common in the early days of information system design, and led, not surprisingly, to considerable user resistance. Moreover, the imposition of a theory usually leads to the wrong problem being solved, as the engineer forces an inappropriate perception of the problem on the user.

The third way of viewing the problems is to consider the artifact and its environment as forming the desired system. In practice, this is usually done for software systems, where

[3]Again we might note the irony of the situation when asked to educate engineers within a system that is being forced to deliver "what the consumer wants". The inconsistencies that such wants analysis present (such as the conflicting wants of students, parents, employees and government) are surely reflected in the number of *ad hoc* solutions, and subsequent problems, of the education system. We might further reflect that if such an education leads graduates to accept the values inherent in the system, then the system might be acting against the education of engineers fitted to solve the software crisis.

functionality is expressed in terms of the running program, even if the target machine is not part of the designed artifact. This is simply explained in our model, for we use theories of program execution in our presentation, importing as instruments theories of the execution environment. The approach is also quite common in control systems, where the specification is written in terms of how the controlled plant is to behave, not how the control program is to behave. We could extend this idea, however, and include, say, users as part of the system. We might, therefore, take our system boundary as the functioning of a department within a corporate structure. In this case, our design task can be seen as the solving of an interface equation [Shi86]:

$$S \equiv (E|X) \setminus I$$

where S is the total system, E is the environment, I is the set of internal communications, and X is the part of the system we are to design. This view might be plausible for a control system, but if E involves a substantial human element it is unlikely that its behaviour will remain constant with the introduction of X into its environment. The only way this view would work is if we could dictate the human behaviour. Current thinking in HCI is coming around to the view that this is not only undesirable, but impossible. Even if we do accept this view for a particular problem, of course, we still have to find a way of constructing a theory of S.

The final way we will consider of viewing the problem, which we shall adopt in what follows, is that the customers and users should be allowed to become part of the design team, who bring with them existing theories that they use for interpreting observations. The task of the Software Engineer is to coordinate the team on a research programme of unification, generalisation and refinement. This allows us to weaken the maxim "the customer is always right" by using Popper's assertion that anyone who holds theories may well be wrong, customers included. It is the duty of the software engineer formulate the theory in such a way that refutation is possible, to manage the refutation/revision cycle, and to move the theory presentation towards an implementable form. In a sense, the software engineer is playing the rôle of teacher here, helping the customer to understand the system. This idea that users and customers should become more involved in the design process is currently being advocated by the proponents of a number of new development "methodologies". One side-effect of this is to emphasise that system design is not usually technologically neutral: by being forced to question their existing theories, many users and managers have discovered inconsistencies in existing company procedures, structures and policies. This again raises the question of honesty, for the approach can only really work if everyone involved accepts scientific standards: a junior member of staff who is not prepared to refute the theory of a director, for example, or a director who refuses to listen to any refutations of a subordinate, is not conforming to these standards[4]. The management structure and ethos in many companies is likely to

[4]We should not assume, however, that scientists automatically conform to these standards either. The history of science offers many examples of junior researchers having their ideas ignored by those in charge of their research programmes.

cause this approach to flounder.

Whichever way we choose to view the process, we can see there are problems. It is not enough for the software engineer to *want* to behave scientifically. If we adopt view one, there is not yet sufficient understanding of the ways in which beliefs change for this to be done. If we adopt view two, then we must accept that we are working with unreliable instruments. View three presents us with both of these problems, and also the task of solving the interface equation. If we adopt view four, then we must accept that it is not enough for the software engineer to want to behave scientifically, but there must be a corresponding willingness on the part of the customers and users. This raises the question of the customers' and users' perceptions of the software engineering process. Unfortunately, these are likely to be formed on the basis of experiences and stories, often apocryphal, of computing disasters. It is hard to persuade people that the process is scientific, with the high ideals this entails, when they have perceptions that include the engineer as a dictator, or as a high priest, or as a back-street trader. User expectations, therefore, must be seen as a contributing factor to the software crisis. Attribution of blame is an unconstructive enterprise, but if this interpretation of affairs is correct, then the crisis can clearly only be resolved over a number of years as expectations change. Hoare's implication that software engineers can change their status from high priests to scientists by their own actions is perhaps a little naive. If the masses still *want* high priests, or continue to see high priests, then they can make such a transition very hard. Naur's view that the status must be both deserved and conferred seems more tenable. This observation is a reflection of Kant's point that art requires observers sufficiently cultured to appreciate it as such [Kan11]: software systems require users and clients who are able to perceive them as engineered artifacts.

4.3 The Refutation of Phenomenological Theories

What does it mean to refute or accept a phenomenological theory? According to Popper, we accept a theory "only in the sense that we select it as worthy to be subjected to further criticism, and to the severest tests we can design"; any theory that survives, however, "is the best theory ...of which we know" [Pop59, page 419]. A major criticism of Popper has been his inability, or reluctance, to explain what he means by "best" in this context. On the surface, he cannot mean "most likely to be true", for bolder, less probable, theories are considered better than probable ones. In fact, however, Popper's philosophy is concerned with the construction of theories, not their use.

When we consider the *construction* of theories, there is a clear distinction between Popper and the inductivists. This distinction seems to disappear when we consider the selection of theories for some use outside of science. Popper does not concern himself directly with such uses of theories, seeing the theory as an end in itself. If we push the point, however, and insist that theories must be capable of application to problems outside of science, then Popper is recommending the selection of the theory that evidence to date has failed to refute, in spite of real attempts. The inductivist is recommending the theory the evidence supports. The difference at the level of selection would appear to be one of terminology. We would argue, however, that there is still a major difference in attitude. The inductivist amasses data, then seeks to fit a theory. The pure refutationist conjectures a theory to explain some minimal set of phenomena, then seeks evidence that will refute it, probably hoping (if hope is allowable to a scientist) that the refutation will fail.

These two approaches to theory construction can be considered as giving rise to alternative strategies for "requirements analysis". Inductivism gives rise to the notion that requirements analysis starts by the process of requirements capture, where data is collected, then moves to forge this data into a generalised theory upon which the design is based. This process may still be driven by theory, for the choice of what data to collect, unless truly random, may be determined, or rationalised, by some plan or theory held by the scientist. Refutationism, however, leads to the view that certain facts will be made available as received knowledge in the problem situation, but then a theory will be constructed. Additional information will then be sought out in attempts to refute the theory, thus the theory construction drives the requirements capture rather than following on from it. It should be stressed here that we are not suggesting the theory will be constructed in all its glory before the refutation is sought. The early stages of theory construction will be exploratory discourse, and it is likely that during this stage many possible sources of refutation will be identified, additional information will be sought, and if a refutation seems likely the theory will be modified. Thus the documented result of both the inductivist and refutationalist approaches are likely to converge to scientific discourse expressed in inductivist terms.

One objection to the adoption of a refutationist strategy is that the process will never stop. The engineer will keep attempting refutations, thus improving understanding of the required system, but never actually building it. It is at this stage that the the maxim "the customer is always right" can safely be invoked, but at the meta-scientific level. The customer is not right in the sense that his or her views are true, but in that he or she *has the right* to call a halt to the process of theory construction. The customer is paying not for truth but for time and expertise. Therefore at any stage the engineer can be told to stop the refutation process and to proceed with the transformation phase: in essence, to stop being a scientist, building theories, and to become an engineer using them. Another objection may be that in most system design activities a purely refutationalist strategy is not an option, for the theory may drift wildly away from the task in hand. In Chapter Six we will suggest that adopting Lakatos's "research programmes" [Lak70] provides a way to reconcile refutation with progress towards a loosely defined goal.

Once the exploratory phase of theory construction has been carried out, a clear contractual boundary must be drawn between the customer and the engineer [Coh82]. Systems that fail to conform to the theory as established at this cut-off point are not acceptable, but if the customer suddenly realises that the theory as agreed is inadequate, that is not the engineer's failing. A professional engineer, of course, has the responsibility to offer good advice to the customer as to when a suitable stage has been reached for the accepting of a theory. When a theory becomes too complex for the customer to follow in an abstract form, certain schemas of the theory may be transformed for use within an idealised world as prototypes. These give the customer a chance to refute the theory by direct experience rather than by having to imagine the system. They also serve as partial existence proofs for aspects of the system, albeit in an idealised world.

There are certain extreme forms that can be observed in this aspect of the model. The customer might, for example, present the engineer with a list of observation statements, and not wish to engage in any further theoretical discussions. In this case any theory that is not refuted by the given statements will suffice. Such an approach is sensible only for trivial systems, and even then usually requires the engineer to use considerable common sense and also some domain specific knowledge if the customer's *needs* are to be satisfied. Alternatively, the customer may view the design process as eternal, but ask the engineer to perform implementations based on partially presented theories on the way through. Large operating systems are often designed in this way, where many versions of the system are released, even though the engineer already knows that the theory they are based on has been refuted, and acknowledges the fact by a list of known bugs and "features". The customer, like the engineer, has to weigh up many factors in deciding when to accept a theory, and this decision is in no way a part of the theory. Such an approach is certainly necessary for the solution of E-Problems.

Note that not all predictions arising from a theory are sufficient to refuteit. Popperian refutation requires a basic sentence to be derived, that is, one that expresses an observable fact. Theories, however, in general comprise generalisations. The conventional way in which to view the derivation of basic statements from theories is to assume some world, expressed in terms of auxiliary statements, and apply the theory to this world, thus producing specifics from generalities. The predictions, therefore, are based upon a conjunction of the theory with the auxiliary statements. A problem of terminology arises here, for scientists usually use the term "theory" to denote this conjunction, rather than the philosophers' pure theory. This allows the scientist to produce laws which it is claimed will be true within this idealised world, and claim them as products of the theory.

These auxiliary statements are crucial in determining the system boundary, for they determine the relevant aspects of the world in which the system is to operate. A system being designed to handle personnel information, for example, might be based on an auxiliary statement that all employees have at most two initials. The engineer and customer may both know this to be false, having met an employee called Cuthbert Jacob Earnest Bottlethwaite. Acceptance of the auxiliary statement will allow laws to be derived, but at the expense of generality. If Cuthbert is to be handled by this system, a way needs to be found of removing one of his initials (thus making him conform to the system) or a patch must be provided to the system (thus rescuing the theory by *ad hoc* means). Acceptance of auxiliary statements forms part of the contractual boundary, and many examples of system maintenance arise as examples of changes to the auxiliary statements. It should be noted that auxiliary statements are accepted in the full knowledge that they are not universally true. They are instrumental, and refutation of the theory is only attempted in worlds in which the auxiliary statements hold.

There is no clear distinction between auxiliary statements and those contained in the theory. At one extreme, all auxiliary statements could be absorbed into the theory proper, in which case those that are not true will lead to immediate refutation. At the other extreme, a theory could be reduced to the rules of some logic, and all other statements could be treated as auxiliary, in which case refutation is not possible. The transition from software design as science to software design as engineering can be considered as the change in status of statements from theoretical to auxiliary, that is, from statements that rationalise the world to statements that define the world of application agreed in the contractual boundary. Moreover, the status of these auxiliary statements cannot change in the subsequent transformations that take place except with the customer's agreement: refutations found in system testing should not lead to changes in the specification, only the program. This aspect of Popper's exposition does not fit well with the task in hand, and is one of the reasons for moving on to consider a more refined notion of "theory" in the next chapter.

This view of the design process allows us to place a different interpretation on the phrase "all right in theory, but in practice ...". If we accept this statement at face value we are allowing comparison of theory and practice, thus admitting them as similar enough to be compared: an approach which Ryle denies, claiming it is a categorical error to carry out such a comparison [Ryl49]. To avoid this problem we will interpret "in theory" and "in practice" as shorthand ways of describing domains of application: thus claiming something is all right in theory, but not in practice is a way of saying that the auxiliary statements adopted by the theoriser do not hold in the world being worked in by the practitioner.

4.4 Schemas

Theories are usually linked with the notion of explanation. The model of explanation that we will adopt is that described by Hempel as the Deductive-Nomological model (the D-N model) [Hem65]. Ryan describes the rules of this model as follows:

> "a successful explanation has to obey three requirements. The first is the formal requirement that the statements laying down the laws and initial conditions should entail the statement laying down the conclusion; the second is the material requirement that the premises should be true—or more cautiously that they should be well corroborated; the last is a consequence of these requirements, that the *explicans* should be empirically testable, by being open to refutation should it predict what is not the case." [Rya84, pages 52-53]

This association of theories with explanation has led to one of the criticisms of the theory building view of programs, namely that the principal purpose of programs is to compute values and not to provide explanations [Joh88]. This criticism arises naturally out of adoption of Popper's philosophy, for explanation in a holistic sense is the only purpose proposed for a theory in this philosophy, hence the reason for refutation of the whole theory if any prediction does not accord with the facts. Kuhn has pointed out, however, that holistic explanation is only the principal purpose of theory in the (unusual) revolutionary phase of science, and that Popper "has characterised the entire scientific enterprise in terms that apply only to its occasional revolutionary parts" [Kuh70a, page 6]. Kuhn maintains that scientific theories in normal science are used for solving "puzzles", such as predicting results given some initial conditions, or determining initial conditions to explain some observed result. An important point to note here is that "refutation" during this normal science is unlikely to result in rejection of the theory, rather they will be attributed to errors on the part of the scientist which will, hopefully, lead to learning by the scientist. Only when cumulative effects of "refutation" are noted by a community at large will the theory be rejected, and corporate learning take place.

The use of theories in puzzle solving leads us to conclude that theories are not, as Johnson suggests, used only to provide propositions, but also to yield values in the form of calculations of initial conditions and consequences. It can always be argued, of course, that such values can be expressed implicitly within propositions, such as "the required values of the initial condition is x", but then Johnson's objection to programming as theory building also dissolves, for we can simply embed every program within a suitable propositional form. Moreover, even if we accept Johnson's point that programs are not theories, this does not deny that programming can be seen as a process of theory building, for we can escape by observing that programs are not the only products of programming. This is the point that Naur makes so forcefully.

These different ways of using theories have been termed "schemas" by Putnam, who identified three principal forms [Put74]. We can illustrate these schemas by appealing to Pythagoras's theorem, as part of the empirical theory of triangular objects (which is how Pythagoras first conceived it, before the theory became instrumental). We will call the first of these the refutational schema. It allows us to make a prediction based on a held theory and a set of auxiliary statements.

$$\frac{\textit{Theory}\ ,\ \textit{auxiliary Statements}}{\textit{Prediction}}$$

As a result of applying this schema, it is the theory that may change. If we observe, for example, that a triangle with sides 6,7 and 8 form a right-angled triangle, then we can predict that $6^2 + 7^2 = 8^2$, which it doesn't, of course. Consequently we need to refute the theory (in this example it is more likely we would rescue the theory by investigating the world in which this triangle exists, and impose auxiliary statements to exclude it).

The second schema, which we will call the explanatory schema, allows us to establish what initial conditions must have held for some fact to be explained by a theory.

$$\frac{\textit{Theory}\ ,\ ????}{\textit{Fact to be explained}}$$

In this case, the fact cannot be false, and the theory will not be immediately refuted if the fact cannot be explained. If we observe, therefore, that $3^2 + 4^2 = 5^2$ we can explain this by stating that there must be in our auxiliary statements something to admit a triangle with sides 3, 4 and 5 as right-angled. There may, of course, be many possible explanations of any given fact. The existence of multiple explanations may give rise to the quest for a generalisation of the theory to provide links between the explanations.

The final schema Putnam considers allows us to compute values given a theory and auxiliary statements, and is effectively expressing the relationship between laws and rules that we considered earlier. We will call this the computational schema.

$$\frac{\textit{Theory}\ ,\ \textit{auxiliary statements}}{????}$$

This schema gives rise to many possible computations, of course: we should not confuse the schemas with their particular use. We might, for example, use Pythagoras's theorem to compute one side of a right-angled triangle given the other two sides. This is a purely instrumental use of the theory. We should note here that if we embody a theory of propositions, for example, in our theory then our calculations may look like predictions (for example, we might compute the proposition $x = y$). It is important to recognise the distinction between the use of logic to express theories, and the theory of logic itself.

This computational schema gives rise to specific formulae. Dijkstra notes that viewing programs as formulae has many corollaries:

> "First, it puts the programmer's task in the proper perspective: he has to devise that formula. Second, it explains why the world of mathematics all but ignored the programming challenge: programs were so much longer than formulae it was used to that it did not even recognise them as such. Now, back to to the programmer's job. He has to derive that formula; he has to derive that program. We know of only one reliable way of doing that, *viz* by means of symbol manipulation. And now the circle is closed. We construct our mathematical symbol manipulations by means of symbol manipulations." [Dij89, page 1401]

Dijkstra fails to observe, however, that a formal system must be found within which the circle can be circumnavigated. This, we would argue, is the formalisation of the theory we are building.

There are two distinct observations that we can make at this point. The first is that computers can be used to implement all three possible schemas. The computerised exploration of the four-colour problem, for example, was an attempt at refutation. Diagnostic systems sometimes attempt an implementation of an explanatory schema (although more commonly, a theory of failure will be constructed). Both of these schemas, however, frequently involve highly complex search strategies, and so are more commonly found under the heading of Artificial Intelligence rather than Software Engineering. The computational schema is still the most commonly implemented one at present, and hence we will concentrate on it here.

The second observation is that the software engineer will use all three types of schema during the course of software design. Refutational schemas will be used in forming the theory of the system under consideration. Explanatory schemas will be used particularly during exploratory discourse, where tentative theories are held, and the engineer is attempting to see if the theory, together with the given auxiliary statements, is sufficient to explain the required phenomena. The computational form will be used with existing theories of third world objects such as programming languages and data structures in circumnavigating Dijkstra's circle.

Causey has criticised the introduction of schemas on the ground that once you start to analyse them carefully you rapidly discover the need for more and more schemas to explain the actual behaviour of scientists [Cau77, page 456]. We shall take this criticism as a warning, and use our schemas only as rationalisations for the plans governing the behaviour of machines.

4.5 Proof Obligations

There are certain properties that we require of both our theory presentations, and also the transitions that can take place between pairs of presentations. These are not independent, for it is the need to preserve properties that gives rise to the transitional requirements. It is a consequence of adopting selfconscious design that the engineer must identify the proof obligations inherent any any approach adopted, be aware of possible ways in which these obligations might be discharged, and select the most appropriate way for carrying this out. Refuge in style is particularly dangerous here, for by adopting a pre-defined method the engineer often relinquishes control over these proof obligations, usually without realising it. The *responsibility* for discharging the obligation, however, *must* still rest with the engineer, and the fact that a chosen tool makes identification and discharge of obligations difficult is no excuse, for the method was selected. If the method is imposed, by customer or higher engineering authority, then the engineer is provided with a refuge from the loss of innocence.

A systematic identification of proof obligations, both those inherent in the process of theory building, and those consequential on the choice of presentation techniques, has not yet taken place in Software Engineering. Cohen and Pitt have started this process, and have suggested a number of obligations that can be identified, and ways in which discharge can be achieved [CP90b][CP90c][CP90a]. Only general properties will be discussed here: those properties relating to specific styles of specification or implementation will be discussed in the next chapter.

Obligations for a Single Theory Presentation

The refutation of a phenomenological theory is an indication that it is not fit for purpose. This notion of fitness for purpose is common in engineering, and captures the idea of a system solving an identified problem. Application of this idea to Software Engineering has been discussed in more detail elsewhere [Loo91]. In our model, however, there are in general two or more theory presentations that need to be considered. The phenomenological theory presentation will, presumably, have been deemed fit for purpose by the customer authority (by failure to refute), and will thus form a contractual boundary. The theory presentation embodied in the delivered system must also be fit for purpose: indeed, this is the only one that really concerns the user. Clearly, one way this could be done is to attempt refutation of the delivered presentation. Since we have an agreed contractual boundary, we do not need to involve the customer in this, we can simply subject the final version to the same body of tests that failed to refute the phenomenological theory. The customer, however, may also insist on acceptance tests before agreeing that the contract has been met. Logically, this approach is perfectly acceptable. Testing is often belittled, but this must be done on methodological grounds, not logical. Indeed, one of the arguments often proposed against testing is that it cannot show the absence of bugs. This is true, but only testing can show fitness for purpose, for deductive proof cannot show the correctness of theories! The key is, of course, to allow the testing of the phenomenological theory that has already been carried out to percolate through the process, so that we do not need to retest at every stage of the design process.

One of the methodological arguments against testing during the later stages of design is that the theory presentations, once embodied in code, become difficult to reason about. This means that it is very hard to find the severe tests deductively, and often testing becomes more of a stochastic process. Moreover, once we have implemented only specific schemas, we usually lose the ability to refute general cases by finding counter examples logically: rather we end up attempting to prove them exhaustively, or more probably, assuming a principle of induction. Tools may offer support to this activity, by "studying" the code and inferring sensible tests, but ultimately we are still looking for needles in haystacks.

The real methodological problem with testing at this stage, however, is that we are trying to solve the wrong problem. If we do succeed in refutation then either our initial phenomenological theory has also been refuted (in which case the contract has to be changed or, more likely, reinterpreted) or the embodied presentation is not a presentation of the phenomenological theory.

Correctness and Refinement between Presentations

Hoare and Gries tackle this problem by suggesting that we attempt to ensure that our delivered system embodies the same theory as that already agreed in the contract. They are suggesting that we *use* computing theories instrumentally, to to prove a correspondence between two theory presentations. They are proposing a meta-theoretical approach. In particular, they are proposing a mathematico-scientific approach, where we take a formal theory presentation in which we deduce appropriate theorems about our specifications and programs (we will adopt these terms to avoid confusion between the levels of theory present here).

This approach, however, requires three conditions to be met. First, we must have a formal presentation of the specification. Second, we require a formal presentation of the implementation. These two conditions amount to the observation that we must be able to formalise our domain of interest (in this case a pair of theory presentations) before we can reason formally about it; this will be considered further in the next section. Third, we need a logic, and associated heuristics, for deducing at least the correctness property that if our specification has not been refuted by a set of experiments then our implementation will not be refuted by the same set. In fact, this condition is not as obvious as it sounds, for it means that the system may be a presentation of a more powerful theory than that inherent in our specification. We are insisting that given a specification theory T_s and an implementation theory T_i then $T_i \Longrightarrow T_s$ rather than $T_i \equiv T_s$. For a more detailed account of the decision to accept implication rather than equivalence, see [Hoa85]. A specification may require some properties for just positive integers, whereas the implementation can provide schemas that work for all integers. If a customer is selling this product on, and intends releasing a more expensive version that works for all integers, then clearly the delivered system will not be fit for purpose, as it will not accord with marketing strategy, although it may be "correct". In this case, we have failed to specify the system tightly enough: it is part of the defined behaviour of the system that it should not respond to negative integers, or rather, should respond with a null response! Similar arguments can be put forward concerning error handling: errors clearly cannot be unforeseen events (otherwise we could not program in ways to handle them), therefore they must be foreseen. If they are foreseen, why are they different to any other event that may occur? We are importing part of the phenomenological vocabulary into the theory. Errors are just a particular class of inputs that need to be handled: their status as errors is irrelevant at best, but quite likely misleading.

Correctness is a safety property, but there are also liveness considerations that need to be met. Every relevant schema in the specification must also be computable within the implementation. An experiment that is inconclusive in the implementation, but conclusive in the specification reduces the liveness of the implementation. The classic example of this is non-termination: procedures in programs that fail to terminate will never *refute* the theory, but will render the artifact unfit for purpose if the specification relies on such a schema embodied in the procedure. This amounts to the statement that our theories must be sufficiently scientific, in the Popperian sense.

Clearly we require some additional properties of our meta-theory, such as its soundness, relative to some intended class of interpretations. We would also like completeness. In practice, however, we can only ask for completeness if certain expressibility constraints are met by the language, and also if we allow oracles from the specification and implementation domains to provide tautologies for use in derivations.

There are several general strategies that can be adopted in solving this problem of proving correspondence. Two important classes can loosely be called generation techniques and verification techniques.

In a generation technique, we seek to generate an implementation from a specification. We might, for example, seek to transform our specification into an equivalent presentation: effectively this is using a computational schema in our meta-theory to compute a new presentation given an existing one. Rather than attempt a strict equivalence, however, it is common to seek a refinement [Mor90] (or reification[Jon86]), for this is sufficient to meet our condition concerning refutation. In addition to transforming it is also common to add extra information to the new presentation in some form or another. Obviously this extra information in the implementation cannot be shown to arise from the specification, in which it was absent, so we should expect to have to test these properties with a view to refutation, unless we can prove non-interference with the phenomenological theory.

A rather particular instance of this generative process is provided by the use of constructive logic to generate the implementation as part of an existence proof associated with the specification. The exemplar of this is the use of Martin-Löf type theory[ML82].

Verification techniques use a computational schema in our meta-theory, but this time one that accepts two presentations and produces the proof of a theorem which we can interpret as a statement of correctness, either using equivalence or refinement. This approach is often criticised, as the task of producing the proof is usually lengthy and complex. This observation is based on a misunderstanding of the scientific process, however, for the proof will actually be generated by an exploratory process during the course of program construction, and not generated after the event. It is the scientific reformulation of the proof that may appear complex and lengthy, when viewed holistically, but it can be argued that the steps involved are no more complex than the task of writing the program. If we want a rationalisation of correctness made public, and if we want it to conform to standard scientific ideals, then we must accept the consequences. The viewing of the process as an extension of exploratory discourse also allows us to see that the proof can help us to write the program, for we can explore ahead, see what intermediate results are required for the proof to succeed, and engineer our theory presentation accordingly.

The adoption of the mathematico-scientific approach, with its instrumental use of theories, amounts to reducing claims about specifications and implementations to claims about the domains in which they are to operate. We deduce that the program is correct with respect to a specification by assuming properties of the environments in which they are to be interpreted. Some of these claims will be restricted to implementation details, such as the assumption that a programming language will be implemented to conform to its Hoare logic. Others will be claims that must hold about both the implementation and specification domains (such as properties of natural numbers). It might be argued that such claims should have been identified in the contractual boundary for the specification domain, and so can be taken instrumentally, but in practice the proof is likely to throw up lemmas that have been taken implicitly. Many of these lemmas will form part of common sense theories, but it is suggested that the engineer ensures that this sense is indeed common on both sides of the contractual boundary. An example of such a case[5] is the lemma that if m buildings contain between them n rooms then $m \leq n$. This is not always the case, for a building that is extended may have just one room after the extension, but the extension itself may be considered as a separate building. Thus two buildings may contain just one room. In the course of a design activity, one software engineering consultant discovered that a particular company used three different notions of the term "building" in its accounting procedures. Individuals in the company could variously consider a single structure to comprise any number of "buildings". Without this analysis, we are faced with the Biblical paradox "In my Father's house there are many mansions" [John, XIV, 2].

There are many possible candidates for theories within which correctness proofs can be carried out. This is not surprising, for we would expect different theories to arise from different pairings of specification and implementation styles and formalisations. Typical of the classes of theories for sequential systems are Hoare logics[Hoa89], predicate transform techniques [Dij76], refinement calculi [Mor90], algebraic approaches based on morphisms [EM85], denotational semantics [Sto77], and term rewriting systems [EM85]. The correctness of concurrent systems has been tackled using formalisms such as Milner's CCS [Mil80b], UNITY [CM88] and temporal logic [Hai82].

As well as discussing particular techniques, it is possible to attempt a general theory of correctness issues, by constructing yet another layer of theory. The most promising candidate for formalising this theory to date has been category theory, which has been used extensively to explain correctness aspects of the theory building approach when expressed in algebraic terms [Bur80]. This formalisation adds little to our discussion of curriculum design, and will be inaccessible to most practitioners, and so will not be included here.

We should not assume that the adoption of mathematics in order to carry out proofs of correctness is universally accepted by the software engineering community at large. Although many years have passed since the famous "debate" between De Millo *et al* and Dijkstra [DLP77] [Dij78], there is still a long way to go before the community is able to embrace formal correctness proofs as a viable option. Reaction to the imposition of such approaches for the development of safety critical systems, for example, shows clearly that there is still a wide gulf between those practising software engineering in the large,

[5]This example was provided during discussions at the ISTIP89 conference.

and those demonstrating the techniques in the small. The key to bridging this gap must lie with the education of software engineers, both in the continuing education of those currently practicing, and also the initial education of those about to enter the profession.

4.6 The Formalisation of Theories

A topic of continuous debate in Software Engineering is the degree of formalisation that should be used in our theory presentations. This debate is just a particular instantiation of the discussions on the formalisation of theories found in the Philosophy of Science. There is little to be gained by repeating all of this discussion here, but we should make a few salient points that are necessary for our task of curriculum design.

First, we must note that there is a difference between *formalisation* and *axiomatisation*. The latter we will take to be the presentation of a theory as a formal system, with the aim of carrying out purely syntactic deductions. Formalisation, on the other hand, whilst including axiomatisation, also includes the semantic techniques of using mathematical models to reason about systems. Predicate calculus, for example, provides a purely formal system, that can be used axiomatically, but if we allow the importation of semantic results such as tautologies and equivalences we do not need to work entirely within the axiomatic framework[6]. It is important to note that we can accept formalisation without insisting upon axiomatisation in all cases. Many of the arguments offered against so-called formal methods are valid only against axiomatisations inherent in particular approaches.

Formalisation is not a pre-requisite of science, only of the mathematico-scientific approach: the logic we need for refutation can be the logic of any scientific discourse, formal or informal. It might be argued that inter-subjectively testable refutations are more easily found using formal presentations, but this is a methodological concern, not a logical pre-requisite. Indeed, the desirability of formalisation in science is by no means universally accepted.

Suppes, a supporter of formalisation, asserts that "the ultimate reason for formalisation is that it provides the best objective way we know to convince an opponent of a conceptual claim" [Sup68, page 663]. Dijkstra makes a similar claim in defence of formalisation in Software Engineering:

> "Eventually a nice formal treatment is always the most concise way of capturing our understanding and the most effective way of conveying the argument with all its convincing power to someone else." [Dij78, page 15].

Both of these claims are unscientific, in the sense of being unrefutable. They both hinge on terms such as "ultimate", "eventually", "best" and "nice". In addition, they both

[6]The fact that completeness and consistency allow us to formalise these semantic importations does not make our formalisation axiomatic. In this case the two approaches are equivalent in power.

introduce the receivers in the discourse process, and yet say nothing about them. It seems unlikely that these claims can be substantiated if read as universally quantified over all individuals, and in the absence of the required properties of these receivers they really say very little. The importance of the claims, however, becomes apparent if we consider them as methodological dogma, for then we must ask how claims can be brought about. What are the properties of "nice formal treatments"? What must we instill in software engineers during their formative education so that they are suitable receivers of such discourse, and how do we achieve this so that they can use mathematics for scientific and exploratory discourse as well?

Even if we accept that under certain circumstances there may be advantages in formalisation, these advantages may be outweighed by the disadvantages, especially if we adopt axiomatisation. Hempel, for example, asserts that:

> "Whatever philosophical illumination may be obtainable by presenting a theory in axiomatic form will come only from axiomatisation of some particular and appropriate kind, rather than just any axiomatisation or even an especially economic or elegant one." [Hem70, page 52]

That is to say, in choosing how to axiomatise, we need to make theoretical decisions in performing our abstraction. Analysis carried out within the chosen axiomatisation will reflect these decisions, as well as those inherent in the formalisation itself. This echoes the more general point made earlier, that decision to delay the introduction of functional attributes into a theory presentation can still influence the design by choice of presentation. What Hempel fails to observe, however, is that informal presentations suffer from the same problem. The choice of natural language, for example, forces certain concepts on an author: certain African tribal languages, for example, lack many of the concepts fundamental to western culture. The fact that this choice is usually made unselfconsciously tends to obscure the problem. Indeed, one of the advantages of formalisation is that it makes this problem explicit, but only if the engineer considers the method of formalisation as a design decision.

Kyburg puts forward the suggestion that this problem can be resolved by observing that the the choice of axiomatisation system and the expression of the theory are not separable concerns [Kyb68]. The choice of system will actually be made during attempts to axiomatise: in essence, exploratory discourse will take place during which no fixed axiomatic system is being adopted, prior to informative and scientific discourse. If we fail to reflect this exploration during subsequent discourse, then we are intentionally abstracting away from the logic of scientific discovery. Criticising formalisation *per se* for a conscious abstraction seems unreasonable.

Another criticism is made by Schwartz, who questions the rôle of mathematics in science in his paper "The Pernicious Influence of Mathematics on Science". Although he is adopting a deliberately provocative stance, he does make the observation that

> "Give a mathematician a situation that is in the least bit ill-defined–he will

> first make it well defined." [Sch62, page 357].

The danger he sees in this is that assumptions made for the convenience of the mathematician become part of the theory, and moreover the scientist may come to accept these simplifications as part of the theory. He goes on to say

> "The physicist rightly dreads precise argument, since an argument which is only convincing if precise loses all its force if the assumptions on which it is based are slightly changed." [Sch62, page 357]

The physicist is seeking generalisations. The software engineer, however, is seeking only sufficient generalisation to cover an identified problem domain, and requires no more. If the formalisation is too specific then we run the risk of our approach not being scientific (for the task in hand), for the theory will be protected from refutation by auxiliary statements preventing us from experimenting in vast areas of the problem domain. As a technologist, however, the engineer knows that the computer will react in a very precise way: we accept that the program will behave in undefined ways if we violate our preconditions (so much so that professional engineers will adopt defensive programming strategies to limit damage should it occur). In this case, formalisation presents no problem that isn't inherent in the design task. This is a balance that the engineer must achieve.

Zemanek observes that the task of formalisation

> "starts in the middle: we are born into an environment [in which] there are suggested and even self-suggesting formal notions, but at the beginning of any investigation, such notions are not precise enough. Their further development frequently leads to a kind of separation–even of two hostile universes, the informal and the formal universe. In order to avoid friction, tension and fights, it is necessary to understand the virtues and limitations of formalisation and how its embedding in the real essential informal world can be done without harm." [Zem75, page 118].

In order to achieve this understanding we will develop a more detailed account of "theory" in the next chapter, one that admits discussion of this relationship between universes. This account will also allow us to discuss the ways in which axiomatisation, model-based formalisation, and the use of analogies fit into our model of design.

This leads us to conjecture a heuristic to accompany the model, that there must be a trend from the less formal to the more formal as the design of a software system proceeds. We are not going so far as to say that formalisms should be used at the outset in constructing our theory, or that they should only be introduced at the stage of writing code in an accepted programming language. We must accept that there is a need to balance the advantages and ultimate necessity of formalisation with the possible restrictions it may place on the problem being solved and the methods being used. This judgement must remain the prerogative of the engineer concerned: prejudging the issue allows the engineer to avoid loss of innocence, and is as flawed as telling a doctor what drugs to prescribe before the patient has been seen. Formalisation must be seen as a tool that can be used: the "can" is important here, for just as the engineer should not be allowed to escape the loss of innocence through imposed dogma, so the route of ignorance should be denied. Choosing not to use formalisation because of inability to handle the tool is on a par with the surgeon who chooses not to operate because he is unable to use a scalpel. One cannot question the decision not to operate, but one might question the right of the individual to the title "surgeon".

The fact that we must move from the informal to the formal, and that this move is forced upon us not by ideology, or even methodology, but by the nature of the artifacts concerned, should be noted. If we ignore comparatively minor problems such as component failure and interference from external devices, and concentrate on an idealised machine, the computer embodies the atomic world of early Wittgenstein's Tractatus. The behaviour of the idealised machine depends only on those bit patterns in its memory, and those in its environment to be presented on its input lines. These bits are interpreted in bit-functional fashion, analogous to the truth functionality of propositional logic. The emergent behaviour of the machine is determined completely by these values (even if it is sometimes convenient to view the process as non-deterministic by abstracting away from detail). Moreover, it is this property of the machine that allows the mathematico-scientific approach to be used so successfully in our problem domain. The rules governing the bit-functionality of the system must be known, since these were the driving force behind the design of the machine itself. The civil engineer's bridge cannot be so easily separated from its environment: its behaviour is not simply that emerging from consideration of its components together with the inputs modelled by the engineer. Factors too complex to model must be taken into account by the inclusion of safety features, using experience and judgement[7].

We must beware, however, of the fallacy of moving this atomism from the target machine to the problem domain. Seeking to see the world at large in such simplistic terms is unlikely to be feasible, let alone practical or helpful. The challenge facing the software engineer is to identify appropriate ways of making this possible for particular, highly restricted, problem domains. This challenge is forced upon us by technology, but there is no need to restrict our exploration of the problem domain to theories expressible in these terms, and no need to dictate when the move to formalisation should take place.

We must also beware of the fallacy of equating formalisation with cooling, in the sense of excluding noise. It is perfectly feasible to think in very warm terms with mathematics, if we are sufficiently fluent, and prepared to use exploratory discourse. It is also quite

[7]In fact, computers are not as easily isolated as we might like to think. A well known brand of ioniser, marketed as ideal to combat fatigue from using computer VDU screens, has the interesting effect of causing the Macintosh Computer to "type by itself". This phenomenon, a wonder to behold, has never been modelled in any specification, formal or otherwise, that the author has seen! It is convenient, and not too misleading, however, to proceed for these discussions as if total separation can be achieved. The task of achieving this separation will be left to the electrical engineer.

easy to think in very cool terms with natural language. It is, we would argue, this false association of cooling with formalisation that leads to the unproductive divisions between the so-called formalists and the non-formalists.

4.7 Summary

In this chapter we have introduced a simple model for the production of software systems. This model has as its basis the idea that system design is primarily the development of theories. The adoption of theories as the central theme of the process is not obvious. We could have considered using specifications or programs as the unifying theme, but then we would have had to forgo the benefits of access to other bodies of knowledge outside of computing.

We should note, however, that this model is not intended to capture only "good" design practices. It can equally well describe the process of sitting at a terminal and solving a problem directly into assembly language. Here there is only one theory presentation. The phenomenological theory is expressed directly in terms of a very computationally biased system: the diagram is constructive, in that the function of the artifact can be deduced from the program and a sufficient knowledge of the theory of the programming language, but the mode of expression is likely to obscure the function of the system.

Like all models, this one is not intended to be a perfect reflection of what actually happens when software design is taking place. More importantly, it is not intended as an exemplar of what should take place. In reality, all projects are likely to progress in different ways. Occasionally an excellent fit with the model might be observed. More often things will be noted that conflict with, or are ignored by, the model. The important thing, however, is that we have established a model so that conformance and deviation can be discussed.

Throughout the rest of this thesis the model outlined above will be refined and used as a basis for discussing aspects of the curriculum. It is crucial, however, that the role of the model should be appreciated. The discourse contained in this thesis is largely exploratory, and not scientific. The discussions it contains, therefore, are enabled by the model, and do not rest upon it as a premise. Refusal to accept the model should lead to progress, as reasons are given, and development, as new models are proposed. Rather than attempt to refute the model here, however, we will depart from Popper and ask the reader to use it, warts and all. This adoption is likely to lead the reader to a catalogue of disagreements by the end of the thesis: such is the nature of exploratory discourse.

Chapter 5

Theories, Models and Analogies

"We have found that where science has progressed the farthest, the mind has but regained from nature that which the mind has put into nature. We have found a strange footprint on the shores of the unknown. We have devised profound theories, one after another, to account for its origins. At last, we have succeeded in reconstructing the creature that made the footprint. And Lo! it is our own."

Aurthur Eddington

In this chapter we will seek to refine our notion of "theory", and to introduce the ideas of models and analogies. This is a non-trivial undertaking, and we will be highly selective in the views we present: no general survey of the issues involved can be undertaken in a document of this size. We will not consider, for example, wider uses of the term theory, such as "the theory of physics", which are intended to denote a wide range of concepts, including a number of more localised theories. This is the usage Gries intends when he refers to "the theory of programming". We will concentrate on individual theories that are intended to refer to a localised clustering of phenomena, and that will serve as specifications or programs. It should be stressed that there is no expectation of answering the question "what is a theory?". This has exercised the minds of the greatest philosophers of science for generations, as evidenced by the famous symposium held in 1969 and the debates that have followed it [Sup77], and there is no point even thinking we might come up with an answer. In fact, the current view seems to be that we must accept many different notions of theory; all useful in their ways, but possibly irreconcilable.

In pursuit of our task, we require a view of theory that is powerful enough to encompass the results of a wide range of activities undertaken by software engineers. Humans, who will be working with these theories, are flexible enough to cope with such a complex domain. We also require a notion of theory that can sensibly be embedded in a machine. The current state of technology suggests that this will need to be a far more rigid type of theory. Anthropomorphically, we might say that if the machine is to "understand" that it is being presented with a theory, the notion of theory involved must be rigidly defined and quite simple. Our computer's view, for example, must admit formalisation, whereas the engineer's earlier views need not. Thus, as we proceed with our development process, not only must our theory presentations become more formal, but there is also the possibility that our understanding of the term "theory" must become more simplistic and precise. In addition, we must accept the shift in the status of our theory. At the outset, the theory is tentative, as the engineer explores the problem and attempts to construct a suitable presentation as the basis of contractual boundaries. There is every expectation that the theory will be refuted, coupled with the hope that such refutations will lead to an improved understanding of the problem. As the design process draws to a close, however, the theory becomes instrumental and is accepted as "sufficiently true" by the customer. There is no *need* therefore, for the final presentations to be efficient aids to understanding, or to facilitate refutation. Of course, if the customer subsequently decides to negotiate a new contract, and the engineer decides that some components of the design, such as parts of theories, can be reused, then a lack of explanatory power in an implementation will be a drawback, and the engineer may want access to other presentations used during the design process. For this reason, it is sometimes suggested that we should not lose this explanatory power as a design proceeds, but embed explanations into code in the form of comments.

Just as our engineers must choose programming languages for implementation, so too they must choose appropriate notions of theory, and suitable presentation methods, for use during the design stages. *Imposition* of any "methodology" relieves the engineer of this responsibility; ignorance of suitable tools and techniques renders the engineer incapable of discharging this responsibility adequately and professionally.

We will start our discussions by presenting an overview of the properties we might expect theories to possess. We will then present two views of theories, first, the received view, and then the semantic conception. The former is no longer widely accepted in the Philosophy of Science because it leads to many problems that in turn lead to a convoluted process of shoring up the view, and inhibiting applicability. We will attempt to apply it to the latter stages of design, however, firstly to see if the criticisms still apply, and secondly because many references to theory in the Software Engineering literature are based on this view. The received view does not admit discussion of the earlier stages of design, for precisely the reasons it has lost acceptance in the Philosophy of Science, and so the semantic conception of theories will be introduced. This seems to fit more naturally with the model of design outlined in the previous chapter, and also allows discussion of concepts such as "model" and "analogy", a discussion necessary for our pedagogical purposes.

Once we have discussed theories generally, we will briefly review theory presentation techniques that are commonly used in Software Engineering. This will not be an exhaustive treatment, since much of this material can be found elsewhere, but we will attempt to show how the techniques can be discussed within the semantic conception of theories. We will also introduce the proof obligations that accompany the use of these techniques when used as part of a theory construction process.

It is appropriate at this point to remind the reader that we are not attempting to *define* terms, far less to dictate how they should be used within Software Engineering. Our intention is simply to come to a better understanding of what constitutes a theory, and how theories can be used by bringing them into contact with problems. It is possible, indeed likely, that by the end of this chapter the reader will feel less comfortable with the terms than before reading it! This is a reflection of challenges being made to the intuitive notion of theory currently held: no apology is made for this. It is an important part of our task to challenge current educational practice, and if this involves disturbing intuitively held notions, so be it. We would hope, of course, that as well as being destructive, by proposing the semantic conception of theories, a framework will be provided within which the reader can subsequently reconstruct an improved understanding. This reconstruction can only be initiated here, however, and not completed. Moreover, the reader will probably want to explore other views of theory before attempting to reconstruct within this view.

5.1 Properties of Theories

Ryle builds a powerful analogy between theories and pathways [Ryl49, pages 272-275]. We will adopt this analogy, and extend it. To have a theory, according to Ryle, is to be aware of a pathway from place to place in such a way as to be able both to use the path, and also to explain its whereabouts to others. During the exploratory phase of theory building, the pathway is being constructed, and the builder must be prepared to stamp up and down the path to establish it. Paths may start out, then become bogged down, or run up against obstacles that cannot be overcome: in such cases they may be abandoned, but probably only if a better path can be found. Once a path has been established, by sufficient stamping, it can be mapped out and made available for others to use. It no longer needs to be established, although a new user may need the map to make use of it until familiarity is achieved. It becomes an instrument, rather than an artifact under construction. It may still unexpectedly subside under use, of course, and then either be abandoned, or need to be shored up before it can be used again. Users may continue to walk subsided paths at their own risk, of course.

In constructing a complex of paths it will almost certainly be more convenient if existing paths can be used instrumentally. This might include constructions that will become redundant once the complex of paths has been built: the logistics of major roadworks shows this clearly. We might also want to include existing paths into our network. Thus our theory builder is also a theory user, both in terms of structure and also infrastructure, so our software engineer needs to be a rambler as well as a pathbuilder. Even if the engineer has a clear goal in mind, the analogy of rambling seems more appropriate than orienteering, because routes using existing pathways are likely to be more efficient if they can be found, even if less direct. The re-use of existing components seems appropriate to theory building as well as programming. Sommerville uses a similar analogy when he talks about the engineer navigating in information hyperspace [Som87]. We are extending the idea to a *theory hyperspace*, because we want to include explication alongside specification and implementation.

What properties do we expect these theories to have[1]? First, we note that we usually admit the possibility of a theory being false. We would not normally call the statement "Paris was the Capital of France in 1989" a theoretical statement as it is just a statement of fact. A theory that comprises just a collection of facts would not normally be accepted as a theory in the conventional sense. Some theories have become so familiar with instrumental use, however, that it seems impossible that they are wrong: possibly we now take them as definitional. We will still accept these as theories, but insist that at least a glimmer of doubt exists, even if the user rarely makes this doubt explicit. Equally, however, we usually entertain the belief that our theory may be true. In short, we accept that our theory is a conjecture.

We also expect our theories to lead to a better "understanding". This may mean several things, such as their acting as explanations of some phenomena, acting as interpretations into a better understood domain, providing a calculus for prediction, or exposing structure. One important aspect of this increased understanding is the use of mechanistic explanations. This is a vague, but important, idea, as Gregory observes:

> "Most curiously, it is difficult to find out just what scientists or philosophers (or indeed the common man) take 'machine' to mean, and yet mechanistic explanations are generally supposed to be the most, or even the only, acceptable kind of explanation in science." [Gre84, page 73]

Gregory goes on to to distinguish between the machine, which embodies purpose or function, and the mechanisms it contains, which embody only causality, although they may assume functions when used. Using this terminology, we can state our theory building aim as finding a mechanistic explanation, or theory, with the purpose of using particular schemas, where the mechanisms upon which the machine depends are provided by our computing devices. This mechanistic explanation alone may not be sufficient, however, for if we wish our theory to evolve we might require explanations of a higher order, where the mechanisms are easier to reason about. Moreover, we are likely to construct such explanations *en route* to our final implementation.

We also expect a theory to tell us what is the case: to make some assertions or propositions. This is not enough, however, for we usually expect some coherence to these assertions. A random collection of propositions is not normally thought of as a theory. We might, of course, discover some model for the collection, and proclaim this collection as a theory of this model. In general, however, we will not call something a theory unless we have some idea both that there is a model, and also that we know roughly what it is. This is a restatement of one of our proof obligations, namely that all theories must be consistent otherwise no such model will exist. Note that again this presupposes selfconscious activity[2]. Inconsistent theories pose no problems for the unselfconscious

[1]This section draws heavily on Achinstein [Ach68, pages 122-129].

[2]For a detailed discussion of the relationship between consistency, completeness and selfconsciousness,

designer, who will simply proceed in ignorance until problems are noticed, then resolve them locally. Some of the assertions in our theory will have different status from others. Moreover, some assertions will probably not be made explicitly, but will accompany our choice of presentation technique. Adoption of equality, for example, will probably not be accompanied by the axioms for equivalence relations, but these will be used; similarly axioms for our deductive apparatus and some of our data types will be assumed. In essence, we will adopt a research program [Lak70], and work within this. As well as our assertions being part of the theory, we also expect some form of logical closure, allowing theorems provable within our adopted logic to be accepted as part of the theory.

We also have the expectation that our theory has some purpose, in the sense that it helps us to meet some identified goal. Thus a random collection of assertions, even if clustered around some phenomenon, will not comprise a theory. Moreover, the arbitrary conjunction of two theories will not, in general, comprise a theory. The conjunction of the theory of natural numbers together with the theory of the digestive system in dogs will not comprise a theory unless we are setting out to study something like the statement that dogs' digestion times are related to the Fibonacci sequence.

Achinstein also suggests that we should expect a theory to be maximal, in the sense that if we already have a general theory we would not usually call the restricted application of that theory a theory in its own right. Whilst this seems sensible when viewed as restriction, it is less obvious that we would require something to lose its status as a theory due to subsequent generalisation or enrichment. We will not insist on theories being maximal, for many of our specification techniques are based upon a process of enrichment, and it is convenient to think of the building blocks as theories throughout. We do not only want our theory of natural numbers to become only a part of the theory of stacks of natural numbers should we present one; we also want it to preserve its individual identity.

5.2 The Received View of Theories

The "received view" is the name given to the view of theories generally held, in some form or another, until the second half of this century. This view has developed through a number of reformulations, each one trying to overcome some of the problems being raised by the previous version. The received view is accompanied by the assertion that all theories can be represented in a canonical form (not necessarily that *they are* all presented in this way). Suppe develops various versions of this canonical form that have been accepted during its development. Presented below is his formulation of the final version given much credence in the literature [Sup77, pages 50-53]. He notes that a theory, according to the received view, comprises:

see [Smu87].

- A first order (possibly modal) language, L, and calculus, K, in terms of which the theory is presented.
- The constants in the language which are partitioned into two classes, V_O, containing the observable terms, and V_T, containing the theoretical, or non-observable, terms.
- L and K give rise to three sublanguages and claculi as follows
  - L_O, which contains V_O, but no quantified forms or modalities, and no terms in V_T. K_O is just the restriction of K to L_O, where any terms arising not in V_O are defined explicitly in K_O. Furthermore, K_O must admit at least one model.
  - L'_O extends L_O by allowing quantifiers and modalities. K'_O is the restriction of K to L'_O
  - L_T is a restriction of L to remove those terms in V_O. K_T is a restriction of K to L_T.

  L is not simply the conjunction of these sublanguages, however, because there will usually be many mixed terms.
- L_O is given a semantic interpretation in which
  - The domain comprises observable events—all objects, properties and relations in this domain must be directly observable.
  - Values of all variables used in L_O must be interpretable as expressions in L_O.

  These interpretations of L_O, and hence of K_O, are partial interpretations of L and K, and will become interpretations of L'_O and K'_O when suitable interpretations of logical terms are added.
- Theoretical terms are partially interpreted by theoretical postulates (axioms in which only terms from V_T occur) and a finite set of correspondence rules, C, which involve terms from V_T and V_O, but no additional, extra-logical, terms.

Before considering the empirical significance of the received view, let us observe that there is a degenerate case of interest. If V_O is empty, the theory ceases to be scientific, in the sense that clearly no observations can be made to refute it. Our terms will all be groundless, and we are committed to working purely theoretically. In this case, we are working in a domain that mathematicians and logicians call model theory [Bri77]. This sense of "theory" is adopted by formalists in Computer Science, particularly those interested in algebraic methods [BG77]. It does not allow us to discuss how theories are grounded in observations, however, so what purpose can it serve? The answer is that once we have made our phenomenological theory instrumental, refutation ceases to be an issue, so we can consider all terms as theoretical without concern. The theory can always be re-interpreted to include observation terms, and thus provide a grounding at a later stage, such as might occur during acceptance testing. This is also the reason that Hoare can embark on the mathematico-scientific method. The theory has been made analytic rather than synthetic, to adopt Kant's terms [Kan29]. We will not endorse the view that analytic theories are meaningless, however, simply that their meaning comes from the relationships imposed between theoretical entities. In particular, we can provide a meaning in terms of other theoretical objects. Thus we can interpret theories as algebras, or classes of algebras, simply by seeking out structures with corresponding relationships between terms, that is, by identifying morphisms between the structures. Once our theory is taken as instrumental, therefore, we can utilise mathematical models with impunity. We should observe that this need not only be during the later stages of design, for we might well consider theories instrumentally during exploratory discourse (during *what if...* type discussions).

Let us suppose that we want to continue to think of our theories as empirical, in the sense that we want some correspondence between their terms and observables in the domain of application. How easily can this be done with the received view? We should note that one of the reasons for the abandonment of the received view was a feeling that this question is inherently unanswerable in true scientific practice, using the given canonical form, so we should not expect any decisive answers here. It seems that a number of fairly arbitrary choices have to be made. In particular, how might we distinguish between observables and unobservables? We will consider just one simple example to illustrate how this might be done, then move on to a richer model where the problems do not arise.

Consider a theory of stacks. First, let us note that such a theory is not really a theory of stacks, in the sense that it tells us how a cluster of stacks might behave collectively, rather it is a theory of "stackness". It expounds those properties we might expect of an object that we are happy to call a stack. Since we need such a theory, and are not happy just to observe stackness as primitive, we can argue that stackness cannot be an observable concept: therefore any expressions that denote stacks (that is, objects we are asserting to have these properties) must be theoretical terms. For the sake of examples, let us suppose that we are interested only in stacks of natural numbers. Are terms involving only numbers theoretical or observable? We might consider that numbers are primitive enough for our customer, who is our archetypical observer, to observe directly. We could, of course, note that numbers have their own theory, and it is a strange notion of theory that allows numbers to be observable, but stacks to be theoretical simply on the basis on customer experience. This is precisely the problem that faced philosophers of science, but they phrased it in terms of instrumentation, rather than customers. Things observed through complex instrumentation (with corresponding theories) were considered theoretical; things observed by eye were observable, even though the eye could be considered an instrument of observation.

If we accept that stackness is a theoretical concept, however, whereas numbers are observable terms, then we can explain some of the interesting features of abstract data types quite coherently. We can consider an object hidden inside a box, with a limited number of actions that can be performed, and a number of observables available to us via experimental method. Consider the theory of stackness, expressed in OBJ, shown in figure 5.1

```
OBJ
             STACK / NAT BOOL

SORTS
             Stack
OPS
             newStack:                         -> Stack
             push:              Nat Stack      -> Stack
             top:               Stack          -> Nat
             delete:            Stack          -> Stack
             isEmpty:           Stack          -> Bool
VARS
             s:Stack
             n:Nat
EQNS
             (top(push(n,s))=n)
             (delete(push(n,s))=s)
             (isEmpty(newStack)=T)
             (isEmpty(push(n,s))=F)
JBO
```

Figure 5.1: A Theory of Stacks

We can note that the term $delete(s)$ is a theoretical term. It is groundless, unless we add a correspondence rule relating it to observables. This is done with the equation $delete(push(n, s)) = s$. We should note, however, that we are obtaining only a partial interpretation of "stackness": we are constraining the meaning of the term to objects that have certain properties pertaining to observables, but we do not have complete knowledge of what is actually inside the box. We can add meaning to our theory, with more correspondence rules, such as grounding the terms denoting stacks in terms of lists, for example, but we expect this extension to be conservative. Note also that terms that are ungrounded by any rules, such as $delete(newstack))$, remain meaningless. If we wish to be able to observe some behaviour corresponding to such terms, we must provide a semantics. Then, of course, it would cease to be meaningless, for our theory of stackness would be asserting that deleting the top of an empty stack is was a meaningful thing to do.

If we wish to consider the design process in terms of the received view, we can attempt something along the following lines. First, we build a theory with observables that correspond to the customer's perception of observables in the problem (objects such as invoices, payroll numbers or sensor readings). We then attempt to refute this theory by appeal to scientific practice, using only the observables we have established. We then produce a new theory in which we allow only observables available to some machine (typically a high-level virtual machine, rather than a primitive bit-functional one), and where all theoretical terms are grounded in terms of these. Thus we express stackness in terms of arrays and pointers, for example, as well as the operations we wish to perform. These will be theoretical terms if the theory is viewed by the user (for arrays and pointers will not be observable) but observable terms for the machine. The machine will then *conform* to this theory, treating assertions as instructions, according to some meta-theory. In OBJ, for example, the equations will be interpreted as left-right rewrite rules.

Such a discussion is of little benefit to the software engineer faced with the task of designing a system, however, for the view being taken of theories is too far removed from the intuitive requirements we have laid down for them. The received view is an extreme one, intended to provide a rationalisation in terms of a canonical form. This canonical form does not seem to sit well with Software Engineering, unless we restrict attention to the instrumental use of theories, in which case it degenerates to mathematical model theory. Rather than attempt to rescue the received view, we will turn our attention to the semantic conception of theories, which has been developed in more recent times to overcome these limitations.

5.3 The Semantic Conception of Theories

Just as the received view was proposed and revised over a number of years, so too the semantic conception is more a programme of ideas than a static entity. We will not attempt an historical reconstruction of this approach, but simply present one of the more recent expositions of it [Sup89]. Central to the semantic conception of theories is the creation of an additional layer between theories and "reality" that will allow us to talk of idealised models of the behaviour of real systems. The advantage for us is that customers' problems are frequently already posed in terms of such models, for their information systems and control systems are defined in terms of abstractions. Customers are not names and account numbers, but flesh and blood, and controlled plant is not just a set of functions, but metal and plastic. Although the case for utilising idealised worlds has to be made quite carefully for *natural* scientists, this case is unnecessary for *information* scientists, as the discipline itself is inherently concerned with a layer above reality. No justification of the approach need be given other than the justification for information science itself, and that will be taken as self-evident.

The semantic conception admits a wide-ranging discussion of models, and allows for several presentations of the same theory. It also allows us to discuss the fact that models and theories are not cleanly distinguishable. Very often a theory is interpreted as a model, and a model is taken as identifying a theory. This accords with scientific practice: many scientific theories are posed as models, and many theories end up being used as models. Whereas the received view rationalises this confusion out of the discussion, by considering only restricted canonical representations, the semantic conception brings such issues to the fore, which is far more useful for our pedagogical purposes.

Central to the semantic conception is the view that theories are not linguistic entities (as in the received view) but extralinguistic. They may be formulated or presented in a number of different linguistic systems, but changes of formulation do not change the underlying theory. With this change of emphasis, theories can now be viewed as models for their linguistic formulations. These models can be thought of as formal structures acting as interpretations of various theory presentations, and these presentations need not be equivalent. We can have, for example, partial formulations of theories, resulting in a number of presentations of different aspects of the theory. In this way we can utilise CCS, VDM and OBJ to present specifications of a system, and these specifications will not be equivalent, yet we can still accept that we are building just one theory. In architectural terms, the theory captures what the building is to be, but it may be formulated using various techniques such as drawing elevations, specifying services, and providing energy equations. The theory is constrained by the totality of these formulations. Theory thus becomes a unify force in our design.

Rather than deal with the behaviour of real systems directly, the semantic conception suggests we consider abstract "physical systems", that are idealised versions of real world phenomena. We will use the term "idealised systems" rather than "physical systems" here, for it seems counter-intuitive to introduce the term "physical" in relation to information systems. We identify the intended scope of the theory, and then extract just those parameters in which we are interested[3]. This makes the assumption that useful problems can be tackled by consideration of behaviours governed by just those parameters, and that for the purpose of solving these problems the effects of all other parameters are negligible. This is precisely what happens when physicists consider frictionless planes, point masses and perfect spheres. It is also what happens when the software engineer assumes that employees can be considered as personnel numbers, or that sensor readings are just real numbers. We cannot subsequently ask how many toes an employee has, or how long the sensor reading takes to stabilise. The idealised system can be completely characterised by the values of the identified parameters at any given time. We will consider these parameters to constitute the state of the system: then the behaviour of an idealised system can be represented as a set of sequences of states. The behaviour of an idealised system under a given set of initial conditions will correspond to a subset of the behaviour, comprising just one sequence if the system is deterministic.

The task of our theory is to constrain the behaviours of idealised systems so that they correspond to how real world phenomena would behave if behaviour were determined only by the parameters reflected in the idealised state. Implicit in this is the notion that the theory identifies what configurations of state are possible, and also what states can result from a given starting state, the latter being deterministic or non-deterministic. This clearly fits very neatly with our assertion that it is useful to consider programs as theories, as a standard view of programs is that they determine machine behaviour through sequences of states.

This constraining is achieved by taking the theory presentation as determining a class of

[3]Analysis of this process is termed measurement theory, and will not be considered further here: more details can be found in [Rob79].

idealised systems, and imposing some structure on them. This structure can be viewed in a number of ways, typically as relational, set theoretic or state-transitional mathematical objects. In fact, these are surface differences, for the three views are easily reconciled. Theory presentations can be construed as comprising a number of laws, constraining the behaviour of the idealised system. These laws might be given as *laws of succession*, describing how the states are related over transitions, *laws of coexistence*, describing how states may be regarded as equivalent, or *laws of interaction*, describing how the system behaves in terms of other theories. These three types of laws can be seen as closely related to state based, algebraic and process based specifications respectively.

We should observe that our theory presentation admits amplified usage, that is, it is capable of referring to more than one thing. This is necessary because we wish our assertions to refer not only to one or more idealised system, but also to be interpretable as the theory itself, or as real world phenomena. We expect to be able to interpret a presentation as laws governing state transitions, constraints on sets of states, or assertions about the real world. A theory of stacks, therefore, should be interpretable as specifying relations between states that must hold, transitions that a typical object might undergo if it is indeed a stack, and whether a real world object is stack-like. We also accept that our presentations can be partial, in the sense that they might refer only to some of the parameters in our idealised world.

We should point out at this stage that the semantic conception, as generally presented, assumes that systems vary with time. This is understandable, because most physical systems are thought of in this way. For our purposes, however, introducing time may be an unnecessary distraction, for we are frequently interested in transitions that arise as results of events such as issuing commands. This need not present us with a problem, for we can observe that what the physicist thinks of as "time" is really nothing more than a sequence of events, typically clock ticks, so we can interpret our sequence of events as denoting the passage of time. Alternatively, we can embed our sequence of events into the state of the idealised system, then rely on the passage of time to invoke their execution. Thus we can internalise or externalise the events driving our system. Moreover, events themselves may be considered as taking parameters: these parameters may be kept in the state, or each parameter can be considered as defining a different event. By a similar argument we can internalise or externalise our programs and data.

Let us illustrate this view so far, as it might apply to the "programming" end of the design process, by using a simple computer architecture, Landin's SECD machine [Lan64]. We observe that the abstraction of the machine to a state comprising a stack, an environment, a control list, and a dump, is inherent in the problem. Our idealised system is thus present in the problem situation. We will see later that there are several problems that can give rise to this abstraction. We could, of course, seek to reformulate the theory utilising a different state model. We will start with a general theory of the machine itself. This comprises laws of succession such as

$$(S)\ (E)\ (LDC\ x\ .C)\ (D) \longrightarrow (x.S)\ (E)\ (C)\ (D)$$

indicating how individual transitions occur over a clock tick (note that we have internalised our program). It defines a relation between pairs of states. We could equally well just consider an SED machine, with events such as

$$(S)\ \ (E)\ \ (D)\ \xrightarrow{LDC_x}\ (x.S)\ \ (E)\ \ (D)$$

We also have laws of equivalence, such as those giving the algebra of objects on the stack.

$$((3+4).S)\ \ (E)\ \ (C)\ \ (D) \equiv (7.S)\ \ (E)\ \ (C)\ \ (D)$$

The relational structure forged by this theory presentation is our theory. The presentation also gives rise to a number of *theory induced* idealised systems, that is, a set of possible execution sequences. What are we capturing with these induced systems? There are several things we might be trying to theorise about. Let us assume that we are trying to capture the execution of LISP S-expressions, which are thus taken as third-world empirical objects. We need to show that a suitable correspondence exists between these expressions and states identified in the idealised system. Each S-Expression to be evaluated needs to be reflected by a state of the SECD machine, together with suitable closures and name bindings in the environment: this is the task of the compiler. Each state with $STOP$ at the head of its control list can be associated with a result, held on the stack. States with control lists not representing S-Expressions, such as will arise after partial evaluation, can simply be given the semantics "evaluating". Any state with an empty stack, and $STOP$ at the head of the control list will represent the null expression. Refutation of our theory at this stage would mean finding an S-Expression that is properly compiled to SECD code, but executes incorrectly. An idealised system now corresponds to the evaluation of a particular S-Expression. Thus our theory presentation gives us as possible models at least:

- The theory which is the relational structure of all possible machine states. In particular, we can always present this as an algebraic structure, using set-theoretic terms.
- The behaviour of idealised machines.
- The representation of phenomena such as the execution of S-Expressions.

We can observe here that our theory also admits as a phenomenological model the architecture of an implemtation of the SECD machine. Thus, via the theory, we can create morphisms between a piece of hardware and the execution of Lisp S-Expressions. This is the key to implementation. The theory building approach applies not only to Software Engineering, but also to hardware design. Adoption of this model of the design process, therefore, will enable us to integrate the curriculum across a wide range of topics, from the "softer" aspects of system design, where exploratory discourse is being used to construct theories, to the "hard" area of logic design. The tools, techniques and modes of discourse used may be different, but we have found a unifying theme for the discipline, as advocated by Gibbs and Tucker [GT86].

The theory outlined above is a theory of SECD programs, but in what sense is an individual "program", just one control list and associated bindings and closures, a theory? Quite simply, it is a restriction of the above theory. In general, a program will admit many behaviours of phenomenological systems dependent on input data (we will consider databases with persistent data as external to the system: they could equally well be considered part of the program), thus a program is a theory determining a restricted class of theory induced idealised systems, and hence a restricted class of phenomenological systems, or computations. Note that in order to take this view, we have violated the maximal notion of theories, that is, we want both the theory of all Pascal programs and also the theory of just one Pascal program to coexist. We could take other views, insisting, for example, that our program is an auxilliary statement, or even that our program is a statement in another theory—the experimental situation—and use laws of interaction to define our behaviours. Such approaches seem artificial, however, so we will abandon the maximal view of theories.

The given example models an inherently functional phenomenon in terms of state transitions, using laws of consequence and coexistence. We should stress that this was only an example, and several styles of theory formulation exist for tackling different kinds of problems. We could formulate a theory of "stackness" for example by laws of coexistence (a typical algebraic specification), by laws of consequence relating pre and post states implicitly (as in Z or VDM), or explicitly (as in Pascal), or even by constructing a state as a number of interacting cells, each of which has its own theory (as in CCS).

5.4 Theory Presentations

Constructing a theory presentation requires at least the following two steps:

- Identification of a suitable state-space to act as the theory. This involves analysis of the problem to ensure that only sensible abstractions are taking place.
- Formulation of laws to restrict the space to those states of interest only, and build the relational structure between these states.

We will delay discussion of selecting a suitable state space, and turn to ask what we expect of a theory presentation language. Clearly we need a language powerful enough to assert properties of our phenomenological system, by capturing properties and relations. A first order logic will usually suffice for this, although we might want to include modalities, to allow the modelling of the passing of time without introducing quantified intervals explicitly, for example. We will, in general, also require extra-logical features, such as the ability to manipulate functions and relations. We will import these into our presentation language as required, just as the physicist feels free to utilise differential equations, for example, without formulating an axiomatic treatment every time.

We will now briefly consider particular styles of theory presentation, and the proof obligations their adoption places on the engineer.

Laws of Coexistence

The style of presentation using laws of coexistence to form equivalence classes of terms is usually termed "algebraic specification" in Software Engineering. This formalisation is founded on the interpretation of variables as components of our state space, and operations as state transitions. Each class is intended to refer to a particular situation in our phenomenological system, but we need to adopt a metatheoretical view here, regarding how our equivalences are to be understood. We could adopt the view that only terms indicated as equivalent by the theory are to be taken as such (an initial view), or we could accept that everything is equivalent unless the theory forces terms to be non-equivalent (a final view), or we could adopt a mixture of the two. The advantage of adopting an initial view is that we can use a theorem of universal algebra to the effect that all algebraic structures interpreted as initial objects of some equational theory are isomorphic to the term algebra of that theory, and consequently to each other [EM85, page 86]. This will allow us to manipulate the theory via its term algebra, or any other initial model, rather than using the theory presentation directly. It is this property that allows us to use set algebra rather than set theory in most mathematical contexts. We will assume initial semantics in all that follows.

It is usual to expect that equational specifications should posses the property that all terms should rewrite, under the equivalence relations, to some canonical form, for this allows us to use the quotient algebra in describing phenomenological systems. If we accept this as a proof obligation, it will be discharged by structural induction over the language used in the presentation.

In addition, we have the proof of consistency as an obligation. This can be discharged by finding a non-trivial model. Clearly the term algebra is such a model, but we will insist that, there are at least two distinct terms in the quotient algebra, and that no two canonical terms are forced to be equal by the theory.

Completeness reduces to the obligation to prove that all terms that should be in the same equivalence class are so placed. Clearly, showing phenomenological completeness can only be done by attempted refutation, or by exhaustive testing, but we can insist that something is said about all possible meaningful constructions in the idealised system. This amounts to the obligation that there should be a set of equations governing the behaviour of all accessors to the state, one for each possible constructor. Stacks, for example, can be constructed, in canonical form, by *newstack* and *push*. Thus we expect two equations governing the behaviour of each accessor, if total, or one for each partial accessor (such as *delete*, which is undefined for *newstack*). If we wish to capture nondeterministic systems algebraically, however, we need to revise this notion of completeness. For a detailed discussion of nondeterminism and algebraic specifications see [Mat90].

If we structure our idealised system into several subsystems, we might find some obligations as result of laws of interaction. Typical of the obligations that might arise here are proofs of conservative extension arising as a result of enrichments, and proofs of type compatibility, in instantiations of generics.

Laws of Consequence

If we present theories using primarily laws of consequence then we have a different manifestation of proof obligations. First, we have to show that our assumption that time can be modelled by sequences of events is well-founded, that is, that suitable sequences exist. In particular, we must show that every event can result in an acceptable state. In a traditional pre-post condition specification, this is discharged by showing that there exists at least one assignment of correctly typed parameters such that

$$(pre - condition \land invariant) \Longrightarrow invariant$$

This prevents an open-ended sequence, where the final "event" does not result in a state within the theory, that is, it shows termination is possible for every event.

It is also sensible to show that all events are necessary, that is, that there exists at least one possible state from which an event can occur. We might argue for a stronger condition, namely that this one state must be reachable for some execution sequence. This condition is, in general, too strong to be provable.

We also have our consistency requirement, which amounts to the assertion that there exists at least one state that satisfies the theory. Again this is not a very strong condition, for a state that meets the requirement for the theory, but from which no transition is defined is a rather sterile theory, but setting proof conditions that are too strong renders the meeting of obligations impossible.

Laws of Interaction

Laws of interaction are clearly of great significance for our purposes, for they will capture the ways in which the theories of mechanisms inside our machine interact to implement our theoretical schemas in mechanistic ways. The law of composition in a Hoare logic, for example, could be viewed as laws of interaction between the theories of two different programming constructs. A more likely use of laws of interaction, however, is the partitioning of state, giving rise to several idealised subsystems, each described by its own theory, where the total idealised system is the emergent behaviour of the interactions of these subsystems.

This raises the important question as to how we view the relationships between our mechanisms and our machines. We could accept the "ghost in the machine" view, which suggests that every mechanism has part of the essence of the machines in which it could be used. Thus every assignment statement would have to contain some of the essence of payroll systems, sorting algorithms, and so on. This has the attraction that understanding of the whole requires nothing more than an understanding of the parts: when the mechanisms are brought together the essence of the system is whole, and the ghost emerges. This seems difficult to accept! The alternative, however, is to accept that the whole machine is more than just the sum of its parts. The laws of interaction contribute to the understanding of the machine as a whole. If we accept this more likely scenario, then we must also accept that teaching details of mechanisms will not be sufficient to lead to an understanding of how these mechanisms might be used in implementing machines. Students will not learn to design systems simply by learning *about* mechanisms such as programming languages.

The identification and discharge of proof obligations here is much more complicated. We have proof obligations pertaining to individual theories, of course, such as consistency, but also obligations relating to the global theory and the interaction effects. Space does not permit a full discussion of these, but for an introduction to the subject, see [CP90a].

Structuring Presentations

One further expected aspect of our presentation language is its ability to structure our theory presentation in some way. This, unfortunately, is not adequately reflected in discussion of the semantic conception. Laws of interaction, intended primarily to allow discussion of the use of experimental methods, with equipment that has its own theory, can be used to discuss some of the required properties, but in a very "flat" fashion. Laws of interaction are intended to discuss the situation where a theory shares parameters with another theory, or forces functional dependencies on another theory. It does not allow, for example, a theory to be parameterised by another theory.

Frawley has carried out some initial work in this area for natural sciences by studying discourse processes amongst scientists, and attempting to identify a number of primitive terms, with the intention of building a computational model of scientific discourse[Fra86, pages 78-89]. His work leads to the suggestion that science is underpinned by a continually changing semantic net, and that this net can be reduced to a computational semantic formalism. Scientific progress corresponds to changes in the structure of the semantic net. In this view, the nature of a program as a theory can be inverted to give the statement that a theory is a program.

Theories encountered in Software Engineering are becoming increasingly complex. This complexity causes no problems for the logic of our model, but the pragmatics do need further consideration. For purely scientific discourse, structure is a convenience but not a necessity: we could treat all of our proofs as flat structures, or even include lemmas and existing theorems in *ad hoc* ways. Similarly programs written in this way cause no problems for the machines they run on. If we consider informative or exploratory discourse, however, with human receivers, then the need for structure becomes obvious. The properties we require of the presentation can only really be decided when we fix a purpose for the discourse, but typically we require recipients of the discourse to gain some understanding of the system, or parts of the system. Such understanding is unlikely to come with a presentation that is simply a collection of unstructured relations and predicates. We must impose structure on our presentation: in a sense, we must build a theory, or at least a model, of our theory presentations.

The need to impose structure does arise in natural sciences, but there it is handled almost exclusively by appeal to the terminology of the discipline (categorisations) or by appeal to mathematics. Physicists, for example, make extensive use of states expressed in terms of vectors, matrices and tensors, which are just ways of building higher order mathematical structures, and then make free use of the properties of these structures, such as tensor calculus or matrix algebra. Similar strategies are used by the Software Engineer, but in general the structures used have not yet become part of a mathematical culture. In part this is because of the youth of the discipline, but it is also due to the fact that the later stages of design are influenced by the need to seek structures available in programming languages. Thus a partially ordered set in a phenomenological "specification" may turn into a list in a " design", and a linked list implemented with pointers in an "implementation", as the theory is presented in more refined and deterministic forms. The software engineer also has to deal with problems that have far more complex state-spaces than does the natural scientist. As a result of this the theories are more complex, and more structure is needed in their presentation. As Dijkstra has observed,

> "The programmer has to be able to think in terms of conceptual hierarchies that are much deeper than any single mind ever needed to face before." [Dij89, page 1400]

The fact that the required structuring mechanisms are frequently not part of mathematical culture means that the software engineer needs to worry about where to get the structures from, a problem not usually facing the physicist. Some structures come with the adoption of specification languages, such as the schemas of Z, extension and combination operators in CLEAR, and the agent constructors and combinators of CCS. Frequently, however, engineers need to design their own structures to meet their perceptions of the problem, in which case they need to provide the theories of these structures explicitly. Moreover, they may perceive areas of regularity, leading to the re-use of certain structures. In this case, they either have to build a theory of such re-use, or select presentation languages that already offer such a capability, such as the image facility in OBJ3 [GW88], parameters in schemas, or inheritance in EIFFEL [Mey88]. We should note that the engineer is interested in re-use in at least three dimensions:

- Re-use between problems, using presentations found in designing one system when designing similar systems, at the level of both specification components and programming language code
- Re-use within a presentation, capturing regularity in the state space.
- Re-use between presentations during the design process, where morphisms allow structures to be transformed rather than thrown away.

We will explore re-use in more detail in the next section, where we consider the rôle of analogies in our model.

Finding suitable structuring mechanisms in theory presentations, at all levels from specifications, code and even user guides (where hypertext is being explored, for example), is a major thrust of Computer Science research. To date, however, most of the research has been very localised, focussing on particular specification styles or programming languages, for example, and little work has been done on generalisations and transference between domains. One advantage of the theory building view is that it makes explicit areas of shared concern, so that we can identify areas of potential technology transfer. This is clearly of great benefit in curriculum design, where major economies can be made if lessons learnt in code design, for example, can be applied to writing user documentation.

5.5 Analogies

Learning to solve problems, and the use of experience in solving novel problems, both need to make use of the transfer of knowledge from one domain to another. This transfer can be called many things, including metaphor, analogy or abstraction. In all of these we can express the process that is taking place in terms of a target domain (the domain of real interest), a source domain (which is already understood to some extent), and a matching system for building correspondences between the two. It is widely accepted that matching systems can operate on two levels, matching surface similarities, such as colour and size, or matching deep similarities, such as the relationships holding between attributes. These two dimensions can be used to give the very rough categorisation of the various terms used to denote the process of transference shown in Figure 5.2 [Gen89, Page 207].

We will restrict attention primarily to analogies, for our aim is to discuss engineering, and we assume that the engineer will require access to relational properties via deep structural mappings in order to reason about systems. Indeed, one of the signs of expertise in problem solving within a given domain is the ability to see problems from that domain in terms of deep structure [Ske82]. Metaphor and other devices might play a part in exploratory discourse in suggesting where to look for this structure, or even in informative discourse in user manuals, but we will not consider these here.

Medin and Orthony [MO89] discuss the problems that arise in matching. In particular, they note that even surface matching is often a manifestation of deep structural properties. They assert that we are usually only interested in matching properties that we perceive to be bound into deeper structural relationships within the domains. We would not, for example, usually base an analogy between suitcases and tennis balls on the fact that both items have no ears. There is unlikely to be any deep structural relations involving the presence of ears in our perception of either object. We might, however, observe that both objects are capable of bouncing if dropped, or that they both enclose space in some way. The views we take will be driven by purpose [KT89, Page 76]. This

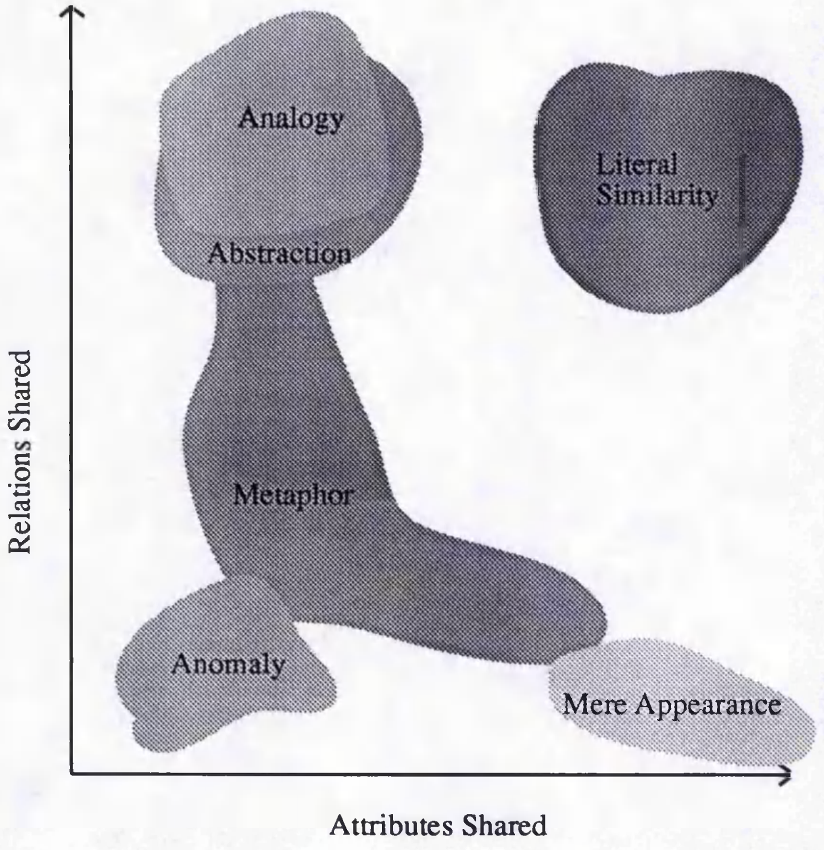

Figure 5.2: Deep and Surface Structure Matching Space

creates a link between analogies and theories, for both are based upon coherence, purpose, and perceived structural views. Viewed procedurally, we can observe that the task of building a theory might well be facilitated by first observing some analogy between two domains, provided that the analogy is based on sufficiently deep structure. If we manage to see a problem in terms of some analogy, which is sufficiently deep for a good relational structure to be matched, then we may be able to carry across a partial theory presentation of aspects of the source domain so that it provides a partial presentation of aspects of the target domain. Simply seeing a metaphor would not be sufficient.

This gives rise to a potential confusion, however, for we we now have both a state space and a source domain playing very similar roles. When we view something as an idealised system, why is this a theory rather than an analogy? If we represent the world as a relational structure of states, why are we not building an analogy with the third world object comprising the theory of relations? We are in grave danger of playing nominalist games here, for, as Anderson and Thompson observe, abstraction (which is what gives rise to our idealised system) is "an analogy in which the model is an abstract description rather than a physical object" [AT89, Page 294]. If we want to consider the algebra of sets as an abstract description, then thinking in terms of sets is abstraction, if we want to think of it as a third world object, then we are using analogy. The distinction is decidedly vague. In essence, both analogy and abstraction are instances of the same process: finding matchings between a source and target domain. The important difference is that between analogy and metaphor, which is based on the degree of deep structure.

We should note that this relationship between abstraction and analogy is by no means obvious. Dijkstra, for example, seems to differentiate between the two terms in quite fundamental ways, when he writes

> One of [the] characteristics [of the middle ages] was that 'reasoning by analogy' was rampant; another characteristic was almost total intellectual stagnation, and we now see why the two went together." [Dij89, Page 1399]

It seems unlikely that Dijkstra is suggesting that formalisation will lead to intellectual stagnation. He goes on to say, however,

> "It really helps to view a program as a formula" [Page 1409]

which seems very much like an appeal to analogical reasoning. In fact, what Dijkstra is really warning against is the use of unwarranted analogies in reasoning, but he fails to develop any sort of account as to what constitutes an appropriate use of analogy, leaving the reader with the notion that formalisations are good, and all other analogies are bad.

We can now see that viewing the world in terms of different analogies, or abstractions, will give rise to different theory presentations, both because different partial views will be taken, and also because different law formulations are likely. If we see the world in terms of algebras, for example, we will abstract away from those facets of our problem which hinder this view, and present the theory in terms of laws of coexistence. If we see it in terms of Kahn Nets, however, we will seek out processes and express the theory not primarily in terms of laws of consequence and coexistence about these processes, but laws of interaction between them. Thus there is a tension between our choice of specification style and our view of the world: VDM, CLEAR and CCS are not simply different formalisms for expressing things, they carry with them different analogies for building the theory. Even individual formalisms can be used to express different world views: Z, for example, can be used purely algebraically, for there is no need to use the state-based conventions [ML91].

It is also possible, of course, to see the world in terms of programmable von Neumann architectures. A true programming expert might well see a problem directly in terms of a programming formalism, constructing an analogy between the problem domain and a Pascal program, for example. Such a person might well produce the program as a first phenomenological theory presentation. This is only likely, however, if the individual has experience of the type of problem being solved, and understands the theory of Pascal well enough to spot the deep structure. We cannot let a novice programmer get away with such a thing, for analogy, with its accompanying theory building, would be replaced by metaphor, creating a matching only between surface similarities, and the programmer would be unable to reason selfconsciously about the artifacts: a requirement for engineering. Moreover, leaping straight to the program renders the task of refutation very difficult: we cannot use the mathematico-scientific approach, but we have to test, and we have no contractual boundary other than the deliverable itself upon which to agree. In general, we suggest that a safer approach is to construct sequences of theories, based on analogies, and hence experience.

Some ways of seeing the world would appear to have a special status, however, providing a common foundation for academic disciplines. Goethe, for example, describes such world views as follows:

> "We call these primordial phenomena, because nothing appreciable by the senses lies beyond them, on the contrary, they are perfectly fit to be considered as a fixed point to which we first ascend [in the process of finding what is fundamental], and from which we may, in like manner, descend to the commonest case of everyday experience." [Goe78, Page 72]

DiSessa discusses the role of these entities, which she refers to as 'phenomenological primitives' (p-prims), in the education of physics students. She observes that novices bring a rich environment of p-prims to bear on problems, but that these are unstructured, and the student has no way of selecting suitable candidates for particular problems. In particular, inappropriate p-prims may dominate, causing difficulties in problem solving. These p-prims need not be complex theoretical entities, but are quite likely to be expressed in terms of common-sense models [diS87]. The simplified models treated in the physical sciences are obvious candidates for useful p-prims. As Trusted has noted, "the sacrifice of comprehensiveness to comprehensibility is seen in the appeal to ideal models which feature in many scientific theories" [Tru87, Page 51].

We can also note here that many of the analogies constructed in software design are pictorial. Despite their widespread use, not only in Software Engineering, very little has been written about how this sort of analogy works. As Gibson notes

> "Nothing even approximating a science of depiction exists. What artists, critics and philosophers of art have to say about pictures has little in common with what photographers, opticists and geometers have to say about them. They do not seem to be talking about the same topic. No one seems to know what a picture *is*." [Gib79, Page 5]

A number of different notions of what a picture is have been proposed. At one extreme is the idea that pictures are just symbols, and like any language we have to learn their meaning for "no degree of resemblance is sufficient to establish the requisite relationship of reference. Nor is resemblance *necessary* for reference; almost anything may stand for anything else" [Goo76, Page 5]. This notion is clearly inadequate, for if pictures were *just* symbols then we would have to learn their meaning, and if presented with a picture we had not seen before, we would not be able to interpret it. There are aspects of pictures that do seem to conform to this idea, however, such as the lines drawn after a cartoon figure to denote motion. This is a convention that we learn through context.

Another common notion is that pictures have meaning by virtue of physical resemblance to the objects they depict. Pictures only represent objects as seen from particular viewpoints, however, and they will only be recognised if the viewer can relate to the given perspective. It is difficult to reconcile this idea with the use of diagrams in Software Engineering, for we do not usually think of "seeing" a computer system. We might argue that systems have a form, and hence the diagram is being used to portray structural similarity. This raises a problem with the use of diagrams, such as dataflow diagrams, for capturing requirements, for if we believe that these should express "what" not "how", what form is the diagram depicting? One possible answer is that we are supposed to view the diagram as giving the form of some system that is analogous to the intended system, but that we are not meant to transfer the structural information inherent in the diagram over to the required system. This does not appear to be the case, however, because most uses of dataflow diagrams, and similar techniques, intentionally carry over the structural information into the design phases of the development process. This leads us to conclude that the decision to use a diagrammatic technique carries with it the commitment to accept the structural information it conveys.

An alternative view would be to accept that such diagrams are purely symbolic. In this case we need to learn what each symbol means, and what the juxtaposition of symbols means. We may be able to consider the diagrams as formal languages, and to give them a formal semantics in terms of our cultural bedrock. This view is commonly adopted of Petri nets and finite state machine representations. Many of the diagrams used in Software Engineering, however, are not described in this way. If viewed as symbolic languages, they have no well-defined semantics. Users of these diagrammatic techniques are free to interpret pictures in different ways. This renders them useless for the purposes of providing theories that are inter-subjectively testable, however, for we cannot tell what interpretations any given diagram is being given. Moreover, a frequently cited advantage of such diagrams is that customers find them easy to understand, so they form a good basis for contractual agreements. If customers find them easier to understand it is almost certainly because they already have conventional meanings for shapes such as boxes (to hold things together), and arrows (to move things about). The contract will only be sensible if the engineer takes the trouble to ensure that all parties to the contract are using the same interpretations.

These discussions lead us to conclude that the possibility of specifying the "what" without the "how" needs to be questioned. In seeking to present a theory of the "what", the phenomenological theory, we will take particular views of the world. Is it possible to assert that the views we take at the outset will never percolate down through our design to influence the "how"? This seems unlikely. If we present a specification in terms of dataflow, for example, it would take vast leaps of the imagination to throw away the underlying analogy with a state-based architecture and produce a Prolog program; similarly if we produce a process oriented view using Kahn nets expressed in CCS we are unlikely to construct monolithic Fortran. The choice of view, if we are able make it selfconsciously, is one of the first design decisions. The real issue is that the purpose of the theory at this stage is scientific, so we need to ensure that it is fit for this purpose, and the introduction of implementation specific detail has too often been observed to mitigate against this. The deep structure of a typical procedural program, for example, is notoriously convoluted, and does not usually lend itself to the task of generalised deductions suitable for refutation.

We should observe that ideally we would like the choice of views taken to be a selfconscious decision, for then the engineer maintains responsibility for the design. In practice, however, such insistence leads to an infinite regression, for this choice will itself involve views. Instead we will insist that the engineer is aware of the problems that such choices can pose, and takes action to ensure that these problems are dealt with safely. Our proof obligations, for example, are manifestations of this practice. We will return to this question in Chapter Seven.

5.6 Summary

In this chapter we have briefly reviewed the notion of "theory", and rapidly refined it to the semantic conception. We have discussed some of the principles underlying theory presentations, and looked at the rôle of analogies in finding suitable presentations. With the benefit of this discussion, we can now refine our model of system design.

Our phenomenological theory building activity involves agreeing with the customer both an idealisation of the problem, giving rise to a state space, and also a set of partial theory presentations governing the behaviour of idealised systems, namely those of interest to the customer. The design task can then be viewed as the activity of changing the state space representation, and also the the theory presentation, so that it captures the problem in terms of structures and transitions available in some target virtual machine, such as a Pascal engine. These changes may be seen as refinements, or as reconstructions with verification conditions. The phenomenological theory presentations must remain partial presentations of the resulting theory.

We should stress that a number of very important problem areas have been identified in this chapter, in particular, the problem of imposing structure onto both the state space and the theory presentation. Resolving these problems, however, is not considered part of this research, for we cannot expect the curriculum designer to solve all the problems of Software Engineering *en passant*. We can expect the teacher to be aware of these problems, however, for one of the tasks of teaching is to help the student navigate around the learning experience, and identification of potential hazard areas is certainly part of this task, as is the indication of where ideas can be carried across from one domain to another. Like the engineer, the teacher must take responsibility for this navigation task until the student is sufficiently well trained.

Chapter 6

Methods, Madness or Mêlée?

"Do all the good you can, by all the means you can, in all the ways you can, in all the places you can, at all the time you can, to all the people you can, as long as ever you can"

John Wesley

One of the most frequently debated topics in Software Engineering is the rôle and merits of "methods" for software development. Such debates are not only common, but they are frequently acrimonious. One reason for this is that much of the discussion rests on dogma, but acknowledging dogmatic views is unfashionable in western scientific culture, so this is often hidden behind a facade of rational argument, usually based on anecdotal evidence. Another reason for the acrimony is that the concept of "method" is a complex one, and the term has at least four common uses within Software Engineering. Many of the debates are really just confusions of the uses of the term. To compound matters, however, many of the protagonists in the debates seem to have sought refuge in the term "methodology", which, far from clarifying matters, simply adds all the possible uses of "theory" to those of "method". Clarification of the idea of a method is essential for our pedagogical purposes, for every scheme teaching Software Engineering teaches methods in some form or another.

In one sense, we have already been discussing methods, for our model of software development suggests a number of procedures to accompany it, and indeed the model would be meaningless without the procedural interpretation the reader gives it. The case has been made elsewhere that a proper treatment of method should be carried out in conjunction with the treatment of representations, and the two should not be separated [Loo86]. We will not adopt this approach here, however, because our primary concern is to explore existing notions of methods, and to try and relate them to our proposed model.

The four uses of the term "method" that we are going to explore are clearly not as separable as we are going to suggest in this discussion, otherwise acrimonious debate could be avoided. Most methods proposed for Software Engineering can be considered as fulfilling, or have been claimed to fulfill, all four uses to some degree. We would argue, as we have done throughout this thesis, that simplification and abstraction are the only ways to come to terms with complex issues, however, and the re-introduction of more complex facets of the problem can wait until a simplified understanding has been achieved.

We will start by considering methods as providing well-defined plans, intended to govern all the actions of engineers and lead to the successful completion of a project. Such methods will be prescriptive. Finding these methods, if they exist, would solve the software crisis. They would also render the engineer little more than a low-level technician, and solve our curriculum design problem by reducing it to the task of devising an appropriate training course in the method. We will argue that such a method cannot exist. Moreover, proceeding as if it might leads to undesirable consequences in our curriculum design that, far from resolving the problems of the discipline, exacerbate them.

Given that methods cannot be reliable plans for solving unknown problems, we will discuss their rôle as rationalisations of the process of problem solving. This is common practice both in the Philosophy of Science and also in science and technology itself, where results are presented as if they were the products of rational processes, governed by plans from the outset. It will be argued that rationalisation, as well as providing a vehicle for historical record, can assist in the process of achieving quality in software design. A number of possible rationalisations for processes accompanying our model of system design will be discussed briefly.

Our third perspective on methods will view them as constraints on the activities of engineers and scientists. These constraints may be justified for a number of reasons, including management of the process, making the engineer more accountable via visible milestones, and facilitating the cooperation of several engineers on one project.

Finally we will present the idea that methods are collections of useful tools available to the engineer for solving localised problems, rather than ways of addressing the whole question of design. These tools may be algorithmic in nature, certain to work if applied correctly, or heuristic, where no such guarantees are given. These tools include mathematical notations with their associated analytical techniques, ways of viewing and representing the problem, heuristics for tackling specific situations, and a plethora of other techniques. The engineer has responsibility for selecting the tools and using them safely.

6.1 Methods as Plans

The question we want to address here is whether or not it is possible to have a method of system design that can be used for the solution of all possible problems. We might rephrase this in a less prejudicial form as asking whether, for any given problem, we can always identify, at the outset, a particular method for solving that problem. These two formulations are equivalent, for the latter involves finding a procedure, or method, for identifying the particular methods, and the conjunction of this procedure together with all the individual methods comprises a universal method. We will tackle the question in its first form. We should observe that there are classes of problem which are so well understood that methods for their solution do appear self-evident. Compiler writing for conventional procedural languages and traditional target architectures, for example, can be tackled in prescribed ways, using tools developed for the purpose. It is important to realise, however, that most problems of Software Engineering do not fall into such classes. Many problems may be seen as similar to the exemplars from the classes, such as sorting and merging type problems, after analysis has taken place, but no general method exists by which this similarity can be detected, and no algorithmic formulation for the requisite analysis has yet been given.

It may seem self-evident that no such generalised method can exist, but this view is not universally held. McCrory, for example, asserts that not only does such a method exist, but that it is common to the whole of design.

> "The design process follows a methodology similar to that of the scientific method, although the design method has not been so carefully defined or historically well established." [McC74, page 161].

It seems likely that similar views can be attributed to many software engineers, who seem to have formed the view that their problems will disappear when the method has been found. Unfortunately a similar view seems to be held by some educators, who seem to believe that they owe it to their students to teach the most complicated version of such a method currently used in industry, on the grounds that this is the closest we have come to finding the Holy Grail, and it will serve the students well when the real thing is found.

Such a view is untenable. The scientific method that McCrory appeals to simply does not exist. He seems to have confused philosophers' rationalisations of the activities of scientists with plans that scientists hold, and even then he seems to have overlooked the vast number of such rationalisations that have been formulated for different purposes. In particular, he has confused science, as presented in scientific discourse, with science as an activity involving exploration and cooperation as well as the presentation of final results. Williams puts this very well, when he writes

> " . . . no paper is ever written with its real, genuine, honest historical introduction; because somewhere in in the middle of it would have to be the statement 'at this point I had an idea'. Editors—and I am one myself—cannot accept this; it removes science from its austere pedestal and makes it into a creative art—which of course it is, but we are not supposed to say so in public." [Wil64, page 54].

It is generally accepted that although there *may* be ways of rationalising the process of scientific justification, scientific discovery can never be a completely rational process. Feyerabend, who takes an admittedly extreme view, observes that

> "...it is of course possible to simplify the historical medium in which a scientist works by simplifying its main actors. The history of science, after all, consists not only of facts and conclusions drawn therefrom. It consists also of

> ideas, interpretations of facts, problems created by a clash of interpretations, actions of scientists, and so on." [Fey70a, page 20]

Furthermore, he goes on to say that it is possible to simplify this state of affairs by "brainwashing" individuals to accept "professional conscience" and "professional integrity", and that

> "An essential part of the training is the inhibition of intuition that might lead to a blurring of boundaries [between science and other activities]. A person's religion, for example, or his metaphysics, or his sense of humour must not have the slightest connection with his scientific activity. His imagination is restrained and even his language will cease to be his own."[Fey70a, page 20]

It is a sobering thought that Hoare's plea for professionalism and scientific attitudes may be interpreted in such an extreme Orwellian fashion. What Hoare doubtlessly intended as a plea for honesty and integrity could be interpreted as a call for the dehumanisation of science. This is significant, for it causes us to reflect that a possible reaction to Hoare's persuasive discourse is to see it as an attempt to start the brainwashing process: this might explain the violent reaction that many anti-formalists show when faced with suggestions that mathematics has a rôle to play in system design.

This is a very important observation. It sets the scene for the discussion of methods as instruments of control, both in the form of limitations of action and of language. Moreover, it highlights the rôle of intuition and individual background in system design. Parnas and Clements have noted that software design is influenced by ideas that do not come from rational consideration of the problem, but "arise spontaneously from other sources"[PC85, page 83]. Naur goes further, when he states that intuition "is the basis on which all activities involved in software development must build" [Nau85, page 60]. Both Parnas and Naur admit, however, that intuition is fallible, and dangerously so because the immediacy of response makes it seem attractive and convenient. Naur goes on to note that intuitive actions can themselves be viewed intuitively by the actor, giving rise to a self-conscious mode of proceeding, echoing Quine's comment that "science is self-conscious common sense" [Qui60].

Naur also develops the argument that the inevitability of intuition in software design means that we should concentrate not on removing it, but on finding ways of detecting and correcting any errors introduced: this requires self-conscious design, albeit a self-consciousness still based on intuition. Moreover he endorses the theory building view put forward here when he writes

> "the problem of high quality software development cannot be solved by rules and methods, which essentially assume that the programmer acts like a machine for producing programs.
>
> ...a view of software development that makes the application of rule-based methods and notations the basic issue is misguided. The deeper problem of

software development is the programmer's building of theories of the computer-based solutions." [Nau85, page 78].

Acceptance of the theory building view, and rejection of the existence of algorithms for carrying out science, forces us to reject the notion of an all-embracing software development method.

Parnas also notes that design will inevitably be beset by unforeseen problems, such as changes in requirements and technology, or the imposition of management decisions. We could, of course, still seek a method capable of admitting change, such as the multiple feedback loops of the typical life-cycle. The actions to be taken in the face of such changes, however, will depend on so many factors, only truly understood at the time of the change, that it is impossible to find a method capable of prescribing actions in a domain independent way[PC85]. Thus even if we do not accept the theory building view, we are forced to reject the notion of a prescriptive method of design.

Rejection of this kind of method, however, leaves us bereft of any framework within which to discuss approaches to a design task, unless we find an alternative explanation. We will adopt Suchman's notion that behaviour should be seen as situated actions:

> "The term ["situated action"] underscores the view that every course of action depends in essential ways upon its material and social circumstances." [Suc87, page 50].

Under this view plans are not prescribed sets of actions, rather their purpose is

> "to orient you in such a way that you can obtain the best possible position from which to use those embodied skills on which, in the final analysis, your success depends" [Suc87, page 52]

It is precisely because the "embodied skills" of the typical computer are so limited, of course, that the software engineer needs to transform the theory underpinning a design into a presentation in terms of a simple programming language.

Interpreted in the theory of situated actions, the rôle of methods is that of a resource to orient the engineer at the outset of the process, and also to provide a frame of reference that the engineer can appeal to when problems arise. The method does not navigate you through the problem, but gives a canonical form against which progress can be measured. There is no need, however, for such methods to be all embracing: the engineer can seek different frameworks as his or her perception of the problem changes.

6.2 Methods as Rationalisations

Given that there cannot be a universal description of how the software design process is actually to be carried out, we will now turn our attention to the matter of finding rationalisations of the process. Whereas a prescriptive method has to foresee all possible events, rationalisations are usually allowed the great benefit of imperfect memory or knowledge. We can select the "significant" features of individual designs and ignore all the others. This selection process is, of course, part of the rationalisation, but it is seldom stated explicitly. In seeking to plan, absolutely, the actions of an engineer we need to foresee the possible consequences of an unsatisfactory lunch; in rationalising the process we can decide that the quality of canteen food has no relevance to the actions taken. If we view plans as controls, of course, we will establish ways of proceeding that attempt to deny culinary factors a rôle in the process.

Why should we even attempt rationalisation? One reason for seeking rational justification of engineering actions is cultural. Technology, because it involves so many diverse factors, cannot form a logically closed system. Western cultures, however, "feel the need for ordering their fields of knowledge so that they are subject to conscious analysis and management" [Deh86, pages 110-111], and hence seeks closed, logical, rationalisations of actions. We will not pass judgement on such motivation, but we should note that many papers proposing rational design processes seem to accept this reason as sufficient. Readers of these papers who reject the cultural values placed on the closed logical structures of science, or at least question their relevance to engineering, are perfectly entitled to reject the papers as worthless. Many readers of Dijkstra and Hoare, unfortunately, attribute only this motivation for rationalisation to the authors, and hence reject the messages contained in the publications out of hand. No doubt, some of the "softer" literature is disregraded, for equally dogmatic reasons, by the "formalists", because it does not acknowledge the importance of closed, logical, structures. Thus important messages in both camps may be lost because of dogma.

Feyerabend argues that this association of rationalisation with closed systems is unfortunate, and the assumption that science, as the exemplar of valued practice in our academic culture, can be rationalised by such systems is untrue.

> "Mature science *unites* two very different traditions which are often separate, the tradition of a pluralist philosophical criticism and a more practical (and less humanitarian ...) tradition which explores the potentialities of a given material (of a theory; of a piece of matter) without being deterred by the difficulties that may arise and without regard to alternative ways of thinking (and acting)." [Fey70b, page 212]

We should not confuse "being scientific" with conforming to just one aspect of a rationalisation of science. We must beware of rationalising only one aspect of the software design process, but then proceeding as if we have finished the task, and also elevated the status of the discipline to that of a "science". This is not to say, of course, that separation of concerns cannot be used, provided the limitations of the technique are acknowledged. Thus Hoare is perfectly entitled to assert that we can prove the correctness of programs, but we must take care to recognise that "prove", "correctness", and "program" are being given meanings within some closed system of rationalisation.

A second reason for rationalisation is to increase understanding of the processes involved in software design. This will allow the discipline to advance by improving both practice and education. This increase in understanding does not come primarily from rationalisations that everyone agrees with, however, but with those that cause discourse, to clarify or dispute the views put forward. In particular, it is perfectly possible for one individual to propose several different rationalisations of some situation in the full knowledge that they are inconsistent. Feyerabend, for example, defends his right to support the rationalisations of both Kuhn and Lakatos, even though there are fundamental disagreements between them that cannot logically be reconciled.

> "Contrary arguments bring out the different features it [science] contains, they challenge us to make a decision, they challenge us to either *accept* this many-faced monster and be devoured by it, or else to *challenge* it in accordance with our wishes." [Fey70b, page 215]

We should note that this is also an excellent defence of exploratory discourse in science and design, where the individual may propose conflicting theories to promote discussion and resolution and, by reflexivity, of this thesis.

Perhaps the most obvious candidate for a rationalisation of the software design process is the life cycle model. We shall argue, in the next section, that life cycles can serve a useful function as controlling mechanisms in software design, but how helpful are they as rationalisations? The answer seems to be that they are of very limited use. Most life cycles say little beyond "before designing something you need to think about what it is you are going to do, then you can do it. If the result turns out to be flawed, redesign it."

The problems arise because life-cycles attempt to satisfy people's need for closed rationalisations, but they propose models that no one can refute. Thus they seek scientific ideals but with pseudo-scientific methods. Pfleeger, for example, proposes a life cycle that comprises a fully connected graph with nine nodes. These nodes are labelled with terms such as "requirements analysis and definition", "system design", "program design", "writing the program", and so on. All we can glean from such models is that nebulous activities such as "writing the program" can go on throughout the design process. How should such an observation be interpreted? Does it mean that "writing the program" is part of "program design", in which case should we conclude from the full connectivity of the graph, that all nine activities are potentially part of each other? Perhaps it means that there will be interleavings of "program design" with "writing the program", but this only tells us something if we can clearly distinguish between the terms. A fundamental criticism of all life-cycle models is that they use jargon such as "design", "specification", and so on, but rely on the model to imply what the terms mean. Such models will never lead to much improvement in understanding, for we can all place our own interpretations on the terms, and either agree with the model, or quibble over the terms and accuse each other of nominalist games. Moreover, any major criticisms of the model are usually greeted by the addition of extra arrows to the diagram, or adjectives to the terms, such as adding "functional", "formal", "requirements" or "system" to the term "specification", none of which really help us to understand what is meant by a specification. Hence the claim that life cycles are usually accompanied by pseudo-scientific methods of defence.

Life cycles are not the only rationalisations of the development process. Another common rationalisation is the transformational view, which sees the process as the transformation of specifications into implementable specifications, or unimplementable programs into runnable programs. These rationalisations can be helpful, for they usually start by explaining what is being understood by a specification or program. The interpretation placed on the terms is often very limited, such as a categorical or algebraic view, but at least we know what we are disagreeing with when we take issue. Another rationalisation that is worthy of note is the notion of a program as an existence proof of its specification, carried out in a constructive logic. Here the rationalisation can be construed as implying that the method of design is similar to the method of proof construction in mathematics. Backhouse, for example, discusses how proof heuristics in Martin-Löf type theory can be viewed as heuristics for the design of procedural programming constructs [Bac90].

The real danger in adopting the life cycle model as a rationalisation is not that it is of little value, that makes it at worse at distraction, but its universal appeal, in the sense that no one can refute it. This has led to its (naive) widespread adoption in structuring the curriculum. We will see in the next section that life cycles have a significant rôle to play as controlling mechanisms in the design process, so if we adopt them as rationalisations of the process as well we are likely to endorse the view that students should conform to these controlling mechanisms. Such a method of teaching is implicitly accepting Durkheim's thesis. Even if the courses within the designed curriculum adopt a critical stance to the life cycle, they are likely to result in the students suggesting patches to the model rather than the overthrow of the model itself, for they will work within the paradigm that the model establishes.

Given this danger with any rationalisation, perhaps we should seek more justification for attempting rationalisation at all. Parnas and Clements note that we do need such rationalisations of the design process, not only for the advancement of knowledge, and because they support the view that we are being scientific, but most importantly because they help with the task of improving software quality. The reasons they give include:

- Rationalisation can act as plans for subsequent projects, providing valuable resources that the engineer can draw upon for initial orientation and also for handling problems when they arise.
- Rationalisations help us to discuss the reasoning that leads to certain courses of action, thus protecting us to some extent from *ad hoc* sequences of actions. This reasoning will be based upon idealisations, but it may still provide guidance.
- Rationalisations help the social process of design by contributing a standard, if idealised, procedure against which progress can be discussed. This allows the identification of milestones, for example, and also the standardisation of discourse activities both within the project and for the purposes of external monitoring.

We might add to this list that rationalisations also serve to support arguments for "best practice" between projects. They might, for example, be used to defend against charges of negligence after the event, or in tendering for contracts. If we accept the line of argument that rationalisation is of value, then we should turn our attention to possible rationalisations of design processes to accompany our proposed model.

In our proposed model the engineer is faced with the task of producing a sequence of theories. An obvious place to start looking for rationalisations, therefore, is the Philosophy of Science. As the development of our model started with Popper's idea of refutation, we will also start our rationalisations from here. The simplest rationalisation we can give is to observe that once a theory has been refuted another one must be found. This naive refutationalism is not particularly helpful, however, for it offers no advice on what might constitute scientific progress, and no guiding heuristics beyond a process of trial and error. Also it does not accord with the facts for there is no doubt that some very successful science has been carried out with theories that had already been refuted. We would end up with little more than a trace of science as a sequence of refuted, and unconnected, theories, and this clearly does not model the software design process as we would like it to be, even if it comes close to the process as it is sometimes observed.

A more sophisticated form of refutationalism is based on Popper's notions that we want to make theories more general and more precise if science is to be progressive. We can translate this into the terms of our software engineering model quite easily, and we end up with a limited form of refinement. If we draw a lattice similar to that of figure 2.1, but using the conventional notation of program correctness, and noting that we want to treat the specification S and the program P as theories, we get the lattice shown in Figure 6.1.

$$\begin{tikzcd}
 & \{x > 1\}\ S\ \{x \geq 2\} \arrow[dl, dash] \arrow[dr, dash] & \\
\{x > 0\}\ S'\ \{x \geq 2\} \arrow[dr, dash] & & \{x > 1\}\ S''\ \{x = 2\} \arrow[dl, dash] \\
 & \{x > 0\}\ P\ \{x = 2\} &
\end{tikzcd}$$

Figure 6.1: Lattice of Refinement

The orderings in this lattice correspond to the refinement ordering, and the diagram above reflects the laws [Mor90, page 8].

$$(post' \Longrightarrow post) \Longrightarrow w : [pre, post] \sqsubseteq w : [pre, post']$$
$$(pre \Longrightarrow pre') \Longrightarrow w : [pre, post] \sqsubseteq w : [pre', post]$$

In this case additional precision was included to make the program deterministic. Precision may also be increased to express the theory in terms of state local to the program, that is, rather than in terms only of input and output components. This would give rise to laws of refinement such as [Mor90, page 53]

> if $pre \Longrightarrow \exists\, c : T.pre'$ and c is a fresh name not occuring in w, pre or $post$ then
> $w : [pre, post] \sqsubseteq con\ c : T.w : [pre', post]$

It seems likely that all the other laws of refinement can be explained in terms of Popper's notion of generalisation or making more precise.

This rationalisation gives a fair idea of what progress in our model might mean, but what of refutation itself? Suppose, whilst attempting a refinement step, we detect an inconsistency, how should we react? Refutationalism, both naive and sophisticated, leaves us floundering here, for it says nothing more than that the theory should be rejected. Software designers certainly do not behave in this fashion, rather they backtrack in quite subtle ways. Given a program that is inconsistent with the requirements, for example, the professional designer is not allowed the option of changing the requirements (although this is a common reaction amongst those lacking professionalism, and also amongst many students). Similarly the designer does not usually reject the theory of the programming language, and replace it by one that removes the inconsistency, for this would require the re-implementation of compilers, and add a number of proof obligations. Both of these options must remain a possibility, of course, for the compiler and requirements have only been tested, and so may contain the inconsistency, but usually the designer will assume that a mistake has been made in producing an aspect of the theory under his or her control, and of immediate concern, and look for corrections in this region.

This idea fits in very neatly with Kuhn's notion of paradigms [Kuh70b]. The engineer will accept certain theories and ways of proceeding, including the use of tools, as forming an irrefutable part of the paradigm, and work within the constraint they impose:

> "...when engaged in a normal research program, the scientist must *premise* current theory as the rules of the game. His object is to solve a puzzle, preferably one at which others have failed, and current theory is required to define that puzzle and to guarantee that, given sufficient brilliance, it can be solved. Of course the practitioner of such an enterprise must often test the conjectured puzzle solution that his integrity suggests. But only his personal conjecture is tested. If it fails the test, only his ability not the corpus of current science is impugned." [Kuh70a, page 70]

The rules assumed constitute a paradigm, and Kuhn distinguishes between normal science, that occurs within a particular paradigm, from periods of scientific revolution, when paradigms themselves are overthrown and replaced. The overthrow of a paradigm is not the direct result of a refutation, therefore, but a reflection of a number of refutations that have occurred in normal science which have not been resolved satisfactorily.

It may also be a reflection of the inadequacy of a paradigm to permit certain puzzles to be attempted at all.

Clearly a situation can occur in software development where, for example, confidence in a compiler is shaken, or when failure to meet a specification causes doubt as to the consistency of the specification itself. In general, however, the notion of a paradigm shift does not seem particularly appropriate to the progress of individual projects. There is more to paradigms than just the theories we adopt, however, for they also include ways of looking at the world, and when paradigm shifts occur

> It is rather as if the professional community had been suddenly transported to another planet where familiar objects are seen in different light and are joined by unfamiliar ones as well. ...we may want to say that after a revolution scientists are responding to a different world." [Kuh79, page 110]

There is little doubt that such paradigm shifts do occur in Software Engineering, but the imposition of control mechanisms usually ensure that these occur only between projects and not during them. Luker, for example, says that

> "I feel strongly that Computer Science is approaching a paradigm shift. This would have staggering financial implications, and many human ones—some may be unable or unwilling to adapt to a new paradigm. The champions of new practices are nearly always treated with great suspicion as they shake the very foundations of a discipline. However, once the shift has been effected, it is regarded as a natural step, and one which should have been taken much earlier." [Luk89, page 255]

The introduction of Object Oriented Design, for example, with its associated tools, causes the designer to take a radically different view of the world [DIL90]. Meyer notes that the move to OOD represents a paradigm shift:

> "For some programmers this change in viewpoint is as much of a shock as may have been for some people, in another time, the idea of the earth orbiting around the sun rather than the reverse." [Mey88, page 50]

Puzzles that could not be explained in procedural pseudocode, can now be addressed, such as the widespread re-use of code. A number of other ways of tackling system design can be seen to form paradigms, including von Neumann based designs, functional programming, logic programming and expert systems. It might be argued that formal and informal approaches to design, or even selfconscious and unselfconscious design, form examples of paradigms. A discussion of this, and its implications for curriculum design has been given elsewhere [Loo90c].

This raises a question that is of significance for our task of curriculum design, namely to what extent should an engineer be responsible for the selection of paradigms? Clearly not every aspect of an adopted paradigm can be subjected to reflection by its proponents; some of the world views, for example, will be tacit, and not available to introspection, but should engineers be expected to attempt selfconscious selection of world views where possible? Shields, for example, demonstrated nearly a decade ago that the paradigm underpinning SDL like systems of specification was unsound. This demonstration has not led to a rejection of the paradigm. We can only assume that this is because those working within the paradigm are not prepared to question its foundations. This is very important, for it raises the question as to whether we should teach within a paradigm, or attempt to introduce and compare many paradigms. Natural science teaching certainly does teach within paradigm, for it uses exemplars drawn from within one paradigm most of the time. Goldberg, for example, has noted that the response of British physicists to Einstein's theory was conditioned by the fact that a "British theoretical physicist ...was trained to do ether mechanics; it is what he had to learn, and it is what he knew best" [Gol70, Page 91]. Moreover, this education is assessed by knowledge of these exemplars, so a successful scientist at the outset of a career is one who works well within the given paradigm. We shall question this attitude further in the next chapter, where we will suggest that it is too early for within-paradigm teaching to be used in Software Engineering, and alternatives have to be found.

We should note in passing that Kuhn has often been dubbed a sociologist by other philosophers because of his introduction of paradigm shifts that depend on the social processes of acceptance, and therefore his rationalisations are not "methodologies in terms of which the historian reconstructs 'internal history' " [Lak71, page 91]. They introduce extraneous factors over which the scientist has no control. Jones, however, rejects this on the grounds that the major influence on this social behaviour is initial education, and this education should be considered part of science.

> "Becoming educated is internal to the professional activity, even if it is, in itself, non-rational" [Jon86, page 450].

Kuhn's notion of paradigms seems to offer a good basis for rationalisation of the shifts that occur between projects, but it does not offer a great deal of enlightenment on the progress of individual projects, other than the negative observation that such shifts would be potentially disastrous for projects within which they occurred, so we should consider controlling scientific activities to discourage that from happening. A more promising approach to the rationalisation of a single project seems to be Lakatos's research programmes [Lak70]. Lakatos suggests that we can construe theory construction as being governed by two types of methodological rules: *negative heuristics* that tell us what routes to avoid, and *positive heuristics* that tell us what paths to pursue.

Science, according to Lakatos, comprises *research programmes* governed by these two types of heuristic. In particular, each research programme has associated with it a *hard core* of theory, and the negative heuristic forbids us from trying to attempt to refute it. We must protect this core by building a protective belt of auxiliary hypotheses if necessary. These auxiliary hypotheses will form the progressive part of the theory, and a "research programme is successful if it leads to a progressive problem shift; unsuccessful if it leads to a degenerating problem shift" [Lak70, page 133]. That is, if the auxiliary statements inhibit the solution of the problems the research programme sets out to solve, the programme will fail. When a programme ceases to be successful, we need to abandon it and find another. Thus the hard core is not simply an adoption of Poincare's conventionalism, for the theories it contains are still vulnerable to rejection.

The positive heuristic of a research programme is what drives us to construct the protective belt, for it encapsulates the purpose behind the programme, and the methods we might adopt to achieve this purpose. The scientist is setting out to solve particular problems, and hence will need to construct additional theoretical statements, but these will be directed more closely than simply by generalisation and precision. Moreover, refutation will be aimed at these additional statements, rather than the programme as a whole. Similarly, the engineer is seeking to solve particular problems, not simply to construct any artifact. The problem to be solved, together with restrictions imposed by the hard core, lead to the selection of the positive heuristic, which thus comprises "a partially articulated set of suggestions or hints on how to change, develop the 'refutable variants' of the research-programme, how to modify, sophisticate, the 'refutable' protective belt" [Lak70, page 135]. It is these "suggestions or hints" that will form the basis of our final use of "method" discussed later in this chapter. This analysis suggests that standards, which if adopted will form part of the hard core of a range of research programes, may have a pernicious rôle to play, for they restrict the options open to the designer. Their use can be justified, however, if seen as part of a larger research programme, whose aim is the unification of systems. The designer must make the value judgements necessary to decide between the heuristics of these two programmes, weighing up factors such as re-use of components against possible distortions that may ensue.

We can now discuss our theory building view of system design in the light of Lakatos's research programmes. This can be done in at least two ways. We can note, for example, that software engineers usually carry a hard core from project to project. This will comprise theories of target environments and also theories inherent in tools and methods adopted for representing world views. In this setting the hard core will only be rejected when it fails to solve problems, that is, when it fails to support current projects. The move from machine code to high level languages, for example, can be seen as a rejection of the hard core based on low level architecture, to be replaced by theories of more abstract machines. Similarly the use of flow charts as world views declined as they failed to support more complex design tasks. Once we have adopted a hard core, however, we will then support it for the rest of the project. Here the software engineer faces similar problems in deciding when to overthrow a hard core that the scientist faces. It is a matter of judgement as to when such a core has ceased to be progressive. Under this interpretation, Kuhn's paradigms can be seen as research programmes that have obtained some kind of monopoly within a community[1].

[1]Lakatos disputes this, claiming that the decision to reject a hard core is rational, being based on the notion of progress, whereas the overthrow of a paradigm is fundamentally irrational. This claim seems

We can also consider an individual project as giving rise to research programmes. In this case we might consider the phase of design up to the establishing of a contractual boundary with the customer as a research programme in its own right. Once the contract has been established, a new research programme is established containing the contractual information in its hard core. This core, which will also contain world views and so on, must be protected. The negative heuristic, therefore, determines that, whatever else may happen, we must meet the customer's requirements. The positive heuristic drives us to seek solutions to the puzzle of how to do this with a theory that can be automated, that is, to extend the phenomenological theory to include the state of some real machine. Note that the specification of a target environment as part of the contractual boundary now contributes to both the negative and positive heuristics. The theories surrounding this machine constitute part of the hard core, but the "suggestions or hints" that comprise the engineers' experience of the machine will contribute to the positive heuristic. With this interpretation, the decision to overthrow a research programme during a project reflects either the engineer's rejection of the methods and tools already chosen for the project, or both the customer and engineer agreeing to a change of contract. We should note, however, that an experienced engineer will not base a research project on a hard core that is too volatile. In particular, not every detail of the customer's requirements will necessarily be included in the core as specific statements: where the engineer anticipates changes occurring in the future, a general statement will be included, and a specific instantiation added to the protective belt. No self-respecting software engineer would base a design on a hard core that contained the specific rate of V.A.T., for example, or on the customer's insistence that no invoices are issued with more than six digits in the number.

One important distinction between refutationism and Lakatos's research programmes is that the dialectic of the latter is no longer simply one of conjecture followed by refutation. We do not test all of our theories, for example, rather we allow the positive heuristic to drive our programme forward, relying on the judgement of the engineer to note progressive situations. In an extreme case, where we embrace a formal system of refinement in our hard core, we might allow the positive heuristic comprising refinement methods to drive the design along, and not test the resulting theory at all. In general, however, testing of theories will be carried out at judicious points, but more to solve local puzzles that to refute the whole theory. Thus although Popper's *ideal* of science is maintained, the methods adopted are significantly different. We know, for example, that the theory of infinite stacks will be incompatible with that of any real, finite, machine used for implementation. Refuting a theory that contained interactions of the two would be pointless, for it is too simple. Rather we shelve the inconsistency, allowing the positive heuristic of the programme to cause it to progress until a sensible point has been reached for the introduction of finite stacks. Refutationalism, both naive and sophisticated, would require us to refute and reject the theory as soon as the inconsistency was noticed, and our scientific attitude would require us to notice it as soon as possible.

difficult to support without a much firmer notion of "progress" than Lakatos ever provided.

6.3 Methods of Control and Coordination

Software Engineering is usually considered to be primarily a social process, involving teams of individuals in group problem-solving activities. A third view of "method" that we can take is as a controlling, or coordinating, influence on these activities. McCrory, for example, writes

> "In addition to being a controlling influence upon the design process, the design method[2] also serves as a communications medium. Management often does not understand the steps through which a design programme must pass between its inception and its completion. ...The design method can provide a 'universal language' understandable to both the designer and his management." [McC74, Page 172]

This is a very complex issue, and a detailed study of the subject would take us into the realms of at least management studies, social psychology and sociology, in addition to a consideration of the ethical issues involved. We cannot avoid some consideration of the matter, however, for the stance we take here will materially effect our curriculum design.

What does it mean when management claims to be using a software design method(ology)? There are at least three possible interpretations that we might place on such a claim. First, it might mean that management has imposed a number of hoops for engineers to jump through in the course of every project. Such imposition is sometimes, erroneously, identified with quality control: as Voss has observed, quality control comes only with a quality culture, and the *imposition* of hoops does not fit well within such a culture [Vos90, Page116]. These hoops may or may not be seen as supportive of the design process by the engineers involved, but this is irrelevant, for they must be jumped none the less. Moreover, these hoops must be jumped in a pre-defined order, and project costing makes the naive assumption (or definition) that when all the hoops have been successfully jumped, the project has been successfully completed. Rejection of the method is not an option for the engineers concerned, for this decision will be taken by a higher authority, with due regard not so much for the ideals of engineering, but for those of corporate finance. In this scenario the engineer cannot assume responsibility for the fitness for purpose of the designed artifact: only management can claim this, and they may not feel inclined to do so. The engineer need only be concerned with the localised task of jumping through the hoops. Moreover, such a scenario has a certain appeal to the engineer as employee. Feyerabend has noted that similar phenomena have arisen in modern science, when he writes

> "...late 20th-century science has given up all philosophical pretensions and has become a powerful *business* that shapes the mentality of its practitioners. Good payment, good standing with the boss and the colleagues in their 'unit'

[2]McCrory believes that a single design method exists.

are the chief aims of these human ants who excel in the solution of tiny problems but who cannot make sense of anything transcending their domain of competence." [Fey75, Page 188]

Why has such a situation arisen, where people are prepared to be treated as "human ants"? Most engineers are well educated and would like to think of themselves as free thinkers. Why would anyone want to become just a cog in a "software factory" [Tul87] or a science machine? Feyerabend has a reason for this:

> " ...it needs only a few well-placed phrases to put the fear of Chaos into the most enlightened audience, and to make them yearn for simple rules and simple dogmas which they can follow without having to reconsider matters at every turn." [Fey75, Page 181].

There are undoubtedly times when the complexity of the software development task is such that the individual flounders, and feels the need for help, but the imposition of sets of hoops is not the answer. What we must do is to structure our approach to each project, with due regard to the problems and resources available, and be prepared to take responsibility for these actions. This is of crucial importance for our curriculum design task, for a rejection of imposed methods *per se* radically alters our the problem. We no longer have the refuge of teaching a subset of common methods from within paradigm, but we must teach the students to select methods. This is not the easy option for it requires a more reflective approach and, as Feyerabend notes

> "I do find it a little astonishing to see with what fervour students and other non-initiates cling to stale phrases and decrepit principles as if a situation in which they bear the full responsibility for *every* action and are the original cause for *every* regularity of the mind were quite unbearable to them." [Fey75, Page 182]

Rejection of this approach to methods, therefore, antagonises some industrialists, who cite the argument of chaos, some educators, who cite the producer-consumer model of education as an excuse for adopting the easy option, and many students, who are terrified of the realisation that in designing faulty software they might kill someone, and much prefer the responsibility to be shifted to someone else. To complicate matters, we cannot argue the case on purely rational grounds, for we cannot claim such an appropriate set of hoops does not exist, the counter will be made that we just have not found it yet. It is a matter of judgement, and, like the students, we are naturally unwilling to take responsibility for the resulting actions. It is much easier to follow the trend in a fear of Chaos, seek support in the job advertisements for software engineers, cite the latest DTI initiatives, and avoid taking unpalatable decisions. Educationalist, being designers, may also try to avoid the loss of innocence by seeking refuge in styles. The discussions on curriculum design in the rest of this thesis, however, are predicated on rejection of this approach.

The phrase "adoption of a method" might also mean, however, that a particular rationalisation of the process is being embraced. This is a very different matter, for it now means the hoops do not actually need to be jumped through, but results must be presented as if they had been. In a sense, this is dishonest, but only in the way that a scientist is dishonest when he writes up his experiments, or a mathematician in presenting a polished proof.

There are many advantages of adopting such rationalisations, including uniformity of documentation, the identification of milestones, and the "scientific" facade it can offer the discipline. This rationalisation, like the hoop jumping, is usually manifest as a series of discursive acts undertaken by the engineers. Rather than a single, final, rationalisation, the engineer will usually show rationalisation by the production of documents at various stages of the design. This is the true rôle of the life cycle. It identifies documents that engineers should produce to rationalise the process. A typical life cycle does not constrain the engineer to think only about requirements analysis until the requirements document has been produced, simply to produce such a document predating the design documents. One of the limitations of the typical life cycle, however, is that it tends to emphasise informative discourse at the expense of exploratory or scientific discourse. The engineer is encouraged to document the "what" of the project, the "how" being simply "what could be done to bring this about", but not to answer the "why" questions, such as "why this presentation of requirements was adopted" or "why this way of proceeding will bring about the desired consequences". Moreover, the progress of a project tends to be judged solely by the stream of documentation that is produced. Brown has noted that this need to produce documentation may adversely effect the project:

> "...we suffered from an excess of design documentation literally driving out design proper. We never had time to notice that there was a simple, elegant, answer to the problem." [Bro84, Pages 59-60]

Norcio and Chmura have suggested that discourse between engineers may be a better indicator of design progress than documentation produced for external consumption [NC86].

It is precisely to redress this balance that the theory view has been proposed, for a theory is informative, but it can also be used for exploration or as the basis for proofs. This rationalisation shows very clearly that we will need to teach students to carry out all three types of reference discourse. Unfortunately, it is unclear how such a rationalisation would be adopted by management for the purposes of control and coordination. Perhaps a "negative heuristic" document is required, identifying the assumptions being made at the outset of the project, which will develop as the user requirements are added, and also a "positive heuristic" document which details the ways in which progress towards the goal has been made. We might insist that the engineer uses particular presentation techniques for informative and scientific discourse, thus making VDM and dataflow diagrams, say, a project standard, or even insist that a tool for keeping track of this progress is used, but then assumptions accompanying these notations, tools and techniques would become part of the hard core. This would have the advantage of forcing management to accept the consequences of imposing paradigms on the designers.

A third possibility is that managers claim to have adopted methods when they believe the engineers are using the methods as the basis for plans. In this case the method is used to orient the project at the outset, and to fall back on during its course. The engineers will not feel bound to the plan, however, and might reject it very early on if they are finding it of little use. In this case, management's commitment to the method is largely one of education: they have placed their engineers in a position to be able to use the method (otherwise claiming to use the method would be dishonest), but they are not claiming that the method is being used in any continuing sense. This sense of "method" allows the engineers to retain control of the project, but identifies a common point of departure. The method will stand or fall on its merits for each individual project.

There is a complication, however, and that is the provision of tools support. Most tools are provided to accompany certain methods, that is, they assume the engineer is following a pre-defined plan, and the tools will become useful only if a particular point in the plan is reached. Moreover, these tools usually capture much of the "what" information about a project, thereby dictating to some extent the informative discourse that will take place in any rationalisation of the project. Once the informative discourse is determined, however, the scientific and exploratory discourse processes will be constrained to fit in. Rather than the exploration driving the process, the tools start to do so. This further confuses the slogan "specify the what before the how", for now we can see that "how" we specify the "what" may be inherent from the outset as a driving force behind the design. Adopting data flow as part of a method, for example, leads inexorably towards a procedural implementation. Once a project adopts a method as a means of orientation, and surrounds this method by an integrated toolset, the method, via its tools, starts to control the process, bringing with it all the dangers of a management imposed method, but with the added confusion that it is unclear who retains responsibility for the design.

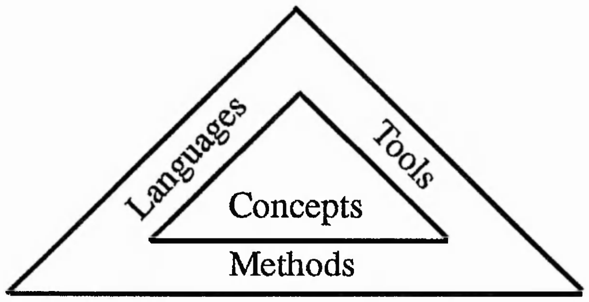

Figure 6.2: Methods, Languages and Tools

Mitchell presents the diagram shown in Figure 6.2, and asserts that it

> "captures the fact that underlying chosen tools, methods and languages are a number of concepts which will vary from paradigm to paradigm. ...[it also suggests that] we must be clear what concepts we are working with when we

design tools, methods and languages."[Mit88, Pages 8-9].

We should also note the converse, however, namely that selection of these things effectively locks us into a research program, or paradigm, by fixing the ways in which we view the world. If methods, languages and tools are forced upon us then we have little control over the world views we take. As Rapp has observed,

> "Because of this characteristic and far reaching interdependence, all individual phenomena are combined to form a comprehensive technological complex into which, it seems, the individual human being is helplessly drawn." [Rap81, Page 48]

Gregory goes even further, suggesting that

> "It is surely true of all tools, that by making some things easier they direct activity and thinking from things that are more difficult; but what is easy and what is difficult are partly set by the available tools, and so we are carried along by a sequence of largely arbitrary and sometimes unfortunate features of our technology, including our language. Human intelligence is very largely Artificial Intelligence, and even our hopes and fears (and our moral commitments, for they are set by possibilities of achievement) are largely set by existing technology." [Gre84, Page 51]

6.4 Methods as Tools

The final sense of the term "method" we want to consider is one commonly encountered in both engineering and mathematics. Asimow observes that

> "The designer encounters a host of problems which are peculiar to the process of design. ... We will speak of the analytical techniques which cope with these problems as the general methods and tools of design." [Asi62, page 3]

In addition to these methods of design, there are also methods associated with particular domains of knowledge, such as mathematical methods, and also methods from the problem domain such as double-entry book keeping. Methods, in this sense, are localised to solving well-identified problems within particular disciplines. They are useful tricks: as Polya and Szegö have said,

> "An idea which can be used only once is a trick. If you use it more than once it becomes a method" Cited in [Arb90, page 499]

It could be argued that design methods in the large, such as SSADM, are attempts to find such localised methods for the discipline of design, but they are clearly different in scale from the methods a mathematician uses when solving an equation. Moreover, the methods we are going to consider here are usually used by one individual, rather than attempting to prescribe actions for teams, and are all underpinned by bodies of theory where possible. Compiler writers, for example, use methods of parser generation based on the theory of LALR parsing. Mathematicians solving equations use the appropriate theory for the equation encountered. There is no theory of design, however, so there can be no general method of design in the sense of method used here, just as there can be no method of mathematics until a unified theory of mathematics is found. It is this use of "method" that gives rise to traditional "methods courses" in engineering and mathematics, that are differ from the methods courses in software engineering.

Consider the problem of solving the quadratic equation

$$(x + 12)(14 - x) = 25$$

A competent mathematician might solve this as follows:

$$\begin{array}{llll} x + 12 = 25 & \textit{and} & x = 13 & \textit{or} \\ 14 - x = 25 & \textit{and} & x = -11 & \end{array}$$

This method (which may surprise less mathematically experienced readers) is based on knowledge of quadratic equations. It hinges on the fact that any equation of the form $(x + u)(v - x) = c$, where $u + v = c + 1$, can be solved in the above fashion. This shows very clearly that *methods are intrinsically linked to knowledge.* Since theories are ways of expressing knowledge, we can see ample support for Rapp's statement that "nothing ...is more practical than a good theory"[Rap81, page 37]. The claim that engineers are practitioners who do not need theories is untenable, for even if we release them from proof obligations, their very methods are based upon theories. Suggestions that computing is a "practical" subject, and so "hands on experience" is more important than theory are equally ludicrous. The cause of the problem seems to be a misunderstanding of the statement that we can learn from our mistakes. Popper claims this as the theme for his book Conjectures and Refutations [Pop63], but what does it mean? Kuhn makes the important observation that

> "The individual can learn from his mistakes only because the group whose practice embodies these rules can isolate the individual's failure in applying them" [Kuh70a, page 11].

This is why we can teach some skills without imparting theory explicitly: the teacher uses theory in correcting actions The software engineer, however, needs to be able to continue to learn after formal education has been completed. It is the selfconscious aspect of design that allows this to happen: by continuously monitoring progress against a theoretical framework, the engineer can continue learning from mistakes. Individuals can construct their own theories to support methods, using tests for refutations. It is equally ludicrous to suggest, however, that modern society can be supported by a learning process where every theory is re-learnt from refutations. Engineers must be able to learn presented theories as well as constructing their own, for these will be the basis for many of the positive heuristics they are to use during the design process.

One complication here is that many engineers do not need to appeal directly to fundamental theory every time they want to use a method. Methods are often neatly encapsulated in rule form, where all the engineer has to do is match the problem to an appropriate rule, and use it. Mathematicians use this strategy, when they look up standard results in tables of integrals, for example. To use such a strategy safely, however, requires a grounding in theory. Although engineers may be excused the task of solving a partial differential equation from first principles by this approach, they must understand enough theory to identify the equation for what it is, and to check that all the conditions for using the rule are met. Moreover, they must possess a good system for classifying their methods if they are to be able to store and find the appropriate rules. The software engineer, on the other hand, has very few such rules available, so there is currently no option but to get to grips with the theory directly.

These methods will provide positive heuristics for our research program. The engineer, being faced with a theory in a particular form, and knowing the aims of the programme, will use the methods of the discipline to make progress. In the early stages of the task, these heuristics will be techniques for interviewing customers and users, methods of discharging consistency and completeness proof obligations for specifications, and so on. In later stages the heuristics will include those of refinement towards particular languages, known algorithms for sorting, and so on. We will not attempt to itemise the sorts of methods that software engineers should have at their disposal, for such a list will be subjective, and bound up in application domains and other such specifics. Rather we will consider what it means for the engineer to have a method available.

One of the distinctions that is usually made between experts and novices in some domain is their ability to solve exemplar problems from that domain. It is no longer considered that this ability is simply a reflection of stages of development, as Piaget proposed. Recent research in psychology suggests that problem solving ability is linked to the amount of knowledge the individual is able to bring to bear on the problem. This, in turn, is not simply a matter of "having" the knowledge, but how of how this knowledge is encoded. This is of crucial importance for curriculum development, for it suggests that the initial encoding of presented information can determine whether or not that information is useful to an engineer in solving problems. It is surprising, therefore, that very little seems to have been said about this in the literature of software engineering education. We should note that "experience" and "expertise" are not necessarily linked. Whether or not experience leads to the development of expertise, and what governs the process, is little understood. Littman *et al.* have shown, however, that there is "virtually no relationship between years of professional programming experience and successfully performing a [program] enhancement task" [LPLS86, page 95]. We should not assume, therefore, that "hands on" experience will necessarily develop expertise in our students. They did show a relationship, however, between successful program enhancement and the ways in which the original program was studied. Programmers who performed a systematic study of the program had more success than those who sought information as they needed it. We can interpret this as another endorsement of the theory building approach to software design.

Chase and Simon forged the link between recall and encoding of information with their studies of chess masters and novices [CS73]. They concluded that masters encode meaningful chess boards in more sophisticated ways than novices. Whereas the novice uses proximity relations to encode a board, the expert uses a number of techniques, including relationships between pieces that are relevant to the goals of the game, such as offensive and defensive scenarios. Chi and Glaser showed that physicists similarly classify problems in terms of significant structure, in this case using the underlying principles of the subject (such as the conservation of energy) rather than the rules of the game, whereas students concentrated on surface similarities (such as the mention of blocks on an inclined plane) [CG85]. They also showed that experts take longer to classify the problems than the students do, presumably as a result of doing more work in uncovering the deeper structure they are seeking. This might explain why weaker students, when given a programming problem, often start work at a terminal before the better students, for they can perform their superficial analysis of the problem more quickly. Gugerty and Olson, on the other hand, have observed that novice programmers study programs for *longer* before attempting to find errors in them than experts [GO86]. One possible explanation for this apparent contradiction is that information can be presented to people implicitly or explicitly. Implicit information requires the reader to expend more effort in finding its structure. Both novices and experts might treat programs as explicit information for the purposes of debugging, consequently the experts' familiarity with the language allows them to complete the task more quickly. This interpretation accords with Bransford's experiment that showed poor readers read documents containing implicit knowledge more quickly that good readers, failing to grasp the significance of what they are reading, but they read documents with explicit knowledge more slowly [BS82]. He concluded similarly that better readers encode with a deeper structure than poor readers, so spend longer seeking out implicit meaning.

Chi, Feltovich and Glaser found that the representation of a problem has a significant impact on the solutions that will be found [CFG81]. This is not surprising, but it adds support to the earlier statement that it is pointless insisting that specifications should be presented independently of implementation details. The choice of theory formulation language for our specification is part of the problem representation process, and hence is likely to influence the designs found. Rather than try to live up to such a slogan, perhaps we should recognise that the choice of specification style and notation will influence our design and make conscious choices of these in recognition of the fact that selfconscious design starts with adoption of a paradigm. As Hirschheim and Klein have noted,

> "the identification of paradigm along with the set of philosophical assumptions which each embraces provides a new vehicle for investigating new the-

> ories about the nature and purpose of information systems development." [HK89, page 1214]

Another implication of this research on encoding is that students may well "know" something, but be unable to apply it to problems because of unsuitable encoding. Mayer's research on learning computer programming, for example, suggests that students learn how to write and understand programs better if they are presented with a model within which to encode their new knowledge [May75]. Furthermore, Mayer noted that using flowcharts as a model was less effective that a more abstract (but still operational) model to which the students could relate. This, he concluded, was because "the [flowchart] symbols themselves provided only a second layer of code (ie. translating statements to arbitrary symbols) rather than an organising superstructure" [page 732]. This causes us to question the wisdom of teaching information systems design using modelling techniques such as dataflow diagrams. The often cited benefit that they are a close representation of the problem, easily understood by the user, may be more of a hindrance to the learning process than an aid.

The importance of finding models for teaching that are more than simply surface encodings seems paramount. For complicated encodings it may be worth seeking to establish intermediate models. Siegler has noted

> "It may sometimes be more desirable to teach towards an intermediate instructional goal than to teach directly towards the final goal. ...It may be more effective to address ...difficulties one at a time as they arise than to try to attain the ultimate instructional objective in one step." [Sie86, page 180].

This is of crucial importance, for it impinges directly on the debate as to whether we should be teaching "what industry wants". The answer may not be a straight "yes" or "no", but a "yes through the use of simplified models". This undoubtedly seems obvious to teachers of subjects other than software engineering, but the much of the literature on whether we should teach Ada as a first programming language shows that many teachers in our discipline have not learnt to separate the end from the means.

Perkins and Martin have carried out a study of novice programmers, and why they fail to solve problems that experts manage to solve. They have concluded that it is not just a matter of experts "having knowledge" that novices do not have. Rather, they suggest that novices may well have the required knowledge, but in too fragile a form for it to be used effectively [PM86]. They suggest that fragile knowledge can be categorised into four types.

Partially Missing Knowledge: here some aspects of the knowledge are missing, leaving the novice with insufficient to solve the problem.

Inert Knowledge: this is possessed, but cannot be retrieved on the cue of the problem. It is suggested that knowledge is inert as a result of the encoding process: the way we learn something determines the circumstances under which we will be able to use it.

Misplaced Knowledge: this occurs when knowledge is invoked that does not assist in solving the problem posed. This may be because missing or inert knowledge leaves the novice in despair, so misplaced knowledge is all there is left. Functional fixedness and the Einstellung effect (the carrying over of solutions from problem to problem) are classic examples of misplacement. This can also occur as a result of over-generalisation of a result or under-differentiation of a problem situation.

Conglomerated Knowledge: this is when two or more partial bits of knowledge are mistakenly joined. In a sense it is a special form of misplacement. Perkins and Martin observed that this is commonplace in novice programmers, and they speculate that it could be because novices are used to conversing with humans, and people are quite adept at overcoming errors in discourse, and unravelling disparate trains of thought. The novice has not yet discovered that computers are not capable of inferring the required program from fragments of code, but require the whole thing in a syntactically correct form. Experts, however, have learnt the lesson that programs cannot be nearly right: they are either right or wrong.

In addition, Perkins and Martin also noted that the problem of fragile knowledge in novices is exacerbated by a lack of general problem solving strategies that would help them to overcome these limitations. The strategies they lacked were often very basic, including techniques such as re-expressing the problem in alternative forms.

6.5 Summary

In this chapter we have explored four uses of the term "method". We have also developed the methodological aspects of our model through the use of positive and negative heuristics, and the idea of research programmes. In particular, we have drawn attention to the limitations of life cycle approaches to methods, and also questioned the rôle of design "methodologies" such as SSADM. We have explored what it means for someone to have a method for solving a problem, and briefly discussed some of the distinctions between experts and novices in their use of methods.

Chapter 7

Personal Construct Theory

"The educational dogma seems to be that everything is fine as long as the student does not notice that he is learning something really new: more often than not, the student's impression is indeed correct"

Edsgar Dijkstra

We have now completed our analysis of Software Engineering design, and arrived at a model around which the technical aspects of our curriculum could be constructed. Production of this model is only one aspect of the task, however, for a curriculum comprises more than just technical content. We have hinted at this idea already, by stressing the selfconscious attitude necessary for modern engineering design. Now it is time to make this explicit by taking a stance on how people will fit into our curriculum. This means we need to add to our discussion some consideration of teaching and learning theory at the very least. In fact, we have also rather neglected the question of how people construct, rather than present, their theories. As theory building and learning are similar, if not identical, concerns, we will allow our discussion to serve a dual purpose.

Using the discussion for this dual purpose, however, presents us with a difficulty of discourse structure, for the discussion of how people construct theories when designing systems would proceed rather differently from the discussion of how they do so in an educational setting. The approach we will adopt, therefore, is to provide a largely neutral presentation of a theory which is pertinent to both tasks, but we will leave the task of applying this theory to our model up to the reader. This is not because it is particularly hard to do, or because the author is too lazy to undertake the task, but simply because we have reached a place in the text where the reader has to take a more active part in the process. Moreover, this is a reflexive application of the theory we will present, so it is also a natural thing to do. Stringer expresses this as follows:

> "The goals of literary work is to make the reader no longer a consumer, but a producer of the text. ... Constructive Alternativism is a cunning built-in device of personal construct psychology to prevent its own texts being closed in a final definition. Seen reflexively, it turns reading into writing. Man-the scientist cannot help but (re)-write *The Psychology of Personal Constructs* as she seeks to make sense of Man's attempt to make sense of the word." [Str85,

pages 212-213]

We hope that this will make more sense as Personal Construct Theory is introduced in the following section.

In addition to discussing the people-related aspects of curriculum design, we should also discuss education in the wider context, by outlining the philosophical stance we are taking on education itself. In practice, however, most philosophies of education are intrinsically linked to the ways in which we view the subject and the people being taught, so rather than separate this out as an additional concern we will treat it implicitly, as indeed we have already been doing.

Although we are not going to consider our philosophy separately, a brief overview will help to set the scene for what follows. The philosophy of education we have adopted as our starting point is one of transformationalism. Students are no longer to be considered as empty vessels to be filled with learning, but as selfconscious people who are to be helped to grow in ways they determine, and in particular, who are to be helped to learn to learn. The job of the teacher is to aid in this process of learning, not to fill a vessel like a petrol pump attendant. Slaughter [Sla89] suggests that the equating of education with the imparting of second-hand "knowledge", which is actually little more than data when it is received, is potentially disastrous for the modern world. For society, the application of this received data, in the absence of true knowledge and wisdom, "can lead to world destroying technologies". For the individual, there is "a systematic frustration of the will to meaning"; people no longer have a personal understanding of the world about them. He also notes that the state of affairs where education is viewed in this way has been perpetuated by "short-termism":

> "The Newtonian/Cartesian synthesis constructed a way of looking at the world which permitted later generations to mistakenly believe that they were 'masters of nature', separate from, or above, natural processes. We are learning the hard way that this is simply not true. But instead of looking at what this implies for the future, our economic, political, and educational systems remain caught up in the business of reproducing an obsolete past. Short-termism is not just jargon for a concept applicable to business and investment; it penetrates our public and private lives too." [Sla89, page 256]

This philosophy sits upon Schumacher's Levels of Being: mineral—existence, plant—life, animal—consciousness, human—self-awareness: he notes that

> The most important insight that follows from the four great Levels of Being [is that] at the level of man there is no discernible limit or ceiling; [self-awareness is] a power of unlimited potential" [Sch77, page 48]

Slaughter adds to this by saying

"It is this [higher consciousness] (rather than new technology *per se*) which leads on to the new human and cultural possibilities." [Sla89, page 267]

If we accept this philosophy, a shift is necessary from the content-driven perception of curriculum currently prevalent in most higher education establishments, towards experiential learning, reflexive modes, and meta-level frameworks of meaning. We will consider this further in section 7.2, where it can be discussed in the light of our theory of learning. First, however, we must develop such a theory. We have selected Kelly's Personal Construct Theory, as this is widely accepted as an excellent basis for the emerging study of student-centred learning, which fits very well with our ideas of selfconscious design and individual responsibility. Harri-Augstein expresses this perfectly:

"It was through Kelly's craft that a breakthrough was achieved into a humanistic technology that allows meaning to emerge in individual terms and yet retain some systematic form." [HA85, page 61]

7.1 Outline of the Theory

Kelly's Personal Construct Theory (PCT), although nearly 40 years old, is still considered radical [Fra88]. It is a complete psychology, with explicit structure which we will retain in our presentation. The similarities between our model of system design and PCT should become apparent as we proceed, but to to aid orientation, we will start by giving an outline of the fundamental similarities, leaving the reader to construe other similarities and points of contact for his or her self. This, as will become apparent, is a reflexive use of PCT itself. We would also argue, that such a presentation is of more use to the curriculum designer than a catalogue of points on which the author has noted similarities. Moreover, we will not go into details of how the ideas presented would be applied to practicalities, such as deciding upon content or teaching method, for these details would involve us in far too many auxiliary discussions to establish context, and are properly the concern of each individual teacher and curriculum designer. The next chapter, however, provides an overview of some designs that have been produced in the light of this analysis.

PCT is an anticipatory, rather than reactive, psychology. Consequently it fits in very well with Popper's philosophy of science, based upon conjectures and refutations. Both the person and the scientist are seen as proactive, rather than active only if provoked by stimuli. This similarity has ben noted and extended by Mancini, who writes:

"[Kelly] represents for psychology what Popper represents for epistemology; that is, the attempt to reconcile constructions to the possibility of knowing the 'real' world, and improving such knowledge, on a firm basis." [MS88, page 69]

As we shall see, the central themes of PCT are very similar to those of our model of system development, and also to those of our (briefly sketched) philosophy of education.

selfconsciousness plays a key rôle in all three, as does the individual's construction of theories (or construct systems) and their validation against experience.

PCT is based on constructive alternativism, a philosophy where our constructions of the world are subject to revision and replacement, and there are always alternative constructs open to us. These constructs, however, may differ in their effectiveness in allowing us to predict and control events in our world. They also allow us to do away with inductive reasoning as a separate device, for our predictions about the world can be seen as deductive, based on currently held constructs. Like Kelly, we will leave aside the bootstrapping task of obtaining the first construct.

PCT is a fully articulated theory, comprising a fundamental postulate and eleven corollaries. These corollaries are not simply deductions from the postulate, but amplifications of it that serve to explain the central ideas behind the theory. In addition to the theory itself, PCT is also closely associated with a methodology of research based on repertory grid analysis. Indeed, the association is so close that Fransella refers to grid analysis as the "security blanket" of personal construct theorists, allowing them to retain the quantitative aspects necessary to be "respectable" psychologists, whilst accepting a radical theory" [Fra88, page 30]. We will break with tradition here and utilise the theory without reference to grid analysis.

Fundamental Postulate

> *A person's processes are psychologically channellised by the ways in which they anticipate events.*

In PCT a person is a fundamental unit, rather than various collections of processes such as cognition, perception and memory, which are the objects of concern in more traditional psychology. Kelly takes the view of "person-as-scientist", and stresses the individuality of the whole person. This allows us to consider theory building and selfconsciousness as primary, which is very difficult to achieve if we remain in the realm of traditional psychology. Although we might construct a notion of selfconsciousness in social psychology, for example, this would not be immediately reconcilable with a similar notion we might build in cognitive psychology. The holistic view of the person is the major distinguishing feature of PCT, and should not be forgotten. It does, however, cause some major problems of presentation, for the theory is highly reflexive and is, therefore, whatever you construe it to be. In particular, the only definition we can give for "events" is that they arise as the result of the individual choosing to "chop up time into manageable lengths" [Kel55, page 52]. Thus an event for Tom may comprise several for Dick, or be considered only part of one by Harry.

The person-as-scientist view also suggests that we could try to apply some of our discussions of science to the theory. Horley, for example, suggests that we could see a person's endeavours not only in terms of events, but as programmes of events [Hor88]: these show a clear similarity with Lakatos's research programmes. McWilliams goes further than the person-as-scientist view, when he argues that we should really construe PCT as advocating the person-as-anarchist. This is a natural development of following Feyerabend's view that science is broader than just the rationalist western traditions usually assumed. It may seem that such a move would be nihilistic; McWilliams argues this is not the case, for

> "as personal consciousness evolves, a natural tendency of the person is to attempt to discover the patterns of nature and live in harmony with them. Anarchist philosophy suggests that when this principle is followed, and constructs are revised to correspond more closely to events, human, social conduct will naturally be 'moral' and cooperative. In contrast, when construing fails to adapt to the nature of events, conflict and 'immoral' behaviour towards others, seen as hostility, occurs." [McW88, pages 17-18]

Hints at this idea of anarchy can be found in the heuristics accompanying such techniques as brainstorming [Osb63] and synectics [Gor61]. We should beware of attributing too much to such techniques, however, particularly the increased flow of ideas that is sometimes claimed. Evidence suggests that although n people in a group produce more ideas in m minutes than a person working alone, they produce significantly fewer ideas than n people working individually. The benefits of group creation seems to come in the evaluation and development of ideas, rather than the initial insights [LT73].

This notion of "hostility" raised above will be considered further as the theory is developed in more detail, but we will not pursue the idea of person-as-anarchist. Although, in retrospect, it might be interesting to replace software engineering as theory building with software engineering as anarchy, we will heed Burkhardt's warning:

> "If innovation is not close to the familiar, few people will buy it; if it is too close, why bother to develop it." [Bur89, page 8]

and accept that our theory building view is probably as radical as we can go without risking rejection. Suffice it to say that as our view of science changes, so too our construal of PCT will change.

Construction Corollary

> *A person anticipates events by construing their replications.*

Constructs for Kelly comprise both similarities and contrasts, thus we cannot have the construct "looping program", but we may have the construct "looping program/non-looping program", although we will often present a construct in terms of one emergent pole, and allow the other to remain implicit. Moreover, constructs need not be representable verbally, so it is perfectly possible for an individual to have ways of construing the world that cannot be made public verbally. Subsequent corollaries expand on the idea of a construct and relate the private nature of constructs to social behaviour.

The constructs we place on events are personal, and not simply a product of the events themselves. Bannister and Fransella provide an excellent example of this when they discuss the problem of conditioning an individual to blink every time a prime number is shown on a screen [BF86, page 9]. Numbers are flashed up, and, if a prime is shown, a jet of air is directed into the eye. After a time, we might expect the subject to anticipate events by blinking when a prime is shown, even if the jet of air is absent. This will only occur, however, if the subject construes the number as prime. If we attempt to condition someone who is ignorant of the notion of primeness, we cannot succeed. We may succeed in inadvertently conditioning a reflex to all odd numbers and two, if that is how the events are construed. People do not respond to stimuli, but to what they perceive them to be. This is clearly a very important message for those charged with the task of carrying out requirements analysis: is is not the requirements that we are analysing but the people making the responses.

This is crucial to teachers, for if we see one aspect of education as conditioning students to respond to events in particular ways, such as solving simple equations almost by reflex, or invoking the idea of an invariant whenever a loop is presented, we need to be aware that it is not enough simply to present examples and leave the student to infer the connections. Much "bottom-up" teaching seems to proceed in this way: numerous examples are presented, and the student is left to infer the greater picture. This can only happen, however, if the student's construct system is already sufficiently developed for the perception replication to take place. Presenting twenty examples of proofs by contradiction will only help the students to anticipate solutions to the twenty first problem, if their construct systems allow them to construe the "proof by contradiction-ness" of the past events, so that they can see the new problem as a replication.

The replications referred to are predicted events, but not in their 'real', concrete, form. Rather they are abstractions of sets of properties, that is, of other constructs. Thus if I construe a program as "badly written" as opposed to "well written", I may predict from this the event that it will be difficult to modify. This difficulty need not have a concrete existence as a "thing", however, but will manifest itself as a number of properties that the event will have, such as a number of errors made when modification is attempted, the need to rewrite several regions of code when making a simple change, and so on.

Individuality Corollary

Persons differ from each other in their constructions of events.

This is Kelly's notion of individual differences. What makes people different is that they will live their lives differently, as a result of seeing the world differently. Software engineers will produce different systems to solve the same problem because they will see the world differently, that is, they will use different construct systems. If we want conformity, we must enable all engineers to see the world through the same set of constructs. It is not enough to impose methods, of course, for these methods may themselves be viewed differently. If we can persuade each engineer to see the world through a construct from which they anticipate being sacked if they do not use the method in some algorithmic way, and if we can also persuade each engineer that they do not want to be sacked, then conformity may occur. It may also occur, of course, if they see other benefits from commonality of behaviour. This is significant, for it shows that constructs do not need to be believed to be used: we can use a construct system because some higher level constructs tell us it is expedient to do so. Of course, if we do this by encouraging students to develop constructs such as "will lead to being sacked" there may be implications. For example, it is likely that such a system will not be compatible with constructs such as "taking pride in work". If we want them to consider using methods, then we must help them develop these higher order constructs so that they can make the choices. In many ways this highlights one of the most significant aspects of this research programme, for we are now faced with the question "can the theory building approach ever work, or will it just lead to Babel"? Clearly one of the reasons that methods are seen as useful in software engineering is because they are construed as a way of avoiding the chaos that would ensue if engineers were allowed to view the world as they wish. We will not attempt to answer this question, but we will suggest that in a selfconscious culture, where the individual is sufficiently well educated, Babel will only follow if the individuals desire it: after all, they are free to control their world. If we do not trust engineers sufficiently to order their world, it is our construct system that is to blame. If we do not educate them sufficiently for them to be able to do so, it is our education system that is at fault.

Organisation Corollary

> *Each person characteristically evolves, for their convenience in anticipating events, a construction system embracing ordinal relationships between constructs.*

Constructs are built into hierarchies, such that each construct may have one or more subordinates and superordinates. This structuring can take two distinct forms. First, the subordinate structure may run in the same dimension as its superordinate, thus if we have the construct "evaluative/descriptive", then the construct "procedural programming language/functional programming language" may be made subordinate, in such a way that procedural languages are seen as evaluative, whereas functional ones are seen as descriptive, as shown in Figure 7.1. Alternatively, we may see the subordinate construct

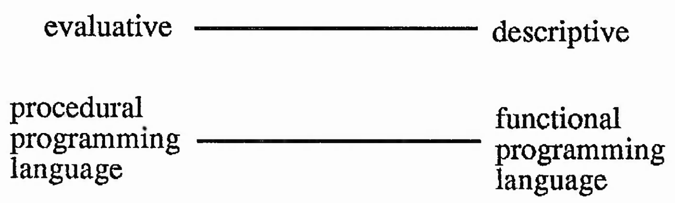
evaluative
descriptive
procedural programming language
functional programming language

Figure 7.1: Longitudinal Subordinate construct

itself as evaluative or descriptive. In this case, the subordinate construct runs across its superordinate. Figure 7.2 shows two fragments of a construct system where the distinction between procedural and functional languages is seen as first as evaluative, and then as descriptive.

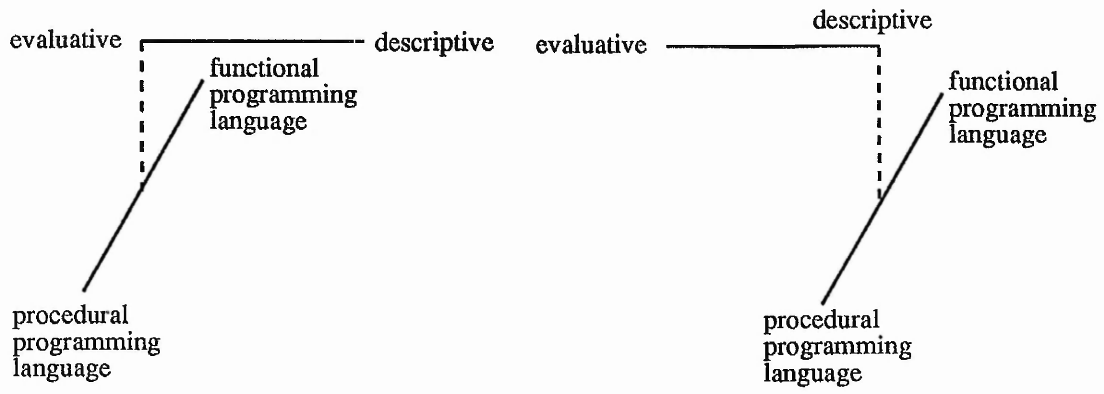
evaluative
descriptive
functional programming language
procedural programming language
evaluative
descriptive
functional programming language
procedural programming language

Figure 7.2: Transverse Subordinate Constructs

It is important to realise that construct systems change, and subsystems can be deliberately selected for particular purposes. Thus, for some purpose we may classify "Fortran" as a procedural language and be using a purely descriptive construct, on another occasion, however, we may be using the construct as a means of evaluation. This can cause considerable confusion to students, particularly if they do not yet have sufficiently well developed construct systems to allow such flexibility.

Consider, for example, a student who construes "bad/good" as having a longitudinal subordinate "worth learning/not worth learning". Such a student will have considerable problems if told, by someone he or she trusts, that Fortran is a bad language but one that is worth learning. The simple option is to re-construe Fortran as good, or to refuse to trust the teacher any more. In the longer term, however, and in the face of persistent events of this nature, the student's construct system may be unable to cope without structural reorganisation. The greatest challenge facing teachers is to aid this reconstruction. In this sense, the teacher is a psycho-analyst: the analyst is helping "abnormal" people to develop more "normal" constructs, and in particular to reach a set of constructs where therapy is no longer required; the teacher is helping the "normal" student to develop more helpful constructs for coping with general and specific tasks, including the task of carrying on the development process itself. Construing the teacher's rôle in this way is potentially quite threatening: it somehow seems much easier to view ourselves as deliverers of information than as analysts responsible for the development of our students. As we shall see when we move on to consider social behaviour, however, the ways in which we construe our own rôle has implications for how the students will construe theirs.

Dichotomy Corollary

A person's construction system is composed of a finite number of dichotomous constructs.

In PCT, Kelly has adopted the view that it is useful to see constructs as demonstrably bipolar, with some objects and events being outside the range of particular constructs. Thus our construct "procedural programming language/functional programming language", for example, must have elements we can use to illustrate each pole. In fact, Kelly insists on at least two at one pole, to show that there is at least some degree of sameness inherent in the construct, and one at the opposite pole, to demonstrate contrast. Moreover, we are free to decide that the element "cabbages" or the element "Prolog" is outside the range of the construct if we wish. The range, like the construct itself, is personal.

Clearly there is a similarity between constructs and predicates (in a typed system), and between construct systems and formal systems, but constructs may be more complex than just predicates. Kelly suggests three types of construct:

Pre-emptive constructs: any event construed under such a construct may be pre-empted from being construed in any other way. These are "nothing but" constructs — "communism is nothing but dictatorship", "SSADM is nothing but pictorial rubbish", "formal methods are nothing but the saviour of Software Engineering". Teaching students in such a way that they see the world in terms of such pre-emptive constructs limits their growth, for subsequent invalidation leaves the student with no option but rejection and major reorganisation. Thus a student who comes to see a working program as nothing but a good program will have considerable problems when faced with feedback saying that a working program has failed to meet certain criteria of assessment. This student will be unable to learn much about programming until a dramatic readjustment of the construct system has removed the pre-emptive construct.

Constellation constructs: these build links between constructs of the form "if it is an A then it is a B". Figure 7.3 shows a constellation where the construer sees all programs as deterministic and executable. Constellations are vital, but inappropriate ones lead to stereotypes and blinkered vision.

Propositional constructs: these are quite free. No subordinate or superordinate structure is imposed upon them, they are free agents. They model predicates, rather than laws, in our formal system. As we shall see, however, they need not form a consistent set,

As well as this classification, it is also useful to consider constructs as permeable or impermeable, depending on how constricted the range of application is seen to be. Thus if we are only prepared to consider traditional programmiung languages under our "procedural/functional" construct it is impermeable: if we are prepared to see Martin-Löf type

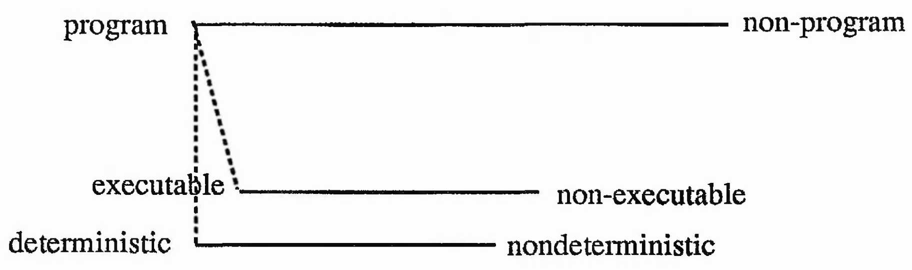

Figure 7.3: Constellation Constructs

theory as within the range of this construct, it is more permeable. If we also see common vegetables as objects in the domain, the construct is very permeable! Permeability is important when we move on to consider educational processes, for it will allow new ideas to be assimilated into existing structures.

Choice Corollary

Persons choose for themselves that alternative in a dichotomised construct through which they anticipate the greater possibility for the elaboration of their system.

This corollary mirrors Popper's searchlight theory of knowledge: we do not construct theories by filling a bucket with miscellaneous facts, but by actively searching out the best facts for our purpose. Kelly is suggesting that "better" choices are those that lead to an elaboration of our construct system. Elaboration comprises two aspects, extension and definition, which are analogous to Popper's generality and precision respectively. Extension is the development of constructs that have wider ranges of applicability; definition is the development of more precise constructs. The choices that people make, based on their constructs, can be deeply significant (such as choosing to see themselves as successful or unsuccessful people), or useful simply for solving a simple problem (such as choosing to see a quartic equation as a quadratic in x^2 to make it amenable to simpler methods of solution).

The choice corollary is very important, for it leads on to some fundamental aspects of the theory. For example, we can choose to take safe, constricted views, anticipating threats to, and the possible destruction of, our construct system if we take risks, or we can take extended views, anticipating maximal growth of our system through the use of bold conjectures. Popper advocates the latter course of action at all times, but then he is only concerned with detached theories, firmly placed into the third world. Construct systems, however, involve not only these scientific theories, but also self-image. PCT, like Kuhn's social view of science, proposes that scientific decisions cannot so easily be divorced from personal decisions. If scientists construe refutation of their theories as destructive of their reputations, then bold conjectures are threatening, and they must weigh up personal progress against scientific progress.

Software engineers need to take extended views in order to see possible solutions to problems: this is threatening, as they may lay themselves open to ridicule. They also need to be able to constrict their views, however, when they are moving towards an implementation. Constriction too early will lead to simple cloning of existing systems, and the inability to construe replications for novel situations. Extension too late may lead to divergence from the solution when effort needs to be expended refining the construct system in order to solve more localised problems. The engineer who walks into a crisis meeting, called to discuss why the final stages of coding are behind schedule, and announces that he has just realised that the system is a cucumber can expect some hostility. The engineer who, during a concept meeting, provides a similar insight, that leads to the realisation that the system should be considered using cylindrical, not cartesian, parameters, may be hailed as a genius. The choices of construct developments we make are therefore governed by constructs just like any other decisions. We must help our students to develop the construct system that governs these choices. This is selfconscious design and learning to learn.

That the choice corollary maintains choice as part of the construct system is crucial. As Kelly notes,

> "Frequently the therapist finds it difficult to understand why his client, in spite of insights which would appear to make it clear how he should behave, continues to make the 'wrong' choices. The therapist, seeing only the single issue which he has helped the client to define, often fails to realise that, within the system of personal constructs which the client has erected, the decision for action is not necessarily based on that issue alone but on a complex of issues." [Kel63, pages 67-68]

We have all experienced this as teachers or parents. Students and children often seem to have all the "information" they need to solve a problem, but persist is in getting it "wrong". A typical scenario might run as follows:

Parent: If you share out a banana equally between two children, how many will they get each?

Child: One.

Parent: Now think, if you share out an apple, how much do they get?

Child: Half.

Parent: Good, and an orange ...

Child: Half!

Parent: Very good, now a banana ...

Child: One!

The subtlety of construing apples, oranges and bananas as things that share out in similar ways probably escapes us, for we are very familiar with it, but children are so often taught to do "sharing problems" with round objects that sharing a banana may well be seen as a completely new problem and not as a replication of past events. Similar scenarios drawn from teaching students programming ("You know how to do this in Pascal, and you know the syntax of Ada, why can't you solve the problem?") or mathematics ("But you know how to do this for relations, and you have just agreed that functions are special sorts of relations, so why can't you do it for functions!") will be familiar to all teachers of Computing. If we don't encourage students to develop flexible construct systems, we must expect these problems of transference.

Range Corollary

> *A construct is convenient for the anticipation of a finite range of events only.*

As well as developing suitable constructs for seeing the world, students need to develop the ranges suitable for the application of these constructs. It is important to realise that the range of a construct can include other constructs, thus the range of the construct "property of a program/not property of a program" may have poles epitomised by the constructs "compiles", "terminates", ''red", or "good program/bad program".

Experience Corollary

> *A person's construction system varies as they successively construe the replication of events.*

This is one the most significant corollaries for our purposes, for it makes learning an integral part of a person. Kelly makes this point explicitly, when he writes

> "Learning is not a special class of psychological processes; it is synonymous with any and all psychological processes. It is not something that happens to a person on occasions; it is what makes him a person in the first place." [Kel63, page 75]

Moreover, this clearly indicates that "development" is not something that happens only to children, as developmental psychology often seems to imply, but a continuing process of life.

It is important to realise that learning is not brought about by events alone, but by the construing of events. Kelly reserves the term "experience" for such construing of events.

> "It is not what is happening around him that makes a man experienced; it is the successive construing and reconstruing of what happens, as it happens, that enriches the experiences of his life. ... it is when man begins to see

> orderliness in a sequence of events that he begins to experience them." [Kel63, page 73-74]

Popper makes precisely the same point:

> "We do not stumble upon our experiences, nor do we let them flow over us like a stream. Rather, we have to be active: we have to '*make*' our experiences. It is we who always formulate the questions to be put to nature; it is we who try again and again to put these questions so as to elicit a clear-cut 'yes' or 'no' (for nature does not give an answer unless pressed for it)." [Pop59, page 59]

This is crucial, for it turns on its head the commonly held notion of learning by experience. As Kelly says, "it is the learning that constitutes experience" [Kel63, page 172] not simply the experience that brings about learning. Thus experience is not primarily a function of time spent on the job, but of our revision of construct systems. We can be in the vicinity of events for years, but experience very little, conversely, we can gain plenty of experience in a short time if the situation is right.

This corollary also shows the flaw in the idea that we should expose students to a wide range of experiences. This cannot be done, for only the students themselves can construe, thus turning events into experiences. What we can do, of course, is to present events which we believe they are capable of turning into experiences, and also attempt to provide the right sort of environment for this to occur. In particular, we need to encourage students to develop permeable constructs, for

> " a person who approaches his world with a repertory of impermeable constructs is likely to find his system unworkable through a wider experience of events. He will, therefore, tend to constrict his experiences to the narrower ranges which he is prepared to understand. On the other hand, if he is prepared to perceive events in new ways, he may accumulate experience rapidly"[Kel63, page 172]

Bannister, in a study [Ban65] which has also been confirmed by Fürst [F̈78], has shown that there is a relation between the sort of feedback received and the resulting changes we make to our construct systems. Validating events are likely to lead to a tightening of the system, so that it is subsequently interpreted as more deterministic; invalidating events lead to a loosening of constructs. If we establish an environment in which students carry out drill exercises, which will validate constructs but rarely challenge them, we will certainly reinforce the system, but we will also change its nature, causing the students to see it as more deterministic. This may serve them well if they are faced with rote problems, but when creativity is required they need to explore events in ways in which their system does not fall apart every time they make an incorrect predication. If they see the world in terms of right answers, however, and their rôle as questing for these answers, they may well flounder. Kelly suggests that good development is encouraged by cycles of tightening and loosening, rather than extended periods of one or the other. This idea seem analogous to the simulated annealing used in the learning stage for neural networks.

Exactly what changes do we make when invalidation occurs? Kelly suggests that our constructs are layered, so that we attempt to change the most exposed constructs, which are likely to be those which our experiment set out to test [Kel55, page 160]. In this respect our personal construct system has a (layered) core, protected by an outer level, similar to the hard core proposed by Lakatos for research programmes. One of the criticisms of PCT is that this layering is not explicit. Duck has suggested that it is possible to express this within the theory itself [[Duc83] cited in [Jah88, pages 9-10]]. This is significant, for it suggests that the theory is powerful enough to express both "content" constructs and "process" constructs.

We should also note that changes to a construct system are not automatic simply because we experience invalidating events. We can choose to ignore inconsistencies (as scientists do in Kuhnian paradigms) for inconsistency is itself a construct that is open to change, and we may perceive inconsistency more in terms of maintaining a consistent wide view than as conflicts between local details. Thus we do not reject a construct system that is serving us well in coping with life simply because of one small inconsistency. Popper's view on refutation can be attributed to his separation of science from the scientist, and the formation of closed scientific theories.

Modulation Corollary

> *The variation in a person's construction system is limited by the permeability of the constructs within whose range the variants lie.*

It is not only our externally visible actions that are governed by our construct systems, but also the changes we make to the system itself.

> "Even the changes which a person attempts within himself must be construed by him. The new outlook which a person gains from experience is itself an event; and being an event in his life, it needs to be construed by him if he is to make any sense out of it." [Kel63, page 79]

If a student has a construct "part of Software Engineering/not part of Software Engineering" such that the discipline is seen as nothing more than writing working programs, this impermeable construct will inhibit changes as a result of experience. If we, as teachers, endorse such impermeable constructs, teaching that design *is defined by* the life cycle, for example, rather than presenting this as one view of the process, then students will have great difficulty learning beyond these constructs. Note that it is not whether we view life cycles in this way that is important, but how our students construe them. If students construe "worth learning" as a longitudinal subordinate of "part of Software

Engineering", and their view of the discipline is limited to a few simple constructs, then they will never develop as software engineers, choosing to learn only those topics that fall within their limited construct systems.

It is not only at the technical levels that this presents a problem, of course, for a student who has an impermeable construct for "true/false" such that "said by lecturer/not said by lecturer" is seen as a subconstruct, may have great difficulty in higher education, where truths are not the order of the day. Such students are unlikely to reconstruct their systems simply in response to technical events, they are more likely to revise the construct pertaining to individual lecturers. To change this aspect of the system requires events targeted at the task.

Fragmentation Corollary

A person may successively employ a variety of construction systems which are inferentially incompatible with each other.

As we develop our construct system, it is generally the lower level constructs that change as a result of experience: we do not, in general, change higher-level constructs every time a detail is out of place. The new subsystems we devise need not be consistent with the old ones, but in a "normal person" they will be consistent with the established higher-level constructs. The fundamental postulate carries connotations of progression, and consistency needs to be seen with respect to this progression. For example, an event may cause us to shift from construing Fortran as a bad programming language to construing it as a good one. This may seem inconsistent, but if it is accompanied by the development of a new construct, such that Fortran is seen as good when applied to numerical analysis type problems, the wider consistency of the system is maintained.

Commonality Corollary

To the extent that one person employs a construction of experience which is similar to that employed by another, their processes are psychologically similar to those of the other person.

This corollary amounts to a rejection of stimulus-response psychology, for Kelly is asserting that it is not events that lead to behaviour, but the construing of events. Thus if two people behave in similar fashion it is because they construe events similarly, not because the events were similar. This is of fundamental importance for education, for it suggests that if we want to produce software engineers with certain behavioural characteristics then a common curriculum will only work if the students all possess similar construct systems on entry. It is noticeable, for example, how mature students behave very differently from the more traditional students who come straight from school. With widening access policies, such issues become crucial. If we accept PCT, and yet we still establish our curriculum in terms of common aims and behavioural objectives, we have to accept as a consequence that the experience for each student should be tailored, to develop their construct systems in such a way that the required behaviour is expected. PCT is closely associated with the idea of student-centred learning, but unfortunately this idea has come to be construed by many academic administrators as "cheaper". In fact, of course, just as individual therapy is likely to cost more than group therapy, so we should expect student-centred learning to cost more than group-centred learning.

With "formal education" in subjects like mathematics and physics, where most of the constructs developed have been formed with the help of a teacher, we can expect some conformity. The teacher carefully controls events, and provides the validation. After several years of experience, the student comes to develop a construct system suitable for algebraic manipulation. This construct system, however, may well depend on a teacher for validation: the pupil may not construe that $x + x = 2x$ for any reason other than the replication of teacher's validations. If we subsequently remove the teacher's validation from the system, the student is lost. When we ask students to make explicit aspects of their construct system, by explaining why they predict that $x + x = 2x$, we do not accept as an explanation "because I got ticks for saying so for many years", although in practice this may well be their reason. It is important, therefore, that if we want students to be able to continue learning after we remove the teacher then we should help them to develop not only a construct system that conforms because of external validation but also a construct system that will continue to function when teacher presence is removed. This amounts to a construct system that admits to the loss of innocence.

We should also remember that similarity in behaviour can arise because people have decided they want to behave in the same way. When twenty students give very similar answers to an examination question, we cannot infer simply that they all see the problem, in the same way because they have common constructs. It may well be the case that they have all seen the problem of passing the examination in the same way, decided upon the sort of answer that is expected, and construed the problem accordingly. Thus commonality leads us naturally to consider problems of sociality What we, as teachers, expect of our students will be construed by them, and can effect their actions. We are part of the problems we set.

Sociality Corollary

To the extent that one person construes the construction processes of another, they may play a rôle in a social process involving the other person.

Commonality is neither necessary nor sufficient for social progress. There are many situations where it is better for people to behave differently in groups. The sociality corollary, therefore, is crucial if we expect software engineers to function within teams, interact with customers and users, and so on. Note that, unlike fragmentary notions of psychology, we do not have to shift from cognitive to social psychology to discuss this aspect of behaviour. PCT is a psychology of the person, and the person can function as part of a group.

Kelly's term "rôle" needs some explanation, for it is not something that is imposed upon a person, but something that comes from within. Thus we cannot say that a software engineer has a rôle to play in a team, only that the engineer has chosen to play the rôle. The rôle chosen corresponds to the set of constructs adopted to fit in with others. Only one person need adopt a rôle in social interaction, although it is not uncommon for more participants to do so. Thus it is sufficient for the engineer to adopt a rôle in discussing requirements with a customer: the customer does not need to adopt a rôle too.

In playing a rôle, the person does not simply adopt the construct systems of other participants, but seeks to construe their construction systems sufficiently to adopt a suitable rôle. The engineer does not necessarily look at the world as a computer user when discussing requirements, but must be able to construe the world as a user would in order to participate in social discourse. This is a high-level skill, that requires a flexible and yet robust construct system. Flexible because the construct system must incorporate the user's constructs readily. Robust because we do not want rôle playing to influence unduly our own construct system. Being able to see the world as a child should not make us become child-like in our constructs. It is commonly noted that teachers, who spend a large amount of time rôle playing, often inherit aspects of their pupils behaviour.

This leads us to a very important point: in construing the construct systems of others we act as a personal scientist, but for complex problems, we need to make public the theory we are devising. The bipolar constructs of PCT are inadequate for this, for they amount to a binary system, and we require much richer structures for expressing typical software engineering problems. These are provided by our theory presentation language, and it is the formal education of software engineers that will permit us to construe these in common ways, thus allowing the theory presentations to act as specifications. Treating the writing of the theory presentation as a sequence of events leads to exploratory discourse, where we debate (possibly with ourselves) what we mean by what we have written or said. Using theory presentations to carry out proofs will lead us to construe properties such as consistency and correctness, which may in turn lead to further constructs such as fitness for purpose. This is crucial, for it forms the missing link between Hoare and Gries, with their scientific theories, and Naur, with his psychological theories (or personal construct systems). Note that it is not sufficient for students to learn *how* to write specifications, they also need to appreciate how they, and others, might construe them.

Since the customer will not, in general, have undergone the same sort of formal education, these external theory presentations will need to be made available in other forms if they are to be used as the basis for contractual boundaries. The forms will typically involve translation into natural language, implementation of prototypes, or interpretation by the engineer. In all of these forms, the customer-as-scientist can use this theory presentation as the basis for experiments, deciding whether invalidation should lead to revision of constructs, or the construal of the theory as "wrong".

Teaching and studying are both social processes. Teachers construe their students' constructs, and vice versa. Thus teachers may adapt to interact with each student, and students adapt to interact with each teacher. If we wish to communicate a useful set of constructs for software engineering, therefore, we need to participate in a social process. We might throw all the onus on the student to adopt a rôle, of course, but typically the teacher, supposedly having the better developed construct system, will adopt a rôle too. This means that the teacher not only has to understand the subject, and be able to make constructs explicit, but also has to understand the constructs that the student will be bringing to bear in the social process, and engineer events that will help the constructs to develop in the desired way. This is widely acknowledged amongst primary school teachers, who realise the importance of understanding how their young pupils see the world, but seems to be largely ignored in higher education. This might be as a result of developmental psychologists giving the impression that development stops at adolescence, so that students are already developed by the time they reach us and so, since they are adults like us, our constructs will be similar. In practice, of course, even if some aspects of their construct systems will lead to commonality, all students, and indeed all teachers, are different.

7.2 Discussion of PCT

In the next chapter we will discuss in detail how we might teach a few aspects of Software Engineering that arise from consideration of our model. Before doing this, however, it is convenient to consider some general educational points that arise from PCT, and how they might influence our curriculum design task.

The first thing we can note is that the nature of "curriculum" itself has to be understood before we can attempt design. If we view the curriculum as a rigid plan for action, whilst embracing PCT, then we must believe that we can predict all of our students' construct systems on entry, and also how they will develop. As this seems untenable, we need a more flexible view of curriculum. Doll sums up the stance we will take, when he writes

> "In a modernist perspective, curriculum plans are to be well articulated, with ends clear and means precise. This is the key to the Tyler-Hunter model. In a post-modern curriculum there must be, as Dewey realised, a sense of indecision and indeterminacy to curriculum planning. The ends perceived are not so much ends as beginnings; they represent ends-in-view, or beacons, which act as guides before the curriculum implementation process begins. But once the course develops its own ethos those ends are themselves part of the transformation; they, too, along with the students, the teachers, the course material, undergo transformation. The locus of power and direction shifts from the external to the internality of the course experience." [Dol89, page 250]

In essence, Doll is noting that curriculum design is an E-Problem, bound into a continually changing environment, where the quest for solutions changes perceptions of the problem significantly, thus the curriculum is not a final product, but an on-going process.

One way of helping curriculum designers to solve this E-Problem is to provide a number of resources upon which they can draw when making situated actions. Most of the Software Engineering education literature, however, contributes to this process in the form of sets of aims, objectives, and course content. That is, it offers informative discourse. Our contribution here is rather more fundamental; it seeks to initiate some of the exploration necessary for the process that currently seems sadly lacking. We would argue that the provision of aims and content is not enough, and is of limited use at the current stage of development of the discipline. Adopting the ACM curriculum [Do89] does not help with the process of curriculum design, it simply defers the real issues until the implementation. Inclusion of "relations" in a published discrete mathematics syllabus is only of real value if it is accompanied by a position on why they are included, how they should be taught, how the student should be encouraged to perceive them, and so on. Similarly, discussion of Ada as a first teaching language is only of value if carried out within a community that has a sufficiently refined notion of what it mans to teach "programming", what a programming language is, and what value systems are being used to make the judgement.

One of the grave dangers in writing about curriculum design at the level of course aims, objectives and content is that some teachers may feel justified in, or coerced into, adopting ideas put forward without gaining a deep understanding of the issues involved. They adopt topics and teaching methods, for example, *on authority*, thus avoiding the loss of innocence. They use group work, practical sessions, and CASE tools not because they have a good reason, which they are prepared to defend, but because someone else has suggested they should. Moreover, they frequently misinterpret these suggestions in quite fundamental ways. Schupp, for example, has noted that the proposals for "New Mathematics" arose in reaction to the Soviet launch of Sputnik in 1957. The end was to be an improvement in the uses of mathematics for science and technology. He goes on to note, however, that

> "the means (structural cleaning up, logical foundations, lingual exactification) became the ends. ...one can state that *New Math* certainly *did not* promote applied mathematics teaching." [Sch89b, page 41]

The same problem of misinterpretation has been noted by Marion with respect to the typical discrete mathematics syllabus in Software Engineering:

> "how does one prevent such a course from degenerating into a collection of discrete topics or from being used as a way to rush through as much material as possible so as to get to the good applications? ...how does one keep uppermost in mind the need for mathematical rigour?" [Mar89, page 276]

The problem of coherence is fundamental to this research programme, for our model has sought to demonstrate that there is at least one way in which many of the topics usually associated with Software Engineering may be reconciled. This model may be used as the basis for teaching, providing constructs for the students to use when assimilating material. It may also be used as the basis for curriculum design, thus providing a framework for rational discussion.

It is the acceptance that the curriculum is not just a text describing what must be done, but a set of constructs that will guide teachers and students through the learning process, that justifies our choice of title for this thesis. There is no intention of producing "the curriculum" for Software Engineering, but hopefully by helping teachers to develop appropriate constructs for Software Engineering education, and by raising events for them to validate or invalidate, a contribution to curriculum design is being made.

PCT suggests that one of the most significant rôles the teacher will play will be to use his or her construal of the curriculum to engineer a suitable environment for students to develop new constructs. Kelly has suggested a number of heuristics for doing this and, although these were originally intended for therapists rather than teachers, we will briefly see how they may be applied to education.

Use of Fresh Elements

It is useful if a fresh set of elements is provided as the context in which new constructs are to emerge.

This is interesting, because is seems to run contrary to conventional educational practice, which is based on teaching by analogy. In fact, however, we must take care to distinguish between developing new constructs and extending the range of existing ones. Analogy often helps us to bring existing constructs to bear on predicting events, but not to develop entirely new constructs. If we are trying to add a new construct such as "formal/informal languages", which students currently do not have, there does seem to be some support for this heuristic, for introducing such a construct with well-understood examples such as arithmetic expressions is fraught with problems. Students want to construe "2+3" and "3+2" as the same string, because they are bringing a rich construct system to bear on the problem, and they construe too much from it. Using abstract examples, in the sense that they are not construed as replications by the students, overcomes this problem, and once the construct has been established, the range can be extended by giving several examples of existing elements already bound into the construct system.

Experimentation

The next condition which is hospitable to the formation of new constructs is an atmosphere of experiment.

An environment should be provided in which the students can try out new constructs in relative isolation, thus we try to avoid establishing situations in which complex constellations occur. To achieve such an environment the effects of the experiment must be limited. This is analogous to the use of simplified models for teaching science, where students are encouraged to develop constructs such as energy conservation by experimenting in worlds consisting of frictionless planes and inextensible strings. Only later do we provide environments where heat is dissipated through friction.

We can use these two heuristics to discuss the debate between teaching bottom-up, where details are introduced before general principles, and top-down, where principles are taught first. It is often said, for example, that one cannot teach what programming is before you have taught a language like Pascal. This, we would argue, involves working against the first heuristic, for once you have taught Pascal you may have great difficulty in teasing out the construct "programming" as distinct from "Pascal programming". Rather, we would suggest a top-down strategy, where the construct is developed directly using simplified models with which the students are not familiar, but keeping the construct as permeable as possible. This is not always possible, but experience suggests that it can work very efficiently. We shall discuss examples of this approach in the next chapter. A curriculum designed on these lines is "inverted", in the sense that it allows the introduction of general principles before details, rather than the more traditional approach [Coh86].

Availability of Validating Data

If returns on the prediction are unavailable or unduly delayed, one is likely to postpone changing the construct under which the prediction was made.

It is important to realise that only the experimenter really knows what the aim of the experiment was, and can construe the experiment as validating or invalidating contructs. If we set exercises for our students, their status should be clearly understood. They might be experiments we are carrying out on the students, to provide feedback on the effectiveness of teaching for example, or they might be suggestions for experiments the students should carry out. The latter use can be problematic, however, as our suggestions for actions will not necessarily lead to the experiments we anticipate, indeed they may not even lead to any real experience for the students. Programming tasks, for example, may well be validated by students using a simple "running/non-running" construct. In this case we may be reinforcing some bad programming habits (as we see them). If we expect students to bring more complex constructs to bear on tasks such as programming or writing specifications, then we must ensure that these constructs are developed first, and that the student sees them as sufficiently permeable to admit the events to their range. If we teach students to "comment code", for example, we are developing a fairly impermeable construct, and it is quite likely that they will not see the lessons learned as applicable to documenting proofs, specifications, or designs. We may end up teaching these as separate concepts, then attempting to develop super-constructs to embrace them all. This, we would argue, is the danger of a fragmented curriculum, and is one of the factors acting against progress in Software Engineering education: as it becomes seen as necessary for the engineer to understand more and more "topics" the fragmentation increases, but the understanding decreases, for the topics are only understood in terms of impermeable constructs.

Avoiding Threat

> *If the elements out of which it is proposed to form a new construct commonly involve threat, that is, if they tend to elicit a contruct or an issue which is basically incompatible with the system upon which the person has come to rely for his living—he may not readily utilize the elements for forming any new contruct.*

Kelly, being primarily interested in therapy as the application for his theory, has a great deal to say about "threat" and also one of the symptoms it evokes, hostility. Hostility is the reaction to threat that is associated with the desire to protect a construct system by interpreting events so that invalidation cannot be observed. This must not be confused with Kelly's use of the term "aggression", however, which is the symptom of a person's active experimentation with the environment. Thus hostility is the result of avoiding experience, whereas aggression is the pursuit of it. It is important to realise that threat not only hinders the development of construct systems, but may also retard it. If a person is being forced to reject the current system in the face of threat a common reaction is to fall back upon earlier systems which can be held even more dogmatically.

We can think of the loss of innocence as posing a threat for many people, because their construct systems tell them that complex design problems will require decisions they are unable to make confidently. The two reactions we noted, pretensions to genius and refuge in style, are both examples of adopting existing construct systems rather than risking the development of new ones. The acrimonious debates that are symptomatic of many discussions in Software Engineering demonstrate the hostility that arises as a result of confronting these reactions, and hence posing threat. Of course, one person's aggression can lead to another person's threat: it is quite possible that one party in a debate is genuinely trying to ask provocative questions to gain experience, but if another person interprets these questions as threatening events, hostility will occur. This is where the sociallity corollary becomes so important, for in gaining experience, whether it be learning in an educational establishment or performing requirements analysis, we must realise the possibility of threating others and act accordingly. Our learning is not a private affair if we make visible our experiments so that they can be construed by others. The software engineer who poses the question "why does this job need to be done" or "why have we been told to use this method" may well be aggressively seeking experience, but could be threatening the job security or reputation of others.

On-the-job training, therefore, requires a degree of personal skill if genuine experience is to be gained. One of the tasks of the teacher is to provide a secure environment where experiments can be carried out without undue threat. Thus students can engage in design tasks without their careers depending on them, or try out new programming styles without the fear of ridicule. Assessment and group work, however, increase the risk of threat, and must be treated with extreme caution. The argument that group working is essential in industry, so students should learn to work in groups, should lead us to conclude that they need to develop constructs that will enable this to happen, not that they should attempt to develop their own construct systems largely within group experiments.

Avoid Pre-occupation with Old Material

Old and familiar material tends to be fixed in place by old and childlike constructs; it is only as we let the client interweave it with new and adult material that he starts bringing his constructs up to date.

This is a particular problem for students who have "done computing before", for they bring with them a set of constructs, possibly developed and validated over a number of years, that are usually quite impermeable. They are also likely to have been subjected to events that validate, rather than invalidate, these constructs, so they are likely interpret their systems as deterministic. If we seek to build on these constructs too directly, by suggesting invalidating experiments, we pose threats: if we suggest further validating experiments we reinforce the constructs still further. This suggests that it may be appropriate to avoid direct confrontation until some new constructs have been developed to replace the old. Teaching "programming" through a language that is unfamiliar to all students, for example, might be more productive than attempting to build on experiences with Pascal, Fortran or Basic. Indeed, avoiding the term "programming" itself might be helpful, except that most students would find such avoidance itself threatening, as their understanding of what they have come to study should probably include programming.

7.3 Summary

In this chapter we have introduced PCT and discussed some of its ramifications for software engineering education. We have not intended to explicitly link this discussion to our model of system design, but we would expect the reader to have formed some constructs concerning how this might be done. One particular rôle this chapter has performed is to forge the link between the individual and social actions, involving both theory building for engineering and construct formation in education. Indeed, we can see that our model of system design, together with the acceptance of PCT, might form an excellent basis to explore the similarities between Education and Software Engineering.

Chapter 8

Curriculum Design Experiments

"In Dip. Ed., as in any fairy tale, just when you think you're out of the woods, there is suddenly more to them than you ever imagined. Just when you think you have escaped to an open quiet place it turns out you're still in them. Even later, the woods you're wandering in turn out to be yourself"

A Student Teacher

We have now reached the stage where we can reap the rewards, and suffer the consequences, of our research programme. Our aim was to increase the understanding of educational practitioners, so that they can improve the quality of software engineering. In this chapter we will discuss some of the consequences of this research for higher education. Boxer develops the analogy of a business as

> "a tangle of conversations which have formed into a knot [which can be thought of as the particular way in which these conversations come together]: they are the history which the business is to those who are in its employ. The knot in this sense is the explanation which governs who can make choices when, where and how about what kinds of things." [Box88, page 421]

This also seems a very apt analogy for the institution of higher education, and it highlights a decision we need to make before proceeding with this chapter: what view are we going to take of the knot?

Our research has been highly reflexive, for by discussing the nature of design we have arrived at some tentative results which we surely should seek to apply to the *design* of the curriculum, as well as to the design of the curriculum to teach *design*: thus process and content once more merge. The former, however, takes us well and truly into the task of untangling the knot. It presents us with all the problems of self referential systems, particularly those of consistency [Smu87]. It also raises issues of the rôles of management, government, students and teachers, issues of control, academic freedom, ethics, and many more facets of education. Our discussion may well lead us in directions that will be interpreted as radical, subversive and threatening. This alone should not worry us, of course, for there was never any guarantee that this enterprise would be comforting for the reader, or the author. There is one reaction we should guard against, however,and that is the "all right in theory" response. If our research leads us to conclusions that can only be utilised in some utopian world, it is likely to be ignored— a fate far worse than refutation. We might argue, of course, that bringing about such an idealised world is part of the teacher's job, but this leads us to an infinite regression, for the "all right in theory" response will doubtless resound again. Fortunately, however, we can escape through the horns of this dilemma by showing that our research has applications even if we are content to leave the knot unexplored, for we can tinker with the loose ends of string that protrude. This may not improve software engineering education as an institution, but it might improve the educational experience for some students.

In the two sections that follow we will start by briefly exploring the implications of our research for the problem of untangling the knot. We will then sketch out some activities that have been carried out in tidying up loose ends.

8.1 Untangling (or Tightening?) the Knot

One of the major results of our research has been the conclusion that software engineering design is, and must be, a selfconscious activity. If we accept this, then it is vital that students are educated in such a way that selfconsciousness develops. Moreover, our model suggests that students need to be able to construct theories to solve novel problems, thus they need to be able to learn without the support of a teacher. We need, therefore, to encourage our students to become selfconscious learners. This poses a major problem for the educational establishment, however, which despite its protestations, is still largely an institution where control is vested centrally, dissipated through resource allocation, common curricula, and teachers, with students coming far down the list. "Academic freedom" means freedom to work within certain norms and constraints. The teacher's role in this is largely to subvert the behaviour of the errant student in an attempt to achieve conformity, measured through examinations of one sort or another. Even "research students" are expected to conform to fairly restricted norms, in spite of the fact that they are supposed to be demonstrating an ability to move the frontiers of knowledge[1].

One major consequence of the shift of emphasis from "teaching" to "learning" is to deprive the researcher of an established methodology for investigating educational problems. Harri-Augstein sums this up perfectly:

> "If educational research concerns itself with learning and the researcher/teacher defines learning in this way, then the priority of scientific objectivity becomes suspect. The pursuit of objectivity normally involves the educational researcher in a detailed control of experimental situations. But studies aimed at increasing self-organisation require that the researcher recognise that the learner has his or her own point of view, and needs the freedom to explore

[1]The author decided, through wisdom or cowardice, not to pursue this line of argument! Reflexivity is all very well, as long as it applies to someone else.

each learning situation" [HA85, page 61]

If this is the case, we are presented immediately with an inconsistency within current educational developments. Industry is currently calling upon us to produce students who can communicate effectively, solve novel problems, manage change, design, and carry out all the activities that our research suggests require selfconsciousness in learning to learn. Government, however, through its various agencies, is calling for more objective metrics, such as standardised testing of student and teacher competence, both of which become increasingly difficult as we move towards producing students who are learning to learn.

We should also observe that the morality of educational experiments, in the traditional scientific sense, is questionable. Subjecting students to experiences with the aim of refuting our theories is, in the author's opinion, unethical. We should, of course, treat our teaching as experimental in the sense that will recognise refutations and take appropriate action, but we should not undertake deliberately risky experiments or establish controls.

A learning to learn curriculum can be considered as a "double-helix" [HA85, page 62]. One component, the learning of the learning process, develops alongside the other, the learning of discipline content. This is complicated for software engineering design, however, by the observation that the content, as suggested by our model, is also a double helix of process and theories. This leads us to ask how the two process components are related. If we help students to become effective learners, will their theory building skills in software design come for free? Conversely, if we teach them how to construct theories in software engineering, will this make them more effective learners? A tentative answer we would offer to both questions is "yes, if we can develop sufficiently permeable constructs". Indeed, both approaches can be used simultaneously, as we shall see in the next section.

Within the double-helix model, the teacher has at least two roles to play. First, that of an experienced researcher helping the less experienced students to develop their methodology. In this role the teacher is a co-student, helping the students to manage their learning, suggesting techniques for improving learning, and also using previous research results to offer ways of approaching the subject. The second role is that of an oracle, or provider of information. Harri-Augstein develops the notion of a "mindpool" as the collected knowledge of the world, similar to aspects of Popper's third world [HA78]. The teacher can act as a window onto the mindpool, or a librarian helping the student to find relevant information, including higher level structures. In this role, the teacher may work in several different modes: lecture mode, offering students pre-defined sets of information; tutorial mode, offering a more flexible set of interrogation procedures; and so on. Obviously these roles interact, for the co-researcher may decide to help by asking the oracle to deliver information. It is crucial that an appropriate balance is struck between these roles, however, for students will not learn to learn if they spend all their time listening to an oracle, but similarly they will not learn to learn if left to flounder by themselves in a vast mindpool with no lifeguard.

It is through these roles, and their interaction, that the teacher exercises control. "Control", however, is simply one pole of a construct, which can have many other ends, including "chaos", "disorganisation", "freedom" or "anarchy". Clearly the way a teacher exercises control will be largely determined by the construct system in use. A teacher that fears chaos as the alternative may exercise tight control, a teacher that fears loss of freedom may exercise very little control. We would argue that exercising tight control of the double-helix is also likely to lead to tight construal of the process component in the helix of content, so that: methods are stressed not as tools to be used but as controls to be obeyed; theories are not personal in oprigin and interpretation, but objective; and selfconsciousness, with its consequences of possible student rebellion, will be played down in favour of received wisdom. Indeed, much of our discussion on method can be reinterpreted to apply to teaching method: is it a plan, a control, a rationalisation, or a set of useful techniques? We should also realise that the controls we exercise as teachers will largely be governed by our personal myths of education, such as "students learn by doing", "students learn when they take notes"," students need to learn bottom up", and so on, most of which are based on our recollection of how we feel we learn most effectively.

Clearly, the way in which controls are exercised forms part of the professional judgment of teachers, and we would not presume to say how this should be done. It is precisely this exercise of control, and its implications, that constitute the teacher's view of curriculum design. Hence the conclusion that we cannot tell another teacher how to design a software engineering curriculum, only offer to engage in discussion and relate experiences. In any particular circumstance, these controls will be affected by those inherent in the knot. Imposition of timetable, scheme structure, assessment methods, cultural norms, "customer" expectations, student backgrounds, laboratory facilities, professional bodies, short-term and long-term industrial requirements will all have a part to play, for the teacher is not free to teach, but free to exercise judgments only within the environment provided by the knot.

8.2 Playing with Loose Ends

In this section we will briefly describe some applications of our discussion to the task of tidying up some loose ends emerging from the establishment knot. These applications have not been carried out in a laboratory, under controlled conditions, so they are not "experiments" in the scientific sense of the term, but they are experimental in the sense that they they have allowed the author to validate, or invalidate, his personal constructs and hence gain experience. It is important to realise that these experiments have been carried out in the ebb and flow of the environment provided by a typical higher educational establishment. They are experiments in "realistic teaching [which is] focussed on what can be achieved in practice by typical teachers in realistic circumstances of work and support" [Bur89, page 9]. The stance we are adopting here has been summed up by Gough, who writes, "I cannot reconcile much of the rhetoric of new-paradigm thinking with the kind of curriculum work that I want to do now and in future (for example, I am suspicious of the quality of life after quest: what does one actually *do* after one has *found* the Holy Grail?). Certainly this sort of rhetoric is too pretentious for the kind of work I am doing here, now. I am not attempting to write a chapter in one of the Great Books, I am writing a work-in-progress report—a short story—with the intention of engaging you, the reader, in a further exploration of the world it signifies: 'What we need is not great works but playful ones—A story is a game someone has played so you can play it too' (Sukenic 1969, quoted in Waugh 84 [Wau84]) " [Gou89, pages 226-227]

The experiments described comprise two types. First, there is the design of a whole scheme for the continuing education of software engineers, leading to the award of an MSc. Second, there is the design of initial education in software engineering for undergraduates.

Continuing Education in Software Engineering

It is important to note the unusual rôle of continuing education in Software Engineering for, as Mills observes,

"In a field more stable than Software Engineering, university education plays a dominant rôle in shaping the principles and values of the field, while industrial education consists of refresher and updating courses in fringe and frontier areas. But university education in Software Engineering was not available to the majority of people who practice and manage it today." [Mil80a, page 1158]

Mills, therefore, proposes that it is necessary for continuing education in this field to consider not only fringe and frontier areas, but also the fundamental principles of the subject as they are now perceived. Although the necessity of continuing education is generally accepted, the fundamental nature it may need to assume is not. Moore, for example, holds up a very different exemplar of continuing education when he writes

"After being on the job for a month, a fellow practicing software engineer invited Kim to a CDR to review the SDDD, STDs, and updates to the STLDD, SRS and SDP. Kim was told that an SQA engineer would be there and to get the most recent version of the SDDD from SCM. Did Kim go into massive panic mode? No! Having attended the Software Engineering Workshop (SEW) two weeks earlier, Kim was able to remember some of the acronyms. Kim referenced the SEW notebook for an explanation of the other acronyms. Kim now knew the meaning of the acronyms, understood how everything interrelated, and what to expect at the CDR." [MP88, page 42]

Mills is totally disparaging of such courses, stating that

> "There are any number of courses which will comfort, rather than educate. They are 'practical', 'easy to understand', 'the latest techniques'. On attendance, programmers discover various new names for common sense, superficial ideas, and thereby conclude, with much comfort and relief, that they have been up to date all the time. But unfortunately for the country, these programmers have not only learned very little, but have been reinforced in the very attitude that they have little to learn." [Mil80a, page 1158]

One of the most significant controls exercised by the knot on continuing education is imposed by industry, who typically fund such activities directly. In 1982, Hatfield Polytechnic was fortunate to be approached by a local company, STC-IDEC, who had seen the longer-term benefits which might accrue from a fundamental continuing education programme for its senior technical staff, and wanted to commission the design, development and implementation of such a programme. The programme was to be concentrated on the technical foundations of the subject, and how these could be brought to bear in solving real problems. This programme, which eventually became a Postgraduate Diploma in Software Principles and Practice, and was subsequently opened to non-STC-IDEC staff, has been described and discussed in more detail elsewhere [JLS86]. This programme was instrumental in bringing to light many of the problems discussed and developed in this research, and has recently been replaced by an MSc in Software Engineering, which has been designed in the light of many of the discussions carried out in this document. Full details of the MSc scheme are available elsewhere [Hat89], but we will briefly discuss how the theory building view developed in this thesis has been used to orientate the design of this major scheme of study.

The author was fortunate to be allowed to play a significant part[2] in the design of this scheme, and very few constraints were imposed. Consequently the whole scheme could be oriented by the proposed model. In particular, the control exerted by the curriculum could be reduced significantly to encourage students to take responsibility for their learning. If control is to be relaxed, however, it is crucial that students are in a position to learn from their experiences, rather than flounder in the mindpool. Perry suggests that we can consider five stages of intellectual development as pertinent to students in higher education [CH82]:

1. "Students see the world in polar terms of terms of right vs wrong. Absolute right answers exist for everything. Problems are solved simply by following the word of an authority."

2. "Students begin to perceive alternative views, as well as uncertainty amongst Authorities, but account for them as unwarranted confusion among poorly qualified Authorities."

3. "Students acknowledge that diversity and uncertainty are legitimate, but still temporary, in areas where Authority "hasn't found the answers yet". They seek relief

[2]The author chaired the development committee.

in hard sciences and mathematics which seem better understood by Authority."

4. "Students perceive legitimate uncertainty, and therefore diversity of opinion, to be pervasive. They are suspicious of any evidence or authorities opinion."

5. "Students perceive all knowledge and value, including authority's, as contextual and relative."

The admissions policy for the designed scheme was intended to ensure that students had reached level three or above, consequently the decision was made to allow them to have complete control over both the process and content spirals of their education. The constraint imposed by the award of an MSc. was overcome by the "teacher-as-senior-researcher" agreeing a programme of study with the student. This programme, proposed by the student, identifies aims, objectives, and assessment tasks to show that the objectives have been met. It also identifies resources that the student requires to complete the programme, including access to pre-defined courses, library facilities, equipment, and so on. In essence, the spirit of the scheme is that of an MPhil., where the emphasis is on personal research and development rather than contributing to the mindpool.

It was agreed, however, that such a radical approach might well leave the students floundering, and so a standard programme was designed to act as a starting point from which the students could deviate. This programme also served to identify a set of eight courses that were felt to be sufficiently fundamental to be useful to students in designing their own programmes. These courses were based entirely on the theory building view of software design, and comprised:

Theories and Formalisms: This course serves to introduce the theory building process together with a discussion of the rôle of formalisation. It also introduces a collection of mathematical topics, useful in model-based theory presentations, in a way that is described later.

Types and State in Computation: This course develops the idea of using laws of consequence and coexistence in presenting theories, and discusses styles of specification with the proof obligations that they entail.

Communicating Systems: The idea of laws of interaction is introduced, and the interpretation of such laws in system design, together with examples of communication primitives in implementation environments that provide models of these laws, is explored.

Performance and Reliability: Here we deal directly with aspects of the target environments that give rise to positive design heuristics, and also discuss the possibility of incorporating these features into the theory itself.

Fitness for Purpose: This course deals more fully with the concept of refutation of a phenomenological theory, and what would make our theories fit for purpose. It also discusses the relationship between fitness for purpose and correctness.

Software Development Strategies: Here we explore the strategies open to the engineer in constructing and transforming theories. In particular, the rôle, uses and dangers of proprietary methods are discussed.

Human Factors in Software Design: At this stage we acknowledge more fully that the design of software systems is usually carried our for humans and by humans. This course discusses topics such as project management, contractual boundaries in law, ethical considerations and human computer interaction, in ways commensurate with the maturity and experience of the students.

Knowledge and Data Representation in Computation: Many application domains now exist in which problems give rise to large amounts of persistent data. This data is often structured in quite complex ways to capture existing theories, such as those inherent in expert systems or temporal databases. This course seeks to unify the approaches in use in these application domains around the theme of theory building.

This design illustrates how our model and discussion has served to influence curriculum design in both process and content parts of the double helix, and how it can act as a structuring mechanism for a whole scheme of study. The reader is invited to distinguish the coverage of this material, which is reasonably familiar, from its structure and motivation, which are unique.

Initial Undergraduate Education

An assumption distinguishing initial education from continuing education is that students embarking upon the latter are likely to be at earlier stages of intellectual development. This is a gross generalisation, of course, particularly with policies of widening access to higher education, but this assumption is implicit in most scheme designs, and underpins the discussions that follow. Given this assumption, it is not reasonable to expect undergraduate students to take the same degree of responsibility for their studies as continuing education students, for they rely too much on authority. Consequently teachers and curriculum designers need to lay down a fairly specific programme of study for the earlier stages of the scheme, whilst at the same time not inhibiting the student from achieving higher levels of intellectual development. It might be argued, of course, that we should not attempt to teach Software Engineering to students who are still at such early stages of intellectual development, but should wait until postgraduate courses. Richardson provides an answer as to why this is not really an option:

> "The more immature the discipline, the more mature the practitioners or learners of that discipline must be. Unfortunately, the reality is that we cannot wait for software engineering. We can neither only have master's level people able to speak confidently about software engineering topics nor can

> we wait for the discipline to mature enough to be easily taught to undergraduates. The future has been forced upon us by advances in other areas and we cannot look the other way." [Ric88, pages 127-128]

In 1990, the BSc in Computer Science at Hatfield Polytechnic was reviewed to its seventh set of regulations[3] [BSC90]. In this section we will discuss the first year courses from that revised scheme, and how they conform to our theory building view. Expressed in theory building terms, the first year of the scheme seeks to develop the students' skills in the following aspects of Software Engineering:

- selfconsciously designing solutions to problems.
- developing theories.
- formalising theories, identifying and discharging proof obligations.
- expressing theories in programming languages in a form suitable for execution.
- understanding the implications of the underlying technological platform for the positive heuristic of the research programme.

To achieve these aims, there are four courses that run throughout the year: Problem Solving, Formal Notations and Models, Programming, and the Organisation of Computer Systems.

Problem Solving

Traditionally, problem solving is taught via the medium of the discipline being studied. This means that students can only tackle problems up to the current limits of their expertise in the discipline. Moreover, this can lead to a very false impression of the processes involved, as Bentley has noted,

> "After 16 years of school, the average college senior has the mistaken notion that all problems come neatly packaged. Of the many wounds inflicted by modern education, this is one of the most tragic: pre-school children come up with wonderfully creative solutions to problems, while graduates seem to have acquired tunnel vision." [Ben88, page 4]

Students do not need to adopt a selfconscious attitude to problem solving when it is presented in this way, for the method of solution is inherent in the problem context. Such tasks are useful, of course, in providing validating events to consolidate the constructs of material introduced, but they do little to help the higher-order problem solving skills, such as the selection and evaluation of methods, languages and tools.

[3]The author was Assistant Head of Division, with special responsibility for subject development, while this review was taking place.

The Hatfield Polytechnic problem solving course was designed to tackle precisely this area[4]. It does not set out to develop methods or notations suitable for any particular aspect of Software Engineering, although some may be encountered in passing. The course involves students participating in a number of experiments, which they carry out largely using existing skills and knowledge, but which they write up in a reflexive manner in a laboratory book. Thus they are, in a very real sense, experimenting on themselves. The problems tackled during the first half of the course have no obvious (to the student) link to topics encountered in other courses. Typical problems include egg-races, system improvement tasks (such as increasing traffic flow around the Polytechnic campus), and pencil and paper puzzles. Most tasks are undertaken in groups, thus developing rôle playing skills and also those necessary for making theories explicit and defending them to peers. Thus exploratory, informative, and scientific discourse are all inherent from the outset of the scheme, and one type of discourse is not given an artificially elevated status.

We would argue that one of the advantages of this approach is that students develop many of the vital constructs for system design without them becoming impermeably linked to ephemeral technological details. Students learn what it means to analyse and specify systems, design solutions, defend their designs, and so on, but without the limited vision often promoted by systems analysis and design courses, programming courses, or even formal methods courses. They also learn that designs come from people, not problems, and come to recognise their rôle in the process. As Slaughter has noted,

> "such [self-reflexive] methodologies and approaches support views of the world in which we recognize our embeddedness in a series of contexts. We begin to see only too clearly that our understanding of 'reality' is dependent upon the quality of the models used. Problem-solving is no longer about about making small, isolated changes. It is about participation and intervention in mutually interacting webs and processes. In this sense, solutions tend not to be 'right', but elegant. And, as Fisher notes, 'the contexts of elegance are dependedent upon the illumination that enables us to see them' [Fis87, page 11]. As ever, the threads which create the world lead back to us." [Sla89, page 265]

By designing, and discussing, a variety of solutions to problems, and the rationale behind them, students come to realise that software engineering design is not a deterministic activity, but one where personal constructs make people see problems in different ways, leading to different solutions. This also motivates the need for presenting designs in ways that can be discussed at a level above these personal constructs, using accepted third world objects taught through formal education, so that they will be similarly construed by all participants.

[4]This course was originally designed by John Sapsford-Francis and the author, but the author has played no part in subsequent developments of the course.

It is interesting to note that this course is viewed as very "difficult" by the students, in spite of the lack of technical content, for they find the task of producing reflexive laboratory reports very hard, being unused to having to explain, analyse and justify their methods, rather than accepting them on authority and applying them to rôte problems. Once they manage to overcome this difficulty, however, we hope that the students are developing the constructs necessary for the confidence and willingness to accept the loss of innocence.

Formalisation

It is important to make the point that when we teach formalisation we are not teaching mathematics, even applied mathematics, in the traditional sense. This point is also made quite frequently amongst teachers of mathematical modelling, which is surely formalisation by another name. Bkouche sums this up, when he writes,

> "Mathematics is proposed to physicists exclusively as a tool and physics is proposed to students of mathematics as a series of applications of mathematical theories. In this narrow perspective, the problem of the use of mathematics (mathematics as a *service subject*) becomes artificial; mathematics is deprived of its actual and historical connection with scientific practice." [Bko89, pages 49-50]

If we are to avoid students seeing the use of mathematics as an artificial device, we must encourage them to see formalisation not simply as the application of an existing body of mathematical knowledge to well-understood problems, but as part of the process of understanding the problems themselves. This way of viewing mathematics runs contrary to that of many mathematicians, however, who subscribe to the "pure" and "applied" partitioning of the subject, and also that of many "users" of mathematics. Fisher sums this up very neatly when he writes

> "Those [mathematicians] who construct the model emotionally are disturbed by the appearance of new aspects or even contradictions. They want to be told once and for all what people want so they can concentrate on solving the problem. In contrast to this traditional orientation the mathematician [who is oriented towards building a description of the problem] understands himself as an *explainer of the problem*, who helps people to articulate their imaginations, who points to alternatives; sometime even as somebody who slows down the process of solving the problem." [Fis89, page 19]

We would argue that it is this view of mathematics and formalisation that needs to be taught to students if we are to promote the theory building view. Students must be encouraged to break away from the notion of mathematics as a medium for scientific discourse alone. Simply extending their constructs so that they also see it as a medium for informative discourse is not sufficient, for they must also come to see the rôle of mathematics in exploratory discourse. This is certainly not the way that mathematics is usually presented to students by most teachers of the subject. Mason observes that

> "Mathematical thinking is like going on a voyage. Doing mathematics with others involves voyaging together, with all the agreements and disagreements of fellow travellers. Writing up mathematics is like reporting back, or writing a travelogue. Teaching mathematics involves being both a tour guide, and an old hand listening critically to fresh reports." [Mas87b, page 77]

He goes on to say elsewhere, however, that this is not an accurate reflection of current practice:

> "Most teachers and students ...think that the formal language *is* the mathematics. ...When this language is taught to students, attention is drawn to the procedural, syntactic aspects, because outer behaviour is desired (solving typical problems), visible and accessible. Thus there is an assessment-induced thrust away from meaning towards mechanical manipulation." [Mas87a, page 209]

The approach adopted to teaching formalisation at Hatfield is that mathematics must be presented not as a static body of knowledge but as a human process. Students do not simply need to learn mathematical knowledge and methods, but need to learn to participate in the mathematical process itself. This is not only an educational perspective, however, but it is also indicative of a shift in the status of mathematics itself, analogous to the recent trends to re-introduce the scientist into science. This idea has been developed elsewhere [Loo90a]. Lakatos, instrumental in bringing about the re-appraisal of science, also proposes this view of mathematics [Lak76] [Lak78]. Ernest suggests that adoption of this new status of mathematics serves to make it more accessible to students, rather than making it more difficult.

> "Having carried out their programmes, but having failed to provide certain foundations, the traditional philosophies of mathematics [logicism, formalism, intuitionism and platonism] are now spent forces, which do not offer tenable accounts of the nature of mathematics.
>
> The new wave in the philosophy of mathematics has set a new agenda for discussion. No longer is it sufficient to consider only the immutable objects and truths of mathematics, that is, its products. The human context must be seen as a legitimate and essential contribution to the Philosophy of Mathematics. ... But this perspective, which includes the nature of mathematical activity, the plurality of methods, the rôle of error and the negation of truth comprises all the factors which most potently contribute to the pupil's view of mathematics." [Ern89, page 559].

Clearly this approach fits very neatly with our model of system design, for we want to develop students' skills in constructing theories using formal modes, rather than simply skills in using existing formalisms. We also want them to become selfconscious participants in the mathematical process, not simply adept in the mechanical application of standard results [Loo90b] [LM88]. This does not imply that we should ignore standard results and existing knowledge, of course, for our students will need to build on these if they are to operate efficiently, and they also provide a common foundation for discourse between engineers. The challenge is to incorporate the traditional material with the process driven view of mathematics.

One way to achieve this is to teach formalisation by developing theories of mathematical objects, that is, to use meta-mathematics. The Formal Notations and Models course[5] begins with a discussion of what it means to be "formal", and an introduction to formal languages and systems. This gives rise to another advantage of the approach, namely that the close connection between languages and machines can be introduced very early on in the scheme, so that the students view programming as just another example of formalisation. Tools are used wherever possible to incorporate formalisation into the computing culture, so that students do not see an artificial divide between "practical" activities on a machine and "theoretical" activities using pencil and paper. Yacc and lex, for example, are used in week one, allowing the students to experiment with formal grammars. This also serves to demonstrate that formalisation can lead to simpler solutions, as well as causing extra work. Formal languages and systems are both introduced using abstract examples to avoid undue interference from existing constructs, although we rapidly move on to providing semantics for the examples and also expressing familiar examples formally, thus developing the range of the constructs.

Once students have achieved some understanding of formal systems, we introduce propositional and predicate calculus, and the notion of a formal theory presentation is discussed. At this stage, formalisation of mathematics itself begins. All students will have met sets before (a self-study package is used to ensure a certain level of background knowledge), so we now apply the permeable constructs of formal systems to the task of constructing a set theory. The task is explored, but eventually ZF set theory is presented as the theory we will build upon. This involves discussing what properties we expect sets to have and how these might be expressed, thus set theory is seen not as a state of knowledge representing the emergent properties of sets, but as a designed theory. This discussion inevitably leads into consideration of related objects such as bags and lists, which the student will also be meeting in the programming course.

When sets have been formalised, model-based theories of functions, relations and sequences are presented. This process is remarkably simple, for the students can see that these are just more examples of theories. They may be overwhelmed by notation, but they seem to accept the important issues very readily. Moreover, because they have been encouraged to see theories as artifacts that have been designed, rather than God-given

[5]This course was designed and taught by the author.

properties that emerge, they also seem to accept that alternative theories are to be expected. The fact that mathematicians tend to consider only total functions as functions, and use functions of several variables rather than curried forms, does not seem to confuse them as one might expect. This is crucial, for it suggests that the students have reached stage 5 in Perry's stages of intellectual development in very quick time (within a term of study). This is evidence only within a very restricted domain, of course, but it is a domain in which authoritative answers usually predominate, and hence we might expect the lower stages to persist longer.

The next part of the course is devoted to developing the students' abilities in constructing, and using, theories to act as specifications. There are two possible ways forward here: we can either encourage the perception of problems in terms of constructs the students already know how to formalise, or we can attempt to formalise the constructs that seem to be suggested naturally by the problem. The former leads us towards model-based specifications in Z, for example, whilst the latter leads us towards a more primitive specification, typically using an equational predicate logic. Since we want our students to be selfconscious, we discuss both methods, but emphasis is placed on model-based techniques as these also develop fluency in constructs which the student will find useful in social processes. A small amount of time is also spent developing a formalism for communication and concurrency, CCS. This typically causes problems because students have usually developed the notion that computers "do what they are told", but this construct is not sufficiently permeable to admit non-deterministic behaviour. They find it very hard to understand that we can have a theory of behaviour which is precise, but from which we cannot predict behaviour. It is important to realise that this is not a problem of formalisation, of course, but the emergence, through the formalism, of a property of the system under consideration. It is often said that "formal methods" are too hard to teach to first year students: we would argue, however, that the difficulty is simply a reflection of the fact that we cannot so easily wave our hands and avoid discussing the difficult facets of our problems. We can, of course, establish simple problems in which these difficulties do not arise. To argue from this that formal methods are only useful for trivial problems, whereas informal approaches allow us to tackle real problems, is fundamentally dishonest: informal approaches simply allow us to avoid tackling real problems more convincingly.

This theory building approach to teaching mathematics itself has been discussed at greater length elsewhere [Loo90a]. Indeed, when the author first presented these ideas [Loo84], albeit without the rationalisation provided by the model, the reaction was that only the better students would be able to cope [Juk85]. It is important to note that this is not the case, for it has also been used to great effect with a group of 180 HND students [LB91]. In addition, a textbook has been written to support this process, as existing books were found to run counter to the approach [WL88].

The final component of this course is to develop the students' appreciation of the rôle of proof in system design. Because the material has all been presented via formal systems and theories, proof is seen as a natural extension to exploration and specification, but some extension of constructs is needed to allow discussion of proofs of correctness. A more detailed account of the view taken of correctness, and how it relates to the construct of fitness for purpose, has been presented elsewhere [Loo91].

Programming

Teaching programming using the theory building view is a delicate operation, for we would argue that it develops poor constructs if students are encouraged to present theories directly in terms of programming languages. This is the approach taken in many textbooks, some of which are even entitled "problem solving with xxx". One solution is to delay the teaching of programming until after the students have developed the requisite constructs for theory building. There are two drawbacks with this approach. First, students, employers, professional bodies, and many other interested parties, have expectations that programming with "real" languages will form part of the initial teaching of Software Engineering. The argument that teaching formalisation actually develops the required constructs more effectively than using a language such as Pascal or Ada is difficult to make persuasively in an atmosphere where "formal methods" are frequently seen as part of an opposing paradigm, and where students expect to carry on an existing paradigm from school or college. The second problem is that the positive heuristics for the research programme in our design process come largely from an understanding of the facilities available in a typical target environment.

The solution that has been adopted at Hatfield in our programming courses for many years is to teach through a "real" functional language[6]. This means that students are learning to formalise theories in a way that corresponds directly to the sort of constructs they are encountering in the Formal Notations and Models course. It is incidental that the notation used permits execution of particular schemas on a real machine. This approach supports the idea that the students retain responsibility for their programs: it is their theories that are being presented, rather than the theory being some emergent property of the code produced. We would argue that teaching programming initially through a real procedural language, with its loss of referential transparency, makes this very hard to achieve, for the programs are hard to reason about, and consequently their status as theories is difficult to appreciate. Lapalme and Cantray, for example, present a very different view of programming from the one outlined here, that suggests they are aiming at developing very different sorts of constructs in their students. They present programming as if it comprises experiments with a machine to discover what a program does:

> "As Ada[7] is one of the first widespread languages to provide a standard way

[6]This approach has been developed over a number of years by various members of staff, but has primarily been promoted by Bob Dickerson. The author claims no credit for this approach.

[7]The author cannot resist including the delightfully ambiguous quotation from Chandok: "Introducing Ada at a late point in the curriculum can lessen its impact on shaping the behaviour of the student, as many bad habits are already well established by this point"! [CG88, page 200]

of describing parallel activity, we tried to use it in a course, but we soon realised that it was impossible to debug those systems short of doing by hand many output statements giving an idea of where the activity of the program is taking place. The output statements have the unfortunate drawback of obscuring the initial code. ...Experienced programmers do not need all that information, but novice ones do." [LC87, page 185].

We would argue that students should never never be encouraged to write programs where they do not "have an idea of where the activity of the program is taking place"; the program represents *their* theory, not an object that has mysteriously appeared from nature. To recommend teaching in an environment where programs can be explored to see what they mean would suggest that students are not being encouraged to take responsibility for their programs, but to treat the results of their "designs" as objects of empirical study. This is not to say that the environment should be devoid of useful feedback, of course, for students need to be able to check the validity of their theories and also that their choice of presentation causes the predicted machine behaviour. The emphasis should be on prediction and attempted refutation, however, rather than on observation and induction.

One of the objections sometimes raised against this approach to teaching programming is that students need to develop competence in "real programming". Experience suggests that this is not a problem, for the construct of programming developed is sufficiently permeable to admit other styles of programming to its range. Indeed, we would suggest that it may be a more efficient way of teaching a procedural style than approaching it directly.

One of the advantages of teaching programming in this way is that it allows us to develop the constructs without binding them inexorably to the operational semantics of some machine. Thus the concept of programming can be broadened to include theory building, rather than being constrained to forming part of the transformation process. This appears to be the suggestion being made by Dijkstra in the 1989 debate on Computer Science Education [Dij89], but strangely Dijkstra goes on to advocate the use of a simplified procedural language. We would argue that this is unnecessary, and, given the existing constructs of many students entering higher education, unhelpful: a point also made by Scherlis [Sch89a], who notes this seeming inconsistency of Dijkstra's arguments.

Computer Systems

Teaching a computer systems course alongside the three courses outlined above poses an interesting challenge. A traditional approach is to teach the material through a simple, but realistic, processor and its machine code. This does not fit well with the teaching of functional programming as a first language, however, for it means that students are developing many of the constructs of procedural programming for the first time in a machine code, which is not supportive of the theory building view.

The approach adopted at Hatfield[8] is to introduce computer systems via finite state machines and pushdown automata. This is a natural extension of the formal language material in the Formal Notations and Models course, and the students are not encouraged to separate the two courses at this stage. Once the idea of a machine and stored program have been established, Landin's SECD machine is then introduced to illustrate a computer system of sufficient power to support the functional programming they are currently learning. This allows the discussion of a number of important topics, but within a very simple model. Stored programmes, memory, parameter passing mechanisms, saving and restoring environments, compiling, and many other issues can all be discussed without the clutter often associated with a "real" machine. This machine is also used to highlight the functional units that may be helpful in implementing a hardware system, and these units are discussed and formally described, with their theories being translated into hardware configurations. Thus the theory building view is supported, and used, in hardware design as well as software, which serves to reduce the problem of an artificial divide. The course continues in a more traditional vein, as the functional units are configured to support more conventional languages.

8.3 Summary

In this chapter we have discussed the application of our model to curriculum design, in the widest sense, and also provided an overview of some experiments that have been carried out in a realistic teaching environment.

[8]The author designed and teaches this course.

Chapter 9

Conclusions

"No one believes an hypothesis except its originator, but everyone believes an experiment except the experimenter"

W. I. B. Beveridge

Keane suggests that

> "There is no real *beginning* to a piece of research (especially doctoral research) but rather a set of false starts and dim gropings; what we may choose to call the *middle* is a complex tangle of issues and cross connections, iterations, blind alleys and mistakes; and the end is not an *end* but a frozen frame of a current set of ideas, which continue to develop." [Kea88, Page 9]

Viewed in this light, the previous chapters of this document have presented the frozen frame. The purpose of this chapter is to stand back from the frame and evaluate the total picture. Ideally we would start this discussion by considering the extent to which our primary aim has been met. Since this aim involves increasing the reader's understanding of the software design process, the extent to which it has been met is difficult to gauge without detailed psychological testing of each individual: and this was not deemed possible. We can, however, discuss the extent to which we have met our subsidiary aims, which we will do in the first section of this chapter.

We can also reflect on the research method, discussing the problems that were encountered and how, with the benefit of hindsight they might have been avoided. Such a discussion may be of value to anyone seeking to build on this research programme, or carry out a similar one. This comprises the second section. Similarly, in section three, we will discuss the presentation of the thesis itself, analysing the document as a piece of discourse and suggesting improvements that could be made if more time were available. Finally we will discuss a number of future research programmes that could usefully be built upon this one, including a few for which the author has already drawn up proposals. This cannot be a definitive list, for virtually all current research in computer science and software engineering can be seen as motivated by our analysis, so we will restrict attention to areas that currently seem to be neglected, possibly due to their interdisciplinary nature.

Before doing so, however, let us develop a simple model, or rather an analogy, of this picture. Consider the problems arising when a massive, complex object appears one day in a valley surrounded by hills. The local inhabitants are curious, and understandably want to discover more about this strange object. They can go up to it, but they have no way of interpreting what their senses detect in any holistic way, so they climb one of the hills for a better overall view. This hill offers a restricted sight of the object, and from only one vantage point, so they move onto another hill offering different restrictions and a new vantage point. There comes a stage when one of the inhabitants believes he understands the object well enough to sketch out a model. Armed with this sketch, the locals can climb new hills, seeking verification and refinement. They can also revisit the original hills, where they will doubtlessly re-interpret many of the glimpses they have of the object in the light of the model. They can also use this model as the basis for their discussion of the object, for it provides common ground, and they may use these discussions as the rationale behind certain actions.

In this research programme we have climbed the hills provided by a number of academic disciplines and their associated bodies of literature in the hope of catching revealing glimpses of software engineering design. We have constructed our model, and reported back on a few of the ensuing discussions and actions. Chapters Two and Three of this document record our glimpses, Chapter Four presents a model, Chapters Five, Six and Seven seek out new vantage points, and revisit some old ones, in order to fill in some details that are obviously missing. Chapter Eight recounts some of the discussions, and ensuing actions, that have resulted from our developing model. Chapter One represents the briefing session that set out the challenge facing us, and this chapter is to be a reflective discussion as to how well these challenges have been met, together with identification of areas that still need exploring, and also guidance for subsequent explorers to enable them to avoid some of the pitfalls and wasted trips that befell us on our journey.

9.1 Meeting the objectives

There were three subsidiary objectives identified for this research programme; the establishing of a framework for the discussion of the software engineering design process (finding the hill to climb and recording the views), constructing and refining a model of the process (bringing together the views and seeking out missing detail), and illustrating how the ensuing discussion can be utilised in curriculum design. We will consider each of these in turn.

Establishing a framework

This objective proved particularly difficult to meet, due largely to the lack of a philosophy of engineering, which it had been anticipated would serve to orientate the search. No claim is made that the framework established is particularly good, although it did serve its purpose in supporting the initial construction of our model. The decision to centre discussion on Popper s philosophy of science proved crucial, for it enabled the subsequent expansion of the framework, as our model developed, to include, for example, the works of Kuhn, Lakatos, Suppe and Kelly. The (embryonic) philosophy of technology and design also contributed significantly, particularly in establishing the idea that the software development process must be one of selfconscious design. The relationship between rules, theories and laws is also crucial, for it leads to the ideas of theory construction and schema implementation found in our model. Kinneavy's notion of the three kinds of reference discourse was also vital, for it provided the framework to reconcile exploration, specification, and explication, all of which are crucial in the design process.

These examples show the value of the eclectic approach, for no reference to these ideas has been found in the Software Engineering literature. The most disappointing aspect of the work on establishing a framework was the failure to integrate the psychology of problem solving to any great extent. With the benefit of hindsight, and an increased understanding of the discipline, the reason for this is easy to see; the research was expecting more from the discipline than it was able to offer. In particular, the desire to be scientific has led to psychologists adopting a very fragmentary approach, whereas the assistance we required was aimed at unification. In retrospect, it might have been more sensible to use Kelly's Personal Construct Theory earlier, when establishing the framework, rather than using it in it's dual rôle later. This would have been fraught with different problems, however, for PCT poses an interesting dichotomy for the researcher: either we accept Kelly's theory as regnant in some absolute sense, in which case we are denying that all constructs are our own, or we reject his construal, replacing it by our own interpretation of the theory, in which case our discourse must be subjective. The author was unprepared to do the former, for this would have meant establishing Kellian constructs as pervasive throughout our framework; and not confident enough to do the latter, for a doctoral thesis that is unashamedly subjective seemed to fly too much in the face of convention.

Construction of the Model

The objectives of constructing a model of the software design process appears to have been fully met. The model is not in any sense complete, of course, but it has proved adequate for our intended purpose, and seems to be sufficiently robust to support further developments. The proposed model has the virtues of simplicity and, in the author's opinion, a certain elegance. The use of theories as the unifying feature permits discussion of many facets of the design process in ways that cut across existing divisions. Soft and hard topics, such as requirement analysis and proofs of correctness, can be reconciled within this model without the need to adopt particular paradigms such as joining the formal methods camp or the CASE camp. Thus the model may serve to reduce hostility, but without removing aggression.

The model is also very useful in establishing links between fitness for purpose and correctness, and between testing and proofs. Notice that we have not sought to abolish testing, but to elevate it to its proper status. We have, however, sought to show ways in which testing may be relocated and reused.

One possible disadvantage of adopting the term "theory" is that it may be seen as neglection of "practice", a reaction that arises as a result of an individual developing a "theory/practice" construct. Such a reaction has already been noted on occasions when the model has been used in conference presentations. Hopefully a careful consideration of the model will show, in a non-threatening way, that such a construct is untenable or, at least, unhelpful.

A significant result has been the identification of the semantic conception of theories as a useful body of knowledge in which to interpret many aspects of system design. Clearly more work needs to be done in making the links explicit, but if this can be achieved there is the possibility of a major area of philosophical literature becoming directly relevant to Software Engineering. In particular, the similarities between Suppe's three kinds of laws and the three types of specification paradigm identified by Cohen and Pitt stand out as requiring deeper analysis.

The use of Lakatos's idea of research programmes is also significant, for it offers a way of talking about the progress of a single project and the transition between projects. It also permits separation of different kinds of requirements into the hard core and the protective belt.

One possible deficiency in the model is its failure to provide a more concrete view of structuring presentations, and the proof obligation this entails. This is not surprising, for the subject is very much a concern of current research in Software Engineering, but only within particular paradigms, so expecting any deep generalisation is over optimistic. The author contemplated introducing category theory to discuss these problems, but decided that this would serve to reinforce the (false) impression that theory constructs and formalisations were one and the same thing, without contributing much to the debate: we can only interpret the categorical presentation if we already have an interpretation of the problem.

Utilisation of the model

The author considers that this objective has been fully met, although this is difficult to demonstrate through this document alone. Perhaps the best evidence has been provided through the use of the model to design and teach a formal notations and systems course to one hundred and eighty H.N.D. students. This has resulted in a course which is well regarded in a number of senses: the students like it (in spite of being a mathematics course), staff are keen to teach on it (although it is a lowly first year H.N.D. course), the BTEC moderator considered it an excellent course, and visiting H.M.I. were also very impressed by the level and standard of work that the students were achieving. We cannot claim that the model deserves all the credit for this, of course, for many factors have played a part, but the author believes that the research undertaken in this programme played a significant part in bringing this about.

A significant result arising from this objective is the realisation that the problem was actually phrased in the wrong terms at the outset. We should have been asking what we could do to help students to become better learners, rather than ourselves better teachers. It is interesting that this observation has arisen from consideration of the content, rather than process, of the curriculum.

9.2 Research Method

This research programme has proved extremely difficult to coordinate and control. The eclectic nature of the research has led the researcher to climb many unfamiliar hills, and when he has reached a vantage point, to interpret the object of interest in the light of the disciplines surrounding him. Goguen observes that

> "To address such issues it is not necessary to be an 'expert', that is, have everything already worked out. Indeed, it is not even desirable, because genuine meaning arises through uncertainty and questioning, even through confusion and error. It is necessary to enter into a dialogue in order for truth to emerge from concealment." [Gog90, Page 25]

Whilst expertise may not be necessary in terms of depth of knowledge, the author would suggest that what is required is possibly a deeper form of understanding: an understanding of the disciplines themselves and the organisation of their literatures. Two examples illustrate this point admirably. First, the philosophical literature has to be read in a very special way; it is steeped in background and the novice reader may well form the impression that the only way to understand a text is to follow every reference back to its source. This rapidly explodes into an impossible task. The trick is to read the literature at a number of levels, first forming some idea of the context in which the author is writing, then extracting the main points, and only seeking out a detailed discussion when this has been done. The style of writing, however, often suggests that every word should be studied in its own right straight away.

Similarly, the problems encountered in unravelling the psychology of problem solving can be attributed to a lack of overview of the discipline. Its fragmentary nature was not appreciated until a substantial part of the research had been undertaken: as a consequence considerable time had been spent searching the literature for the reconciliation of ideas. This reconciliation, it is now being acknowledged, has been neglected by psychologists in favour of tightly controlled pockets of study.

A consequence of this need to develop an overview of the disciplines visited before their literature became usefully available was that the opening stages of the research seemed very unproductive. Many items were studied that were of relevance to subsequent study, but of no direct relevance to the problem in hand, thus the early stages were a development of method. In retrospect, some of these activities could have been simplified by seeking help from experts, but at the time but at the time the researcher was unable to phrase the right questions: asking for someone to "tell me about the philosophy of science" is actually the right sort of request, but such requests were always interpreted as "tell me about some of the details within the philosophy of science".

Another problem with eclecticism is the feeling that you are visiting a hill on the off-chance that the view may be helpful. Sometimes, of course, someone points out the hill to you, but often when you arrive you are unable to interpret what you see. Consequently a number of disciplines and bodies of knowledge were explored that have not been included in this thesis either because the views they gave were too restricted to be of much use, or because the researcher was unable to interpret them sufficiently to incorporate them in the design of the model. Topics such as the philosophy of mathematics and category theory were discounted for the first reason, the use of anthropology for the second. Here we must acknowledge the rôle of accident, for had Suchman's book on situated actions been discovered a year earlier this would probably have provided the insights the author needed to make use of anthropology. This leads to another problem, namely that it is not possible to carry out systematic searches for literature that "might cast a light on the software engineering design process". It was only with the development of the model that a systematic approach became tenable. Expressed in Popper s terms, during the early stages we were forced to seek knowledge with a bucket, collecting items as we chanced across them, but the design of our model allowed us to switch to a searchlight strategy. This suggests that the global strategy of establishing a framework, to constrain the bucket filling, then designing a model to focus the search was a good choice.

Another problem has been that the world does not stand still to allow doctoral research to be completed. This is true in all disciplines, of course, but when working on the cusp formed by the intersection of many changing disciplines the problems are compounded, particularly when the research has to be carried out part-time over a number of years. In addition, the use of an authentic problem causes difficulty here, for the problem shifts as the researcher develops, although it was, to a certain extent, frozen by adopting the three subsidiary objectives. It should be noted that this is not such a problem for the research programme *per se*, but of trying to structure it in a way suitable for a doctoral dissertation.

Authenticity has also caused a methodological difficulty with respect to our third objective, for every opportunity was exploited to solve problems using the model. This has led to the model being used to structure meetings, write papers and books, give lectures, design schemes and courses, explore teaching methods, and probably many other purposes. Reporting back on the third objective, however, would have been facilitated if some experimental control had been exercised. No apology is made for this lack of restraint, however, for the author considers the education of students more important than the simplification of the documentation task.

Perhaps the biggest problem, but one of the most exciting observations for the researcher, is the extent to which the development of the framework and model have fed into the quest for a suitable research method. It posed a problem, because the researcher started out with the traditional, but impoverished, view of science, in which "method" gives rise to "results", rather than being part of the results: thus the early stages of the research programme were decidedly threatening. It was exciting, however, because it blew apart the research programme in the most spectacular ways. Reading a research paper ostensibly on an aspect of teaching method, for example, could lead to the most powerful insights into requirements analysis, proof techniques, or the research method itself. Thomas and Harri-Augstein sum this up when they write

> "For us as researchers, the interaction between theory and method is central to what we do. Sometimes the idea system is ahead of the methods, and one looks around for (or is forced to invent) methods that allow one to operationalise one's ideas. Then suddenly one finds oneself with methods that produce all kinds of surprising results that do not seem related to the ideas that gave rise to them. Often this is because one develops all kinds of insights and skills in using the techniques." [THA88, page 98]

Even more exciting was the realisation that our research method itself feeds back into the model: exciting, but very hard to capture in a sequential document of this nature.

9.3 Presentation of Results

The major difficulty in presenting the results of this research is that these "results" represent a frozen frame of exploratory rather than scientific or informative discourse. This leads to a major problem of honesty, for the author is confronted by the dilemma as to whether a true account should be given, or whether a facade should be created to present the researcher as more clever than he really was. This is accepted practice in scientific discourse, but the author took the view that the objective of the research would be better met by an honest presentation, as this would involve the reader in the exploration process rather than relegating him or her to an observer. In particular, Chapters Two and Three were developed as a resource upon which to draw in building our model: they have not been rewritten in the light of the model to add resources that were missing or to remove those that were not used.

A similar problem arose in selecting the style of presentation. The norms of scientific discourse tend to discourage use of the first person. Philosophical literature, however, uses the first person quite liberally, as philosophers frequently present their opinions as such, and not as theories detached from themselves. The author decided to adopt the first person plural to encourage the view of exploration: "we will ..." having connotations of "let us agree to ...", leaving the reader free to rebel and write their own alternative versions. This seemed preferable to "I think ..." which simply reports a fact. This point was not made in the introduction as the author believes that such explicit references to the style of writing can detract from the pleasure of reading.

In one sense, it is essential that the author should remain dissatisfied with a text of this nature. If it is to serve its purpose, it must provoke an aggressive reaction from its readers (its author included), otherwise it is inert, a piece of history: and since, unlike a railway timetable, it contains no "facts" or "truths", it is useless. In another sense, the author is very satisfied that he is unable to re-read this document without thinking of better ways to express things, or of points that need clarification, or of further avenues to explore. It is certainly the intention that the material should continue to develop, subsets being extracted and presented for publication in different forms.

One interesting possibility would be the development of a hypertext version of this thesis, allowing the readers to climb the hills as they wish, contributing their own notes to the text not as marginal items but as valid contributions to the main corpus. Possibly higher educational institutions should consider admitting such works as doctoral theses, even if it would mean relaxing their regulations on binding!

A source of disappointment is that the author was unable to capture the high degree of reflexivity inherent in this research. No way could be found of achieving this without either allowing the research method to intrude constantly into the text, thus creating an historical account of the programme, or developing several strands of discussion simultaneously. Although some key points of reflexivity have been highlighted, it must be left to the reader to imagine where others might have occurred.

9.4 Future Research directions

There are many possible research programmes that can be seen as emerging from this exploratory work. We will restrict attention to avenues that currently seem poorly represented in current research, and suggest possible developments that could take place.

There is research that needs to be carried out to increase understanding of aspects of the software design process. In particular, our model suggests that a key question involves the relationships between the structures our theory presentations assume at various stages in the process. If, as we have claimed, the structure of a specification has implications for the subsequent implementation structure, then clearly some research needs to be done to establish these relationships. Mitchell and Loomes, for example, have suggested that the structure of a specification has implications for the maintainability of the resulting system [ML91]. Refinement and correctness proof techniques exploit such relationships, but discussion is always carried out within a paradigm containing particular specification and implementation languages. The challenge is to generalise these ideas, possibly using Suppe's notion of laws of coexistence, consequence and interaction. This work could also cover the proof obligations that arise out of different styles of specification, seeking to integrate the work of Cohen and Pitt into a philosophical framework. This research programme would allow us to substantiate, or deny, the claim we have made that it is impossible to present specifications in such a way that they do not contain implementation details. Substantiation, moreover, would focus attention on the need to teach the design of specifications, rather than suggesting that design starts after specification, as most life-cycles can be interpreted.

Another avenue to explore is the use of Kellian techniques for requirements analysis. Clearly bipolar constructs will not be sufficient for presenting our theory, but possibly repertory grid analysis could be utilised during the early stages of analysis to focus attention on those constructs the users consider important. This would be particularly useful in domains where articulation of the problem is difficult. Such an approach might lead to the development of some of the computer-based tools currently used for grid analysis into tools for the initial stages of requirements capture.

A similar programme of research has been suggested by Michie[1] for developing systems to emulate human experts, such as helicopter pilots. He suggests that automatic rule induction could be used to infer theories that would explain actions of human experts by observing them in controlled, experimental, situations. This idea has been tested in simple laboratory studies and has yielded encouraging results, but the issue as to whether the technique will scale up is an open question. Another question is can such an approach be made acceptable for application to the development of safety critical systems? An affirmative answer to the second question is more likely if a way can be found to formalise the assumptions underlying the rule induction system, and the resulting theory it produces, in a manner that conforms to requirements for the development of such systems.

Our model has also highlighted the various views that can be taken of methods in the software design process. An interesting question is: how do engineers claim to be using methods, and are these claims consistent with their actions? This is an important question because companies are increasingly spending large sums of money on "methods", typically through investment in training and tools support. In order to judge the wisdom of continued investment, however, they need to evaluate the improvements that these methods have brought about. If the engineer is not really using the method, or is using it in ways not envisaged by management, this judgment may be ill-founded. An interdisciplinary research project comprising of computer scientists, psychologists and philosophers of science has been proposed to explore this question, and was short listed for funding by the Tri-council. Unfortunately, the proposal was finally rejected on the grounds of an insufficiently developed methodology. This highlights a meta-level of research that needs to be done before exploration of some implications of our model can be explored. In particular, acceptable methodologies for interdisciplinary research need to be devised.

In addition to the numerous research activities that could refine and develop our model, we could also seek to utilise our model in new ways for curriculum design. Of particular interest would be to explore the ramifications of bringing together the theory-building view of design, PCT, and a transformational view of education. We have already noted how close these three components are in a number of ways, and an in depth investigation

[1]In a personal correspondance.

of the implications of adopting their conjunction might lead to significant advances in technical education. Such an investigation would have to be liberated from short-term goals, such as designing a particular course or scheme, for these would impose too many constraints on the process. A suggested starting point would be a detailed discussion of the rôle of the designer, carried out in terms of PCT.

We could also attempt to find ways of orienting and evaluating design education in terms of the constructs being developed. Such psychometric approaches to job analysis are becoming more common, and repertory grid techniques appear to be useful tools in this regard. This opens up an interesting possibility, for such a research programme could be linked to the use of grid analysis for requirements capture discussed earlier. This would truly reconcile software design with education, for the constructs to be developed, whether these are to be educated into man or designed into machine , become the basis for both processes. This would force us to ask some very hard (in all senses) questions, such as who provides the requirements for education: that is, whose constructs are significant, and whose constructs prove dominant in the social processes of education and design.

Appendix A

Related Publications

This appendix lists publications by the researcher that are related to this research programme, together with a brief comment on the rôle they have played.

1. *Mathematics and computer science: coming together again?*: Invited presentation at the Standing Conference for Heads of Mathematics, Statistics and Computing, Scarborough, 1984.

   This presentation discussed the approach to teaching mathematics that the author was developing at Hatfield. It was largely because of the difficulties in defending this approach in a rational way that this research programme was undertaken.

2. *An education programme for software engineers*: Proceedings of the first British software engineering conference, Brighton, 1986. Published by Springer-Verlag. Joint authors: J. Jones and R. Shaw.

   This paper discusses the design and implementation of the Postgraduate Diploma in Software Principles and Practice, and sets the scene for some of the experiments described in this thesis.

3. *Using OBJ for Concurrency*: Invited paper at the BCS-FACS Colloquium on OBJ, London, 1986.

   This paper discusses the interpretation of algebraic specifications, and attempts to capture laws of interaction within such a paradigm.

4. *A paradigm for the development of distributed systems*: proceedings of the 14th, IFAC/IFIP workshop on real time programming, Hungary, 1986. Joint authors: G. Bull and R. Mitchell

   This paper discusses the notion of a paradigm, and the rôle that methods, languages and tools play in establishing a paradigm for real time systems. It was the catalyst for investigating the work of Kuhn.

5. *Future Research Directions*: Esprit deliverable for project Peacock, 1986.

This paper raises many of the questions that are still identified as open research issues within this thesis, but without the benefit of the framework established here in which to phrase them.

6. *Essential Mathematics for Software Engineers*: Published by Peter Peregrinus, on behalf of the IEE, 1987 Joint Authors: Slater et. al.

   This comprises a set of books and a video, produced as part of an Alvey project. The approach taken was not based on theory construction, however, but on a more traditional concept of applied mathematics. It was partly in reaction to this work that the author set out to establish explicitly the theory building approach to software design.

7. *Software Engineering Mathematics*: Published by Pitmans, 1988 (Also published by Adison Wesley in the USA, 1989, and in german translation by Springer-Verlag in 1990) Joint author: J Woodcock

   This textbook was written explicitly to support the teaching of the theory building view of design.

8. *Mathematics not method*: presentation at the IMA conference on software engineering mathematics, London, 1988. Joint author: R. Mitchell

   This paper was structured around the theory building view, and illustrated the use of partial theory presentations in specifications.

9. *Selfconscious or unselfconscious design*: Jounal of Information Technology, 5(1), March 1990.

   This paper discusses the two paradigms established by selfconscious and unselfconscious design, and their implications for teachers.

10. *Putting mathematics to use*: invited chapter in "Managing Complexity in Software Engineering, Edited by R. J. Mitchell, Published by Peter Peregrinus, 1990.

    This paper puts forward the theory building view to an industrial audience.

11. *Mathematics for software engineers*: proceedings of mathematics in a changing culture, April, 1990, Glasgow College.

    This paper discusses the Hatfield approach to formal methods in the context of changes to perceptions of mathematics generally. It also makes explicit the theory building approach.

12. *Humane mathematics for software engineers*: presented at the IEE colloquium on software engineering education, February, 1991. Joint author: J. Brown

    This paper presents experiences of adopting the theory building approach with HND students.

13. *Logic and Correctness Proofs*: invited chapter in the Software engineering reference book, published by Butterworths, 1991.

This chapter discusses the rôle of correctness proofs in establishing fitness for purpose, and provides a background to the task of carrying out such proofs.

14. *Structuring Specifications*: submitted to Formal Aspects of Computer Science, 1991. Joint author: R. Mitchell

    This paper discusses the implications of specification structure on system maintainability, and proposes an alternative style for Z specifications.

15. *The mathematical revolution inspired by computing*: Editing of the proceedings of the conference, published by OUP, 1991,Co-editor: J. Johnson

    This conference raised many of the issues covered in this thesis, but from the mathematician's perspective.

Bibliography

[Ach68] Peter Achinstein. *Concepts of Science.* John Hopkins Press, 1968.

[Aga86] Joseph Agassi. Science and the interpersonal. In Mark Amsler, editor, *The Languages of Creativity*, pages 30–46. University of Delaware Press, 1986.

[Ale64] Christopher Alexander. *Notes on the Synthesis of Form.* Harvard University Press, 1964.

[Alt89] J. L. Alty. Machine expertise. In K. J. Gilhooly, editor, *Human and Machine Problem Solving*, pages 123–160. Plenum Press, 1989.

[AM90] Igor Aleksander and Helen Morton. *An Introduction to Neural Computing.* Chapman and Hall, 1990.

[App87] *HyperCard User Guide.* Apple Computers, 1987.

[Arb90] Beno Arbel. From tricks to strategies for problem solving. *International Journal of Mathematics Education for Science and Technology*, 21(3):429–438, 1990.

[Ars34] Arsitotle. *Nicomachean Ethics.* William Heinemann, 1934. Translated by H. Rackham.

[Asi62] Morris Asimow. *Introduction to Design.* Prentice Hall, 1962.

[Asi74] M. Asimov. A philosophy of engineering design. In Friedrich Rapp, editor, *Contributions to a Philosophy of Technology*, pages 150–157. D.Reidel Publishing Company, 1974.

[AT89] John R. Anderson and Ross Thompson. Use of analogy in production system architecture. In Stell Vosniadou and Andrew Ortony, editors, *Similarity and Analogical Reasoning*, pages 267–297. Cambridge University Press, 1989.

[Aus68] D. P. Ausubel. *Educational Psychology: A Cognitive View.* Holt, Reinhart and Winston, 1968.

[Bac90] Roland C. Backhouse. Constructive type theory—a view from computer science. In Edsger W. Dijkstra, editor, *Formal Development of Programs and Proofs*, pages 1–32. Addison Wesley, 1990.

[Ban65] Don Bannister. The genesis of schizophrenic thought disorder: Re-test of the serial invalidation hypothesis. *British Journal of Psychiatry*, 111:377–382, 1965.

[Ben88] Jon Bentley. Teaching the tricks of the trade. In G. Goos and J. Hartmanis, editors, *Software Engineering Education: Proceedings of the 1988 S.E.I Conference*, pages 1–8. Springer-Verlag, 1988.

[BF86] Don Bannister and Fay Fransella. *Inquiring Man:The Psychology of Personal Constructs (Third Edition).* Croom Helm, 1986.

[BG77] R. M. Burstall and J. A. Goguen. Putting theories together to make specifications. In *Proceeding of the Fifth International Joint Conference on Artificial Intelligence*, 1977.

[BJ72] J. D. Bransford and M. K. Johnson. Contextual prerequisites for understanding: Some investigations of comprehension and recall. *Journal of Verbal Learning and Behaviour*, (61):717–726, 1972.

[Bko89] R. Bkouche. Mathematics and physics: where is the difference? In Blum, Berry, Bihler, Huntley, Kaiser-Messmer, and Proske, editors, *Application of Modelling in Learning and Teaching Mathematics*, pages 49–54. Ellis-Horwood, 1989.

[Box88] Philip J. Boxer. Regnancy: a shadow over personal construing. In Fay Fransella and Laurie Thomas, editors, *Experimenting with Personal Construct Psychology*, pages 418–425. Routledge and Kegan Paul, 1988.

[Bra69] Theodore Brameld. The educational philosopher as "liaison officer". In Christopher J. Lucas, editor, *What is Philosophy of Education?*, pages 217–220. MacMillan, 1969.

[Bri77] Jane Bridge. *Beginning Model Theory.* Oxford University Press, 1977.

[Bro69] Harry S. Broudy. How philosophical can philosophy of education be? In Christopher J. Lucas, editor, *What is Philosophy of Education?*, pages 114–122. MacMillan, 1969.

[Bro84] Dave Brown. My accordian's stuffed full of paper. *ACM SIGSOFT Software Engineering Notes*, 9(4):58–60, July 1984.

[BS77] R. Bhaskar and H. A. Simon. Problem solving in semantically rich domains. *Cognitive Science*, (1), 1977.

[BS82] John D. Bransford and Barry S. Stein. Differences in approaches to learning: an overview. *Journal of Experimental Psychology: General*, 111(4):390–398, 1982.

[BSC90] Bsc in Computer Science: scheme booklet, 1990. School of Information Science, Hatfield Polytechnic.

[Bun74] M. Bunge. Technology as applied science. In Friedrich Rapp, editor, *Contributions to a Philosophy of Technology*, pages 19–39. D.Reidel Publishing Company, 1974.

[Bur80] R. M. Burstall. Electronic category theory. In *Proceedings of the Ninth Annual Symposium on the Mathematical Foundations of Computer Science, Rydzyua, Poland*, 1980.

[Bur89] H. Burkhardt. Mathematical modelling in the curriculum. In Blum, Berry, Bihler, Huntley, Kaiser-Messmer, and Proske, editors, *Application of Modelling in Learning and Teaching Mathematics*, pages 1–11. Ellis-Horwood, 1989.

[Cau77] Robert L. Causey. Relpy to Hilary Putnam. In *The Structure of Scientific Theories*, pages 456–357. University of Illinois Press, 1977.

[CFG81] M. T. H. Chi, P. J. Feltovich, and R. Glaser. Categorization and representation of physics problems by experts and novices. *Cognitive Science*, 5:121–152, 1981.

[CG85] M. T. H. Chi and R. Glaser. Problem solving ability. In Sternberg R, editor, *Human Abilities: An Information Processing Approach*, pages 227–250. Freeman, 1985.

[CG88] Ravinder Chandok and Terry A. Gill. Adaedu project: supporting the use of ada in introductory computer science. In G. Goos and J. Hartmanis, editors, *Software Engineering Education: Proceedings of the 1988 S.E.I Conference*, pages 199–207. Springer-Verlag, 1988.

[CH82] R. S. Culver and J. T. Hackos. Perry's model of intellectual development. *Engineering Education*, 1982.

[Chu81] P. Churchland. Eliminative materialism and the propositional attitudes. *Journal of Psychology*, 78(2):67–90, 1981.

[Cla89] Andy Clark. *Microcognition.* MIT Press, 1989.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design.* Addison Wesley, 1988.

[Coh82] B. Cohen. Justification of formal methods for system specification. *Software and Microsystems*, 1(65):119–127, August 1982.

[Coh83] Robert S. Cohen. Social implications of recent technological innovations. In Paul T. Durbin andFriedrich Rapp, editor, *Philosophy and Technology*, pages 35–47. D.Reidel Publishing Company, 1983.

[Coh86] B. Cohen. Total curriculum inversion. *IEE News*, August 1986.

[CP90a] B. Cohen and D. Pitt. Proof obligations 3: Concurrent systems. In *Safetynet90*, December 1990.

[CP90b] Bernie Cohen and David Pitt. The identification and discharge of proof obligations. In *Proceedings of the Conference on New Directions in Software Development.* Wolverhampton Polytechnic, 1990.

[CP90c] Bernie Cohen and David Pitt. Proof obligations 2—state-based systems. In *Proceedings of the Colloquium on High-Integrity Systems.* University of Warick, 1990.

[CS73] Chase and Simon. Perception in chess. *Cognitive Psychology*, 4:55–81, 1973.

[Cye80] Richard M. Cyert. Problem solving and educational policy. In D. T. Tuma and F. Reif, editors, *Problem Solving and Education: Issues in Teaching and Research*, pages 3–8. Laurence Elbaum, 1980.

[dC77] L. Sprague de Camp. *Ancient Engineers.* Tandem, 1977.

[Deh86] Edmund Dehnert. The dialectic of technology and culture. In Mark Amsler, editor, *The Languages of Creativity*, pages 109–141. University of Delaware Press, 1986.

[Den88] Peter J. Denning. Computing as a discipline. *ACM SIGCSE Bulletin*, 21(1):41, February 1988.

[Dew69] John Dewey. Philosophy as the general theory of education. In Christopher J. Lucas, editor, *What is Philosophy of Education?*, pages 74–80. MacMillan, 1969.

[Die83] George Dieter. *Engineering Design.* McGraw-Hill, 1983.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[Dij78] E. W. Dijkstra. On a political pamphlet from the middle ages. *ACM Software Engineering Notes*, 3(2):14–16, April 1978.

[Dij82] Edsger W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.

[Dij89] Edsger Dijkstra. On the cruelty of really teaching computer science. *Comunications of the ACM*, 32(12):1398–1404, 1989.

[DIL90] N. W. Davis, M. Irving, and J. E. Lee. The evolution of object-oriented design form concept to method. In R. J. Mitchell, editor, *Managing Complexity in Software Engineering*, pages 21–50. Peter Peregrinus, 1990.

[diS87] Andrea diSessa. Phenomenology and the evolution of intuition. In Claude Janvier, editor, *Problems of Representation in the Teaching and Learning of Mathematics*, pages 83–96. Lawrence Erlbaum Associates, 1987.

[DLH65] C. B. DeSoto, M. London, and S. Handel. Social reasoning and spatial logic. *Journal of Personality and Social Psychology*, (2):513–521, 1965.

[DLP77] Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Langugages*, pages 206–214, January 1977.

[Do89] Denning and others;. Computing as a discipline. *Communications of the ACM*, 32(1):9–23, January 1989.

[Dol89] William Doll. Foundations for a post-modern curriculum. *Journal of Curriculum Studies*, 21(3):243–253, 1989.

[Duc69] C. J. Ducasse. On the function and nature of the philosophy of education. In Christopher J. Lucas, editor, *What is Philosophy of Education?*, pages 167–175. MacMillan, 1969.

[Duc83] S. Duck. Sociality and cognition in personal construct theory. In J. R. Adam-Webber and J. C. Mancuso, editors, *Applications of Personal Construct Theory.* Academic Press, 1983.

[Dun45] Duncker. On problem solving. Monograph 279, American Psychological Society, 1945.

[Dye86] James Wayne Dye. The poetization of science. In Mark Amsler, editor, *The Languages of Creativity*, pages 92–108. University of Delaware Press, 1986.

[Ell72] Jacques Ellul. The technological order. In Carl Mitcham and Robert Mackey, editors, *Philosophy and Technology*, pages 86–109. The Free Press, New York, 1972.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1.* Springer-Verlag, 1985.

[EN69] G. W. Ernst and A. Newell. *A Case Study in Generality and Problem Solving.* Academic Press, 1969.

[Ern89] P. Ernest. Philosophy, mathematics and education. *International Journal of Mathematical Education in Science and Technology*, 20(4):555–559, 1989.

[F̈78] H. Fürst. Modes of constructions and their change through validation and invalidation. *Acta Universitatis Upsaliensis*, 5, 1978.

[Fei72] James K. Feibleman. Pure science, applied science, and technology: An attempt at definitions. In Carl Mitcham and Robert Mackey, editors, *Philosophy and Technology*, pages 33–41. The Free Press, New York, 1972.

[Fei82] James K. Feibleman. *Technology and Reality.* Martinus Nijhoff Publishers, 1982.

[Feu69] Lewis S. Feuer. American philosophy is dead. In Christopher J. Lucas, editor, *What is Philosophy of Education?*, pages 34–41. MacMillan, 1969.

[Fey70a] Paul K. Feyerabend. Against method: Outline of an anarchistic theory of knowledge. In Michael Radnor and Stephen Winohur, editors, *Minnesota Studies in the Philosophy of Science: Volume IV*, pages 17–130. University of Minnesota Press, 1970.

[Fey70b] Paul K. Feyerabend. Consolations for the specialist. In Imre Lakatos and Alan Musgrave, editors, *Criticism and the Growth of Knowledge*, pages 197–230. Cambridge University Press, 1970.

[Fey75] Paul Feyerabend. *Against method : outline of an anarchistic theory of knowledge.* NLB, 1975.

[Fey87] Paul Feyerabend. *Farewell to Reason.* Verso, 1987.

[Fin86] Maurice A. Finocchiaro. The two cultures today: A third look. In Mark Amsler, editor, *The Languages of Creativity*, pages 13–29. University of Delaware Press, 1986.

[Fis87] F. Fisher. Ways of knowing and the ecology of change. In *Proceedings of Barriers to Change, Unesco Network for Appropriate Technology Seminar, University of Melbourne, 11-16 November*, 1987.

[Fis89] R. Fischer. Social change and mathematics. In Blum, Berry, Bihler, Huntley, Kaiser-Messmer, and Proske, editors, *Application of Modelling in Learning and Teaching Mathematics*, pages 12–21. Ellis-Horwood, 1989.

[Fra86] William Frawley. Science, discourse, and knowledge representation. In Mark Amsler, editor, *The Languages of Creativity*, pages 68–91. University of Delaware Press, 1986.

[Fra88] Fay Fransella. PCT: still radical thirty years on? In Fay Fransella and Laurie Thomas, editors, *Experimenting with Personal Construct Psychology*, pages 26–35. Routledge and Kegan Paul, 1988.

[Gas72] Jose Ortega y Gasset. Thoughts on technology. In Carl Mitcham and Robert Mackey, editors, *Philosophy and Technology*, pages 290–316. The Free Press, New York, 1972.

[GB61] R. M. Gagné and L. T. Brown. Some factors in the programming of conceptual learning. *Journal of Experimental Psychology*, 69:313–321, 1961.

[Gen89] Dedre Gentner. The mechanisms of analogical learning. In Stell Vosniadou and Andrew Ortony, editors, *Similarity and Analogical Reasoning*, pages 199–241. Cambridge University Press, 1989.

[Gib79] J. J. Gibson. *The Ecological Approach to Visual Perception.* Houghton Mifflin, 1979.

[Gil81] R. Giles. Lukasiewicz logic and fuzzy set theory. In E. H. Mamdani and B. R. Gaines, editors, *Fuzzy Reasoning and its Applications*, pages 117–131. Academic Press, 1981.

[GO86] Leo Gugerty and Gary M. Olson. Comprehension differences in debugging by skilled and novice programmers. In Elliot Soloway and Sitharama Iyengar, editors, *Empirical Studies of Programmers*, pages 13–27. Ablex Publishing Corporation, 1986.

[Goe78] W. Goethe. *Theory of Colours.* MIT Press, 1978.

[Gog90] Joseph A. Goguen. Truth and meaning beyond formalism. In *Four Pieces on Error, Truth and Reality,* pages 18–27. Programming Research Group, Okford University, 1990. Monograph PRG-89.

[Gol70] Goldberg. In defense of ether. In R. McCormmach, editor, *Historical Studies in the Philosophy of Science*, pages 89–125. 1970.

[Goo76] N. Goodman. *Languages of Art (Second Edition).* Hackett, 1976.

[Gor61] W. J. J. Gordon. *Synectics.* Harper and Row, 1961.

[Gou89] Noel Gough. From epistemology to ecopolitice: renewing a paradigm for curriculum. *Journal of Curriculum Studies*, 21(3):225–241, 1989.

[Gre73] J. G. Greeno. The structure of memory and the process of solving problems. In R. L. Solso, editor, *Contempory Issues in Cognitive Psychology.* Winston, 1973.

[Gre84] Richard L. Gregory. *Mind in Science.* Peregrine, 1984.

[Gri81] David Gries. *The Science of Programming.* Springer-Verlag, 1981.

[GT86] N. E. Gibbs and A. B. Tucker. *Communications of the ACM*, 29(3), March 1986.

[GW88] Joseph A. Goguen and Timothy Winkle. Introducing OBJ3. Report SRI-CSL-88-9, SRI International, August 1988.

[HA78] E. S. Harri-Augstein. Reflecting on structures of meaning. a processes of learning-to-learn. In Fay Fransella, editor, *Personal Construct Psychology 1977.* Academic Press, 1978.

[HA85] Sheila Harri-Augstein. Learning-to learn languages: New perspectives for personal observer. In D. Bannister, editor, *Issues and Approaches in Personal Construct Theory*, pages 47–66. Academic Press, 1985.

[Hai82] Brent Hailpern. *Verifying Concurrent Processes Using Temporal Logic.* Springer-Verlag, 1982.

[Hat89] Hatfield. Msc in Software Engineering: scheme booklet, 1989. School of Information Science, Hatfield Polytechnic.

[Hay78] Patrick J. Hayes. The naive physics manifesto. In D. Michie, editor, *Expert Systems in the Micro-electronic Age.* Edinburgh University Press, 1978.

[Hay85] Patrick J. Hayes. The second naive physics manifesto. In Jerry R. Hobbs and Robert C. Moore, editors, *Formal Theories of the Commonsense World*, pages 1–36. Ablex Publishing Corporation, 1985.

[Hei59] Martin Heidegger. *Introduction to Metaphysics.* O.U.P, 1959. Translated by Ralph Manheim.

[Hei62] Martin Heidegger. *Being and Time.* Blackwell, 1962. Translated by Macqarrie and Robinson.

[Hei77] Martin Heidegger. The question concerning technology. In *Basic Writings*, pages 296–304. Harper and Row, 1977. Translated by D. Krell.

[Hem65] C. G. Hempel. *Scientific Explanation.* Collier-Macmillan, 1965.

[Hem70] C. Hempel. On the "standard conception" of scientific theories. In M. Radnor and S. Winokur, editors, *Minnesota Studies in the Philosophy of Science*, pages 142–163. University of Minnesota Press, 1970.

[Hir69] Paul H. Hirst. Philosophy and educational theory. In Christopher J. Lucas, editor, *What is Philosophy of Education?*, pages 175–187. MacMillan, 1969.

[HK89] Rudy Hirschheim and Heinz K. Klein. Four paradigms of information systems development. *Communications of the ACM*, 32(10):1199–1216, October 1989.

[Hoa84] C. A. R. Hoare. Programming: Sorcery or science. *IEEE Software*, pages 5–16, April 1984.

[Hoa85] C. A. R. Hoare. Programs are predicates. In C. A. R. Hoare and J. C. Sheperdson, editors, *Mathematical Logic and Programming Languages*, pages 141–154. Prentice Hall, 1985.

[Hoa89] C. A. R. Hoare. An axiomatic basis for computer programming. In C. A. R. Hoare and C. B. Jones, editors, *Essays in Computing Science.* Prentice Hall, 1989.

[Hor88] James Horley. Construal of events: Personal constructs versus personal projects. In Fay Fransella and Laurie Thomas, editors, *Experimenting with Personal Construct Psychology*, pages 359–360. Routledge and Kegan Paul, 1988.

[Hun83] Alois Huning. Technology and human rights. In Paul T. Durbin andFriedrich Rapp, editor, *Philosophy and Technology*, pages 49–57. D.Reidel Publishing Company, 1983.

[Ihd83] Don Ihde. The historical-ontological priority of technology over science. In Paul T. Durbin andFriedrich Rapp, editor, *Philosophy and Technology*, pages 235–252. D.Reidel Publishing Company, 1983.

[Jah88] Marie Jahoda. The range of convenience of personal construct psychology—an outsider's view. In Fay Fransella and Laurie Thomas, editors, *Experimenting with Personal Construct Psychology*, pages 1–14. Routledge and Kegan Paul, 1988.

[Jam90] W. James. *The Principles of Psychology.* Holt, Rinehart and Winston, 1890.

[Jan87] A. D. Jankowicz. Whatever became of George Kelly? *American Psychologist*, 42(5):481–487, May 1987.

[Jar74] I. C. Jarvie. The social character of technological problems. In Friedrich Rapp, editor, *Contributions to a Philosophy of Technology*, pages 86–92. D.Reidel Publishing Company, 1974.

[JLS86] John Jones, Martin Loomes, and Roger Shaw. An education programme for practising software engineers. In *Proceedings of Software Engineering 86*, September 1986.

[Joh88] L. Johnson. Programs as psychological theories. Personal communication of unpublished paper, 1988.

[Jon86] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice Hall, 1986.

[Juk85] K. A. Jukes. Conference report. *IMA Bulletin*, 9:157–158, 1985.

[Kan11] Immanuel Kant. *Critique of Aesthetic Judgement.* Clarendon Press, 1911. Translated by J. C. Meredith.

[Kan29] Immanuel Kant. *Critique of Pure Reason.* MacMillan, 1929. Translated by N. Smith.

[Kea88] Mark T. Keane. *Analogical Problem Solving.* Ellis Horwood, 1988.

[Kel55] George A Kelly. *The Psychology of Personal Constructs, Vols I and II.* Norton, 1955.

[Kel57] J. W. Keltner. *Group Discussion Processes.* Longman, Green and Co., 1957.

[Kel63] George A Kelly. *A Theory of Personality.* Norton, 1963.

[Kid82] Tracy Kidder. *The Soul of a New Machine.* Penguin, 1982.

[Kin74] W. Kintsch. *Three Representation of Meaning in Memory.* Erlbaum, 1974.

[Kin84] J. L. Kinneavy. *A Theory of Discourse.* Prentice-Hall, 1984.

[Kne71] George Kneller. *Introduction to the Philosophy of Education.* Wiley, 1971.

[Knu74] Donald E. Knuth. Computer programming as an art. In *ACM Turing Award Lectures*, pages 33–46. Addison Wesley, 1974. Book Published in 1987.

[Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[Koz80] George Kozmetsky. The significant role of problem solving in education. In D. T. Tuma and F. Reif, editors, *Problem Solving and Education: Issues in Teaching and Research*, pages 151–157. Laurence Erlbaum, 1980.

[KT89] Keith J. Kolyoak and Paul R. Thagard. A computational model of analogical problem solving. In Stell Vosniadou and Andrew Ortony, editors, *Similarity and Analogical Reasoning*, pages 242–266. Cambridge University Press, 1989.

[Kuh70a] T. S. Kuhn. Logic of discovery or psychology of research? In Imre Lakatos and Alan Musgrave, editors, *Criticism and the Growth of Knowledge*, pages 1–24. Cambridge University Press, 1970.

[Kuh70b] Thomas S. Kuhn. *The Structure of Scientific Revolutions (Second Enlarged Edition).* University of Chicago Press, 1970.

[Kuh79] T. S. Kuhn. History of science. In P. Asquith and H. Kyburg, editors, *Current Research in Philosophy of Science.* Ann Arbor, 1979.

[Kyb68] H. E. Kyburg. *Philosophy of Science: A Formal Approach.* Macmillan, 1968.

[Lak70] I. Lakatos. Falsification and the methodology of scientific research programmes. In Imre Lakatos and Alan Musgrave, editors, *Criticism and the Growth of Knowledge*, pages 91–196. Cambridge University Press, 1970.

[Lak71] I. Lakatos. History of science and its rational reconstruction. In R. C. Buck and R. S. Cohen, editors, *Boston Studies in the Philosophy of Science, Volume 8*, pages 91–136. Reidel, 1971.

[Lak74] Imre Lakatos. Popper on demarcation and induction. In Paul Arthur Schlipp, editor, *The Philosophy of Karl Popper, Part 1*, pages 241–273. The Open Court Publishing Company, 1974.

[Lak76] Imre Lakatos. *Proofs and Refutations.* Cambridge University Press, 1976.

[Lak78] Imre Lakatos. *Mathematics, Science and Epistemology: Philosophical Parers Volume 2.* Cambridge University Press, 1978.

[Lan64] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.

[LB91] M. Loomes and J. Brown. Humane mathematics for software engineers. In *Proceedings of IEE Colloquium on Software Engineering Education*, 1991.

[LC87] Guy Lapalme and Pierre Chartray. An education system for the study of tasking in ada. *IEE Transactions on Education*, 30(3):185–191, August 1987.

[Leh80] M. M. Lehman. Programs, life cycles and laws of software evolution. *Proceedings of the IEEE*, 68(9), September 1980.

[LM88] M. Loomes and R. Mitchell. Mathematics not method. In *Proceedings of IMA Colloquium on Software Engineering Education*, 1988.

[Loo84] M. Loomes. Mathematics and computer science: coming together again? Presented at the Standing Conference for Heads of Maths, Stats, and Computing, Scarborough, 1984, 1984.

[Loo86] Martin Loomes. Future research directions. In *Esprit Project Report*. 1986.

[Loo90a] M. Loomes. Mathematics for software engineers. In *Proceedings of Mathematics in a Changine Culture, Glasgow College, April 1990*, 1990.

[Loo90b] Martin Loomes. Putting mathematics to use. In R. J. Mitchell, editor, *Managing Complexity in Software Engineering*, pages 87–96. Peter Peregrinus, 1990.

[Loo90c] Martin Loomes. Selfconscious or unselfconscious software design? *Journal of Information Technology*, 5(1):33–36, March 1990.

[Loo91] Martin J. Loomes. Logics and program correctness. In J. McDermid, editor, *Software Engineers Reference Book*. Butterworths, 1991. In Press.

[LPLS86] David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. Mental models and software maintenance. In Elliot Soloway and Sitharama Iyengar, editors, *Empirical Studies of Programmers*, pages 80–98. Ablex Publishing Corporation, 1986.

[LR79] Hans Lenk and Gunther Ropohl. Towards an interdisciplinary and pragmatic philosophy of technology. In P. Durbin, editor, *Research in Philosophy and Technology*, volume 2. JAI Press, 1979.

[LT73] H. Lamm and G. Trommsdorff. Group versus individual performance on tasks requiring ideational proficiency: A review. *European Journal of Social Psychology*, 3:361–388, 1973.

[Luk89] Paul A. Luker. Never mind the language, what about the paradigm? *ACM SIGCSE Bulletin*, 21(1):253–256, February 1989.

[Mag73] Bryan Magee. *Popper*. Fontana, 1973.

[Mal55] I. Maltzmann. Thinking: from a behaviouristic point of view. *Psychological Review*, (62):275–286, 1955.

[Mar59] Karl Marx. *Economic and philosophic manuscript of 1844*. Progress Publishers, 1959.

[Mar89] William Marion. Discrete mathematics for computer science majors— where are we? how do we proceed? *ACM SIGCSE Bulletin*, 21(1), February 1989.

[Mas87a] John Mason. Representing representing: notes following the conference. In Claude Janvier, editor, *Problems of Representation in the Teaching and Learning of Mathematics*, pages 207–214. Lawrence Erlbaum Associates, 1987.

[Mas87b] John Mason. What do symbols represent? In Claude Janvier, editor, *Problems of Representation in the Teaching and Learning of Mathematics*, pages 73–81. Lawrence Erlbaum Associates, 1987.

[Mat90] Stuart R. Matthews. *Nondeterministic Abstract Data Types and their Implementation.* PhD thesis, Hatfield Polytechnic, 1990.

[May75] Richard E. Mayer. Different problem-solving competencies established in learning computer programming with and without meaningful models. *Journal of Educational Psychology*, 67(6):725–734, 1975.

[May83] Richard E. Mayer. *Thinking, Problem Solving, Cognition.* Freeman, 1983.

[McC74] R. J. McCrory. The design method–a scientific approach to valid design. In Friedrich Rapp, editor, *Contributions to a Philosophy of Technology*, pages 158–173. D.Reidel Publishing Company, 1974.

[McW88] Spencer A. McWilliams. On becoming a personal anarchist. In Fay Fransella and Laurie Thomas, editors, *Experimenting with Personal Construct Psychology*, pages 17–25. Routledge and Kegan Paul, 1988.

[Med64] Peter Medawar. Is the scientific paper a fraud? In David Edge, editor, *Experiment*, pages 7–12. BBC Publications, 1964.

[Mey75] B. J. F. Meyer. *The Organisation of Prose and its Effects on Memory.* North-Holland, 1975.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.

[MGP60] G. Miller, E. Galanter, and K. Pribram. *Plans and the Structure of Behaviour.* Rinehart and Winston, 1960.

[Mic90] Donald Michie. Human and machine learning of descriptive concepts. *ICOT Journal*, (27):2–20, March 1990.

[Mil76] J. Millendorfer. Konturen einer dritten industriellen revolution. *Stimmer der Zeit*, (194), 1976.

[Mil80a] Harlan D. Mills. Software engineering education. *Proceedings of the IEEE*, 68(9):1158–1162, September 1980.

[Mil80b] Robin Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1980.

[Mit88] Richard J. Mitchell. *Literate Programming.* PhD thesis, Hatfield Polytechnic, 1988.

[ML82] P. Martin-Löf. Constructive mathematics and computer programming. In L. J. Cohen, editor, *Proceedings of the Sixth International Congress for Logic, Methodology and Philosophy of Science*, pages 153–175. North-Holland, 1982.

[ML91] Richard Mitchell and Martin Loomes. Structuring specifications. Submitted to Formal Aspects of Computer Science, 1991.

[MO89] Douglas Medin and Andrew Ortony. Psychological essentialism. In Stell Vosniadou and Andrew Ortony, editors, *Similarity and Analogical Reasoning*, pages 179–196. Cambridge University Press, 1989.

[Moo85] Robert C. Moore. A formal theory of knowledge and action. Report CSLI-85-31, Center for the Study of Language and Information, 1985.

[Mor90] Carroll Morgan. *Programming from Specifications.* Prentice-Hall, 1990.

[MP88] Freeman L. Moore and Phillip R. Purvis. Meeting the training needs of software engineers at texas instruments. In G. Goos and J. Hartmanis, editors, *Software Engineering Education: Proceedings of the 1988 S.E.I Conference*, pages 32–44. Springer-Verlag, 1988.

[MS88] Francesco Mancini and Antonio Semerari. Kelly and popper: A constructivist view of knowledge. In Fay Fransella and Laurie Thomas, editors, *Experimenting with Personal Construct Psychology*, pages 69–79. Routledge and Kegan Paul, 1988.

[Mun88] Phil Munson. Some thoughts on problem solving. In *Problem Solving*, pages 11–13. SCDC Publications, 1988.

[Nau85] Peter Naur. Programming as theory building. *Microprocessing and Microprogramming*, 15:253–261, 1985. Invited keynote address at Euromicro 84, Copenhagen, Denmark.

[NC86] A. F. Norcio and L. J. Chmura. Design activity in developing modules for complex software. In Elliot Soloway and Sitharama Iyengar, editors, *Empirical Studies of Programmers*, pages 99–116. Ablex Publishing Corporation, 1986.

[Neu85] Peter G. Neumann. Letter from the editor. *ACM SIGSOFT Software Engineering Notes*, 10(3):3–16, July 1985.

[New69] George L. Newsome, Jr. Educational philosophy and the educational philosopher. In Christopher J. Lucas, editor, *What is Philosophy of Education?*, pages 160–167. MacMillan, 1969.

[Osb63] A. F. Osborn. *Applied Imagination.* Scribner's, 1963.

[PC85] D. L. Parnas and P. C. Clements. A rational design process: How and why to fake it. In Hartmut Ehrig, Christiane Floyd, Maurice Nivat, and James Thatcher, editors, *Formal Methods and Software Development, Proceedings of the Joint Conference on Theory and Practice of Software Development (TAP-SOFT), Berlin, March 1985. Volume 2*, pages 80–100. Springer-Verlag, Berlin, 1985. LNCS 186.

[Pea11] Karl Pearson. *The Grammar of Science, Third Edition.* Adam and Charles Black, 1911.

[Pet82] Henry Petroski. *To Engineer is Human.* Macmillan, 1982.

[Pfl87] Shari Lawrence Pfleeger. *Software Engineering.* Macmillan, 1987.

[PM86] D. N. Perkins and Fay Martin. Fragile knowledge and neglected strategies in novice programmers. In Elliot Soloway and Sitharama Iyengar, editors, *Empirical Studies of Programmers*, pages 213–229. Ablex Publishing Corporation, 1986.

[Poi13] Henri Poincare. *Science and Hypothesis.* The Science Press, 1913.

[Pol68] George Polya. *Mathematics and Plausible Reasoning, Volume 2: Patterns of Plausible Inference.* Oxford University Press, 1968.

[Pop59] Karl R. Popper. *The Logic of Scientific Discovery.* Hutchinson and Co., 1959.

[Pop63] Karl R. Popper. *Conjectures and Refutations.* Routledge and Kegan Paul, 1963.

[Pop72] Karl R. Popper. *Objective Knowledge.* Oxford University Press, 1972.

[Put74] Hilary Putnam. The "corroboration" of theories. In Paul Arthur Schlipp, editor, *The Philosophy of Karl Popper, Part 1*, pages 221–240. The Open Court Publishing Company, 1974.

[Qui60] W. V. O. Quine. *Word and Object.* MIT Press, 1960.

[Ran60] J. H. Randall. *Aristotle.* Columbia University Press, 1960.

[Rap74] F. Rapp. Technology and natural sciences–a methodological investigation. In Friedrich Rapp, editor, *Contributions to a Philosophy of Technology*, pages 93–114. D.Reidel Publishing Company, 1974.

[Rap81] Friedrich Rapp. *Analytical Philosophy of Technology.* D. Reidel Publishing Company, 1981.

[Rei65] W. Reitman. *Cognition and Thought.* Wiley, 1965.

[Ric88] William E. Richardson. Undergraduate software engineering education. In G. Goos and J. Hartmanis, editors, *Software Engineering Education: Proceedings of the 1988 S.E.I Conference*, pages 121–144. Springer-Verlag, 1988.

[Rid85] Riddle. Letter from the chairman. *ACM SIGSOFT*, 10(2):1–3, April 1985.

[Ris86] Robert S. Rist. Plans in program definition, demonstration, and development. In Elliot Soloway and Sitharama Iyengar, editors, *Empirical Studies of Programmers*, pages 28–47. Ablex Publishing Corporation, 1986.

[Rob79] Roberts. *Measurement Theory with Applications to Decision Making, Utility and the Social Sciences.* Addison Wesley, 1979.

[Rop83] Gunther Ropohl. A critique of technological determinism. In Paul T. Durbin andFriedrich Rapp, editor, *Philosophy and Technology*, pages 83–96. D.Reidel Publishing Company, 1983.

[RS68] W. G. Roughead and J. M. Scandura. What is learned in mathematical discovery. *Journal of Educational Psychology*, 69:283–289, 1968.

[Rum75] D. E. Rumelhart. Notes on a schema for stories. In D. G. Brown and A. Collins, editors, *Representation and Understanding: Studies in Cognitive Science.* Academic Press, 1975.

[Rya84] A. Ryan. *The Philosophy of the Social Sciences.* Macmillan, 1984.

[Ryl49] Gilbert Ryle. *The Concept of Mind.* Peregrin Books, 1949.

[Sch62] J. Schwartz. The pernicious influence of mathematics on science. In E. Nagel, P. Suppes, and A. Tarski, editors, *Logic, Methodology and Philosophy of Science: Proceedings of the 1960 International Congress*, pages 356–360. Stanford University Press, 1962.

[Sch77] E. F. Schumacher. *A Guide for the Perplexed.* Cape, 1977.

[Sch81] Peter Schefe. On foundations of reasoning with uncertain facts and vague concepts. In E. H. Mamdani and B. R. Gaines, editors, *Fuzzy Reasoning and its Applications*, pages 189–216. Academic Press, 1981.

[Sch89a] W. L. Scherlis. Response to dijkstra. *Comunications of the ACM*, 32(12):1406–1407, 1989.

[Sch89b] H. Schupp. Applied mathematics instruction in the lower secondary level—between traditional and new approaches. In Blum, Berry, Bihler, Huntley, Kaiser-Messmer, and Proske, editors, *Application of Modelling in Learning and Teaching Mathematics*, pages 37–48. Ellis-Horwood, 1989.

[Shi86] Mike Shields. Solving the interface equation. *FACS FACTS*, 8(6):7–26, October 1986.

[SI86] Elliot Soloway and Sitharama Iyengar, editors. Ablex Publishing Corporation, 1986. Papers Presented at the First Workshop on Empirical Studies of Programmers, Washington DC, July 1986.

[Sie86] Robert S. Siegler. Encoding and the development of problem solving. In S. Chipman, J. Segal, and R. Glaser, editors, *Thinking and Learning Skills, Volume 2*, pages 161–185. Laurence Erlbaum Associates, 1986.

[Sim69] Herbert A. Simon. *The Sciences of the Artificial.* M.I.T. Press, 1969.

[Ske82] R. Skemp. Surface structure and deep structure. *Visible Language*, 16(3):281–288, 1982.

[Sko72] Henryk Skolimowski. The structure of thinking in technology. In Carl Mitcham and Robert Mackey, editors, *Philosophy and Technology*, pages 42–49. The Free Press, New York, 1972.

[Sla89] Richard A. Slaughter. Cultural reconstruction in the post-modern world. *Journal of Curriculum Studies*, 21(3):255–270, 1989.

[Smu87] Raymond Smullyan. *Forever Undecided.* Oxford University Press, 1987.

[Sno34] C. P. Snow. *The Search.* Gollanz Press, 1934.

[Som87] Ian Sommerville. Starships in information space. Working Paper ISF/11.0, from the IED ISF study, 1987.

[SS82] V. Stewart and A. Stewart. *Business Applications of Repertory.* McGraw-Hill, 1982.

[Sti83] S. Stitch. *From Folk Psychology to Cognitive Science.* MIT Press, 1983.

[Sto77] J. E. Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[Str83] Elisabeth Stroker. Philosophy of technology: Problems of a philosophical discipline. In Paul T. Durbin andFriedrich Rapp, editor, *Philosophy and Technology*, pages 323–336. D.Reidel Publishing Company, 1983.

[Str85] Peter Stringer. You decide what your title is to be and [read] write to that title. In Don Bannister, editor, *Issues and Approaches in Personal Construct Theory*, pages 210–232. Academic Press, 1985.

[Suc87] Lucy A. Suchman. *Plans and situated actions.* Cambridge University Press, 1987.

[Sup68] P. Suppes. The desirability of formalisation in science. *Journal of Philosophy*, 65:651–664, 1968.

[Sup77] Frederick Suppe. *The Structure of Scientific Theories.* University of Illinois Press, 1977. A collection of papers with an extensive forward by Suppe.

[Sup89] Frederick Suppe. *The Semantic Conception of Theories and Scientific Realism.* University of Illinois Press, 1989.

[Tar56] A. Tarski. Concept of truth. In A. Tarski, editor, *Logic, Semantics and Mathematics*. 1956.

[Tay69] Albert J. Taylor. What is philosophy of education? In Christopher J. Lucas, editor, *What is Philosophy of Education?*, pages 199–209. MacMillan, 1969.

[TDM81] Ryan D. Tweney, Michael E. Doherty, and Clifford R. Mynatt. *On Scientific Thinking*. Columbia University Press, 1981.

[THA88] Laurie F. Thomas and E. Shela Harri-Augstein. Constructing environments that enable self-organised learning: the principles of intelligent support. In Fay Fransella and Laurie Thomas, editors, *Experimenting with Personal Construct Psychology*, pages 92–110. Routledge and Kegan Paul, 1988.

[Tho11] E. L. Thorndike. *Animal Intelligence*. Macmillan, 1911.

[Tho77] P. W. Thorndyke. Cognitive structures in comprehension and memory of narrative discourse. *Cognitive Psychology*, 9:77–110, 1977.

[Tho89] J. C. Thomas. Problem solving by human–machine interaction. In K. J. Gilhooly, editor, *Human and Machine Problem Solving*, pages 317–362. Plenum Press, 1989.

[TM87] Wladyslaw M. Turski and Thomas S. E. Maibaum. *The Specification of Computer Programs*. Addison-Wesley, 1987.

[Tru87] Jennifer Trusted. *Inquiry and Understanding*. Macmillan Educational, 1987.

[Tul87] Colin Tully. What is an ISF? Preliminary verion of document ISF/8.0, from the IED Software Factory Study, 1987.

[Tur81] Władysław M. Turski. Software stability. In *Systems Architecture: Proceedings of the Sixth ACM European Regional Conference*, pages 107–116. Westbury House, 1981.

[Tur84] S. Turkle. *The Second Self*. Simon and Schuster, 1984.

[Vos90] A. J. R. Voss. The need for a quality culture for software development. In R. J. Mitchell, editor, *Managing Complexity in Software Engineering*, pages 113–126. Peter Peregrinus, 1990.

[Wau84] P. Waugh. *Metafiction: The Theory and Practice of Self-Conscious Fiction*. Methuen, 1984.

[Wer59] M. Wertheimer. *Productive Thinking*. Harper and Row, 1959.

[Wil64] W. T. Williams. The computer botonist. In David Edge, editor, *Experiment*, pages 48–54. BBC Publications, 1964.

[WL88] James Woodcock and Martin Loomes. *Software Engineering Mathematics*. Pitmans, 1988.

[Wun73] W. Wundt. *An Introduction to Psychology.* Arno Press, 1973. Originally Published in German in 1911.

[Zem75] Heinz Zemanek. Formalization - history, present and future. In *Lecture Notes on Computer Science 23 Programming Methodology*, pages 477–501. Springer, 1975.