Understanding Software Engineering:

From Analogies With Other Disciplines To Philosophical Foundations.

Submitted by

Jason Baragry B.Sc(Hons).

A thesis submitted in total fulfilment

of the requirements for the degree of

Doctor of Philosophy.

Department of Computer Science & Computer Engineering.

School of Engineering.

Faculty of Science, Technology & Engineering.

La Trobe University.

Bundoora, Victoria. 3083.

Australia.

July 2000

Table Of Contents

List of Figures

List of Tables

Summary

Since its origin, software engineering research has sought to improve the practice of software development based on analogies with traditional engineering disciplines. Those disciplines are perceived to have more rigorous, predictable, and mature development methods and similar techniques would be beneficial for software developers. The underlying assumption is that the systems built by the respective disciplines are similar enough for the transfer of ideas to be possible. That is, software engineers have an artefact engineering view of software development.

However, a detailed comparison of the way software developers and traditional engineers address an identical problem shows that significant differences exist between the respective design approaches. Traditional engineers design and build corporeal artefacts to solve real-world problems. In contrast, software developers solve real-world problems by implementing models of reality that explain and automate a perceived process.

These differences help explain many research issues that are currently the source of debate, for example, the area of software architecture. A case study of a large-scale system highlights the issues involved and a chronological review of the theory shows existing theory is based on specious analogies with traditional notions of architecture that fail to consider significant differences between software development and other engineering disciplines.

A foundation for the understanding of software systems is then proposed based on an examination of research performed in other disciplines that are concerned with model building. They include metaphysics and epistemology, the history and philosophy of science, and the psychology of cognition. The conclusion is that models of reality cannot be understood using the same principles as other built artefacts and that software engineers must consider the role of subjective interpretation in human understanding.

Finally, an examination of how research-based disciplines progress shows that the understanding of the phenomena under investigation is significantly influenced by guiding assumptions that can change over time. The conjecture of this thesis is that the artefact engineering view has been the established guiding assumption in software engineering research. However, a better understanding of the underlying principles of software systems leads to an improved approach to software engineering – the model building view. That different way of understanding is exemplified by conjecturing alternate explanations for software reuse, architecture, and design patterns. The objective for software engineering research now becomes how to engineer those explanatory models of reality.

Statement of Authorship

Except where reference is made in the text of the thesis, this thesis contains no material published elsewhere or extracted in whole or in part from a thesis by which I have qualified for or been awarded another degree or diploma.

No other person's work has been used without due acknowledgment in the main text of the thesis.

This thesis has not been submitted for the award of any degree or diploma in any other tertiary institution.

7th July 2000

Acknowledgements

The ideas presented in this thesis have taken a long time to coalesce and get down on paper and I'd like to thank the many people who have helped me, both academically and personally, during that time. My supervisor, Karl Reed, has been the primary interlocutor for discussing my ideas and though our debates have sometimes been passionate, my understanding of the issues has benefited significantly from them. I have acknowledged his input as 'personal communications' where appropriate, but I also thank him for his endless anecdotes of bygone eras that helped to clarify some issues.

I'd also like to thank him from supporting me when I changed research topic from the one originally suggested. As I moved from the specific software engineering topic of *Graphical Component Based Software Engineering* to an examination of philosophical foundations of the discipline as a whole, he repeatedly warned me of the inherent dangers of dealing with such a topic in a Ph.D. thesis. Nevertheless, he continued to let me follow my own path and provided the necessary feedback and guidance as required. Now that I can sit back at the end of this process I have a better idea of what he was talking about and realise how much easier it would have been had I stayed on the original topic. That seems to be all the more relevant at a time when higher education in this country appears to be making it harder and harder to explore these more open-ended research issues. To make a contribution by addressing some focused problem is of course worthy, but, for me, to have made a foray into the fundamental problems of a discipline, and feel one may have added to its understanding, is an experience of an entirely different kind.

Finally, I'd like to thank him for taking an interest and offering support during the personal issues that cropped up during the last few years.

I'd also like to thank the other members of the Amdahl Australian Intelligent Tools (AAITP) program of which this research formed a part. In particular David Cleary, whom I shared an office with for a few years, provided many useful discussions on my ideas as they were emerging. Jacob Cyubulski also provided useful comments and Fred Brkich, who listened to what I was doing and suggested I read some Aristotle and Feyerabend, thereby adding an extra couple of years to my work but making it all the more worthwhile. I'd also like to thank the AAITP project for its financial support and the many individuals who made the program possible.

The academic and administrative staff and my fellow postgraduate students in the Department of Computer Science and Computer Engineering provided plenty of help over the years and the department provided teaching opportunities when my AAITP funding ran out. I'd also like to thank the Research and Graduate Studies people of La Trobe University for not throwing me out when they could have. Thanks to John Fox and Geoff Cumming, from the departments of Philosophy and Psychology respectively, for helping me with my understanding of those relevant issues and to those friends who looked over sections of my thesis and helped with my often loose interpretation of the English language.

Finally, thanks to my family and friends who have stuck by me over the years and offered their support when required, especially over the last few months when it seemed I had vanished without trace. Also thanks to my housemates who have had to put up with my research all over the house and to those friends who helped me keep an appropriate balance between work and play.

If we knew what it was we were doing,

it wouldn't be called research,

would it?

– Albert Einstein.

1. Introduction

Since its emergence 30 years ago as a distinct research discipline, "Software Engineering" has developed many tools and techniques designed to help the software development community improve the way it builds systems. In that time, debate has contained many conflicting theories, ideas, conjectures, proposals, and analogies, concerning the nature of software development and how it can be improved. A central tenet of software engineering research is that the process of software development can benefit by learning from the development techniques of traditional engineering disciplines. That gained currency in the first software engineering conference in 1968 and it is still evident in the most recent research publications. See for example, the latest Annals of Software Engineering issue – *Comparative Studies of Engineering Approaches for Software Engineering* (Patel, Wang et al. 2000). For the entire duration of our young discipline, analogies with traditional engineering disciplines have served as the inspiration, evidence, and justification for many of our research ideas. They have pervaded our thought both explicitly and implicitly. Yet despite the massive investment in software engineering research, many fundamental goals have eluded us. For instance, software component reuse has not yet been achieved to the same extent as in other engineering disciplines; object-oriented development has failed to meet its initial claims; researchers still cannot agree on a definition of software architecture; and, finally, software developers still do not enjoy the benefits of rigorous mathematical analysis that other engineering disciplines do. Software engineering research has proposed many solutions in those particular areas, however, none of them have been universally accepted. Moreover, until very recently, there has been little, if any, effort made to understand the philosophical basis of those issues or to assess whether our expectations are reasonable. The assumption has been that solutions are possible because analogous approaches are presumed to exist in other engineering disciplines and all that is required is more research.

This thesis contributes to that debate by developing an understanding of software engineering based on philosophical and psychological foundations which can be used to analyse the contradictions between the practice and evolution of software development and the theoretical expectations of software engineering. The premise of this thesis is that

software engineering theories have been dominated by the guiding assumption that software system development can be understood as being analogous to the development of traditionally engineered artefacts and systems. That is, software engineers have an artefact engineering view of software development. While that assumption has enabled software engineering research to make a significant contribution to development practices, a thorough analysis reveals it leads to numerous anomalies. By examining the differences between software systems and traditionally engineered artefacts, a better understanding of the fundamental nature of software systems has been achieved and that can be used to improve software engineering research. The aim of the thesis is to convince software engineering researchers of the potential benefits of moving from an artefact engineering view of software development to a model building view.

The quote by Einstein that begins this thesis captures both how the ideas within were discovered and how the resulting conclusions should be interpreted. Because the content of the thesis is philosophical in nature, it is necessary to explain how it came about. When my Ph.D. research began, I had just completed an undergraduate degree comprising training in both computer science and electronic engineering. My initial research topic was to develop a graphical approach to what is now referred to as component-based software engineering. The project was part of the Amdahl Intelligent Tools Programme (AAITP), a collaborative research effort between Amdahl Australia and La Trobe University (see (Cybulski and Reed 1992)). Within the project group, my supervisor and project Director, Karl Reed, and I, were definitely of the view that software development could be understood as analogous to traditional engineering development. In contrast, other project members, specifically the deputy director, Jacob Cybulski, and another Ph.D. student, David Cleary, were of the opinion that software development was significantly unique that the analogies could not be easily applied. That led to some lively debates between the group members and allowed a close examination of the issues. As my research progressed, I examined contemporary theories in software reuse, design methods, graphical representations, visual languages, software architecture, and research in traditional engineering design. It was the analysis of theories in software architecture that initiated the change in research direction – specifically, reading Mary Shaw's 1994 technical report, *Making Choices: A Comparison of Styles for Software Architecture* (Shaw 1994) (later published as (Shaw 1995c)). In that report, Shaw compared and

analysed eleven different software designs for automotive cruise control systems. While discussing many issues about that report with Reed, he suggested that an interesting exercise would be to compare the software designs with published engineering designs of automotive cruise control systems. Even if it failed to provide a definitive answer to the "should software development be like traditional engineering?" debate, it was something that had never been done before and would be a worthwhile research contribution. Already 'knowing' that software development was like engineering development, I suggested the comparison would be pointless and that perhaps he didn't know what he was talking about. Nevertheless, I eventually went ahead with it. The result was a comprehensive study of twenty two software design examples and fourteen traditional engineering examples of the cruise control problem, which culminated in a series of questions that would lead to a profound change in my understanding of software systems and software development. Those questions led me from the specific software engineering topic of graphical component-based design to a more encompassing theory of the philosophical understanding of the discipline as a whole.

The thesis begins with that comparison of software designs and engineering designs for automotive cruise control systems. A selection of designs from each discipline is presented in detail to exemplify the approaches used and to capture the design rationale. The descriptions include the design steps taken, the models used, the representations depicted, and the analysis and design techniques utilised. Because so much published and informal conjecture exists concerning the relationship between the disciplines, a significant amount of detail is included to counter any preconceived biases. That detail includes a brief introduction to control theory to explain the engineering designs. From the analyses, a generic design approach for each discipline is developed which accounts for the similarities and differences noted between the respective disciplines. Those generic design approaches are summarised as follows. When the software developers approached the problem they each began with a pre-selected design formalism, either a particular design method or software architecture style, which served as the basis for performing the system analysis. That analysis resulted in an initial representation of the system, that is, a conceptual model or system architecture. As the designs proceeded, the components of the initial representation were refined and the model was augmented with additional models of the system until all aspects of the required functionality, behaviour,

and structure were depicted. From that point, the system designs could be analysed for various design, implementation, and testing properties.

The traditional engineering examples also exhibited a common approach to the design process. The engineers all began with a similar feedback control architecture, which is a well-known structural arrangement in that discipline, and which consisted of generic engineering components that are also well-known to the practitioners of the discipline. The required behaviour of each of the components in that architecture was then specified using mathematical models. The design process proceeded by using a combination of mathematical and experimental techniques to solve the unknown parameters of those mathematical models of the generic component behaviour. Finally, system testing ensured the implemented system performed as predicted by the models.

The generic approaches of the two disciplines were then compared. The most striking aspect of the comparison was the immense difference between the design approaches of the respective disciplines – especially for someone whose pedagogic training had led to a predisposition to thinking about software engineering in terms of analogies with traditional engineering. A staunch advocate of the artefact engineering view of software development would argue, "Of course they are different! The purpose of software engineering research is to make software development more like the engineering development approach". However, it has been almost 25 years since the goal of engineering software had been set and the software design examples were still doing something completely different to 'engineering'.

Rather than blindly accept that argument I tried to determine why they were so different. Perhaps the reason why the design approaches were so different was due to fundamental differences between what was being built? Perhaps those differences, which have never been systematically examined, invalidate the artefact engineering view of software development? Therefore, subsequent research was performed to investigate and identify significant differences between engineering and software components; investigate how those components evolved; and determine how designers think about them during the development process. For example, traditional engineers can only construct their systems using the building blocks of their discipline. Those building blocks are aggregations of functionality determined by the underlying physical properties upon which the discipline is founded. As the design proceeds, both the problem requirements and the components of

the proposed solution space are modelled, mathematically, to allow the comparison and evaluation of common properties. That allows the designer to predict the ability of the proposed solution to satisfy the given requirements.

Conversely, software developers work with an implementation medium that enables them to implement almost any concept. The initial system representation depicts the concepts and relationships in the designer's perception of how the solution should work. That model can take any form and the concepts within the model can come from many different views of the problem domain and many levels of generality of those views. The components and relationships in that model are then decomposed, refined, and finally implemented in code. The remainder of the cruise control comparison explores those ideas and develops the following conjecture to explain the differences between the respective disciplines.

> **The Model Building Conjecture:** Traditional engineers design corporeal artefacts to solve some real-world problem. In contrast, software developers solve a problem by implementing a model of some real-world phenomena.

More precisely, traditional engineers utilise the properties of physical components and materials to build corporeal artefacts that solve real-world problems, mathematically modelling the underlying functional properties of the materials and components of their discipline during the design process. In contrast, software developers solve real-world problems by developing and then implementing models of reality that explain and automate a perceived process. That model must be conceived by the developer and implemented using the constructs provided by the software language(s) being utilised and the virtual machine(s) used to execute the resulting implementation.

From that study, my research agenda changed significantly. I could not continue to develop ideas based on an artefact engineering view of software development when that view could be completely inappropriate. Based on questions posed at the conclusion to the cruise control study, the goal became an exploration of the fundamental nature of what software developers build and that resulted in a model building view of software development. Those questions include:

- Why can engineering components be modelled mathematically and can that be achieved in software?

- What does it mean to build explanatory theories of reality?
- Can those model building issues explain areas of contention in contemporary software engineering research?
- And if the design approaches are so different, why have analogies with other disciplines persisted in being used as the inspiration and justification of research ideas for so long?

Those questions form the basis of the rest of the thesis and the last question serves as the starting point for the third chapter. It presents an examination of the history of engineering analogies in software engineering research and an analysis of their validity.

That history begins with the NATO conferences on software engineering from the late 1960s (NATO 1976a; NATO 1976b). Although the idea that software development could be understood as analogous to traditional engineering development may have occurred to practitioners before that time, the NATO conferences saw the first formal expression of that goal and it serves as a useful starting point for analysis. The published transcripts and selected reports from those conferences provide an invaluable historical record from which the understanding of software development that researchers and practitioners had at that time can be explored. While the transcripts are no substitute for actually being there, they capture significantly more debate than is provided in present-day conference proceedings. Indeed, substantial quotations have been included so that the precise flavour of the arguments and debates is clear.

The label 'software engineering' was suggested by the NATO science committee as a starting point for generating ideas for improving the software development process. The perception was that traditional engineering development could provide useful insights for software development researchers. The 1968 conference report details many comments, insights, analogies, and conjectures that were used for and against the appropriateness of 'software engineering' and they are presented in detail in the chapter. Importantly, they show there was no clear consensus concerning the appropriateness of the engineering metaphor. However, by the 1969 conference a significant change in emphasis was evident in the transcripts and reports. The belief that software development could be understood as analogous to engineering development appears to have been accepted despite the recognition of many anomalies generated by the assumption. Subsequent debate turned to

the application of those analogies to the specifics of software development. Although the artefact engineering view of software development had not been substantiated by detailed analysis, it became the dominant view of software engineering research.

The investigation then examines the way in which this view was perceived in subsequent years. A selection of publications are presented and analysed. They range from McIlroy's *Mass Produced Software Components* paper from the 1968 NATO conference through to contemporary publications on component-based software engineering. The examination attempts to determine the understanding of software engineering researchers by presenting the arguments and analogies they use and analysing them in detail to determine their validity. What becomes clear is that software engineering researchers did not have a thorough understanding of the fundamental nature of software or software development, nor did they have a thorough understanding of the engineering approaches they were using as the source of software engineering ideas. This is not meant as a criticism of those researchers, indeed much of their research has led to significant progress in the discipline. However, with the benefit of hindsight, numerous anomalies become evident that require explanation.

One anomaly that repeatedly appeared, yet was never satisfactorily explained, concerned the underlying principles of software systems. Analogies had highlighted the importance of the underlying principles of engineering disciplines for developing the science required to support the mathematical treatment of engineering design. However, the nature of analogous principles in software development never became clear. Researchers concluded that to become an engineering discipline software engineering research needed to identify those principles and to develop an appropriate 'software science'. However, they did not question whether their identification would invalidate the goal of software engineering – that assumption was firmly entrenched. Research had to ensure that a 'software science' was achieved somehow. Formal methods are a good example. Software engineering researchers have developed formal methods to apply rigorous mathematical techniques to software development. Moreover, arguments have used the following syllogism to justify those methods in software engineering: *Engineers use mathematical techniques. Formal methods allow the mathematical treatment of software development. Therefore, the use of formal methods can lead to software engineering.* However, the study shows the role of

mathematics in the design approaches of the respective disciplines is quite different, and therefore, the reasoning is false.

The analysis of the engineering analogies used by researchers and the elicitation of their understanding of software engineering highlighted many issues for further investigation. Specifically, a number of additional anomalies were identified that may be causing misunderstanding between the theory and practice of software engineering. Therefore, before examining the underlying principles of software systems, the impact of those anomalies were examined in the specific context of software architecture research. That served as the basis for the fourth chapter.

The chapter on software architecture begins with a case study of the design, implementation, and maintenance of one of the applications developed as part of the AAITP project. That system, HyperEdit, is a graphical editor generator system that can be used to graphically create the diagram editors used to construct software diagrams. Its development occurred over a number of years with different designers and maintainers, and provided practical evidence of how designers use architecture principles in the development process. The goal of the chapter is to identify the large-scale structures identified during the development and maintenance processes, identify the issues that affected the decision making process at the architecture level of design, and to compare those with existing theory.

The case study identified many different factors that influenced the architecture decision making process of the project group that are not adequately accounted for in conventional software architecture theory. They included the influence of changing requirements on the architecture, how knowledge of available architectures and their practical consequences affected decisions, and how architectures were developed to take advantage of pre-existing knowledge of structures in the implementation environment. Those issues are discussed in detail, however significant analysis is reserved for a discussion of the different large-scale system representations identified during the design and maintenance process and how they relate to existing theory.

Many different large-scale representations of the HyperEdit system were identified. Conventional theory claims those different representations are analogous to the different architecture views that exist in the development of traditionally built artefacts. In those disciplines the different views provide graphical subsets of the complex, underlying

implementation detail. However, analysis of the identified software architectures shows they are not all subsets of the same underlying implementation detail. A review of the history of ideas in software architecture research shows our understanding of architecture views are based on comparisons with traditionally built artefacts. In those disciplines, systems exhibit an immutable architecture, set early in the design process, which is visibly recognisable in the physically implemented form. However, differences that exist between software systems and traditionally built artefacts serve to undermine the idea that the two notions of 'architecture' are sufficiently analogous for concepts from one to be automatically applicable in the other. Those differences concern the notion of system form and the fact that to realise the required software system a computer must execute its implementation. The conclusion is that the different large-scale software representations are not analogous to the different architecture views of built artefacts. Rather, their explanation is somehow related to the unique, fundamental nature of software systems. To determine how they should be understood it is necessary to identify and explore the underlying principles of software systems. That is the topic of the fifth chapter of the thesis.

The underlying principles of software systems are related, somehow, to concepts, models, abstractions, theories, and how they are used by the human mind to understand reality and solve problems. Those issues have been explored by other disciplines for many years and their theories serve as a starting point for uncovering the foundations of software engineering. Philosophy, especially in the fields of epistemology and metaphysics, has a long history of identifying the concepts that constitute reality and how they are represented in knowledge. Additionally, psychologists, especially in the fields of conceptual development and cognition, have devised experiments and theories to explain how concepts are used to capture reality, how they are devised, and how they evolve. Finally, theories in the history and philosophy of science explain how models and theories are used to explain the world, how those theories can be verified, and how they evolve over time.

Unfortunately, these disciplines do not offer ready-made explanations of the underlying principles of software engineering. Nevertheless, different theories from these fields have been cited in software engineering research as justification for proposed ideas. To ensure this treatise does not simply adopt one of the many different philosophical and

psychological positions to support a presupposed understanding of software engineering, it presents, in detail, the major theories from those disciplines that are related to the underlying principles of software systems. An attempt to compress two thousand years of thought into a handful of pages is, perhaps, over ambitious. However, without this material, the basis for the conclusions is unlikely to be clear. In addition, this is the path that was trodden to reach the final point, and the reader is entitled to see the chain of reasoning that occurred.

The conclusion of that presentation is the identification of two main phases of thinking about the underlying issues. The classical way of understanding concepts and theories dates back to the philosophies of Plato and Aristotle and begins with the assumption that people can be considered as separate from their environment and that all things can be classified in terms of essential attributes. That assumption results in a belief that all people observe the same objective reality and our concepts are derived by inferring abstractions from that reality. As people operate in the world, they associate objects with known concepts by identifying the essential attributes. Furthermore, because reality is objective, theories used to explain phenomena capture the natural order of the world.

However, as progress occurred in both philosophy and psychology, a different way of thinking about the issues emerged. Many philosophical arguments and psychological experiments highlighted anomalies in the classical way of understanding. Subsequent research showed that people's conception of reality cannot be considered as separate from some objective reality. As people interact with the world, they automatically and subconsciously apply their accrued concepts and theories to the observed phenomena in order to understand it. Consequently, people's explanatory theories do not capture the natural order of the world. Rather, they are subjective to the person using them and different theories cannot be evaluated as being better or worse depictions of reality. Each can only be evaluated in terms of the usefulness it provides a person in their attempt to understand and operate in the world. An additional contradiction to the classical way of understanding concerns the definition of concepts. Experiments and dialectic debate have shown that people do not classify phenomena into different classes of concepts based on the existence of essential attributes. Instead, concepts are defined in terms of the roles the play within people's explanatory theories of the world. Although those conclusions contradict the classical way of understanding, researchers note that the classical way of

understanding still pervades the philosophical assumptions of people who have not studied the relevant philosophical and psychological research. That is also evident in the justification of the artefact engineering view of software development.

The chapter continues by applying those philosophical foundations to our understanding of the software development process. It does not provide a comprehensive explanation in terms of those foundations – that would require another thesis. The aim is to show the applicability of those foundations to our understanding to software engineering and to advocate its use as the basis for future research. The conclusion is that software engineering researchers should explicitly consider the potential benefits of moving from an artefact engineering view of software development to a model building view.

Finally, the thesis argues that these foundations can be used to improve software engineering research by providing a basis with which to evaluate and justify research ideas. However, to develop that ability, researchers must become aware of how research in general is evaluated and justified. The history and philosophy of science has devised many theories to explain the role of research in the progress of a discipline. While philosophers hold different opinions about many of the issues, some consensus has emerged. As a discipline performs research, guiding assumptions dictate how researchers understand the phenomena under investigation, even though those researchers might not be explicitly aware of them. Those guiding assumptions change as a discipline evolves, and radically new ways of understanding the same phenomena occur. That new way of understanding is difficult to evaluate at first but is soon used to explain problems and anomalies with existing theories and provides the potential for significant progress in the discipline. The contention of this thesis is that an artefact engineering view of software development has provided the dominant set of guiding assumptions for software engineering research. However, an alternative model building view offers the potential to explain existing problems and anomalies and can provide new insights for improvement.

Unfortunately, changes in guiding assumptions do not occur smoothly. Proponents of the existing way of understanding may refuse to see the benefits of an alternative approach, despite the anomalies present in the original. Moreover, it is not always easy to compare two sets of guiding assumptions. Some researchers have already presented research based on a model building view of software engineering, although their theories are rarely cited

in mainstream research. Aspects of their work are presented – specifically the theories of Peter Naur and Bruce Blum.

This thesis does not claim that the artefact engineering view should be completely replaced by a model building view. Not enough research has been performed to illustrate the benefits of this new way of understanding to convince the mainstream research community of the benefits of such a complete change. However, the goal has been to provide enough detail to make the issues explicit for other researchers and to convince them that it is worthy of more detailed investigation. Moreover, research based on an alternate way of understanding should not be dismissed simply because it is different. To facilitate research based on this different way of understanding, the concluding chapter includes a number of conjectures about how the model building view of understanding can provide explanatory theories for software reuse, patterns, and architecture.

This thesis has been the result of the author's research into a number of areas. It has been an adventure, a journey into parts unknown, and into regions I never expected to visit when my task began. My view is that the reader will have a better understanding of the results if the journey is laid out, and they can experience it in part for themselves, travelling with me, as it were.

The thesis examines the way in which software engineering researchers understand the phenomena they investigate and the history of that "understanding". As a consequence, many of the ideas presented are philosophical in nature. These ideas are rarely discussed explicitly in the research literature and, in practice, debating such issues can be very difficult. Moreover, a succinct linear presentation is difficult to achieve. The decision to present the issues in roughly the same order as my own progression through them results in a spiral-like presentation in which some of the discussion may appear to be repetitious. However, what we have is merely the recurrence of themes that are presented in successively greater detail as the narrative develops. It is hoped therefore, that the style of presentation will assist in the clarification of what are, after all, complex and contentious issues.

Before entering the thesis, let me remind the reader of Einstein's comment on the nature of research and plant a seed to help interpret what is to be encountered. The audience of this work is software engineering researchers, and the reason we do research is because we do not know exactly what it is we are doing. What is required is a continual

questioning of our guiding assumptions, especially the implicit ones that may not be immediate obvious, and a commitment to remain open-minded to radically different ways of understanding the same phenomena we may have been studying for a very long time.

2. Is Software Development Analogous to Traditional Engineering? A Comparison Of Designs for Automotive Cruise Control

2.1 Introduction

The idea that the discipline of software development could become an engineering discipline received its first formal expression at the NATO Conference on Software Engineering in 1968 (NATO 1976a). The motivation was that software development was perceived to be unstructured and unpredictable while engineering was perceived to be disciplined, predictable, and well structured. Over the years, this goal has been questioned by some, while others, committed to the goal, have argued that we still have a long way to go. Part of the problem is that the exact relationship between software development and traditional engineering disciplines is not fully understood. This study seeks to clarify the issue by comparing the design approaches of software developers and traditional engineers when confronted with the same problem – automotive cruise control. The detailed study of multiple, published design examples from both disciplines shows that while analogies between the disciplines appear to be valid, there exists significant differences between the design approaches taken by the respective disciplines. Traditional engineers build corporeal artefacts to solve real-world problems, mathematically modelling the materials and components of the discipline during the design process. In contrast, software developers solve real-world problems by implementing models of reality that explain a perceived process. This difference between the fundamental natures of the systems developed by the respective disciplines is caused by differences between the implementation mediums used by them.

The study consists of a number of stages. After analysing the cruise control requirements from both disciplines to ensure the study is comparing designs of a similar problem, a selection of specific design examples from both disciplines is presented. The design examples detail the steps taken by each designer by stating the models developed, the techniques utilised, and by reproducing significant design diagrams. To present a comparison free of any preconceived biases towards similarities between the disciplines, a considerable amount of detail is included. In addition, because some readers may be unfamiliar with the design concepts and techniques used in the engineering examples,

they are preceded by a brief discussion of control system theory and design techniques that are relevant to the cruise control problem.

Generic design approaches are then developed for each discipline and they are compared to identify the similarities and differences between the disciplines. That comparison results in the model-building conjecture to explain the differences between the design approaches. That conjecture is then analysed to determine why the differences and similarities exist.

The conclusion recommends further avenues of research that have been made apparent by the results of the comparison. Finally, a number of comments are made about the discipline of software engineering as a whole.

An initial comparison of the cruise control designs was published previously (Baragry 1996). This chapter provides considerably expanded detail in terms of the presented designs, the analysis and comparison of those designs, and the implications for software engineering research than in the paper in which the study was first reported[1].

2.2 The Cruise Control Requirements

A cruise control system is a device for keeping an automobile at a constant speed over varying terrain. Although those requirements are relatively simple, it is necessary to detail their expression by both disciplines to ensure the problem is an appropriate one for a comparison of design approaches. While the scope of this study does not include the formulation of requirements, it does makes several observations that relate to the differences that exist.

The software designs all contained a basic set of cruise control requirements.

- The cruise control system can only operate when the engine is running.
- When the driver activates the system the vehicle speed should remain at the current value until the system is interrupted or is de-activated.
- Pressing the brake pedal suspends the system operation until the driver resumes cruising.

[1] This study has been submitted for publication in IEEE Transactions on Software Engineering.

- When the driver resumes operation, the system should bring the vehicle speed to the previously set value.

The specific requirements of individual examples differed slightly with some specifing a small amount of additional functionality. The following requirements, specified by Booch (Booch 1986) as a collection of system inputs and outputs, formed the basis of many of the examples.

- System on/off: If on, denotes that the cruise control system should maintain the speed of the car.
- Engine on/off: If on, denotes that the car engine is turned on; the cruise-control system is only active if the engine is on.

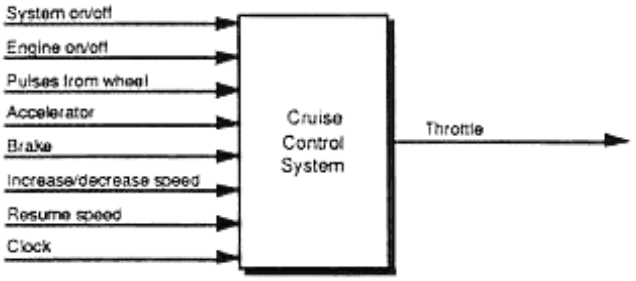


Figure 2-1: Booch Requirements.

- Pulses from the wheel: A pulse is sent for every revolution of the wheel.
- Accelerator: Indication of how far the accelerator has been pressed.
- Brake: On when the brake is pressed; the cruise-control system temporarily reverts to manual control if the brake is pressed.
- Increase/Decrease speed: Increase or decrease the maintained speed; only applicable if the cruise-control system is on.
- Resume: Resume the last maintained speed; only applicable if the cruise control system is on.
- Clock: Timing pulse every millisecond.
- Throttle: Digital value for the engine throttle setting.

Some of the software designs based their system requirements on those presented in a published example by Brackett (Brackett 1987). That paper could not be located in the literature, however the details are evident in the work of others – e.g., (Smith and Gerhart 1988). Brackett's core requirements were the same as those presented by Booch with additional functionality specified to provide auxiliary monitoring functions such as monitoring the average speed, fuel economy, and maintenance requirements of the vehicle.

The hardware design examples also had a common set of core requirements for the cruise control system that were the same as those presented for the software designs. They included the ability to engage/disengage the system, the ability to set the current speed as the cruising speed, the ability to temporarily disengage the system when the brake pedal is pressed, and the ability to resume the last recorded cruising speed (see (Shaout and Jarrah 1997)). However, in contrast to the software examples, the requirements were often stated in an 'engineering language'. For example Rutland's design requirements were stated as:

> "The primary objectives of an automatic vehicle control system is to maintain a constant vehicle speed and acceptable ride comfort, for the set of all possible load forces. A particular load force, possible or fictitious, is said to be tolerable if it keeps the speed, and comfort levels, within predetermined margins. … The only input [external to the control system] to be considered is a load force due to the changing gradient of the road profile and the varying wind speed. The important outputs are the error from the required speed and the outputs relating to a comfortable ride." (Rutland 1992)

While the core requirements of the engineering examples remained the same, additional functional and performance requirements were introduced as cruise control technology evolved. Functionally, these included the ability to slightly increase or decrease the set cruising speed, which is similar to the software design requirements. However, from a performance perspective, those additional requirements included "smooth and minimal throttle movement", "universality: the same control module must meet the performance requirements for different vehicle lines without recalibration", and "simplicity: design concepts and diagrams should be understandable to automotive engineers with basic control [theory] background." (Liubakka, Rhode et al. 1994). Figure 2-2 depicts Muller's representation of the requirements for the 'man-machine interface' (Muller and Nocker 1994).

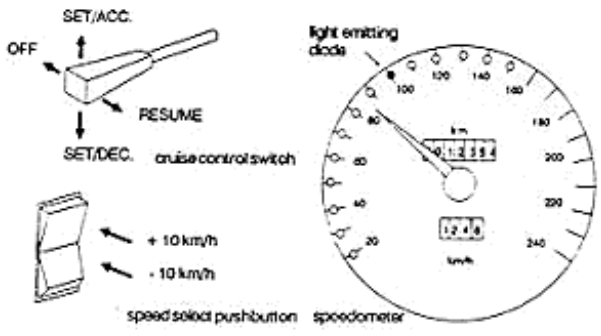


Figure 2-2: Man-Machine Interface.

More recently, the desire to alleviate congestion on urban highways has led to the development of autonomous intelligent cruise control systems (AICC). These are

designed to provide automatic vehicle following where "the throttle and brake are controlled by the computer, and only steering is under manual control. The sensor on board of the vehicle senses relative distance and velocity of the immediate vehicle in front, and the computer control system sends the appropriate commands to the throttle and brake" (Ioannou and Chen 1993). The requirements of AICC systems encompass those of traditional cruise control systems and provide valid examples for the design comparison.

Traditional cruise control systems have been commercially available since the late 1950s and AICC systems have recently been introduced in commercial vehicles by, for example, Mercedes Benz (Shaout and Jarrah 1997).

Despite slight variations in the requirements, both between and within the respective disciplines, it can be seen that a common, core set of requirements exists for all design examples. The variations that exist do not undermine the ability to develop a useful comparison of the design approaches and any design reasoning that is influenced by those variations is detailed in the comparison.

2.3 The Software Designs

The problem of automotive cruise control has been used numerous times as a design example in software engineering research literature – both for detailing and comparing the use of proposed design methods or techniques. Twenty-two examples were located for use in this study, though that list is not exhaustive. In addition, Shaw used many of the designs in a comparison of software architecture styles (Shaw 1994; Shaw 1995c). Indeed, part of the impetus for this study came from reading that comparison.

This study uses the examples to generate a generic software design approach that is subsequently compared with the design approach of traditional engineering development. The software designs are loosely categorised in terms of the methods or techniques they are used to exemplify.

- Object-Oriented (OO) Design: Booch (Booch 1986) used the cruise control problem to present his initial approach to object-oriented design. Birchenough and Cameron (Birchenough and Cameron 1989) used it to exemplify the application of Jackson System Development (JSD) to software design. Yin and Tanik (Yin and Tanik 1991) used it in their exploration of reusability in Ada. Wasserman et al

(Wasserman, Pircher et al. 1989) presented an OO extension to structured analysis and design using cruise control to highlight their architectural design method. Gomaa (Gomaa 1989; Gomaa 1993) used cruise control to analyse different real-time system design methods. His study examined a Booch approach and a JSD approach to OO design. He also analysed the Naval Research Laboratory software cost reduction method (NRL/SCR) using the same problem and used it to present his ADARTS approach, which extends DARTS for Ada systems by using an information-hiding structuring step. Finally, Appelbe and Abowd (Appelbe and Abowd 1995) respond to Shaw's comparison of her process control model of cruise control with Booch's OO design by presenting an updated version based on a more recent approach to OO design by Booch.

- State or Control Based Design: Smith and Gerhart (Smith and Gerhart 1988) illustrated the application of a functional decomposition approach to the Statemate interactive development environment (Harel et al 1990). While the approach employed functional decomposition, the emphasis on state-based design allows it to grouped in this category. Atlee and Gannon (Atlee and Gannon 1993) investigated the safety of a cruise control design by model-checking an event-based representation of the problem.

- Process Control Feedback Loops: Shaw (Shaw 1995b) represented the cruise control problem as a process control architecture using the design technique of feedback control from traditional engineering disciplines. Higgins (Higgins 1987) used a similar approach to his feedback control model resulting, however, in a different representation. Jones (Jones 1994) produced a distinctive design example, again using feedback control, by claiming to design a system from a traditional engineer's mindset rather than a software developer's one.

- Real Time Structured Analysis and Design (RTSAD): Ward and Keskar (Ward and Keskar 1987) used the problem as an example to compare the Ward/Mellor and Boeing/Hatley real-time extensions to DeMarco structured analysis and design. Gomaa (Gomaa 1993) presented an example of an RTSAD cruise control system and compared it with his design approach for real-time systems (DARTS). It extends RTSAD by partitioning the design into concurrent tasks. Booch (Booch 1986) also produced a design based on traditional structured analysis and design

to compare with his OO design. However, it does not include any real-time analysis.

- Concurrent Object-Oriented Design: Four published examples depicted the cruise control problem when decomposed and implemented using concurrent, object-oriented system methodologies. Saksena et al (Saksena, Freedman et al. 1997) presented the ROOM method (real-time object-oriented modelling). Gomaa (Gomaa 1993) presented the CODARTS method (concurrent design approach for real-time systems). Awad et al (Awad, Kuusela et al. 1996) presented the Octopus method. Lastly, Caromel (Caromel 1993) presented his own approach to concurrent, OO design.

Those publications provide examples of cruise control design produced using different design methodologies, using different styles of the same design methodology, and by different designers using the exact same design method. A single example from each category is chosen to illustrate the way in which software developers actually perform 'design'. The descriptions contain a brief account of the general approach and concentrate on the concepts elicited by the designers to model the problem and solution. Those concepts are most apparent in their graphical representation of the designs. The similarities and differences between the various software approaches are then highlighted and a generic design strategy is identified. Finally, it is worth noting that none of the software designs describes a commercially available system.

2.3.1 Object-Oriented Design

Booch (Booch 1986) used the design of a cruise control system to illustrate the differences between object-oriented and traditional functional decomposition development techniques. Initially, both techniques require the identification or creation of a model of the problem space. Booch developed a data flow diagram (DFD) from the requirements (figure 2-3) that acts as a starting point for both designs. The object-oriented design is created by extracting the objects of the system from the DFD and depicting the dependencies between them (figure 2-4). Booch provides very simple guidelines for how objects were extracted from the DFD,

> "Typically, the objects we identify in this step derived from the nouns we use in describing the problem space." (Booch 1986)

However, this is a very early example of OO design. More recent analysis techniques, including Booch's own (Booch 1991), provide more detailed rules and guidelines for identifying objects in the problem space.

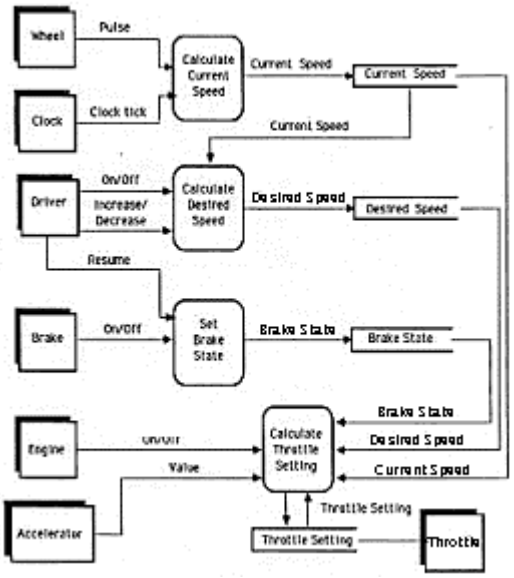


Figure 2-3: Booch DFD

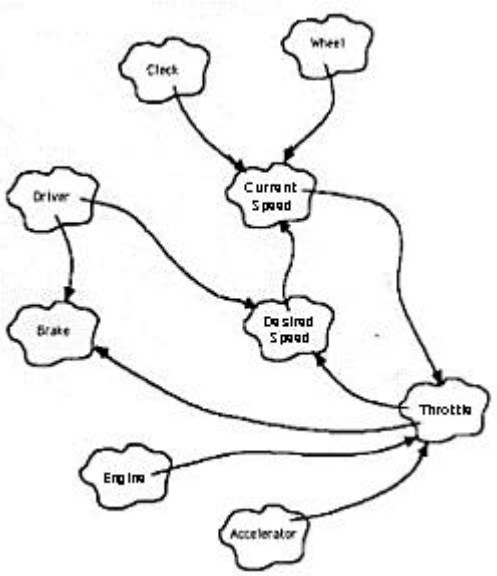


Figure 2-4: Booch Object Model

The implementation of the design proceeds by identifying the attributes of each object, the methods performed and suffered by it, establishing their visibility with respect to each other, establishing their interfaces, and finally, realising them in code.

Booch notes that due to the autonomous nature of objects it is not possible to identify a central thread of control from the object diagram. While it was not explicitly developed in Booch's example, complex object diagrams need to be complemented with a representation that denotes the dynamic interaction between them. State transition diagrams are often used in OO design methods, including Booch's more recent technique, to achieve that purpose.

2.3.2 State Based Design

Smith and Gerhart (Smith and Gerhart 1988) used the problem to analyse the effectiveness of the Statemate development environment (Harel et al 1990). Statemate uses activity charts to represent the functional properties of the system and statecharts to represent the controlling mechanism between those activity charts. A further representation, module charts, depict how those state and activity charts are realised in implemented components. Different design methodologies can be utilised with Statemate depending on the order and emphasis used when creating the respective charts. In their

example, Smith and Gerhart used a functional decomposition method, based on Brackett's formulation of the problem statement, and compared their results to a traditional, structured-analysis and design approach to the same problem.

The design begins by modelling the requirements as three separate types of activities to be monitored by the system:

1. Standard driving events (e.g.: braking and acceleration).
2. Cruise control actions (e.g.: activate, inactivate, etc).
3. Monitor features (e.g.: speed calculation, etc).

These lead directly to the top-level activity chart (figure 2-5). The controlling statechart is then developed for the top-level activity chart. The design continues by recursively developing activity charts and the corresponding controlling statecharts for the activities in the highest-level activity chart. Figures 2-6 & 2-7 show the activity chart and statechart for the 'Control Speed' activity from figure 2-5.

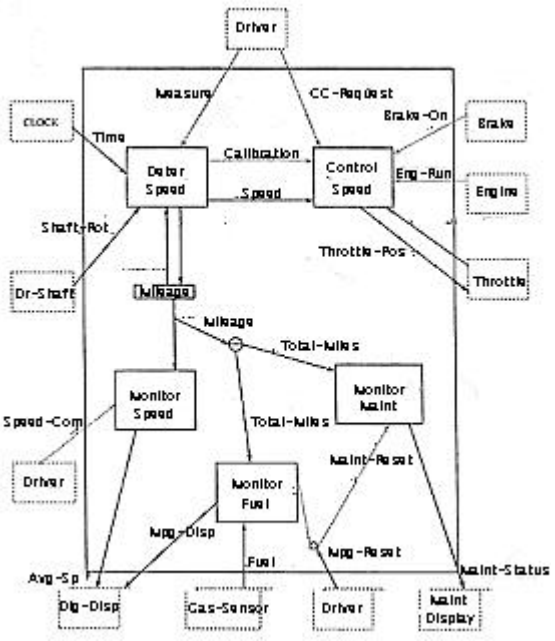


Figure 2-5: Top-Level Activity Chart

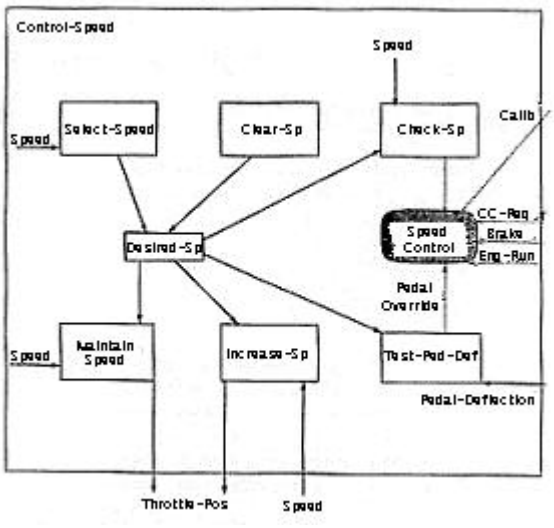


Figure 2-6: Control Speed Activity Chart

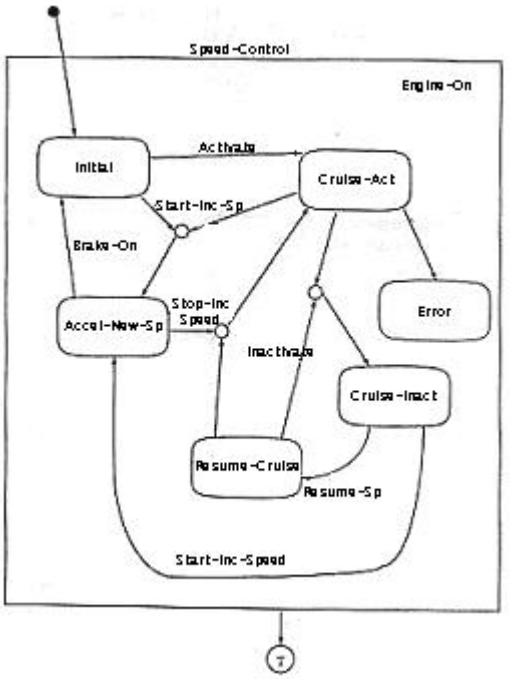


Figure 2-7: Control Speed State Chart

2.3.3 Process Control Feedback Loops

Traditionally, software design approaches have begun by eliciting the important functions, objects, states, or events from the system requirements or problem domain. In contrast, Shaw (Shaw 1995b) began by recognising the cruise control problem as a specialisation of the generic control problem. Consequently, the high level requirements could be satisfied using the process control feedback model, which is a well-known approach to design in conventional engineering disciplines (e.g., (Stefani, Clement J. Savant et al. 1994)). By comparing the current output with a predetermined reference point, the generic feedback model modifies its operation to maintain a stable output to a system when the environment or the system input may be unstable.

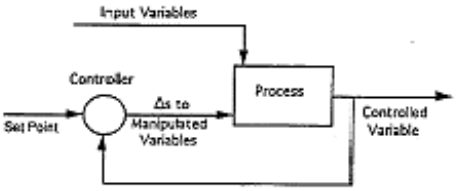


Figure 2-8: Generic Feedback Model

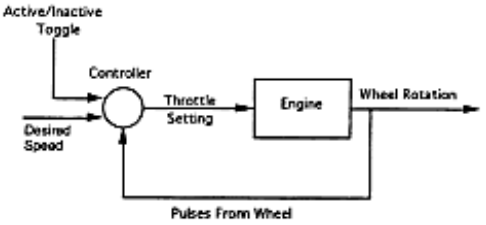


Figure 2-9: Specific Feedback Model

Starting with the generic feedback system architecture (figure 2-8), Shaw matched the entities of the cruise control requirements with the concepts of the generic feedback model – e.g., 'set speed' to 'set point', 'throttle setting' to 'manipulated variable', and 'current speed' to 'controlled variable' (figure 2-9). Shaw then used a state transition technique to represent the system's activation and control (figure 2-10) and an event table technique to represent how the set point is determined. Those three sub-architectures were combined to realise the global system architecture (figure 2-11), which contains all the functional and behavioural information necessary to proceed with the implementation.

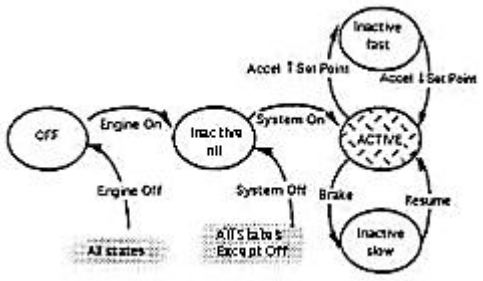


Figure 2-10: Feedback STD

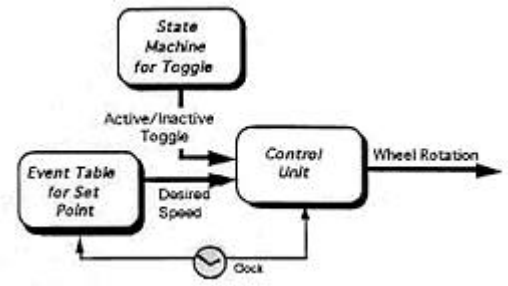


Figure 2-11: Global Architecture

2.3.4 Real Time Structured Analysis and Design

Ward and Keskar (Ward and Keskar 1987) compared two methods used for real time system development – Ward/Mellor and Boeing/Hatley. Both approaches are extensions of DeMarco Structured Analysis. This section details the Ward/Mellor approach with aspects of the Boeing/Hatley approach used for comparison in a later section.

Their requirements for the cruise control system are similar to those of Booch and that description highlights the core functions that were subsequently used to derive the initial data-flow diagram.

> "Increase speed at a uniform rate by gradually opening the throttle; capture and store the instantaneous speed for subsequent use as the desired speed; compare the instantaneous speed with the desired speed and adjust the throttle to minimize the deviation." (Ward and Keskar 1987)

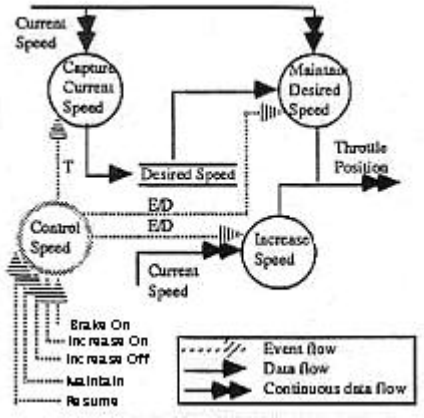


Figure 2-12: Ward/Mellor CTD

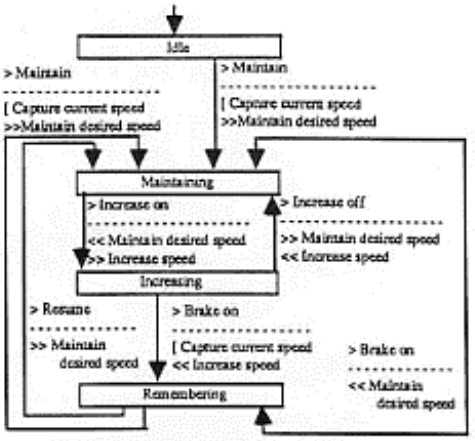


Figure 2-13: Ward/Mellor STD

Because the traditional structured analysis notations fail to capture time-dependant behaviour, the Ward/Mellor approach provides extensions to the basic DFD. A transformation schema extends the basic DFD with events and control processes. The control transformation diagram (figure 2-12) contains the information of the original DFD in conjunction with the control and event information (dotted lines). It depicts how the control process 'Control Speed' receives events from internal and external interfaces and how they enable, disable, or trigger the basic functions. A state transition diagram (figure 2-13) complements the control transformation diagram by describing its logical operation. Slight modifications to these two diagrams were performed by the designers to correct an anomaly in the logic discovered during analysis.

2.3.5 Concurrent Object-Oriented Design

Saksena et al (Saksena, Freedman et al. 1997) presented their experience in applying real-time scheduling theory to embedded control system designs. The particular design example is developed using his ROOM (real-time object-oriented modelling) methodology, though the authors note their guidelines could be applied to other methodologies such as UML (unified modeling language). The representation of the problem using the ROOM methodology provides the information required for this study.

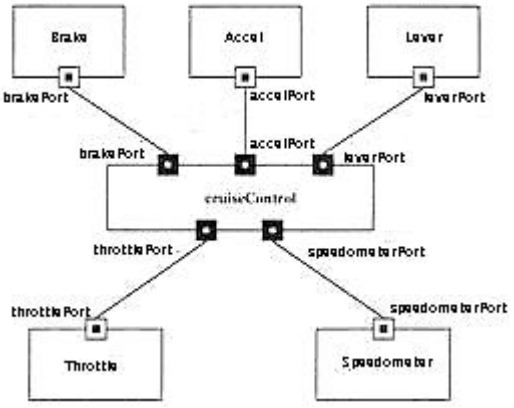


Figure 2-14: Actor Structure

The first step in the methodology was the identification of 'actors', which are encapsulated concurrent objects that communicate via point-to-point links. For the cruise control example, the actors encapsulate the input/output hardware interfaces and the cruise control logic (figure 2-14).

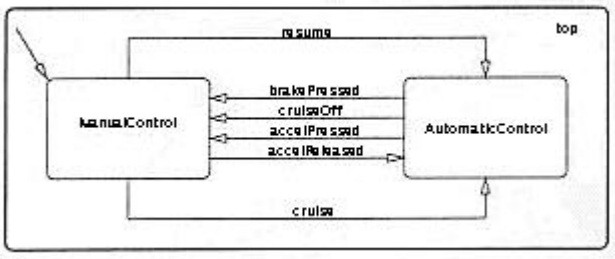


Figure 2-15: Top-Level ROOMchart

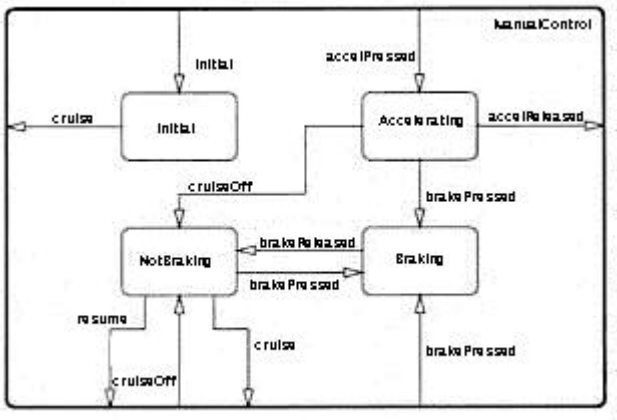


Figure 2-16: Manual Control ROOMchart

The designers then specified the behaviour of the actors using ROOMcharts, a formalism based on statecharts. The high-level ROOMchart consists of a ManualControl state and an AutomaticControl state with the appropriate transitions between them (figure 2-15). Those states were then decomposed using refinement ROOMcharts (figure 2-16 & 2-17). The real-time behaviour of the design was analysed by specifying the timing constraints of the identified transactions and representing them using message charts. Figure 2-18 depicts the initial message chart for the control loop transaction.

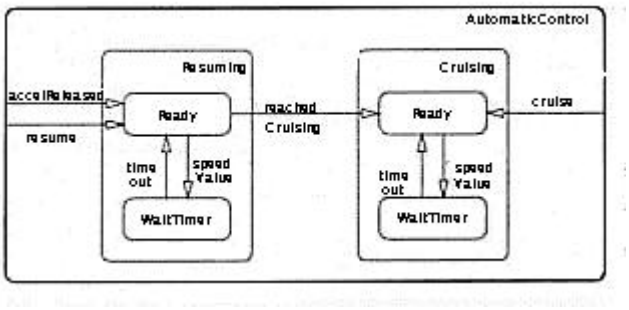


Figure 2-17: Automatic Control ROOMchart

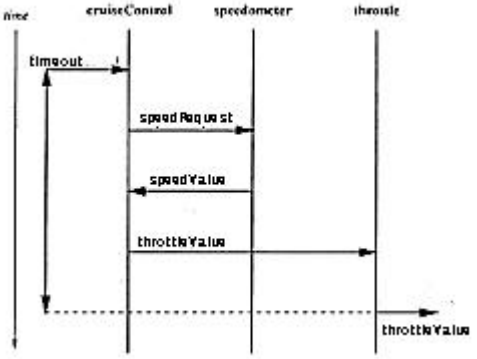


Figure 2-18: Top-Level Message Chart

The final step in the design was to map the actors onto the operating system abstractions that support concurrent functions. The two typical abstractions are operating system processes and separate threads of control within a single process. For models with high inter-actor communication, the designers recommend using individual threads for realising the actors.

2.4 The Hardware Designs

Engineers have investigated the problem of automotive cruise control since the 1950s. Fourteen examples of, or directly relating to, cruise control design were gathered from the engineering design literature (Takasaki and Fenton 1977; Nakamura, Ochiai et al. 1983; Koning 1984; Ellinger 1985; Oda, Takeuchi et al. 1991; Rutland 1991; Tsujii, Takeuchi et al. 1991; Rutland 1992; Ioannou, Xu et al. 1993; Ioannou and Chen 1993; Lee, Kim et al. 1993; Germann and Isermann 1994; Liubakka, Rhode et al. 1994; Muller and Nocker 1994; Shaout and Jarrah 1997). In addition, Shaout (Shaout and Jarrah 1997) provides a historical review of cruise control technology and Liubakka (Liubakka, Rhode et al. 1994) discusses different design strategies before the presentation of his own design. More examples are available, especially in the relatively new area of autonomous intelligent cruise control. However, for the purposes of this study, these examples provide a broad spectrum of design strategies that are detailed in an attempt to develop a generic engineering design approach.

The individual designs could not be categorised into distinct design methodologies as easily as the software design examples. Rather than being used to exemplify different design methodologies, the hardware examples used more advanced design strategies in an attempt to provide an improved solution to the problem. In that sense they are more

evolutionary in their development than the software designs. To gain an understanding of the evolutionary nature of the solutions and to provide data for developing a generic engineering approach to system design this section presents the processes used to design five of the examples. Those examples range from the earliest mechanical feedback implementations, to the introduction of microprocessor controlled solutions, to adaptive design strategies, and finally, to fuzzy logic and autonomous intelligent cruise control systems. The remaining examples are used in later sections to highlight issues that impact upon the comparison of software and traditional engineering design approaches. Unlike the software examples, at least four of the designs are actual cruise control products.

To provide an effective comparison of software and traditional engineering design approaches it is necessary to present sufficient detail of what engineers do to elicit how they approached the design process and, subsequently, why that approach was required. Because some software engineering researchers may be unfamiliar with the engineering techniques and terminology used the design examples are preceded by a brief introduction to control system design theory and analysis techniques.

2.4.1 Basic Control System Analysis and Design

Traditional engineering disciplines have a long history of design and implementation of control systems. This introduction serves to clarify what is happening in the cruise control design examples and, in the later comparison, why it happened that way. In control theory, the manipulation of electronic signals is represented in terms of *system transfer functions* and system designs must be *linear* and *stable* for both efficient design and satisfactory functioning. Those core concepts, plus the fundamental concepts of feedback control systems such as *proportional feedback*, *PID control* etc, are described. In addition, standard analysis techniques such as *Bode*, *Nyquist*, *root-locus*, etc. and some of the more specific design techniques used by the individual example are also described. The reader is referred to the following references from which the summary has been developed: (Gray and Meyer 1984; Smith 1984; D'Azzo and Houpis 1988; Horowitz and Hill 1989; Sedra and Smith 1991; Stefani, Clement J. Savant et al. 1994).

The functionality of an electronic circuit is nearly always described in terms of how the applied input, usually a voltage, is transformed into some corresponding output, which again is usually a voltage. Engineers speak of the *transfer function*, which is the ratio of the output signal to the applied input. For example, an audio amplifier might produce an

output voltage that is 100 times as large as the input voltage. In this case, the transfer function is simply a constant of 100.

Those signals are a time varying quantity that can be represented by a graph such as the one shown in figure 2-19. Using the technique of Fourier analysis, the waveforms can be represented as a sum of sine waves of different amplitudes and frequencies. That allows circuit analysis and design to be performed in terms of mathematical representations of sine wave input signals, which greatly simplifies the process (figure 2-20).

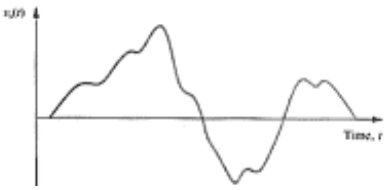
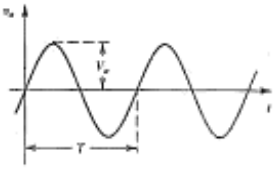


Figure 2-19: Time-varying input signal



$$u_a(t) = V_a \sin wt$$

V denotes the peak amplitude in volts and ω denotes the angular frequency in radians per second ($\omega = 2\pi f$ and $f = 1/T$ Hz).

Figure 2-20: Sine-wave signal

The system is linear when the system transfer function does not vary with respect to the amplitude of the input signal. The output signal is just an amplified replication of the input signal. The size of that amplification or gain may vary with respect to the frequency of the input signal. For example, the gain of the system may remain constant only over a particular frequency range or bandwidth. However, it remains a linear system. The ability to keep a system linear is important because linear systems are considerably easier to design than nonlinear ones due to the complicated mathematics required to represent nonlinear systems. In practice, most systems are nonlinear, although in most cases the nonlinearity is small enough to be neglected or the limits of operation are small enough to allow a linear approximation to be made. Due to the nature of the physical components and the materials that constitute electronic components, the characteristics of an amplifying system remain linear only over a limited range of input and output voltages. Once the input or output signal exceeds that boundary, the transfer function cannot be used to predict the output signal from the applied input signal.

Using these foundation concepts, system design and analysis can be performed in terms of block diagrams of transfer functions. For control systems like the cruise control example, the system must provide the correct output for a varying input signal and

operating environment. However, environmental conditions and variations in the physical components can make that impossible. For example, the gain of active components, such as transistors that amplify the input signal, can vary due to voltage supply variations, temperature changes, or device ageing. The technique of system feedback can eliminate the effects of those variations on the transfer function by providing the following characteristics:

4. Increased accuracy: the closed loop system may be used to drive the difference between the actual and desired response to zero.
5. Reduced sensitivity to changes in components.
6. Reduced effects of external disturbances.
7. Increased speed of response and bandwidth.

For the purposes of control systems, the basic feedback block diagram is represented in figure 2-21, where G represents the transfer function of the system and H is the transfer function of the feedback path. Using simple block diagram algebra:

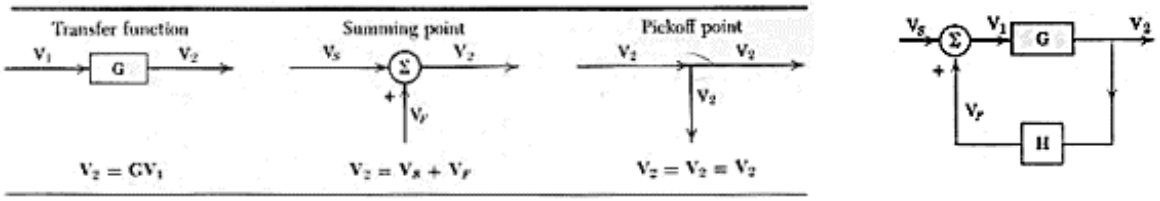


Figure 2-21: Control System Transfer Functions

$G = V_2/V_1$, $H = V_F/V_2$, and $V_1 = V_s + V_F$.

$V_2 = GV_1 = G(V_s + V_F) = G(V_s + HV_2) = GV_s + GHV_2$.

The global transfer function, G_F, then becomes:

$G_F = V_2/ V_s = G/(1-GH)$

This has many consequences for system design.

- If the quantity GH is positive, as it approaches unity the global system gain becomes infinite. The system produces an output with no input. This is an unstable system.

- If GH is negative, but the magnitude of the gain is large, then $G_F \cong -G/(GH) = -1/H$. The system transfer function becomes solely dependent on the gain of the feedback network. This is called negative feedback.

Because the global feedback system transfer function (G_F) is independent of the original, system transfer function (G), it becomes immune to variations in the gain of the original system and other external disturbances. The system output is directly proportional to the system input and because the feedback path consists of passive components, which do not have variable gain, it remains immune to the previously mentioned variations. Although negative feedback systems do suffer from low gain, the improvement in system performance and stability far outweighs this drawback. Moreover, gain can easily be obtained with an amplification stage after the feedback network.

For the cruise control problem, the feedback system needs to accurately track the desired speed. This requires a system with a large gain to respond quickly to fluctuations in the actual speed of the vehicle. With a large gain however, the system may become unstable. Stability can be restored with additional compensatory components, however these add to the cost and reduced reliability of the system. Figure 2-22 depicts the ability of a system to track a desired value. When the gain of the system is high, the system quickly reaches the desired value but oscillates around that value before tracking accurately. With lower system gain, the response time of the system to reach the desired value is much larger, however it then tracks the value more accurately. The problem for the designer of control systems is to provide the necessary accuracy and speed of response with adequate stability and reliability at a minimum cost.

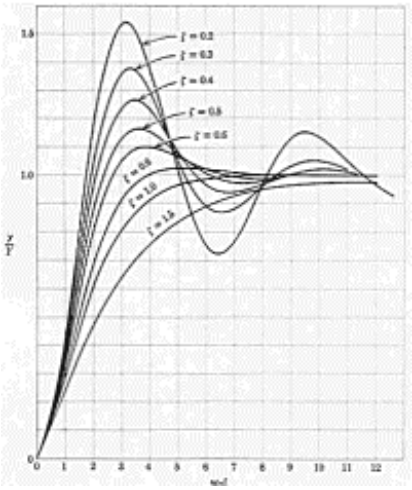


Figure 2-22: System Response

There are various methods of analysis for determining the stability of feedback control systems. Traditional methods rely on *root-locus* and *steady-state frequency response* analysis. If these analyses show the basic system does not meet the desired specifications it can be improved using compensators. Modern control theory provides analysis techniques and methods to ensure stability using *state-variable feedback*, *sensitivity functions*, *parameter optimisation*, and other mathematically intensive techniques. Often a designer will have to use multiple methods to solve the problem. The techniques

mentioned in the cruise control design examples to be detailed are briefly discussed to assist understanding.

A designer can calculate if the design meets its specification by determining the time response of the controlled variable. Laplace transforms simplify the calculation of time response transfer functions by allowing the solution to differential equations to be performed using simpler algebraic operations in conjunction with a table of transforms. The Laplace transform of the function, f(t), is designated F(s), where the parameter 's' is a complex quantity. Because the transfer functions of feedback system components are a function of time, they can be represented as $G_F(s) = G(s)/(1-G(s)H(s))$. The previous discussion noted that the system will be unstable when the denominator of that transfer function becomes zero. Therefore, to determine if a negative feedback system will be unstable, the designer needs to solve the characteristic equation $1 + G(s)H(s) = 0$. The root-locus method provides a graphical representation that is used to determine the system stability. It plots, on the complex plane, the roots of the characteristic equation for a closed loop system as a function of gain. It provides an easy way of determining if the system will be stable. Figure 2-23 depicts the relationship between the root-locus and the system response. If the roots of the equation lie on the left-hand side of the s plane (imaginary axis), the system is stable. If the roots are on the right hand side the system is unstable and the output will grow exponentially. If the roots lie on the axis of s plane, the system remains in constant oscillation.

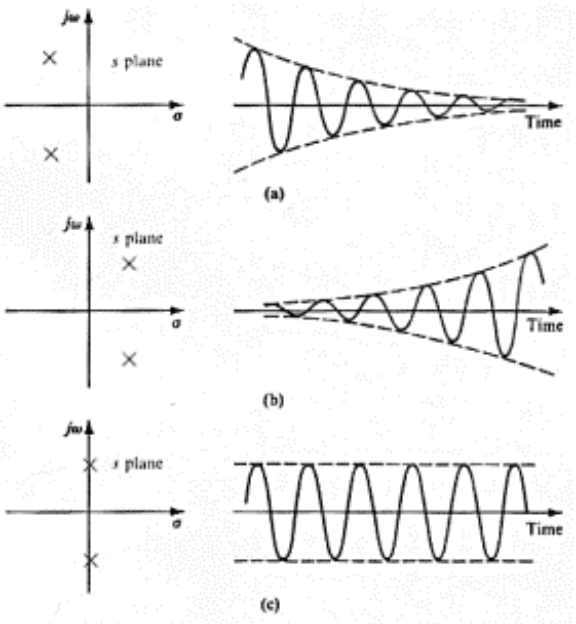


Figure 2-23: Root-Locus Analysis

The other method for determining the system stability is by analysing the frequency response of the system. Two types of graphical plot of the frequency response are used – Bode and Nyquist diagrams. The Bode plot (figure 2-24) graphs the magnitude of the gain and

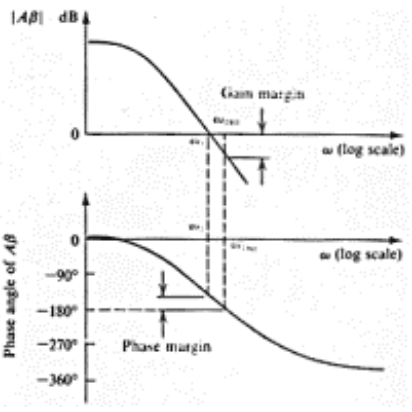


Figure 2-24: Bode Plot Analysis

phase of the system (amplification in decibels) versus the frequency of the input signal. The gain is plotted in decibels, the phase in degrees, while the frequency is plotted on a logarithmic scale. The Bode representation allows the designer to view the system's response to various variable values and determine their appropriateness. Stability can be determined by examining where the magnitude plot crosses the 0-dB line. If at this frequency the phase angle is less (in magnitude) than 180°, the amplifier is stable. The Nyquist diagram plots the frequency response of the system in the imaginary plane using polar coordinates. This provides the designer with a different representation of the system response. The advantage of the Nyquist representation is that the designer can immediately recognise if the system is unstable using the Nyquist criterion – if the Nyquist plot encircles the point (-1, 0) the system is unstable.

If initial analysis shows the design will result in an unstable system, the introduction of additional components can be used to reshape the root-locus. This is system compensation. The types of compensation used in the cruise control examples are proportional plus integral control and proportional plus integral plus derivative control. The frequency response of a system can be considered in terms of how it responds to a sudden step input (its transient response) and how it responds to a constant sinusoidal input (its steady-state response). The transient response details how quickly the system reaches the desired value and the steady-state response details how well it tracks the value once it has reached it. If the transient response is satisfactory but the steady-state error is too large, the system can be improved by operating on the actuating signal to produce one that is proportional to both the magnitude and the integral of this signal. The proportional plus integral controller (PI) is depicted in figure 2-25. This compensation changes the root locus to improve the steady state error without appreciably affecting the response time. Proportional plus derivative compensation (PD control) improves the transient response by moving the root-locus farther to the left of the imaginary axis. This is achieved by operating on the actuating signal to produce one that is proportional to both the magnitude and the derivative (rate of change) of the signal. Finally, the PI and PD controllers can be combined in a single controller to produce a proportional plus integral plus derivative controller (PID).

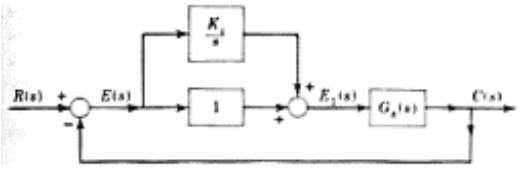


Figure 2-25: Integral Compensation

Advances in mathematical representation techniques, such as representing control systems in terms of state variables, has led to modern control theory methods. Techniques such as *parameter optimisation* and *recursive least squares* analysis, provide the designer with improved means of analysing the stability of control systems allowing the problem of optimal control to be addressed. The specifics of those techniques are mathematically intensive and are not required to understand their usage in the cruise control design examples.

2.4.2 Mechanical Cruise Control Systems

Ellinger (Ellinger 1985) and Koning (Koning 1984) provide implementation descriptions of the original analogue or mechanical cruise control systems that were used before the introduction of more advanced microprocessor based systems. Written for automobile mechanics and electronic hobbyists respectively, they do not discuss the design of the system explicitly, however they include sufficient detail to infer the design rationale.

Ellinger's description begins by detailing the basic feedback loop architecture. The driver engages the system at the desired speed, a sensor is used to determine the current speed, and a controller is used to reduce the difference between the current and desired speeds. The controller unit achieves that function by connecting to a power servo actuator unit that directly affects the throttle.

Details of the speed sensor, controller unit, and power servo actuator components required to implement the design were then discussed. The discussion details different types of speed sensors that could be used (flyweight governor or rotating magnet) and the locations in the vehicle where they could be positioned (speedometer cable or driveshaft). The internal operation of the speed controller unit and its connection to the power servo unit to manipulate the throttle provides a description of how the properties of mechanical components were used to implement the feedback network (figure 2-26).

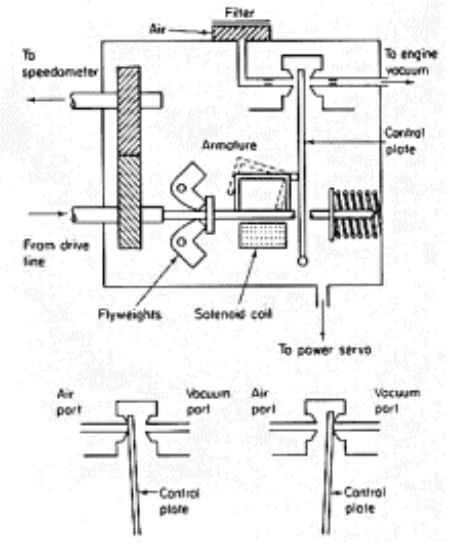


Figure 2-26: Mechanical Flyweight Governor Speed Control Unit

> "Centrifugal flyweights mechanically move a shaft against a spring. ... When the speed control is engaged a solenoid will clamp an armature against this shaft. ... When speed is slowed about 1/4 mph below the set speed, the plate closes the air port. This increases the vacuum in the controller housing. The housing is connected to the power servo so that the vacuum is increased in the power servo ... Increased vacuum in the power servo begins to open the throttle. When the speed increases about 1/4 mph above the set speed, the plate will move to close the vacuum port. Air comes into the housing through the air port to lower the vacuum. This gradually reduces the throttle opening to slow the automobile. The speed controller moves the control plate back and forth to adjust the vacuum so that it will hold the set speed within +/- 2 mph. ... Only enough vacuum is supplied by the speed controller [to the power servo] to position the throttle correctly. Air and vacuum are balanced while the automobile is running at the set speed". (Ellinger 1985)

2.4.3 Microprocessor Based Control

With the increase in availability and decrease in price of microchips, automotive companies began to centre their cruise control designs on the microprocessor (Shaout and Jarrah 1997). Nakumura, Ochiai, & Tanigawa (Nakamura, Ochiai et al. 1983) utilised a microprocessor in their design and claimed significant improvements over conventional analogue designs. In addition to the basic cruise control requirements they were able to realise increased functionality, such as system preset, transmission control, and fail-safe functions – functionality that existing analogue designs could rarely obtain. Moreover, the microprocessor-based design was able to rectify deficiencies identified in the analogue designs. Those deficiencies include compensating the feedback system for improved stability, coping with degrading factors such as vehicle characteristics, and handling nonlinearity due accelerator link hysteresis.

Nakamura began with a standard feedback control arrangement (figure 2-27) with the controller implemented in an electronic controller unit (ECU). The ECU has the ability

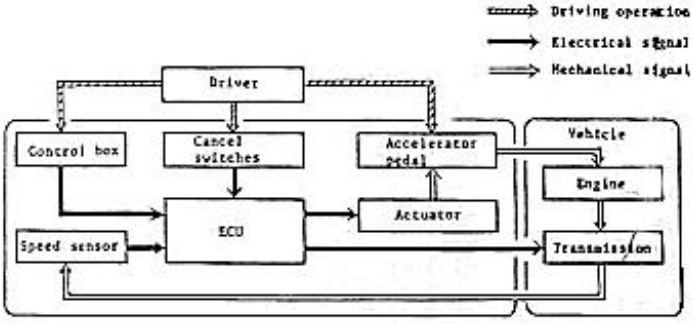


Figure 2-27: Nakamura Block Diagram

to manipulate the automatic transmission unit in addition to the actuator-controlled throttle. To improve on previously developed cruise control designs, Nakamura identified issues in the physical environment that may degrade system performance. He observed that "the cruise control system is a nonlinear control system in which the accelerator link has the hysteresis characteristics" (Nakamura, Ochiai et al. 1983). Hysteresis refers to "a retardation of the effect when forces acting upon a body are changed (as if from viscosity or internal friction)" (Miriam-Webster Dictionary 1997). In the context of the cruise control system this refers the force required to manipulate the throttle. The actuator is required to exert a force on the accelerator to keep the vehicle at a constant speed, for example during an incline in the road. However, because of the nature of the physical link, the force required to keep the car at a constant speed once the road has become level again may be different than it was previously. This does not change the generic feedback architecture devised to meet the solution, however the functionality of the components which comprise the architecture need to be devised with respect to those characteristics. Nakamura also noted that the driving speed of the vehicle periodically fluctuated while the vehicle was under control of the cruise-control system and this variable needs to be considered in the design of the feedback components.

> "Since the speed signal is transferred via the speedometer cable, if angular velocity fluctuation caused by cable torsional resonance exceeds the ECU linear calculation limit, constant speed control becomes impossible." (Nakamura, Ochiai et al. 1983)

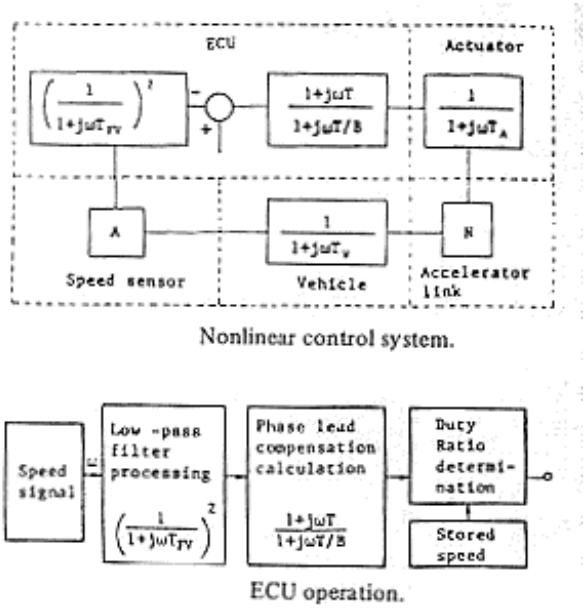


Figure 2-28: System Transfer Functions

The design proceeds by analysing the requirements of the system using an event-based approach. The results of the analysis were represented using a simple flowchart. Using the results of the requirements analysis and the previously identified system issues, the components of the generic feedback system (figure 2-27) were replaced by transfer functions (figure 2-28) to mathematically express the elements to be controlled. The ECU, actuator, throttle-valve connection, and

vehicle characteristics were each modelled as transfer functions. That included using standard transfer functions for generic components such as the well-known Butterworth filter that was employed in the ECU design.

The design process ensued by solving the transfer function equations to ensure the implemented system remained stable. Utilising the root-locus analysis technique, with parameter optimisation, graphical and mathematical tools were used to determine the appropriate values for the parameters of the transfer functions needed to meet the requirements. The system was then tested experimentally to ensure the design met the required objectives. The designers also note, "in the general running test, we enjoyed satisfactory driving without any problem." (Nakamura, Ochiai et al. 1983).

2.4.4 Considering the External Inputs in More Detail

The design by Rutland (Rutland 1992) begins by suggesting that previously published cruise control designs do not fully consider the effects of the external environment when determining what disturbances should be considered in the analysis of the feedback system. For example, wind velocity and angle of the road could affect the rate of acceleration required to bring the vehicle to the desired speed. Moreover, he suggests it is not possible to design the system to consider those disturbances using the traditional stability analysis techniques such as Bode and Nyquist. Utilising a more recent design analysis technique called 'matching' (Zakian 1991), which employs a set theoretic approach and results in quantitative design criteria in terms of the inputs, he proposes to design a better cruise control system.

Rutland notes that the principle of feedback control is required to satisfy the design criteria and explains that "an electrical feedback implementation incorporating a microcontroller" is the most appropriate generic system architecture to begin with. Moreover, he decides to utilise the same system configuration as Nakamura (figure 2-29) because it minimised cost and complexity. The process of implementing the generic architecture begins with the

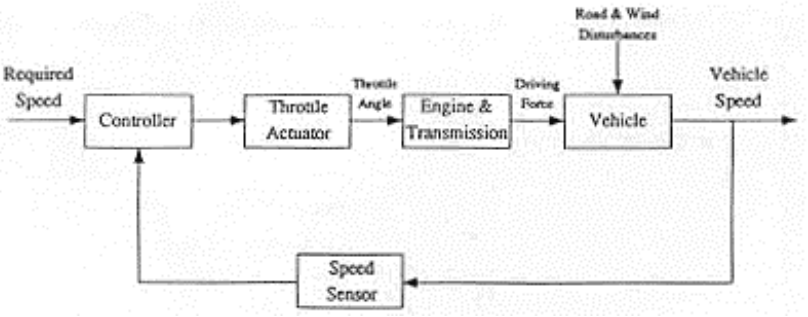


Figure 2-29: Feedback Block Diagram

selection of a pneumatic type actuator component, because it "gives the best compromise between the conflicting criteria of performance, cost, and reliability" (Rutland 1992). The pneumatic actuator connects to the throttle using a bead chain rather than a cable. This type of connection eliminated the problem of link hysteresis that was faced in the Nakamura design. An inexpensive speed sensor, connected to the speedometer cable for reliability, was selected as being suitable for the design.

The block diagram of the system (figure 2-29) was modelled as a collection of appropriate transfer functions that could be solved to satisfy the required functionality. To develop the transfer functions, Rutland draws on the work of a number of published researches. They included Takaskai and Fenton's (Takasaki and Fenton 1977) published models of vehicle longitudinal dynamics, the previously discussed design of Nakamura's, and a number of other published designs. The resulting mathematical model (figure 2-30) is more detailed than Nakamura's and includes variables to represent external inputs such as the angle of the road to the horizontal (θ) and the wind velocity (v_W). They are in addition to the variables required for the driving forces, velocities, and throttle settings.

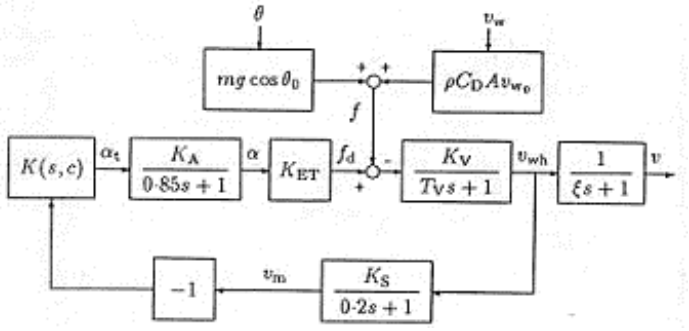


Figure 2-30: Block Diagram of Transfer Functions

In addition to the more detailed models, Rutland's and Nakamura's designs use significantly different techniques to solve the transfer functions. The traditional techniques for solving transfer functions were considered inadequate for representing the full set of external factors. Rutland recognised that if the design requirements were conceptualised differently, Zakian's matching technique (Zakian 1991), which uses numerical methods on sets of values, could be utilised to resolve the conflicting design criteria. The primary objectives of the cruise control problem were then restated as:

> "To maintain a constant vehicle speed and acceptable ride comfort, for the set of all possible load force. A particular load force ... is said to be tolerable if it keeps the speed and ride comfort levels, within predetermined margins. … Using this definition, an essential objective of control system design is to

> ensure the set of all possible load forces is a subset of the set of tolerable ones." (Rutland 1992)

The design then became a matter of adjusting the two sets to obtain a good match.

The major difficulty with the technique, and the source of a great deal of Rutland's design work (Rutland 1991), was the conceptualisation of the sets of possible and tolerable inputs so they could be compared. For example, the effect of the wind velocity can be modelled as having a persistent component and a superimposed transient component that produces the familiar gusts. The solution proceeds by utilising known mathematical analysis techniques to solve the equations. They were the method of inequalities to formulate the problem as a set of inequalities and the moving boundaries numerical search algorithm, for its simplicity and robustness, to solve those inequalities.

2.4.5 Adaptive Speed Control

The motivation for the design by Liubakka (Liubakka, Rhode et al. 1994) came from the manufacturer's point of view that a design for mass production differs substantially from an academic study of competing theories. The cruise control system should consist of a single control module that provides acceptable performance over a wide range of vehicle lines. Moreover, it should do so without need for recalibration. The complexity of speed control design strategies had increased to meet more stringent customer expectations, however for "commonly used proportional feedback controllers, no single controller gain is adequate for all vehicles and all operating conditions. Such simple controllers no longer have the level of performance expected by customers." (Liubakka, Rhode et al. 1994). For example, to accelerate to a desired speed, "low power cars will generally need higher [system] gains than high power cars. This suggests a need for adaptation to vehicle parameters." (Liubakka, Rhode et al. 1994). For an individual car, the best performance on flat roads is achieved with slow response to input fluctuations, while rolling hill terrain requires a faster response. In control design terms, slow system response requires high gain in the integral compensator, while faster response requires a lower integral compensation gain. "This suggests the need for adaptation to disturbances." (Liubakka, Rhode et al. 1994)

The designer reviews the history of cruise control design strategies and concludes that a well-tuned proportional integral (PI) controller achieves the best performance. The

difficulty with the PI controller, however, is how to keep it well tuned when the system and operating conditions vary so greatly. The controller gain is dependent on:

- Vehicle parameters (engine, transmission, weight, load, etc).
- Vehicle speed.
- Torque disturbances (road slope, wind, etc.).

Figure 2-31 (from (Germann and Isermann 1994)) depicts how those conditions affect the vehicle model.

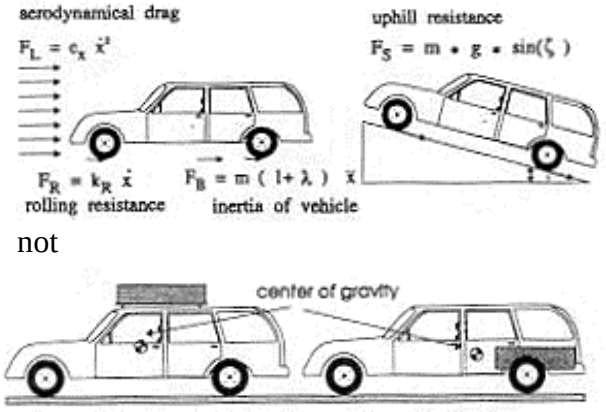


Figure 2-31: Vehicle Modelling

Because the vehicle parameters are not constant and torque disturbances are not measurable, it is not possible to automatically set the system gain. Considerable testing and calibration work is required to ascertain the system gain of a PI controller for one type of vehicle. If the controller is then applied to a different model in the company's range of vehicles, or to different variations of the original vehicles, such as a larger engine capacity or automatic transmission, the controller gain must be retuned. The goal for Liubakka was to design a "an adaptive controller that outperforms its fixed gain competitors, yet retains their simplicity and robustness." (Liubakka, Rhode et al. 1994)

Starting with the dynamics that are relevant to the design problem, a block diagram of the system transfer functions was developed (figure 2-32). A note is made that individual blocks contain different parameters and, possibly, a slightly different structure to represent the variations among different vehicle lines. Additionally, a separate input is made to the vehicle dynamics module to represent the road disturbance input such as a constant slope road or rolling hills.

The distinguishing feature of this example is the design of the adaptive control algorithm used to optimise the parameters of the system for the particular operating conditions. The algorithm is based on an approach to feedback analysis that minimises the sensitivity of the output response to parameter variation. Liubakka notes that the sensitivity analysis method was original proposed in engineering literature in the 1960s, however it was

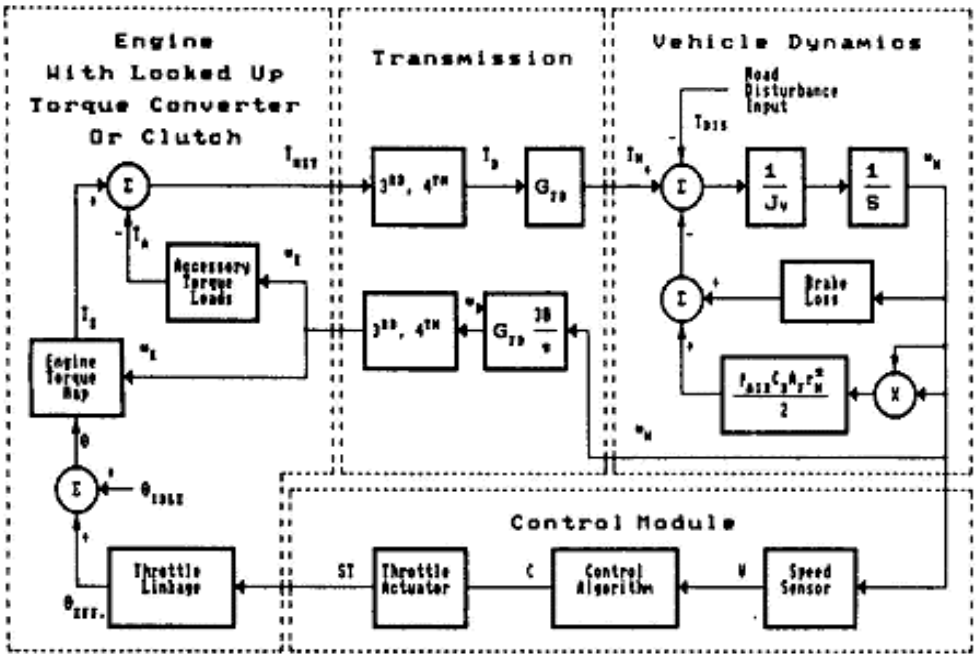


Figure 2-32: Transfer Functions of Adaptive Model

abandoned because it led to instability when used in systems that required fast adaptation. The designers re-evaluated the strategy based on analysis work from the late 1980s that proved the sensitivity-based approach could result in stable system design if it matched a 'pseudo-gradient condition'.

> "From the known bounds on vehicle parameters and torque disturbances, we evaluate, in the frequency domain, a 'phase-uncertainty envelope'. Then we design a sensitivity filter to guarantee that the pseudo-gradient condition is satisfied at all points encompassed by the envelope." (Liubakka, Rhode et al. 1994)

Beginning with a well-tuned, fixed-gain PID controller designed for a reference vehicle, a sensitivity filter was developed to adaptively tune the system within the bounds of an uncertainty envelope around that model. The PID controller for the reference vehicle was tuned to satisfy the conflicting requirements of providing a small speed error to accurately track the desired speed, and of minimising the amount of throttle motion to provide acceptable driver comfort. This was achieved by including the throttle position in the transfer function of the integral section of the controller. Adaptive control was attained by passing the speed error to the sensitivity filter, which produces a sensitivity function based on partial derivatives of the signal with respect to the proportional gain and the

integral gain. Those partial derivatives are passed onto the gradient filter to develop the optimal proportional and integral gains using a gradient optimisation technique.

The final testing and implementation of the system required subjective analysis based on experience with tuning standard PI controllers to determine the free parameters of the system.

2.4.6 A Fuzzy Approach to Autonomous Intelligent Cruise Control

Autonomous intelligent cruise control (AICC) systems are the subject of considerable attention in the control system community. Engineers at Mercedes-Benz, for example, have designed an AICC system that maintains speed and distance from preceding vehicles (Muller and Nocker 1994). The system operates as a traditional cruise control system by keeping the vehicle at the desired set speed as long as no preceding vehicle is detected. If a preceding vehicle is detected, the AICC switches automatically to distance control, driving with the same speed as the preceding vehicle at an 'optimal distance'. If the preceding vehicle accelerates over the pre-selected maximum or leaves the lane, the system reverts to the original speed control. Finally, to handle emergencies, the driver can override the system at any time by braking or accelerating.

The general cruise control feedback structure is augmented with deceleration control and a distance sensor (figure 2-33). Accurate distance sensing is notoriously difficult due to sharp road curvature, driver intentions, oncoming traffic, weather conditions, and particular driving situations. Those disturbances can never be eliminated and they are difficult to model accurately and implement in traditional PID control schemes (see Shaout's review of cruise control technology (Shaout and Jarrah 1997)). Moreover, PID controllers fail to adequately handle steep ascents and descents because of their poor transmission shifting strategies. Experiments by the designers with PID control resulted in an unsatisfactory, jerky ride. To provide a more intelligent type of cruise control design, which would more accurately mimic driver behaviour, the designers chose to utilise a fuzzy logic control unit.

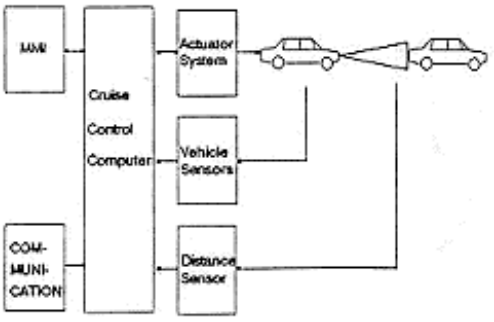


Figure 2-33: AICC Block Diagram

In general, a fuzzy control unit accepts two inputs: the difference between the desired and current speeds, and the vehicle acceleration. Different categories of input value combination are then given a membership value. From that table of values a mapping of the output is generated from if-then control rules. For this particular design, AICC controller inputs were required for the distance error from the 'optimal distance', which was based solely on the speed of the vehicle, and the speed relative to the preceding vehicle. These were used to determine the acceleration or deceleration correction output.

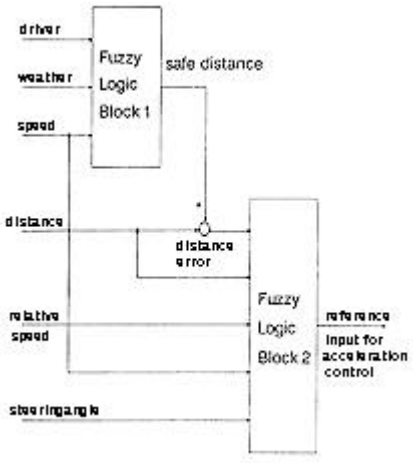


Figure 2-34: Fuzzy Controller

Experiments with this type of controller resulted in good speed tracking, however it was criticised for its inflexibility by human testers. Drivers wanted the 'optimal distance' to also depend on weather conditions (rain, ice, etc) and driver behaviour (sporty, neutral, comfortable, etc). The fuzzy distance controller was subsequently revised (figure 2-34) to contain two fuzzy blocks. The first is used to determine the 'optimal (safe) distance'. That stage uses sensors for the outside temperature, wiper action, and friction of the wheels to determine the weather conditions. The driver behaviour can be set using a potentiometer. The second stage uses the output from the first stage and the measured distance between the vehicles to determine the acceleration output. It extends the original distance controller by utilising the speed, distance, and steering angle to determine if the vehicle is in a curve and changes the required acceleration accordingly. Furthermore, the output is influenced by the driver's behaviour, which is determined by measuring the brake and accelerator actions.

The final implementation consisted of four fuzzy controllers with approximately two hundred rules. Experimentation and test-drives showed the system satisfied the design objectives and achieved near-human behaviour.

2.5 The Software Design Approach

2.5.1 Differences Between Designs That Used Different Methodologies

All of the software-based cruise control design examples began by utilising a particular design methodology to develop a conceptual view of the problem. The terms 'conceptual view', 'architectural style', 'domain model', and 'initial level of abstraction' are all, to some degree, synonyms. They represent the act of depicting a view of the world, system context, or domain model, using a collection of separately identifiable concepts and relationships. Beginning with that view of the problem, the software designs proceeded through many stages until the concepts contained within the initial model were implemented. The gross structure of those initial models varied based on the chosen design formalism, architecture style, or design methodology. Significant variations also exist between designs that utilised the same design formalism or style. This section details those differences to develop a generic approach to software development that will be compared with a corresponding generic approach to traditional engineering.

The designer's initial model of the problem, based on the chosen formalism, highlights certain aspects of the problem at the expense of others.

- State Based Modelling: Models the dynamics of the system, the modes in which it will operate, and the conditions that cause transitions between those states.
- Functional Modelling: Identifies the major functional components required and extends the model by incorporating the control signals that highlight the sequence of operation of that functionality.
- Feedback Control Modelling: Models the problem as a generic control system and instantiates the constituent concepts with specific concepts from the problem domain.
- Object-Oriented Modelling: Allows the direct modelling of concepts identified in the developer's perception of the problem domain. Different OO analysis methods, for example JSD, provide different guidelines for identifying boundaries around the entities (objects) that should be treated as first-class objects.
- Real-time and Concurrent Modelling: Used in extension to one of the preceding styles, they provide concepts to explicitly considered timing constraints. They also

allow the representation of concepts that can be executed in parallel by the computer.

These models only highlight a subset of the system properties required in a software implementation. To provide a complete view of the system design, the initial models were complemented with other design representations that emphasised the aspects not depicted in the initial model or abstraction.

All designs must contain formalisms to represent the desired functionality and behaviour of the proposed system, and a means of representing the structure of how that functionality and behaviour will be implemented in software (Harel 1992). The software designs exemplify that observation.

- The state-based designs began with the behaviour and then developed the functionality and structure.
- The object-oriented notation encapsulates the functionality and structure in a single concept, the object, and then augments the design with a representation of the control flow. This is usually done with a state transition diagram.
- The traditional structured analysis and design examples represent the functionality and behaviour with data and control flow notations. A structure chart notation represents the system structure.
- Finally, the feedback control representation models the functionality of the design using a generic pattern that includes its own internal means of control. An additional behavioural notation is then required to depict how the feedback system is controlled within a global context. The structure representation depends on how the concepts in the feedback model will be implemented in the chosen programming language, for example object-oriented or procedural.

Both Shaw (Shaw 1995c) and Gomaa (Gomaa 1993) performed an analysis of the primary design formalisms provided by the different methodologies, using cruise control as an example. Shaw, concentrating on properties of the architectures derived using those methodologies, reviewed designs from each of the categories mentioned. The designs reviewed by Shaw were (Booch 1986; Higgins 1987; Ward and Keskar 1987; Smith and Gerhart 1988; Birchenough and Cameron 1989; Yin and Tanik 1991; Atlee and Gannon 1993; Shaw 1995b) and an NRL/SCR design by Kirby that could not be located in the

research literature. Shaw also notes the existence of some of the other designs used in this study though they were not used in her own. Shaw's analysis evaluated the software architectures using the following criteria:

- Locality and separation of concerns.
- Perspicuity of the design.
- Analysability and checkability.
- Abstraction power.
- Safety.
- Integration with the vehicle.

Gomaa, rather than review a set of existing cruise control designs, analysed the ability of existing methodologies to represent real-time systems by developing his own cruise control designs. The methodologies used where RTSAD, DARTS, JSD, NRL/SCR, and OO. The analysis evaluated the methodologies in terms of the following real-time system issues:

- Provision of concepts for representing concurrent tasks.
- Realisation of information hiding/object structuring to support modifiability and reusability.
- The definition of control aspects of the system using state machines.
- The handling of timing constraints for real-time issues.

Both of these research efforts provide useful insights for software development. However, for the purposes of this study, only portions of Shaw's analysis are relevant. Those portions are summarised now and a critical examination presented in the comparison of software and hardware design approaches.

Shaw's discussion begins by noting the substantial differences that exist between the software designs, attributing them to variations among individual designers and the way each architecture led the designer to view the world. Each designer chose a particular architecture style and modelled the solution based on that style. Additional representations were then used to provide complementary views of the system. Those views were decomposed and combined to create the system, though not all of those

models could persist until runtime. Shaw notes that most models provide associated techniques to ensure the correctness of the aspects of design they are intended to highlight. However, when multiple views were used to provide decompositions from one representation to another or when concepts from one model were added to another model, confusion could arise. When considering perspicuity of the design, each of the examples could be defended as matching some view of reality. However, Shaw claims that styles such as object-oriented and process control are more explicit in their modelling of the 'real world' than, for example, functional decomposition. The particular modelling styles allow the designer to decompose the problem into parts that localise decisions. In addition, they facilitate the ability to identify and implement components that can be reused in similar applications. Finally, Shaw suggests a guiding factor in choosing a particular style is the degree to which it allows the identification of entities that are most important to the client.

2.5.2 Differences Between Designs That Used The Same Methodology

In addition to differences between designs caused by the use of different methodologies, significant differences also exist between designs that used the same methodology or architecture style. That was most evident in the collection of object-oriented designs, though it also exist in the other design methodologies.

Booch's object-orientated analysis (figure 2-4) was a result of 'objectifying' the inputs and internal states of the data flow diagram (figure 2-3) that was used to conceptualise the original requirements. Yin and Tanik (Yin and Tanik 1991) used the same data-flow

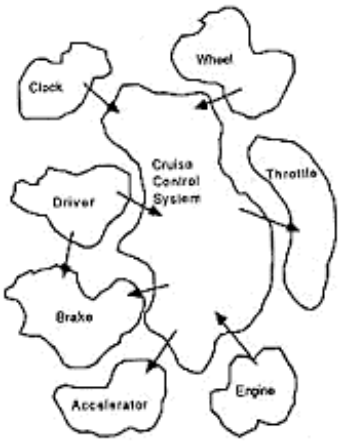


Figure 2-35: Yin & Tanik Object Model

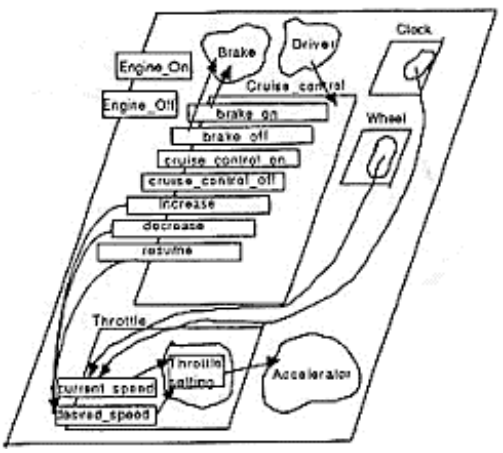


Figure 2-36: Yin & Tanik Architecture

diagram as Booch to represent the system requirements. However, from the same representation they created a different object model by identifying objects to represent the external elements and a single object for the entire cruise control system that encompasses all other elements (figure 2-35). The generated system architecture discriminates between active and passive objects and incorporates system operations (figure 2-36).

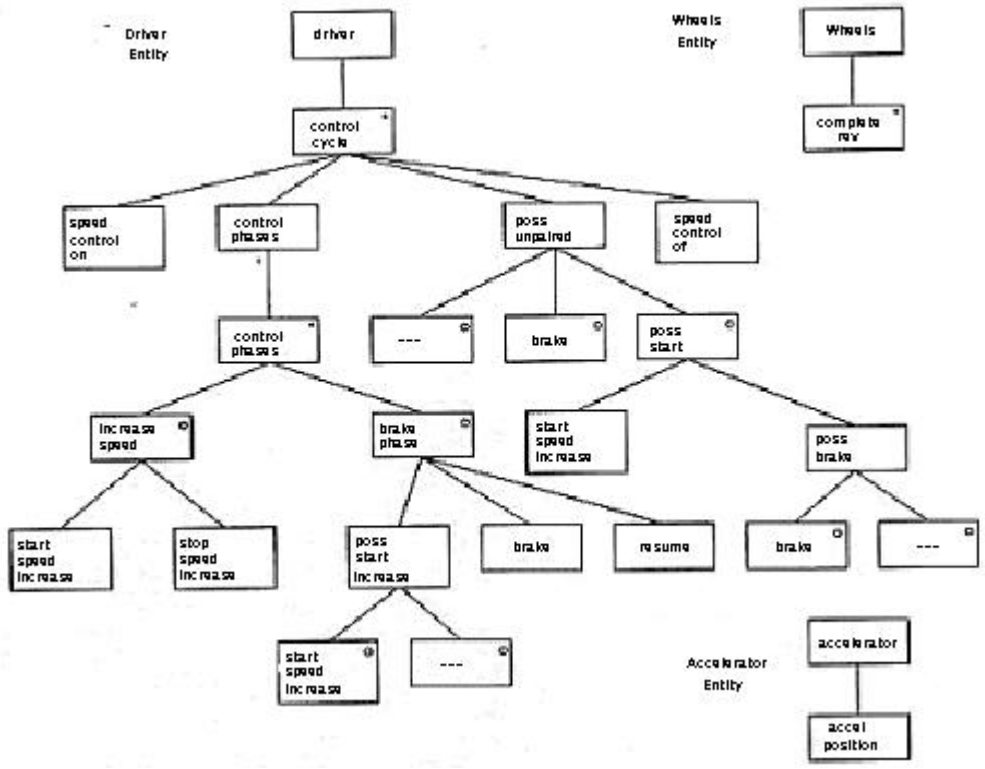


Figure 2-37: Birchenough JSD Entities

The central tenet of the JSD method is similar to that of object-oriented design, "the key to software quality lies in the structuring of the solution to a problem in such a way as to reflect the problem itself" (Birchenough and Cameron 1989). However, in JSD objects are referred to as 'entities' and their methods are called 'actions'. The JSD approach to analysis emphasises the identification of actions before entities, with objects becoming entities only if they suffer time-ordered or state-changing actions. The analysis by Birchenough and Cameron results in an 'object-model' that consists of three objects: 'driver',

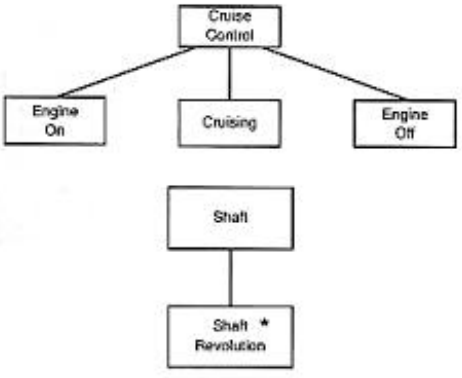


Figure 2-38: Gomaa JSD Entities

'wheels', and 'accelerator'. Figure 2-37 depicts the JSD entities and their actions, with time orderings, in structure chart notation. In contrast, Gomaa uses the same JSD methodology and identifies 'cruise control', 'calibration', and (drive) 'shaft' as the highest level entities. Figure 2-38 shows Gomaa's 'cruise control' and 'shaft' structure diagrams. In effect, Gomaa models the operations of Birchenough's 'driver' entity within his 'cruise control' entity. Gomaa also identifies a 'buttons' entity, however that entity represents the monitoring functions of the Brackett requirements, which Birchenough did not consider.

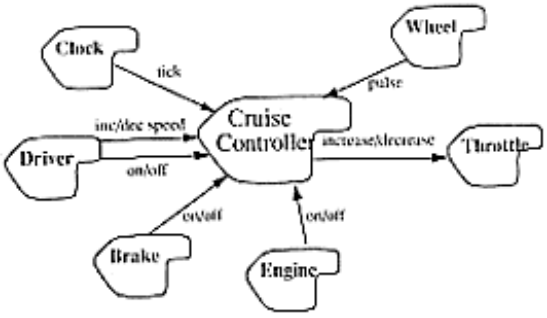


Figure 2-39: Appelbe Object Model

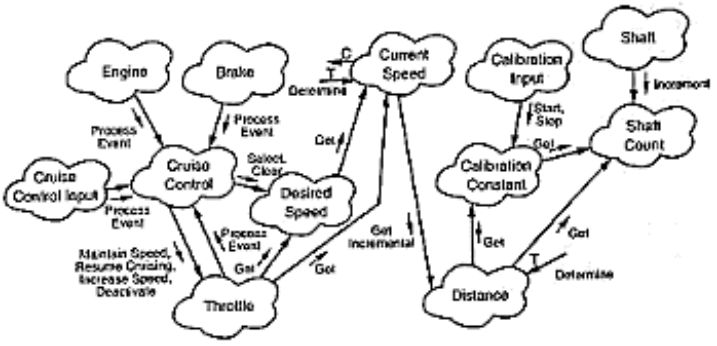


Figure 2-40: Gomaa Object Model

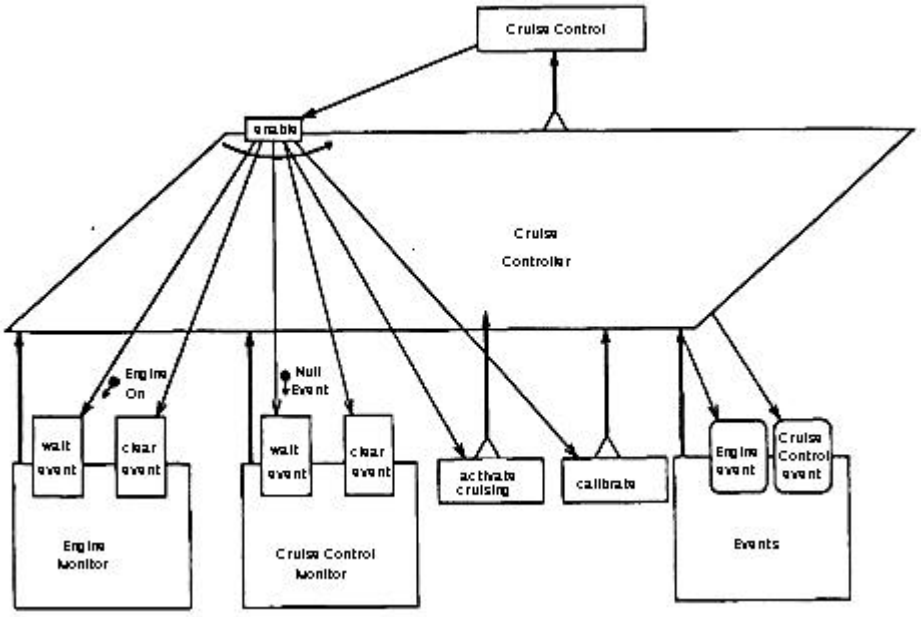


Figure 2-41: Wasserman High-Level Architecture

Appelbe and Abowd (Appelbe and Abowd 1995) note that Booch's design was presented from one of the earliest articles on object-oriented design and provide a new design based on Booch's more recent OO design guidelines (Booch 1991). Those guidelines require candidate objects meet the criteria of having state, behaviour, and identity. When applied,

the guidelines result in an object model of the system that is depicted in figure 2-39. Curiously, Gomaa uses the same set of Booch guidelines to develop his object-oriented design (Gomaa 1993) and develops a completely different object-model to Appelbe and Abowd. His design is shown in figure 2-40.

Wasserman's approach to design (Wasserman, Pircher et al. 1989) extends traditional structured analysis with additional notations, one of them being the object-like notion of information clusters that encapsulate data and functionality in a single structure. That approach results in a hierarchical collection of diagrams depicting the information clusters, operations, control & data flow, and asynchronous processes in the design. The high level system, depicted in Wasserman's OOSD notation, is shown in figure 2-41.

The object-oriented designs result in vastly different collections of objects. Table 2-1 depicts the collections of objects identified by the different designers.

Design Example	Objects Identified.
Booch	Driver, Brake, Engine, Clock, Wheel, Current speed, Desired speed, Throttle, Accelerator. (9)
Yin & Tanik	Driver, Brake, Engine, Clock, Wheel, Cruise control system, Throttle, Accelerator. (8)
Birchenough	Driver, Wheels, Accelerator. (3)
Gomaa (JSD)	Cruise control, Calibration, Drive shaft. (3)
Wasserman	Cruise controller, Engine monitor, Cruise monitor, Brake pedal monitor, Engine events, Cruise events, Brake events, Speed, Throttle actuator, Drive shaft sensor. (10)
Appelbe & Abowd	Driver, Brake, Engine, Clock, Wheel, Cruise controller, Throttle. (7)
Gomaa (Booch OO)	Brake, Engine, Cruise control input, Cruise control, Desired speed, Throttle, Current speed, Distance, Calibration input, Calibration constant, Shaft, Shaft Count. (12)

Table 2-1: Cruise Control 'Objects'

The different object-oriented or object-like methodologies all attempt to achieve the same purpose – the identification of the important objects in the designer's perception of the problem.

> "With an object orientated approach … we instead structure our system around the objects that exist in our model of reality. By extracting the objects from the data flow diagram … immediately we can see that the object orientated decomposition closely matches our model of reality." (Booch 1986)[2]

Similar claims can be found in almost any reference on object-orientated development. Those claims are often used as justification for the belief that object-oriented techniques promote reusability of implemented concepts – a claim based on the assumption that people identify similar objects in their models of reality. However, the enormous differences between the collections of objects used by the designers to model a problem as small and well-defined as the cruise control system clearly highlights the differences in the way similarly trained people can view the same reality. This observation alone calls into question the 'reusability' claims which pervade object-oriented design reasoning[3].

The differences between design examples created using the same design methodology or architecture style were not limited to the object-oriented designs. The design of Smith and Gerhart depicts a state-based approach to development that begins by identifying the control events that exist in the problem (figures 2-5, 2-6, & 2-7). Their design identifies standard driving events, cruise control actions, and monitor features (e.g. speed calculation) as the important activities. In contrast, Atlee and Gannon also produced a state-based design (Atlee and Gannon 1993). They utilised Brackett's formulation of the cruise control problem to demonstrate their state-based model checking technique for verifying the requirements of event-driven systems. They recognise that a state machine representation of a system can serve as a temporal logic model, which can be tested for safety by presenting the properties as temporal formulae. The result of the process is a state-based representation of the system, expressed in computational tree logic (CTL),

---

[2] More recent design guidelines provide rules of increased complexity for identifying those objects, however their purpose is still the same.

[3] This point is discussed in more detail in chapter 5.

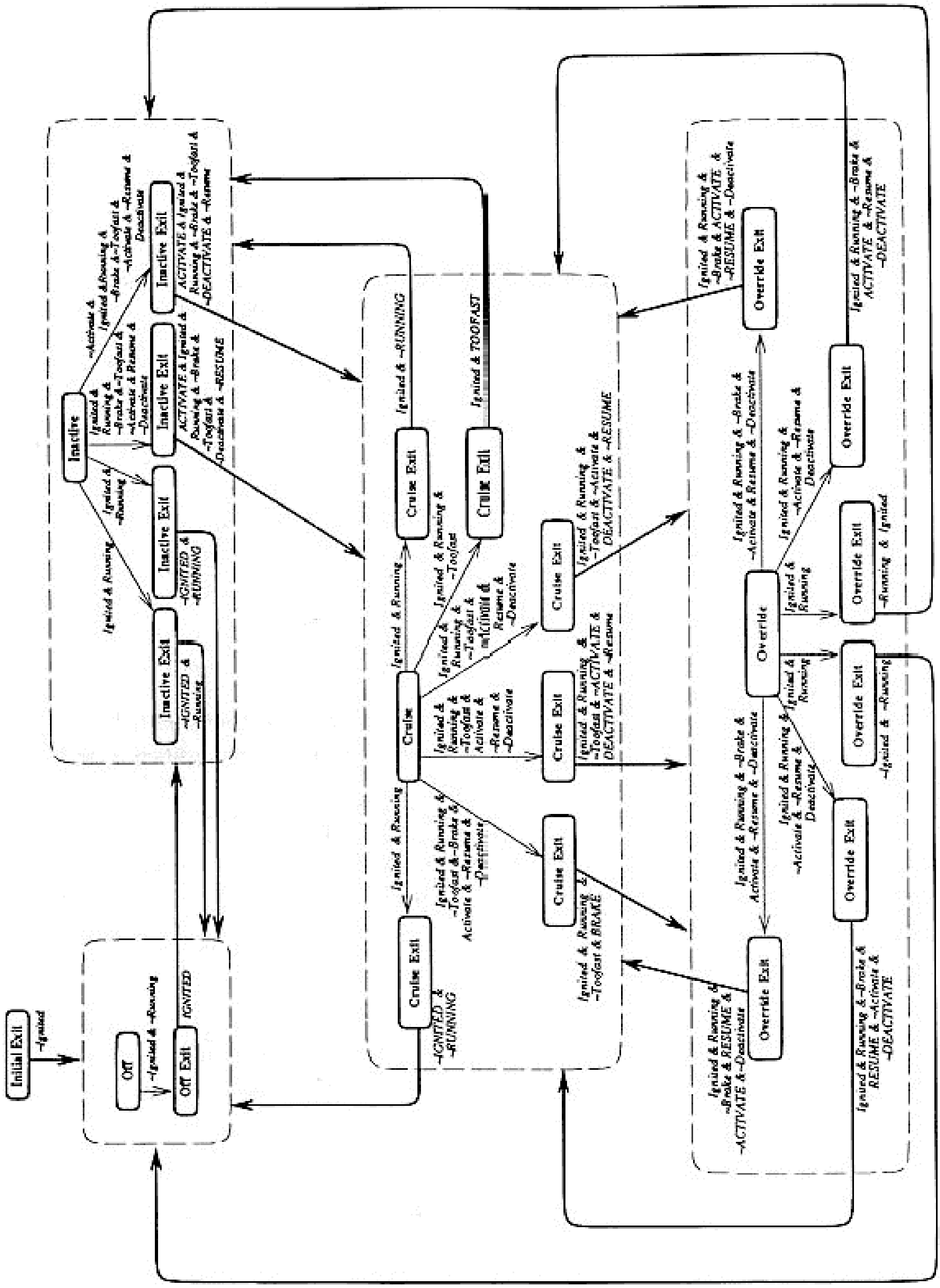


Figure 2-42: Atlee & Gannon CTL Model

which is detailed enough to serve as a system architecture. That model does not provide enough design rationale to analyse the design process to completion, however there is sufficient information to show how the state-based or event-driven approach to design drives the initial architecture and how that architecture sets the path for the subsequent design

The initial step in the design process is the partitioning of the system into four possible modes: off, inactive, cruise, and override (on but not in control). A transition table represents those modes and the events that cause transitions between them. A number of operations and transformation algorithms were performed on the requirements to determine the validity of the requirements or initial system conception and, finally, the system was represented as a CTL model (figure 2-42). The Atlee and Gannon design stops at that point, however to complete the design of the system the functionality required to generate the identified events and to implement the functionality in each state would need to be addressed.

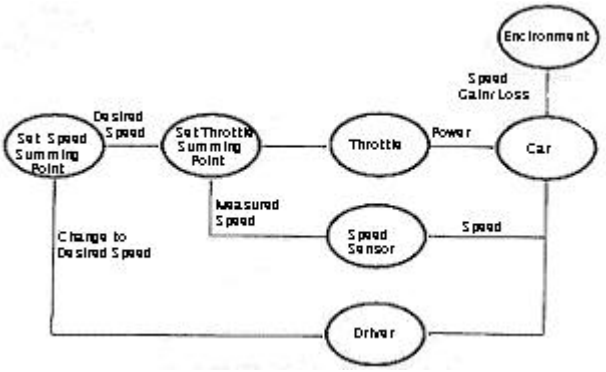


Figure 2-43: Higgins Generic Feedback

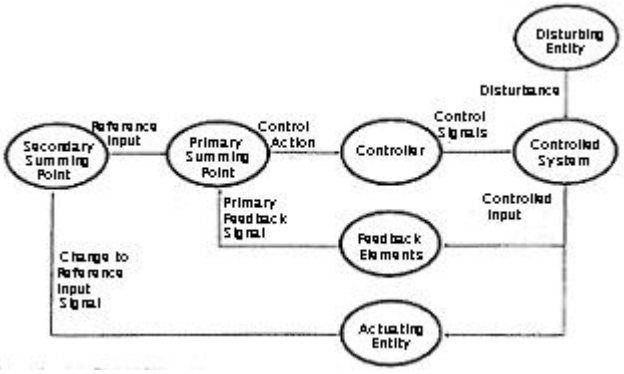


Figure 2-44: Higgins Specific Feedback

The different process control based designs also exhibited significant variations. Shaw's design begins with the abstraction of a 'classic feedback loop' and proceeds to match the concepts of the problem domain to the concepts of the generic feedback architecture. Higgins uses a similar approach (Higgins 1987) but develops a different system representation. His design begins with a more complex representation of the generic feedback pattern that contains a secondary feedback loop (figure 2-43). The process then matches the problem domain concepts to the feedback pattern concepts to depict the system representation (figure 2-44). Higgins then considers the rest of the system operations as feedback concepts and derives another, more complex feedback arrangement (figure 2-45).

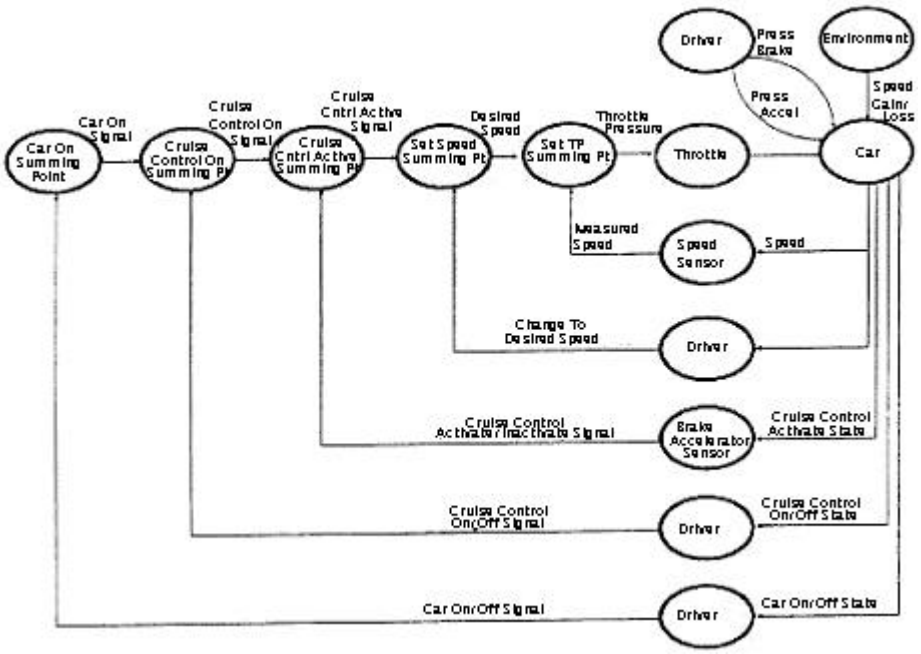


Figure 2-45: Higgin's Complex Feedback Model

Shaw and Higgins use the same design approach, beginning with a specific conceptual model (feedback control) rather than a particular methodology. This is in fact a design pattern strategy. In this approach, a generic arrangement is recognised as a suitable means of solving the problem and the pattern is instantiated by realising the generic pattern components with specific concepts from the problem domain (Gamma, Helm et al. 1994). The relationship between design patterns and software architecture styles is not well defined with researchers debating whether or not they are the same. A special issue of IEEE Software (Mellor and Johnson 1997) provides many papers discussing the issues.

Jones (Jones 1994) also begins his design by recognising the need to employ the generic feedback architecture of traditional engineering design. Trained as an electronic engineer, his design differs from both Shaw's and Higgin's because of his more detailed knowledge of feedback control systems. His design also matches the concepts of the cruise control domain to those of the generic control architecture. However, his pattern-matching process goes to a deeper level of detail to consider feedback concepts such as 'sampling frequency', 'system gain', and 'system stability'. Interestingly, his design process is not different to the approach of Shaw and Higgins, however it goes to a greater level of detail.

The oldest of the design methodologies used in the examples, structured analysis and design, also shows variations when used to model the cruise control problem. The Ward/Mellor SADT design of Ward and Keskar (figures 2-12 & 2-13) can be compared

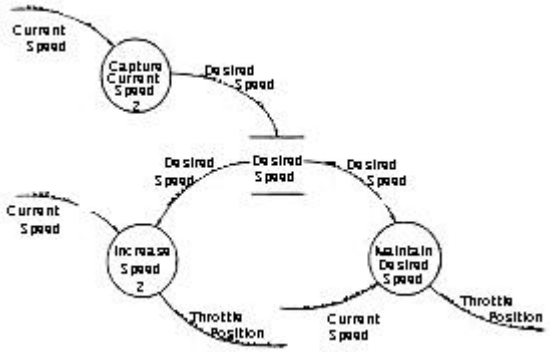


Figure 2-46: Ward & Keskar's BH DFD

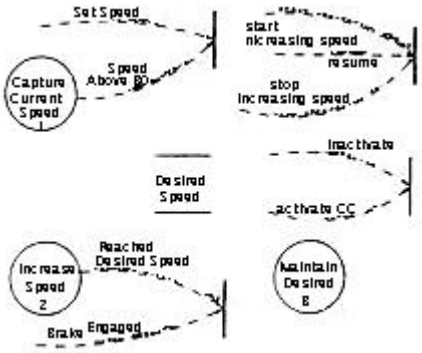


Figure 2-47: Ward & Keskar's BH STD

with their Boeing/Hatley representation and with the functional decomposition examples of Booch and the SADT design of Gomaa. Ward and Keskar's examination of the Boeing/Hatley technique begins with a data flow diagram (figure 2-46) and a representation to depict the control flow (figure 2-47).

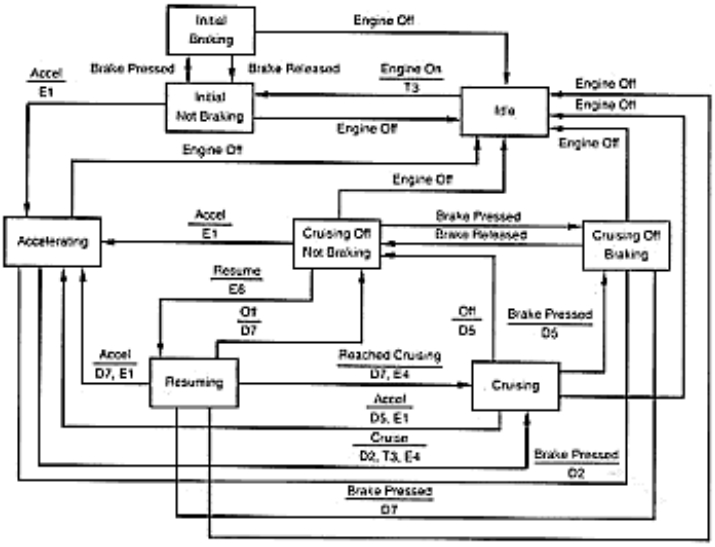


Figure 2-49: Gomaa STD

Booch uses a functional decomposition design to evaluate his object-oriented approach. The data flow diagram was given in figure 2-4 and is quite similar to that developed by Ward and Keskar.

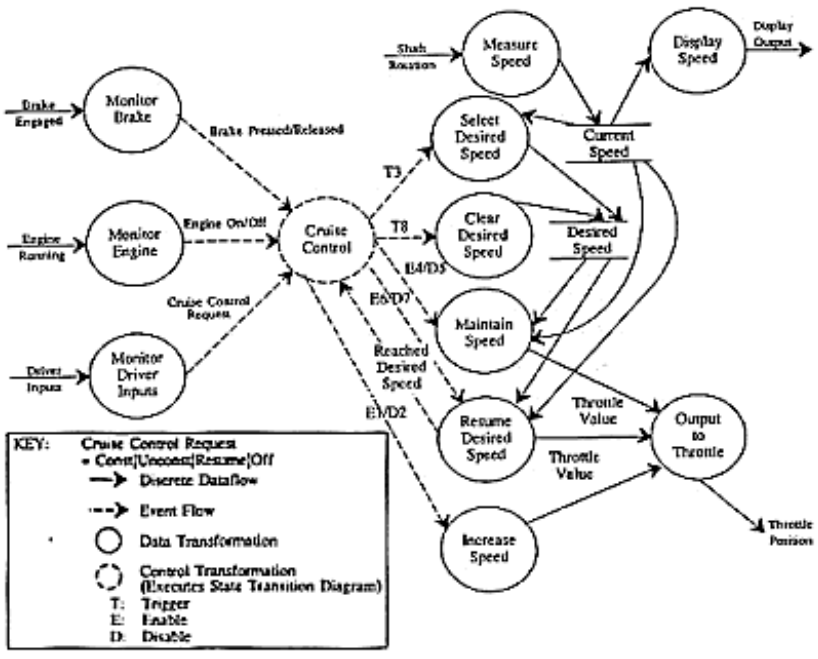


Figure 2-48: Gomaa DFD

Gomaa, using RTSAD to evaluate real-time modeling techniques, develops a detailed, hierarchical, structured

design using the Ward/Mellor notation. Beginning with the system context diagram, the design is partitioned into the 'Perform Cruise Control' and 'Perform Automobile Monitoring' functions, where the later is used for the monitoring functions of the Brackett requirements. The data flow diagram and state-transition diagram for high-level cruise control functions are shown in figures (2-48 & 2-49). The data flow diagrams are refined to successive levels to complete the design of the system before the structure charts are created.

2.5.3 Discussion

A generic approach for the software design examples was identified. That approach begins by identifying some means of representing the problem 'on paper'. Modelling the problem using a known design methodology or using a known architectural style achieves the same purpose. It represents the designer's mental conception of the problem in terms of entities and relationships that can be eventually implemented in a computer program. Those methodologies constrain the types of entities that can be chosen for the initial model. Design methods such as object-oriented or state-based constrain the entities and the relationships to those that can be represented in the chosen formalism. Styles of software architecture similarly limit the entities that exist at the highest level of abstraction to entity and relationship types of well-known large-scale system arrangements such as 'pipe and filter' or 'process control feedback loops'. Although the developer is constrained by the choice of formalism used to represent the problem, the presented examples show that there can be many variations within a particular style of development – even for such a small problem as automotive cruise control.

The reasons why those differences occurred, both between particular design styles or methodological formalisms, and within those formalisms, are examined later during the comparison with the generic hardware approach to design.

Before presenting that generic hardware approach however, there is one more observation that influences any attempt to make analogies between the two disciplines. That observation concerns the notion of feedback. Only the designs of Shaw, Higgins, and Jones identify themselves as being feedback or process control systems. Indeed, Shaw stresses the explicitness of her process control approach using Booch's object-oriented design as a counter example. However, the definition of a feedback control system is simply the implementation of a "path or loop from the output back to the controller. Some

or all of the outputs are measured and used by the controller … [It] may then compare a desired plant output with the actual output and act to reduce the difference between the two." (Stefani, Clement J. Savant et al. 1994). All of the software design examples, regardless of the methodology used to model the system, use the current speed to affect the system control, thereby making them feedback systems. The three 'feedback' based designs make the concepts of the generic feedback loop explicit during their design process. However, those same concepts are also evident in the other examples. They are not stated explicitly, but they are still evident.

This observation highlights questions concerning the discipline's understanding of what a system representation actually is. Those questions are examined in detail during the later comparison.

2.6 The Hardware Design Approach

All of the engineering designs exhibited a similar pattern. Beginning with a set of system specifications, a feedback control architecture consisting of well-known generic components was created. The designer then created models, usually mathematical transfer functions, of the specific functionality required of those generic components. Combinations of analytic and experimental techniques were used to solve the unknown parameters of those transfer functions and to implement the functionality of the system. Finally, experimental testing of the system fine-tuned the variables and ensured it complied with the original performance specifications. Each of the steps in this generic process is discussed, detailing the variations that exist. The discussion of feedback control architecture, design reuse, component reuse, standard analysis techniques, use of mathematical models, and differences between the logical and implementation views of a system are used to develop an in-depth understanding of the engineering development process. That understanding is subsequently used to compare the traditional engineering approach with the software development process.

2.6.1 The Evolutionary Nature of the Designs

Starting with the establishment of the specification of the system, the first observation concerns the evolutionary nature of the system designs. The problem of cruise control has stayed essentially the same since the first cruise control systems in the 1950s. Even the most recent variants, autonomous intelligent cruise control systems, are merely extensions

of the original principle – maintaining the constant speed of a vehicle even over varying terrain. The need for new designs did not change through the development of new requirements. Rather, the requirements were considered in more detail by demanding improvements over previous solutions. Those improvements were considered from the customer's point of view not the designer's. The examples did not suggest a new way of designing or analysing the system simply because it was easier than a previously used technique. It was done to produce a system that performed better for the customer or was cheaper to manufacture for the producer. For instance, Nakamura developed a microprocessor based design to provide functionality that traditional mechanical designs could not and Liubakka produced a design that worked better across a range of vehicles. Traditional methods of control were originally adequate, however as customer expectations changed, and better design techniques were applied, improved, rather than new, design solutions were possible. The designs, presented in chronological order, demonstrate that evolutionary nature.

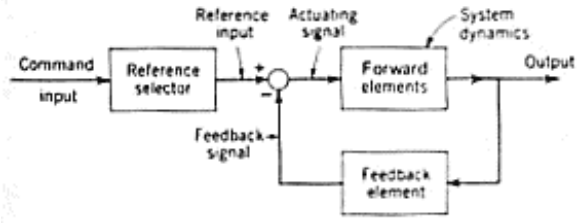


Figure 2-50: Generic Feedback Model

From the system specifications, the problem was immediately recognised as a control problem – an area of engineering design with a long history of theory and experimentation. That research has led to a number of system configurations for dealing with control problems, most notably feedback control (figure 2-50). All of the engineering designs, regardless of the major area of concern, the complexity of their controlling mechanism, or the nature of the components used (mechanical or microprocessor based) utilised the principle of feedback control in their system. The choice of feedback control as the global system configuration appears to be such an obvious choice that only one paper provided any design rationale for the decision:

> "Feedback control is needed to achieve the desired design criteria in the presence of disturbances, and uncertainty in the plant model [a production car with a carburetted internal combustion engine and manual transmission] and parameters. ... The modern trend seems to be to implement digitally with microcompomputers ... For the reasons above, an electrical feedback implementation incorporating a microcontroller was chosen. The very simple

> control system configuration ... was picked to minimize cost and complexity." (Rutland 1992)

The individual feedback configuration was depicted for each of the presented examples, however the use of feedback in the remaining cruise control examples provides additional evidence of design reasoning in the engineering disciplines. For instance, an alternative approach to the adaptive controller of Liubakka was presented by Oda et al (Oda, Takeuchi et al. 1991; Tsujii, Takeuchi et al. 1991). Their design used a feedback network within the global feedback structure to self-tune the system to changes in the vehicle operating model (figure 2-51).

Similarly, in contrast to the fuzzy system approach of Muller, Ioannou (Ioannou, Xu et al. 1993) used a conventional PID controller arrangement to produce an AICC system (figure 2-52). Moreover, he used feedback control systems to model human driving behaviour and then used those models to compare and evaluate his PID based design.

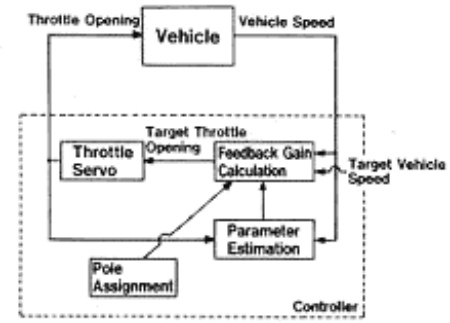


Figure 2-51: Adaptive Control

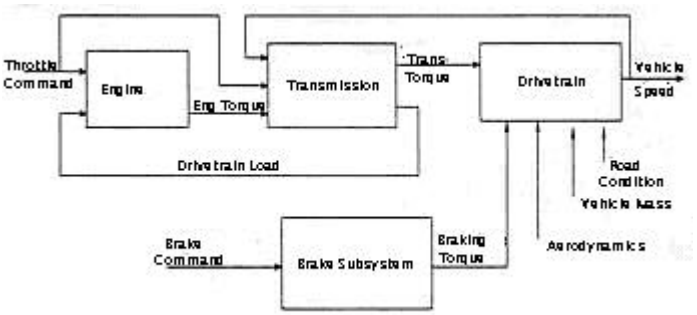


Figure 2-52: AICC PID System

Three other designers used the feedback system configuration slightly differently to those previously presented. Takasaki (Takasaki and Fenton 1977) developed mathematical models of longitudinal vehicle dynamics that were used by other designers to develop their cruise control transfer functions. The derivation of those models involves the use of feedback models to identify particular parameter values. Furthermore, the design of Lee (Lee, Kim et al. 1993) concentrated on just one aspect of the design – controlling the movement of the throttle in a pneumatic cruise control system. Traditional techniques used a 'bang-bang' method of control,

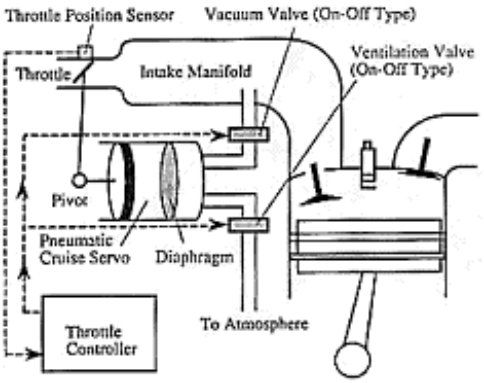


Figure 2-53: Throttle Control

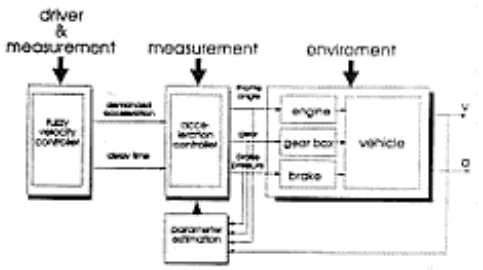


Figure 2-54: St Germann Model

where the control direction is switched after the error crosses a desired value. Lee develops a 'sliding mode' control by modelling the system using the anticipatory band in the phase plane. This allowed the system to begin switching the control direction before the error reached the desired value, thereby allowing a smoother and more accurate tracking of the required throttle opening. The resulting system (figure 2-53) uses a feedback network to control the throttle within the global feedback network of the overall cruise control system. Finally, St. Germann and Isermann discuss model-based methods for controlling all longitudinal vehicle dynamics. Obtaining the required functionality is a more general design problem than the cruise control system and requires the ability to bring the vehicle from any operating point to any other. The result was a feedback system incorporating a linear controller and a fuzzy velocity controller (figure 2-54).

2.6.2 The Reuse of Existing Designs and Components

In all of the engineering designs the development of the system architecture entailed considerable design and component reuse. In fact, the use of feedback control is itself a form of design reasoning reuse of the kind that software engineering researchers have sought to bring to software development. It is standard practice for engineers to publish their designs and utilise the designs of others. For instance, Rutland used the initial system architecture of Nakamura's design and developed his model of transfer functions by utilising the work of three other published designs. Liubakka reviewed the published history of cruise control examples and identified PID control as the most robust design strategy with the best tracking performance. He then improved on previous PID designs by using an adaptive control technique to overcome the deficiency with PID controllers – their vehicle dependant system gain.

The hardware cruise control examples provide many examples of component reuse. That reuse involves both specific components, such as Nakamura's use of the Butterworth filter, and generic components such as 'actuators', 'speed sensors' and 'controllers'.

> "The actual acceleration of the vehicle can be determined using numerical differentiation. There is a wide range of nonlinear controllers that can be

> chosen as suitable candidates for this application." (Germann and Isermann 1994).
>
> "A pneumatic type actuator was selected as it gives the best compromise between the conflicting criteria of performance, cost, and reliability." (Rutland 1992).

The ability to reuse previous system structures, especially for routine design problems, is enhanced by their capacity to communicate designs graphically. Diagrammatic representations serve as a visual communication medium that allows engineers to convey their ideas, not only to others, but also to reflect on their own work. The ability of engineers to convey their designs and ideas in a uniform, well-understood manner across the entire discipline has helped their profession to evolve by successfully building on the ideas of others. The graphical communication medium has provided the infrastructure for that evolution. Training in technical drawing is required in virtually every engineering school in the world. Disciplines have their own standards for graphical representation (for example: (The Institution of Engineers 1973)) and all engineers need to understand the fundamental principles, the grammar, of their graphical language. In fact, someone lacking that understanding is considered professionally illiterate (Giesecke, Mitchell et al. 1974).

Shaout's summary of cruise control technology (Shaout and Jarrah 1997) shows clearly the level of design reuse, generic component reuse, and specific component reuse that is achieved in the engineering designs. He states that present day cruise control systems are all feedback systems consisting of four generic elements: a vehicle speed sensor, a user interface, an electronic control module, and a throttle actuator (figure 2-55). Shaout discusses and provides additional detail for each of these generic components. However, the specifics of the components are superfluous. For engineers, knowledge of the generic system architecture and functionality of the generic components is enough to understand the design.

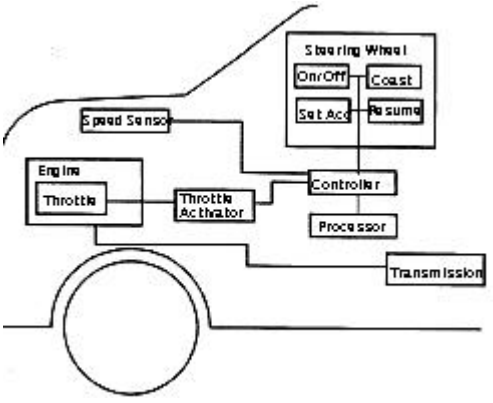


Figure 2-55: Generic Cruise Control

To summarise, the architectures of the cruise control examples exhibit four types of reuse. (1) Design reuse of feedback architectures; (2) reuse of components of functionality that need to be specified though further decomposition; (3) reuse of generic components that need further specification through parameter identification; and (4) reuse of specific components that need no additional design detail.

2.6.3 The Mathematical Modelling of System Requirements and Component Behaviour

Using the architecture of the system, the engineering designs represented the constituent components, both generic and specific, using mathematical models. Those mathematical models describe the required behaviour of the components, based on the specifications, which must be met to successfully implement the system.

> "In order to analyze a dynamic system, an accurate mathematical model that describes the system completely must be determined. The derivation of this model can be based upon the fact that the dynamic system can be completely described by known differential equations or by experimental test data. Thus the ability to analyze the system and determine its performance depends on how well the characteristics can be expressed mathematically." ((D'Azzo and Houpis 1988) pp. 21).

The engineering design examples highlight different methods of modelling the cruise control problem as the performance requirements changed and as new analysis techniques became available. Those variations include modelling system and component properties, modelling vehicle motion as a whole, modelling environmental factors, and representing models of human driving behaviour.

The first microprocessor based design, by Nakamura, provides sections explaining how the individual components of his system architecture were modelled (figure 2-28). Those models are frequency dependent, time varying transfer functions incorporating the elements to be controlled. Unknown parameters represent the values that need to be determined through analysis and experimentation to provide the specifics of the functionality – functionality that must meet the performance and stability requirements. In addition to the core control-problem specification, Nakamura's model of transfer functions incorporated aspects to represent external disturbances that must be tolerated – a model of the accelerator-link hysteresis. Unlike the models produced by software

designers, that engineering model does not represent the designer's perception of the functionality that an engineering component should perform. It is a model of the physical properties of an actual accelerator link. Similarly, the other component transfer functions are not models of the functionality that generic components should perform. They are models of the relevant behavioural properties of physical materials, which have been determined through experimental testing, and which engineering components can be made to match. The physical components exhibit properties that designers utilise as functionality[4].

Rutland utilised the same architecture as Nakamura, yet represented the components of the system using a different collection of transfer functions (figure 2-30). By realising one of the generic components in the architecture with a different physical component he eliminated the need for considering accelerator-link hysteresis, thereby removing the requirement to represent it in the system transfer function. However, because he recognized the need to consider additional environmental factors in the performance specification, his mathematical model had to extend Nakamura's model to incorporate them. The parameters of the model represent system variables such as angle of the road to the horizontal, wind velocity, load force, driving force at the wheels, throttle angle command, and the measured velocity. Interestingly, the final mathematical model involves another example of design or design reasoning reuse by incorporating the mathematical models of others.

> "A linear perturbation model was considered adequate, since the design is restricted to speed regulation at some constant value. According to Takasaki and Fenton, under small signal conditions a vehicle's longitudinal dynamics are quite velocity dependent, but for speeds greater than 77 km h^{-1}, a second order fixed parameter model can be employed to represent the dynamics between the throttle angle command and the velocity of the vehicle's wheels." (Rutland 1992).

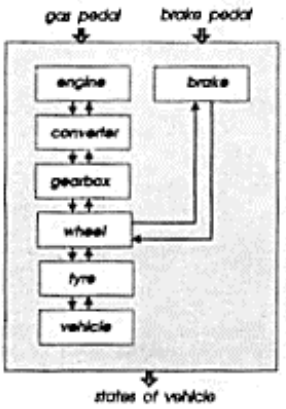


Figure 2-56: St Germann Components

Rutland reused the mathematical model of Takasaki and Fenton as the foundation of his own improved model because

[4] This is discussed in more detail in the next chapter.

it had been published as a separate, self-contained piece of research (Takasaki and Fenton 1977). Engineers recognise the need for models of systems and the environment to develop applications to exist within them. Similarly, St. Germann's design to control the complete longitudinal dynamics of a vehicle began with the rigorous development of a mathematical model for the global system. Because the model was described by large, non-linear differential equations, which are to complex to be implemented in hardware and operate in real time, the model was divided into sections each with its own separate model. The powertrain, which includes the intake manifold, engine, and hydrodynamic coupling; and the gearbox, driveshaft, tyres, and the motion of the vehicle, each had to be modeled. Those models were augmented with models of the brake and a damping system to represent air drag and rolling resistance (figure 2-56).

Finally, Ioannou evaluated his PID-based AICC system by modelling a platoon of vehicles in motion (figure 2-57) and by comparing his design with different mathematical models of human driving behaviour. Figure 2-58 (Ioannou and Chen 1993) shows the block diagram of the generic human driver model. Ioannou utilises three different models of human driving behaviour (a linear follow-the-leader-model, a linear optimal control model, and a look-ahead model) to derive the transfer functions of the block diagram. He concludes:

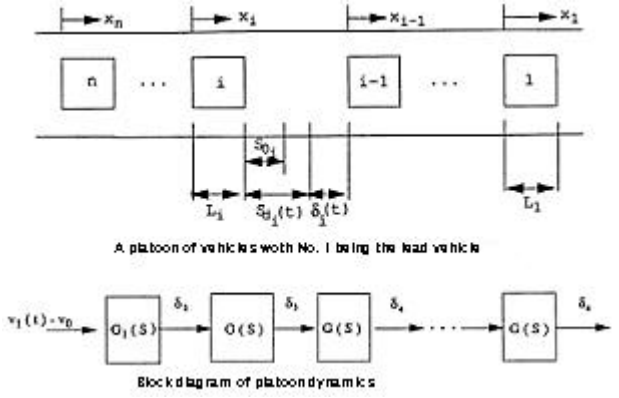


Figure 2-57: Vehicle Following Model

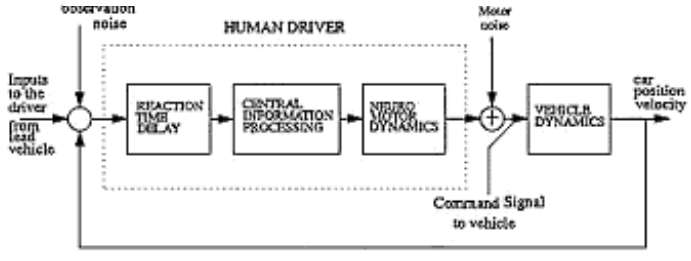


Figure 2-58: Human Driver Model

"A human driver controller can be replaced with a more sophisticated one that is based on a more realistic model of vehicle dynamics and driven by a computer and physical sensors. Computer control will eliminate human reaction time, be more accurate, and be capable of achieving much better performance. Better performance will translate into smoother traffic flows, improved flowrate, less

pollution, and safer driving." (Ioannou and Chen 1993)

These examples highlight the importance of the designer's ability to produce mathematical models of real world artefacts during the design process. The engineer can formulate the problem in terms of mathematical functionality – functionality that is known to be achievable using the components of the discipline. Solutions can then be found through iteration, analysis, and experimentation to produce an implementation that exhibits the required performance attributes. Performing the design in terms of mathematical properties results in context or domain independent development.

> "Throughout the various phases of linear analysis … mathematical models are used. Once a physical system has been described by a set of mathematical equations, they are manipulated to achieve an appropriate mathematical format. When this has been done, the subsequent method of analysis is independent of the nature of the physical system; i.e., it does not matter whether the system is electrical, mechanical, etc … [or] whether the controlled variable has the physical form of position, speed, temperature, pressure, etc…. This technique helps the designer to spot similarities from previous experience." ((D'Azzo and Houpis 1988) pp. 16&191).

The designer models particular properties of the problem space and selected generic components of the solution space. As the development proceeds, a collection of artefacts that exhibit the required component properties are combined into a complete system. That system then demonstrates the required properties of the global performance specifications. The engineer can model many properties of the system and environment and part of the skill of engineering design is to determine which properties are important to consider and to what level of detail those properties should be modelled.

2.6.4 The Use of Standard Techniques During the Design Process

Standard techniques were used to solve the mathematical models of the cruise control designs to ensure they met the performance requirements and remained stable during operation. The performance of physical systems, networks and devices can be described using appropriate differential equations. However, the classical solution to those differential equations is mathematically intensive and tedious. Moreover, if the design does not meet the specifications, it is not easy to determine from the solution of those differential equations just what physical parameters in the system should be changed to

improve the response. To facilitate the solution of differential equations and to analyse and improve the system performance, standard techniques are used to manipulate the equations, analyse the system for stability, and represent the performance characteristics in a more useful format.

The cruise control designs provide a broad range of examples that highlight the use of standard analysis techniques in the traditional engineering approach to design. Throughout the design examples, those techniques were used to validate solutions, compare alternatives, and prove the proposed solutions met the specified requirements.

Nakamura graphs the characteristics of the individual components comprising his cruise control design to ensure each one met its prescribed specification. Global system stability was then checked using traditional Bode and Nyquist methods and the parameters of the system were optimised based on those representations. Rutland recognised that the Bode and Nyquist representations could not account for system equations based on a model that incorporated variables to represent complex external disturbances. He subsequently used a different analysis method, matching, which solved the equations using a set theoretic approach. Oda et al (Oda, Takeuchi et al. 1991; Tsujii, Takeuchi et al. 1991) designed a self-tuning cruise control system that could be used to operate over a range of vehicle models and operating conditions. The system automatically adjusted the feedback gain continuously in terms of changes in a vehicle model. That vehicle model was estimated using the recursive least squares method to derive the model parameters. It worked by placing the estimated model in parallel with the vehicle and using the square of the deviation between the vehicle output and the estimated one to reduce the estimated parameters. That technique was used because it was the most basic known algorithm, requiring the least calculations. It also assured safety and reflected the accumulated design experience in the cruise control field.

In addition to using standard analysis techniques to check the performance and stability characteristics of the design, Liubakka uses standard techniques to improve the performance of the generic control system. Noting that proportional feedback control has conflicting requirements of fast response time and constant error tracking, the proportional controller needs to be augmented with integral and derivative compensation. Those techniques are recognised in the control system domain as standard methods for adjusting the system response to reduce the compromise required between the conflicting performance criteria.

Engineers realise that although many standard techniques exist,

> "… no single method is to be used to the exclusion of the others. Depending on the known factors and the simplicity or complexity of the control-system problem, a designer may use one method exclusively or in combination. With experience in the design of feedback control systems comes the ability to use the advantages of each method to a greater extent." ((D'Azzo and Houpis 1988) p. 16).

Indeed the choice of which technique to use is a notoriously difficult problem. Cruise control is seen as a benchmark problem for control system researchers to evaluate techniques. Mehra, in an introduction to a 'real-life, control design problem session' at the 1995 *Conference on Decision & Control*, made the following remarks:

> "The field of control theory is conspicuous by its lack of meaningful benchmark problems. This has led to confusion in the field regarding the relative performance of different design approaches [figure 2-59]. … Various permutations and combinations of different paradigms lead to a bewildering plethora of control design approaches. The most difficult question for an application engineer is to decide which approach or paradigm to use for his specific application and the control field offers very little guidance to the practitioner in this area." (Mehra and Baheti 1995).

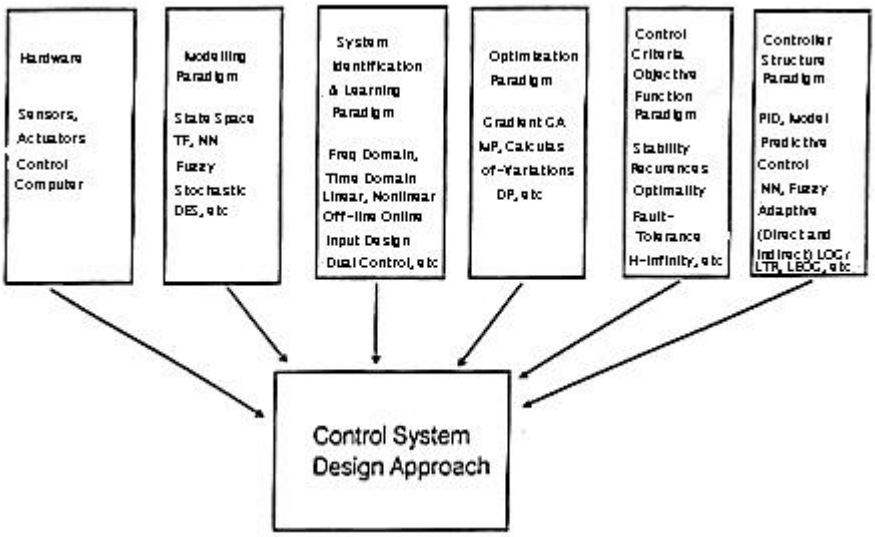


Figure 2-59: Control System Design

2.6.5 The Amount of Assumed Design and Component Knowledge

The final observation to make about the engineering development process concerns what was missing from the engineering design examples – the implementation detail. Only the publications by Ellinger and Koning provided any detail of the system implementation and that was because they were explaining the implementation rather than detailing the design. Nakamura provides a picture of the implementation of his logical ECU design (figure 2-60) and the component name of the microprocessor used to implement it. However, his design provides no additional description of the implementation detail.

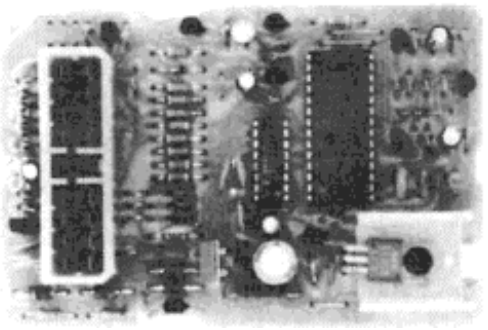


Figure 2-60: ECU Implementation

There is an identifiable division between the design and implementation phases of engineering development. The representation of the system at the end of the design process requires no further design decisions to be made before it can be implemented – even by somebody other than the original designer (Reed, 1994 in Reed 2000). Moreover, the designers are able to assume the readers of their designs have knowledge of other aspects of the system or generic engineering components so that system analysis can proceed without providing details of those aspects. The members of the engineering profession can assume their peers possess a particular level of knowledge of design, generic functionality, specific components, and analysis techniques to allow information not explicit in the published design to be inferred.

2.6.6 Summary

Summarising the process, the designer begins by developing a system architecture. The architecture is based on a generic feedback control system incorporating a number of generic and specific components from the discipline. The designer analyses the requirements and represents them as mathematical transfer functions. Those transfer functions are based on the ability to represent, mathematically, properties or functionality of generic control systems – an ability that is part of traditional engineering education. The process of design subsequently becomes the process of determining the specific functional properties of the generic components, which when synthesised, combine to emulate the properties of the global transfer function. The models used to represent the

generic components contain many unknown variables that are determined using standard analytical techniques and experimentation and provide validation of the design.

2.7 Comparison of Design Approaches

The most striking observation that comes from the presentation of the specific design examples and the generation of the generic approaches is just how different the process of design is in the respective disciplines. The software designs all exhibit a similar pattern. The initial solution was presented in a particular modelling formalism. The designer then proceeds to implement the concepts and relationships that constitute that model using auxiliary models to capture the properties not present in the primary representation. The engineering designs also exhibit a common pattern. The engineer developed an initial representation of the solution that consists of generic functionality known to the discipline. Moreover, those components are organised in a generic structural arrangement that is also well known in the discipline. The components of the architecture were modelled mathematically to depict precisely the required functionality of the proposed design. Those representations were then refined using further modelling and analysis tools to detail, identify, and validate the components required to implement that specific functionality.

There are many aspects of the engineering design approach that are appealing for software developers and it is easy to see why considerable research effort has been spent in search of an engineering approach to software design. The engineering designs incorporate substantial amounts of design and component reuse. They use rigorous mathematical techniques to provide quantitative analysis of their designs. There is evidence of a knowledge base of engineering design expertise that can be assumed of their fellow practitioners and relied upon to facilitate the understanding of design rationale. Finally, the engineering designs are able to achieve context independent development. The successful application of these aspects of design would produce significant benefits for the discipline of software development.

The terms ‘component’, ‘design’, ‘system’, ‘architecture’, ‘feedback’, ‘modelling’, ‘implementation’, and ‘reuse’[5] are all evident in both disciplines and their existence

[5] Interestingly, a mechanical engineer who works in automotive design reviewed this chapter for me. He had difficulty understanding the notion ‘reuse’ in software engineering. The use of existing

makes analogies between the disciplines sound plausible. Consider the following syllogism as an example,

> Traditional engineering disciplines design systems incorporating considerable design and component reuse by utilising quantitative, mathematical analysis techniques. The discipline of software engineering designs systems and recognises the potential for considerable design and component reuse. Therefore, software engineering can improve its development process by incorporating quantitative, mathematical analysis techniques.

Subscription to these types of syllogisms is fraught with danger. Important differences exist between the disciplines and they need to be considered before any conclusions can be made based on perceived analogies. Moreover, before determining if software development can or should be like traditional engineering development, the reasons why those differences exist between the disciplines must be examined.

One essential difference between the engineering and software development designs concerns the utilisation of modelling techniques in the design process. By viewing the design processes from the point of view of how they utilise models and what properties they represent, it is possible to see how the disciplines differ.

Summarising the definitions of (Cybernetica ; Tjalve, Andreasen et al. 1979), a model is an object, process, device, scheme, or procedure that shares crucial properties of an original, modelled object or process but is easier to manipulate or understand because it highlights the properties of interest, whilst omitting the remainder. These models consist of a type (mathematical, causal, dynamic, stochastic, etc), the properties that it highlights (function, structure, dynamics, etc), and its use (simulation, verification, investigation, etc). However, a model may also contain properties not found in the original object, for instance, the physical form of a functional model may be different to that of the original object. When modelling it is important to be conscious of the differences so that the most appropriate model for the situation is chosen. During systems analysis, the model usually aspires to represent the real world or the relation between some observed phenomena in the real world and those important properties should be specifically highlighted in the model.

---

components was such a natural thing for engineers that he couldn't imagine us having such problems with it.

The models used in the engineering design examples are mathematical representations of the functionality of their discipline. The unknown parameters of those models represent the system variables of the problem domain but the mathematical relations themselves represent achievable functionality of the engineering components. The designer knows that engineering components can be used to exhibit the functionality of particular mathematical relationships. The goal is to represent the problem in a similar mathematical relationship and solve the parameters of those equations. For example, Rutland's control system architecture was refined with a collection of transfer functions. Those transfer functions represent mathematical manipulations of input signals that can be implemented using known engineering components. The parameters of the transfer functions represent the domain concepts of velocity, wind speed, throttle angle, and angle of the road to the horizontal, etc. However, once those domain concepts are represented as parameters of mathematical relationships, the design process becomes a matter of developing functionality to solve the mathematical problem. The design becomes independent of the problem domain.

The reason engineers work in this fashion is because they can only build using the artefacts provided by their discipline. The evolution of those engineering building blocks began as simple functional manipulations of the underlying properties of the discipline. For the electronics domain of the microprocessor-based cruise control designs, those underlying properties are electronic signals and the concepts of voltage, current, resistance, inductance, and capacitance. The intricate operations of modern electronic components are merely complex aggregations of a small number of functional operations that can be applied to this small set of properties. Furthermore, because the interactions between these small number of underlying concepts can be represented by mathematical relationships, the complex aggregations can also be represented mathematically. This is evident in the structure of the design textbooks used by engineers. For example, D'Azzo's text, *Linear Control System Analysis and Design* (D'Azzo and Houpis 1988), begins with a presentation of the mathematics of simple electronic circuits. It details how simple configurations of resistors, capacitors, inductors, voltage sources and current sources are modeled using mathematical relationships.

> "In order to analyse a dynamic system, an accurate mathematical model that describes the system completely must be determined." (D'Azzo and Houpis 1988)

The level of detail of D'Azzo's mathematical analysis of physical systems is extensive and provides many ways of using mathematics to represent systems, system properties, and techniques to simplify their analysis. With this mathematical foundation the text proceeds to the theory of linear control systems. The mathematical understanding of complex control systems is presented in terms of the mathematical foundations that were developed for the simple systems. Indeed, all of the engineering design texts used in this study began with sections concerned with mathematical modelling of basic electronic circuits before they proceeded to more complex design theory. Furthermore, the basis of engineering education courses is also the development of mathematical representations for the simplest components and systems of the discipline.

In contrast to the engineering designs, the software developers used different design methodologies, software architectures, or patterns for development as a means of developing an initial view of the problem. The software engineer does not produce models of existing designs or components. The first step of the software development process is the creation of a model of reality, specifically, the reality confined by the bounds of the problem to be solved.

Software developers are not constrained by their implementation medium. They work with a medium that allows them to implement almost any concept, so long as it can be sufficiently well defined. This is in contrast with traditional engineers who can only implement using the components of the particular engineering discipline and whose functionality is constrained to the possible aggregations of underlying properties of that discipline. For the software developer, there are many ways of modelling aspects of reality and there are many levels of generality for those models. Moreover, because of the flexibility of the software implementation medium, almost any of those models can be successfully implemented. The purpose of design methods, software architecture styles, and design patterns is to assist in the creation of the initial model of the problem by constraining the designer to developing one that is relatively easy to implement in our implementation medium.

A speculative hypothesis may be that it is possible to view the actions of software developers as similarly modelling the problem to match the properties of the implementation medium. The difference between the disciplines is that the properties of the respective implementation mediums are different. The software designs model the problem using functional and behavioural viewpoints and the implementation structure of

those viewpoints. The implementation medium of the software developer is the programming language. It could be speculated that design methods assist the problem solving process by providing functional and behavioural representations because they match the information flow constructs of programming languages: sequence, iteration, and selection. Furthermore, the structural properties of the implementation depend on the concept structures of the programming language, such as procedures, rules, and objects, and the execution constructs of the operating environment, such as threads, processes, and distributed systems. This hypothesis however, requires further investigation.

In summary, the difference between the respective disciplines is due to the fundamental nature of the implementation mediums they build with and, subsequently, the fundamental nature of the systems they construct. Engineers design physical artefacts. Software developers implement models of reality. Traditional engineers use modelling techniques to represent the properties of a possible implementation, based on known, generic functionality. Software developers use modelling techniques to develop models of reality that can be implemented in computer programs. This conjecture explains the differences observed during the case study. Those differences include the purpose of the published designs, the differences between the initial models and the differences between designs that utilised similar modelling formalisms. These are now briefly described.

The intended purpose of the design publications in the respective disciplines was also different. The software designs were used to illustrate a particular design methodology or style of software architecture. Conversely, the intention of the engineering publications was to present a solution to the cruise control problem. The authors of the majority of the engineering designs came from automotive engineering companies, not academia. Furthermore, in Shaout's review of cruise control technology, the majority of the references came from patent applications. In the software design examples, the use of the cruise control problem was secondary to the explanation of the design methodology. The exception was the design of Jones (Jones 1994). That design is interesting because it is a software design performed by an electronic engineer. It begins with an attack on the software approach to cruise control design and illustrates the difference in emphasis between the designs of the respective disciplines.

> "I attended a software conference where one of the speakers gave a presentation prescribing a commonly accepted software development method … by designing an automotive cruise control. The design was terrible but he

> was proud of it! … His academic expertise was an excellent example of how software development often puts the emphasis in the wrong place, ignoring the real problems that result in catastrophic failure.
>
> The speaker who provoked my wrath obviously knew a lot about computer science, but very little about physical science, system engineering, testing, economics, safety, and closed-loop servo systems. He approached the cruise control design as if it were a software design problem. His design method stressed how functions should be grouped with other functions in tasks or packages based on temporal sequence or information-hiding criteria. He drew a context diagram. He drew a state-transition diagram, and then drew several levels of data-flow diagrams. He created a task-structure diagram. Then he drew a system architecture diagram. He partitioned the architecture into task and package structures. By the time he was finished, he had an impressive looking but totally impractical design consisting of seven tasks, a message queue, and four asynchronous interrupts. …
>
> The problem is viewed as one of information processing or program structuring instead of system engineering." (Jones 1994)

This difference in approaches stems from a fundamental difference between the disciplines. The task of the engineer is to utilise materials, components, and systems of the discipline to solve real-world problems. Their research attempts to develop improved methods of modelling their problems so they can be solved using functional properties of those materials, components, and systems. Alternatively, the task of the software developer is to implement models of reality to automate some perceived process. Software development research is concerned with improved ways of modelling reality for more effective implementation, maintenance, and reuse. Methodologies are generic problem-solving processes that facilitate the implementation of theories that are devised to explain 'real-world' processes. The difference is exemplified in the implementation of the controller mechanisms of the respective cruise control systems. Ellinger describes the mechanical flyweight governor speed control unit (figure 2-26) as follows:

> The act of setting the desired speed causes the activation of an electromagnetic solenoid. The activated solenoid causes an armature to block off an air flow port. The restriction of the air flow causes vacuum to increase

> in the controller housing. The increased vacuum in the controller housing results in increased vacuum in a power servo unit. The increase in vacuum in the power servo unit causes the throttle to open further.

This description of the controller unit's operation shows how the properties of the discipline (electromagnetic force, mechanical motion, air vacuum) are utilised by the designer to achieve the desired result. In contrast, the following pseudo-code fragment could represent the logic for a software based controller unit:

```
If (system_activated)
   Error = Desired_Speed – Current_Speed;
If (Error > 0)
      Increase_Speed(Error);
```

The conjecture that engineers build artefacts while software developers build models of reality also explains the observation made concerning the labelling of some software systems as feedback control systems. All of the engineering designs used a similar generic feedback architecture. The designs differed in terms of (1) the types of components used to realise the generic functionality; (2) the issues considered when modelling the system mathematically; and (3) the mathematical formalism used to model the problem. In contrast, the software designs differed in the primary formalism used to develop the initial model and in the specifics of that initial model when the chosen formalism was applied. All of the software designs were feedback systems because they all used the current speed to manipulate the system control. The difference was in the types of concepts used to represent the problem (the design formalism) and the specificity of the concepts identified within that formalism. For example, in an object-oriented formalism, the designer is free to choose which objects constitute the designer's perception of the problem. The designer can choose between the specifics of the problem domain, as Booch did, or a mental model of a more general representation of the problem, for example Shaw's feedback control system. They are all feedback loops, they are just at different levels of generality.

No software design is more perspicuous than the others are. Shaw's claims that object-oriented and process control feedback loops provide a closer match of reality than functional decomposition is ill-founded. They appear more perspicuous when the understanding of software systems begins from the assumption that software systems are

analogous to traditionally engineered corporeal systems. However, by understanding software systems as implemented models of reality, all of the software designs implement the generic notion of a feedback system, they simply use different collections of mental concepts to do it. As a final observation, the model-building conjecture shows that the generally accepted notion that object-orientation is beneficial because it allows developers to implement their models of reality is also ill-founded. The discussion of philosophical foundations in Chapter 5 explains these comments in greater detail.

2.8 Conclusion

Can analogies with traditional engineering disciplines be used to improve the process of software development? The answer is neither yes nor no. This study has shown that the question is too simplistic. The study presented what developers do when they design. It then developed a generic approach to how the design proceeds. Finally, it presented why the respective disciplines designed using that approach, explaining the observed differences. Compared with traditional engineers, who build physical artefacts, software developers implement models of reality – explanatory theories of real-world processes. Software developers would like to 'engineer' their 'systems' using an analogous approach to traditional engineering development. They would like to use the same how approach to design. However, this detailed study shows that significant differences between what the respective disciplines design makes it extremely difficult to make valid analogies, regardless of how plausible they may sound.

It may be possible to improve the software development process by examining how traditional engineers work. However, any attempt to do so must consider the differences that exist. What we build is nothing like traditionally engineered systems. They are models of reality, theories of how concepts and relationships should interact to solve a problem. The concepts that comprise our systems are limited only by our imaginations. Engineers do not model reality in the same way. They are constrained to developing systems using the components of their discipline. Those materials and components are aggregations of a small set of possible manipulations to a handful of domain-level concepts.

This study has based a conjecture about the fundamental differences between traditional engineering and software development on a relatively small case study – the automotive cruise control system. The design of linear control systems is a small discipline within the

global sphere of engineering design and it would be foolish to base such an all-encompassing conjecture on the design approach of one engineering discipline. However, the conjecture is not based on the difference between the design approaches, it is based on the reasons why those approaches were used. Those reasons are applicable to all fields of engineering. Some engineering disciplines, such as civil engineering or chemical engineering, are more materials-based than the component-based nature of the control system domain (Reed, 1996 in Reed 2000). However, the design techniques they rely on are still based on mathematical models of the properties of the underlying materials of their discipline. For example, see Currie (Currie and Sharpe 1990) for mathematical models of civil engineering materials and their structural arrangements. Software development implements models of reality in computer programs. Traditional engineers utilise properties of physical materials and their structural arrangement to suit a desired purpose.

To use traditional engineering disciplines as a source of ideas for improving the software development process we must consider the disciplines in term of what they design and build as well as how they design and build. Software developers want the ability to use rigorous mathematical techniques to analyse quantified design criteria. Furthermore, they want to achieve the same high-levels of design and component reuse, if possible, through context-independent development. Are these issues achievable given the fundamental nature of the systems built by software developers? This study highlights new research questions that need to be examined before that question can be answered. They are:

- Why can engineering components be modelled mathematically, and can a similar approach be achieved in the software implementation medium?
- What does it mean to build explanatory theories of reality?
- What affects our ability to create models and how can that be utilised to improve the software development process?
- If the initial model, or software architecture, sets the path for subsequent development, what model-building issues affect its creation? Can it be influenced by qualitative design criteria, such as design-for-modifiability or design-for-performance, or is it a subconscious process that cannot be manipulated?

- Are there areas of contention in software engineering research that can be solved by examining the philosophical foundations those theories are based on?
- If the design approaches of the respective disciplines are so different, why have analogies between them been used as the source and validation for software engineering research ideas?

These questions can be answered using research from other disciplines. An understanding of the evolution of traditional engineering disciplines and their systems/components can be used to determine if software components can be devised to allow the same design approach. The history and philosophy of science can be used to develop an understanding of the process required to create and validate models of reality. Metaphysics and epistemology provide theories for explaining why people have different perceptions of reality. Finally, theories about conceptual development in the discipline of psychology can be used to understand how we devise the concepts that comprise those models/theories.

Finally, a study of traditional engineering may allow researchers to improve the way in which software developers approach the design process. However to validate any attempt to use analogies with traditional engineering, software development must develop an improved understanding of the fundamental nature of the systems that it builds. Indeed, it needs an improved understanding of the philosophical underpinnings of the discipline as a whole. Research in the philosophy of science not only provides illumination concerning the nature of the systems we build but also how the discipline of software engineering is progressing. Different philosophy of science theories explain how scientific disciplines evolve through phases of progress as the discipline changes its philosophical understanding of the systems it attempts to understand[6]. For example, Kuhn explains scientific progress through a series of evolutions and revolutions (Kuhn 1962). Though Kuhn's explanation is not the only one, researchers in the philosophy of science agree that progressive scientific theories are relative, to varying extents, on the underlying guiding assumptions of the discipline. The discipline of software engineering has progressed to its current state based on an implicit understanding that software

[6] see Chapter Six for more information.

development is analogous to traditional engineering development. This study has shown that assumption is inadequate. To improve the progress of software engineering it is time to develop a better understanding of our discipline.

3. A History of the Artefact Engineering View of Software Development

3.1 Introduction

The cruise control comparison identified many significant differences between the design approaches of software development and traditional engineering disciplines. It then conjectured the reason for those differences was due to fundamental differences between the types of systems built and the implementation mediums used to build them. It concluded by highlighting many questions to be answered to evaluate that conjecture. One of those was, if the design approaches of the respective disciplines are so different, why have analogies between them been consistently used as the source and justification for software engineering research ideas? To answer that question this chapter presents and analyses a history of the artefact engineering view of software development. That analysis examines the arguments used by software engineering researchers to determine their understanding of both disciplines.

Wegner notes that as early 1950 software developers recognised the importance of subroutine libraries for capturing and reusing subprograms in software development (Wegner 1984). In the early 1960s, Fred Brooks and Jerry Weinberg discussed the appropriateness of the term 'architecture' for describing structural design issues in computer systems. Brooks had been working in the area of computer architecture (Brooks 1962) and was worried about the appropriateness of the analogy. However, as their discussion progressed it seemed to hold (Coplien 1999b). At that time their discussion considered computer systems as both hardware and software, in contrast to the more software-centric analogies used in recent times (Weinberg 2000). In addition, their concept of software architecture included the interface with the computer operator as well as the large-scale system structure (Weinberg 2000). That aspect is also evident in Brooks' later comments on the integrity of the system architecture.

> "By architecture of a system, I mean the complete and detailed specification of the user interface." (Brooks 1975)

Coplien notes therefore, that as early as 1965 the discipline of software development was already enough on its feet to consider the influence of design theories in other artefact construction disciplines (Coplien 1999a). Nevertheless, it was the NATO conferences on

software engineering in the late 1960s that provides the first formal expression and debate of software engineering ideas. The transcripts of those debates provide the starting point for the history and analysis presented. The analogies and insights used by those conference participants are analysed to determine the validity of the analogies used and to identify the participants understanding of the fundamental nature of both software systems and hardware systems. A selection of the research presented between that time and the present, which promotes the view of software development as an artefact engineering discipline, is then presented and evaluated in the same way.

The conclusion, and answer to the originally posed question, is that the label 'software engineering' was proposed as a starting point for discussion at the 1968 NATO conference. Its suggestion was intended to provoke ideas for improving software development. However, despite numerous unresolved questions concerning the applicability of that metaphor, its implied way of understanding software development was tacitly accepted. The series editor of the NATO conference noted,

> "This book points the way to the future of software. It examines our shortcomings in software practice and technique and suggests alternatives that could overcome many of the problems. But most important, it lays out, by implication, the frame of mind that we take to produce dependable software." (NATO 1976a)

That frame of mind is the artefact engineering view of software development.

As the discipline of software engineering progressed, its proponents rarely questioned the artefact engineering view although numerous anomalies appeared when using that view to develop research ideas concerning a discipline of software engineering. The most persistent anomaly, which has never been satisfactorily explained, concerns the underlying principles of software systems. The assumption made by the relevant researchers is that the underlying principles would eventually be discovered or artificial intelligence techniques would be developed to overcome the anomalies identified.

From the analysis presented it is clear that researchers promoting the artefact engineering view of software development did not have a thorough understanding of software development, traditional engineering, or the relationship between them. That is not a criticism of those researchers. It is simply a recognition of the growing body of knowledge concerning the relevant issues that can now be applied with the benefit of

hindsight. Moreover, fundamentally important decisions concerning the understanding of a discipline are made in similar ways in most professional disciplines[7]. However, to develop a better understanding of software development and its relationship with traditional engineering, a thorough understanding of the underlying principles of both disciplines, and their relationship, is required.

3.2 In the Beginning: The NATO Conferences

The NATO conferences on software engineering originated in 1967 when a study group on computer science, set up by the NATO science committee, recommended holding a working conference on software engineering that would focus on the problems of software. Specifically it would address issues pertaining to the design, implementation, and maintenance of software.

> "The Science Committee conferences are deliberately designed and structured to focus expert attention on what is *not* known rather than on what is known. The participants are carefully selected to bring together a variety of complementary viewpoints. Through intensive group discussion, they seek to reach agreement on conclusions and recommendations for future research that will be of value to the scientific community." (In the Preface. All quotes from the 1968 and 1969 conferences are taken from (NATO 1976a; NATO 1976b))

The background of the first conference and the working papers generated by it give the first indication of the software development community's attempt to understand the fundamental nature of the discipline. The first is the choice of the term 'software engineering' itself.

> "[It was] deliberately chosen as being provocative, in implying the need for software manufacture to be used on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering." ((NATO 1976a) p. 5)

At that stage in the discipline's evolution, many large software systems had been developed, for example the OS/360 project, and many lessons learned. The community's perception was that the rapidly increasing importance of software systems in many

[7] This is discussed in detail in chapter six.

activities of society, and the increasing size and complexity of those systems, required significant improvement in the way those systems were produced.

The difficulty in establishing the philosophical assumptions of the NATO conference participants, and present day software engineering researchers, is that these philosophical issues are very rarely discussed explicitly. Therefore, in the absence of unambiguous statements, the only means of capturing their fundamental conception of software and its development is to extract it from the ideas that were presented. In the NATO conference reports, two types of comments can be used to gain an indication of the variety of conceptions that researchers had of software and software systems. They are analogies and insightful observations. In the 1968 conference alone, analogies were made between software development and aircraft design, civil engineering, mathematics, logic, automobile design, frame stressing, and even musical education. For those analogies to be perceived as valid, certain assumptions about the nature of software and engineering systems and their development must be made. For example, the following extract, taken from McIlroy's often quoted *Mass Produced Software Components* paper, is indicative of many analogies used in software engineering research – both then and now.

> "McIlroy: We undoubtedly produce software by backward techniques. We undoubtedly get the short end of the stick in confrontations with hardware people because they are industrialists and we are often the crofters. Software production today appears in the scale of industrialization somewhere below the more backward construction industries." ((NATO 1976b) p. 89)

McIlroy's decision to make an analogy between the production of software systems and those systems produced by 'hardware people' was based on the desire to have a software development process that exhibits the same tractability that traditional engineering development appears to have. However, for the comparison to be valid, the nature of software must contain characteristics that make it possible for systems to be constructed in a similar fashion to hardware systems. That is, the respective materials, components and their means of interaction, must be analogous. The question that needs to be asked is: Is that in fact the case?

It is interesting to note that the conference report also contains the first analogies between software design and the design theories of Christopher Alexander, whose theories of

design in building architecture are quite popular in present day software engineering research.

During the course of the conference the participants made many philosophically insightful observations about the nature of a particular topic of discussion. That may have been about a specific design method or a comment made by the editors about the state of the current discussion. Those insightful comments were often followed by vigorous discussion of a more theoretical nature rather than the usual pragmatic issues. For example, Fraser made the following comment:

> "Fraser: One of the problems that is central to the software production process is to identify the nature of progress and to find some way of measuring it. Only one thing seems clear right now. It is that program construction is not always a simple progression in which each act of assembly represents a distinct forward step and that the final product can be described simply as the sum of many sub-assemblies." ((NATO 1976a) p. 7)

That comment provides an indication that the nature of software and software systems is not something which is easy to grasp. While it may be assumed that engineering systems can be designed and implemented in terms of pre-existing components, this insight shows that those concepts cannot easily be transferred to software systems. Something uniquely fundamental about the nature of software systems, which is not quite explicitly understood, is brought closer to the surface of our understanding by these comments.

Many of the discussions associated with those insights resulted in confusion between the participants. As the discussion delved into more philosophical issues there was a lack of common understanding between the participants concerning the exact meaning of words. Confusion also arose concerning how those theoretical concepts were applicable to specific aspects of software development. For instance, the discussion concerning the logical completeness of software systems resulted in a discussion of what the participants meant by the term logical completeness. After much discussion the following comments were made:

> "Genuys: I think I would just prefer another term because this one has a certain logical flavor, and I'm not certain that...

> Perlis: (Interrupting) The word 'logical' has a meaning outside the realm of logic, just as the word 'complete' does. I refuse to abrogate to the specialist in mathematics the word 'completeness' and in logic, the word 'logical'.
>
> Bauer: The concept seems to be clear by now. It has been defined several times by examples of what it is not." (op. cit p. 26)

The concept of 'logical completeness' was identified as an important design criterion and discussion ensued to determine how to apply that concept to the actual practice of system design. However, the discussion eventually veered away from that goal and centred on the semantics of the words 'logical' and 'completeness'. Whilst that diversion into semantics was necessary, the debate failed to return to the original goal, that of determining how to apply 'logical completeness', in any of its possible meanings, to specific design situations. The best that was achieved were generic statements such as "ensuring that a system was capable of performing a 'basic' set of operations".

Those debates, and the insightful observations about pragmatic issues, highlighted the need for a better theoretical understanding of the discipline. However, as the discussion worked towards important answers the issue would often be abandoned due to the difficulties inherent in debating such esoteric concepts. That abandonment of the theoretical debates gives the impression of placing the philosophical foundations of software engineering into the 'too hard basket'.

Attempting to discover what others believe are the philosophical underpinnings of the discipline is a difficult task, though one that must be addressed. Interpreting the views of others is naturally fraught with danger due to the ambiguity of natural language and the problem of access to the actual people concerned. In addition, there is the potential for the community-wide understanding of terms such as 'design method', 'high-level language', and 'module', to change over time as the discipline evolves and matures. Moreover, using those terms and theories as the basis for extracting people's philosophical views is subject to the prejudices and philosophical biases of the researcher. The only means of minimising the risk of misinterpretation is by identifying those potential sources of error and maintaining a vigilant watch to ensure their effects do not taint the conclusions reached. That effort was consciously made, however to what level of success is hard to gauge.

The nature of the source material also provides a potential source of error. The conference proceedings provide edited transcripts of the conference discussions as well as copies of the invited addresses and, unfortunately, only a few of the working papers in full. The editors decided to structure the proceedings following the "normal sequence of steps in the development of a software product, from project start, through design, production or development, to distribution and maintenance" (op. cit p. 3). Those steps were augmented with sections about 'Software Engineering and Society', 'The Nature of Software Engineering', 'Education', and 'Software Pricing'. The editors used transcripts of the discussions, which were recorded by stenographers and captured on magnetic tape, to place content into that structure. The printed comments are also interspersed with excerpts from all of the working papers where they were relevant. While the editors attempted to retain the "spirit and liveliness" of the conference by retaining the original wording and context as much as possible, any attempt to extract the philosophical assumptions of the participants must also consider the editors influence on the source material. Despite these inherent difficulties, the analogies and insights provided useful material for showing how the philosophical assumptions of the delegates at the NATO conferences shaped the origins of software engineering understanding.

The insights and analogies are presented in the order depicted in the proceedings, using additional comment to put them in context. They are then examined to determine what assumptions they make about the fundamental nature of software and software systems and later in the chapter they are analysed in detail using additional information from the engineering disciplines that they purport to identify similarities between. That analysis questions the validity of those analogies and consequently begins to question the assumptions made about the fundamental nature of software systems.

3.2.1 The 1968 NATO Conference

The first section of the proceedings, *Software Engineering and Society*, collates discussions concerning the growing importance of software systems and the more general problems faced by the research community. The first excerpt, by Graham, is part of a discussion about the nature of progress in software development and the inability to successfully predict and measure it.

> "Graham: Today we tend to go on for years, with tremendous investments to find that the system, which was not well understood to start with, does not

> work as anticipated. We build systems like the Wright brothers built airplanes – build the whole thing, push it off the cliff, let it crash, and start over again." (op. cit p. 7).

The analogy with the Wright brothers highlights a belief that is evident in the editorial comments included in that section.

> "*There was general agreement that 'software engineering' is in a very rudimentary stage of development as compared with the established branches of engineering.*" (op. cit p. 7)

That belief assumes we *can* consider software development to be an engineering discipline and that the early phases of those engineering disciplines were rudimentary in an analogous manner to the early stages of software engineering. Gillette explicitly expresses this belief in the ensuing comment. He suggests that, like the aircraft industry, as the software development discipline evolves it will become better at specifying its systems and estimating its development schedules.

> "Gillette: We are in many ways in an analogous position to the aircraft industry, which also has problems producing systems on schedule and to specification. We perhaps have more examples of bad large systems than good, but we are a young industry and are learning how to do better." (op. cit p. 7).

The validity of the analogies concerning the early stages of the disciplines is analysed later in this chapter. However, in the same discussion Fraser provides the comment, which was used as an example earlier, that suggests the nature of the software systems may not be similar enough to those other disciplines for the analogies to be well-founded. He notes that the nature of software systems cannot be described "simply as the sum of many sub-assemblies", however, no further discussion is presented that debates the differences between software designs and, in these cases, aircraft designs.

Later comments continue to highlight specific aspects about the nature of software systems that may not be present in traditionally engineered systems. Kinslow makes the first while discussing management issues in software development.

> "Kinslow: There are two classes of systems designers. The first, if given five problems will solve them one at a time. The second will come back and announce that these aren't the real problems, and will eventually propose a

> solution to the single problem which underlies the original five." (op. cit p. 13).

Why do these different types of designers exist? Believers in the engineering discipline of software development, including Kinslow himself, describe the first type of the designer as the "system-type" who employs what is considered to be an 'engineering-mindset' to the problem solving process. However, the analysis of cruise control systems presented in Chapter 2 suggests the nature of the systems built by software developers and traditional engineers are fundamentally different. Therefore, an alternative reason why these different types of developers exist may be based on the fundamental nature of software systems rather than the 'system-type' and 'engineering-mindset' labels. Furthermore, the role of training in the respective disciplines may also exert a powerful influence (Reed, 1993 in Reed 2000). Unfortunately, because no thorough comparison of the fundamental natures of software and traditionally engineered systems exists in the research literature, arguments for particular hypotheses are difficult to justify and objectively compare.

Kinslow's remark is followed in the proceedings by an excerpt from the working paper by Berghuis. That excerpt identifies a significant difference between software systems and those produced by other 'system-engineering' disciplines.

> "Berghuis: Independent software packages don't exist; they run on an equipment (hardware), they need procedures by which to be operated and that indicates that we have to define what a system, project, phase of a project, releases, versions, etc., mean. Also we have to consider the organisation from the point of view of developing systems and in fact we are faced with the differences between functional and project organisation. We are also faced with the difficulties of system-engineering." ((NATO 1976a) p. 13).

That comment highlights the principle of 'system execution' – a principle that does not exist in any other engineering discipline. Software systems are the only 'engineered' systems that do not realise the original design requirements until the statements of the implemented system are executed by a machine[8]. Unfortunately, no further discussion of this important insight by Berghuis is evident in the proceedings. Moreover, the full text of his working paper was not included.

---

[8] This issue is discussed in more detail in Chapter 4.

A similar distinction between software systems and engineered systems is highlighted later in the proceedings. The editors collated a number of discussion points related to the distinction between design and production (implementation) of software systems. Significantly, the editor's note that the "appropriateness of the distinction between design and production was contested by several participants" (op. cit. p. 18), however the distinction was retained in the conference report to expedite the publication of the report. The discussion begins with an excerpt from the working paper of Naur that attempted to clarify the distinction.

> "Naur: … The distinction between design and production is essentially a practical one, imposed by the need for a division of labour. In fact, there is no essential difference between design and production, since even the production will include decisions which will influence the performance of the software systems, and thus properly belong in the design phase. For the distinction to be useful, the design work is charged with the specific responsibility that it is pursued to a level of detail where the decisions remaining to be made during production are known to be insignificant to the performance of the system." (op. cit p. 18)

Naur's comments highlight the implied analogy with traditionally engineered systems – the need for a division of labour between design and implementation. No further comment is made by Naur regarding why a division of labour should occur in an analogous manner to other engineering disciplines. It is assumed that because the division is useful for software project management reasons, the nature of software systems should allow such a distinction to be possible even though it is recognised that the distinction is more arbitrary than in other disciplines. Unfortunately, the full working paper from which the extract is taken, *The Profiles of Software Designers and Producers*, is not published in the report. Other participants however, did highlight aspects of software system production that question the ability to implement the distinction, even if management forces require it. First, Dijkstra notes the difficulty in implementing the distinction because the correctness of the program cannot be guaranteed until the structure of the system is implemented. Therefore, any artificially enforced distinction between design and implementation will merely hinder the ability to "do a decent job" (op. cit p. 18). Furthermore, Kinslow identifies the iterative process required to develop systems as

another reason why the distinction is difficult, if not impossible, to successfully implement.

> "Kinslow: … If you are writing a large production project, trying to build a big system, you have a deadline to write the specifications and for someone else to write the code. Unless you have been through this before you unconsciously skip over some specifications, saying to yourself: I will fill that in later. You know you are going to iterate, so you don't do a complete job the first time. Unfortunately, what happens is that 200 people start writing code. Now you start through the second iteration, with a better understanding of the problem, and it is too late. That is why there is a version 0, version 1, … version N. If you are building a big system and you are writing specifications, you don't have the chance to iterate, the iteration is cut short by an arbitrary deadline. This is a fact that must be changed." (op. cit p. 18)

Kinslow notes that iteration is required because designers need to develop a correct understanding of the problem and this cannot be achieved until the design process has been traversed on more than one occasion. It may be possible to argue that this iteration is an iteration of the design phase and comes before implementation (Reed, 1993 in Reed 2000). However, the previous point discussed that the distinction between the design and implementation phases of software development is not as clear as it is in traditional system development. A separate part of Naur's comment exemplifies this insight in the software development process. Although the term 'flowchart' may be outdated, it can be successfully replaced with any current design method and still be relevant.

> "[Naur] In my terms design consists of:
>
> Flowchart until you think you understand the problem.
>
> Write code until you realise that you don't.
>
> Go back and re-do the flowchart.
>
> Write some more code and iterate to what you feel is the correct solution." ((NATO 1976a) p. 18)

Unfortunately, no debate is generated to determine why this iteration process is required in the software development process yet the distinction between design and implementation can be maintained in engineering disciplines. Presumably, it is because

these issues would be resolved as the 'software engineering' discipline matured[9]. Interestingly, Ross concludes the discussion in the conference report with the most explicit comment about the difference between software and other engineered systems. Again though, it appears not to have lead to any subsequent discussion that may have uncovered the significant differences between software and other engineered systems.

> "Ross: The most deadly thing in software is the concept, which almost universally seems to be followed, that you are going to specify what you are going to do, and then do it. And that is where most of our troubles come from. The projects that are called successful have met their specifications. But those specifications were based upon the designers' ignorance before they were started." (op. cit. p. 19).

The observation by Ross about the importance of the 'concept' is similar to the one made later by Brooks concerning the importance of conceptual integrity in the design process (see *Aristocracy, Democracy, and System Design* in (Brooks 1975)).

The topic of the proceedings then changed to the mindset required by a software developer and this provided the next analogy with traditionally engineered systems. The editors quote another excerpt from the previously discussed working paper by Naur. This provides an analogy with the large and complex systems designed by architects and civil engineers and, in turn, provided the first reference to the design theories of Christopher Alexander in software engineering research.

> "Naur: … software designers are in a similar position to architects and civil engineers, particularly those concerned with the design of large heterogeneous constructions, such as towns and industrial plants. It therefore seems natural that we should turn to these subjects for ideas about how to attack the design problem. As one single example of such a source of ideas I would like to mention, Christopher Alexander: Notes on the Synthesis of Form." ((NATO 1976a) p. 20)

It appears obvious to Naur that software developers should look at the design methods of other engineers, in this case civil engineers and architects. Those disciplines have developed techniques to design and build large, complex, heterogeneous systems.

[9] Chapter 5 discusses these issues in detail from a philosophical perspective.

Software designers also design and build large, complex, heterogeneous systems. Therefore, we should look at those disciplines for guidance. For the analogy to be valid, it is assumed that the meanings of 'large', 'complex', 'heterogeneous', and 'system' must be similar enough for the techniques to be applicable, with appropriate modifications, across domains. That assumption is analysed in more detail later in this chapter to determine its validity. However, as an initial thought, consider the remarks of Alexander from of the source suggested by Naur.

> "The ultimate object of design is form … Every design problem begins with an effort to achieve fitness between two entities: the forces in question and its context. The form is the solution to the problem, the context defines the problem. … The rightness of the form depends ... on the degree to which it fits the rest of the ensemble." (Chapter 2: *Goodness of Fit* in (Alexander 1964)).

Many of the theories of Alexander are based on the notion of 'form' and the relationship between it and the human beings who interact with those built forms. Without thoroughly considering the differences between the built form of corporeal artefacts and the nature of software structures, it is impossible to successfully determine the appropriateness of Alexander's theories to software engineering design. Yet research literature in software engineering shows researchers continue to apply them based on perceived similarities with little regard to a thorough examination of the differences.

The participants then, as well as researchers now, were prepared to draw analogies between software development and established engineering disciplines without fully considering the differences between them. However, subsequent observations suggest the participants were admittedly not fully aware of the exact nature of software systems. Perlis, Bauer, and Kolence highlight this in an exchange on the relationship between software design and mathematics.

> "Perlis: Software systems are mathematical in nature. A mathematical background is not necessary for a designer, but can only add to the elegance of the design.
>
> Bauer: What we need is not *classical* mathematics, but *mathematics*. Systems should be built in levels and modules, which form a mathematical structure.

> Kolence: At the abstract level a concise mathematical notation is required by which to express the essential structures and relationships irrespective of the particular software product being implemented." ((NATO 1976a) pp. 21-22).

These comments identify the importance of system structure and the precision and order of mathematical expression, however the participants could not quite identify the exact relationship between the two. What was missing was a precise understanding of exactly what a software system is and the process required to design one. The ensuing comment by Smith on the nature of design criteria typifies that, though his comments may no longer be agreed with.

> "Smith: There is a tendency that designers use fuzzy terms, like 'elegant' or 'powerful' or 'flexible'. Designers do not describe how the design works, or the way it may be used, or the way it should operate. What is lacking is discipline, which is caused by people falling back on fuzzy concepts, instead of the razors of Occam, which they can really use to make design decisions. Also designers don't seem to realize what mental processes they go through when they design. Later they can neither explain, nor justify, nor even rationalize, the processes they used to build a particular system. I think that a few of Occam's razors floating around can lead to great simplifications in building software…" (op. cit p. 22).

The conference report then turns to the area of design strategies and techniques. Once again, Naur presents an analogy with traditionally engineered systems, this time with the automotive industry, to explain the importance of what is now known as software architecture. His comment concerns the importance of partitioning the major subsystems of a design to minimise their dependencies and to allow the correct ordering of their detailed design and implementation.

> "Naur: In the design of automobiles, the knowledge that you can design the motor more or less independently of the wheels is an important insight, an important part of an automobile designer's trade. In our field, if there are a few specific things to be produced, such as compilers, assemblers, monitors, and a few more, then it would be very important to decide what are their parts and what is the proper sequence of deciding on their parts. That is really the essential thing, what should you decide first." (op. cit p. 26)

Naur continues by quoting the design ideas of Christopher Alexander and suggests they are promising starting points for specific software design strategies. However, the suggestion Naur is making is that because designers partition the subsystems of automobiles in a particular way, there exist minimal dependencies between the identified subsystems. If software developers could partition their systems in an analogous way, they could achieve similar benefits that are "an important part of the automobile designer's trade". For the analogy to be valid, the necessary implication is that automobile designers have a choice in the large-scale partitioning of the automobile 'system' that is similar to the choice available to software designers. Using knowledge gained from interviews with automobile engineers, the validity of that assumption is analysed later in this chapter. Importantly, that analysis uncovers aspects of engineering design that have no direct analogue in software development. Furthermore, it suggests comments such as "you can design the motor more or less independently of the wheels" appears to be true, however many dependencies do actually exist that are not usually considered by software engineering researchers.

The discussion of design strategies moved onto debates about the importance and influence of 'top-down' and 'bottom-up' approaches. The core of the debate was captured in an excerpt from a working paper by Gill and was then analysed by the other participants including Fraser's analogy with frame stressing.

> "Gill: The obvious danger in either approach is that certain features will be propagated through the layers and will finally cause trouble by proving undesirable and difficult to remove, when they should have been eliminated in the middle layers. … In practice neither approach is ever adopted completely; design proceeds from top and bottom, to meet somewhere in between, though the height of the meeting point varies with circumstance." (op. cit p. 28)
>
> "Fraser: In designs I have been involved with, and which have not involved too many people, I have not been able to identify whether these have been 'top-down' or 'bottom-up'. They seem to be more like frame stressing, where one is trying to stress a structure with welded joints. You fix all the joints but one, and see what happens to the one, then fix that joint and free another and see what happens with that. It's a sort of iterative process which follows an arbitrary pattern through the structure. Perhaps this only holds for small

> designs, with few people and with good communication. About large designs, I don't know." (op. cit pp. 28-29)

These comments, and the others in the report, highlight interesting aspects of the software development process that are uncovered when considering the nature of 'top-down' and 'bottom-up' development. The questions that are not answered however, are: What is it about software systems and the process required to develop them that results in a mixture of top-down and bottom-up design? Moreover, what is it about the nature of software systems that result in one design decision constraining the alternatives for the remaining decisions?

Finally, Dijkstra uses an analogy with musical education to address the discussion of educating students about software design strategies.

> "Dijkstra: If I look for someone with a position analogous to the way in which I experience my own position, I can think of the teacher of composition at a school of music. When you have got a class of 30 pupils at a school of music, you cannot turn the crank and produce 30 gifted composers after one year. The best thing you can do it to make them, well, say, sensitive to the pleasing aspects of harmony. What I can do as a teacher is to try to make them sensitive to, well, say, useful aspects of structure as a thinking aid, and the rest they have to do themselves." (op. cit pp. 29-30)

Like all analogies used in software engineering research, Dijkstra's general comment on the harmony of system structure appears reasonable. Music has a substantial amount of theory that details which notes and chords can be played together to produce sounds that are pleasing to the ear. Similarly, music theory notes many combinations that should not be played together. Moreover, there exist patterns of notes and their combination that specify certain styles of music. All of these can be taught to students in pedagogic education. On closer examination however, the precise application of Dijkstra's analogy to software is considerably more difficult. What exactly are the harmonious structures of software composition? Contemporary research in the area of software architecture styles and design patterns are concerned with the structure of software systems. However, work in those areas is also based on analogies with other disciplines – specifically, theories of architecture and the theories of Christopher Alexander. No research in those areas however, details why the analogies are valid. All of their theories are justified based on

perceived similarities with those other disciplines rather than by examining the fundamental nature of software. Furthermore, they fail to systematically consider the differences between software systems and those other disciplines.

Later sections of the conference report deal with management aspects of the development process. As the discussion moved into the area of software quality and how it could be tracked and measured, McIlroy drew an analogy between software documents and engineering drawings.

> "McIlroy: I think we should consider patterning our management methods after those used in the preparation of engineering drawings. A drawing in a large organization is usually signed by the draftsman, and then after that by a draughting supervisor when he agrees that it looks nice. In programming efforts you usually do not see that second signature – nor even the first, for that matter. Clarity and style seem to count for nothing – the only thing that counts is whether the program works when put in place…" (op. cit p. 57)

Without questioning McIlroy's desire to be able to track and evaluate the quality of software designs, the analogy highlights the question of why it is possible to evaluate engineering designs as they proceed yet it is difficult to do the same for software systems. One fundamental difference is implied in the comment by Smith that precedes McIlroy's comment in the conference report.

> "Smith: … All documents associated with software are classified as engineering drawings. They begin with planning specification, go through functional specifications, implementation specifications, etc., etc. This activity is represent by a PERT-chart with many nodes. If you look down a PERT-chart you discover that all the nodes on it up until the last one produce nothing but paper. It is unfortunately true that in my organisation people confuse the menu with the meal." (op. cit pp. 57-58)

The topic of discussion moved back to education and the concepts that software developers should be trained in. That debate uncovered aspects of the fundamental nature of software systems and how they are different to traditionally engineered systems. However, it soon moved into a discussion of semantics and terminology and the examination of the differences between the disciplines was not completed. The debate appeared to flow on from the previous comment by Ross who notes the difficulty in

dealing with the 'concept' of software. Indeed the title of that section in the conference report is *Concepts* and the editors note the importance of the discussion and the difficulty faced by the participants when discussing it.

> "Editors: *The above title [Concepts] has been chosen, perhaps somewhat arbitrarily, for a report on a discussion about the basic techniques or ways of thinking, that software engineers should be trained in.*
>
> *It is perhaps indicative of the present state of software production, that this topic was one of the most difficult to report on*." (op. cit p. 62)

Ross begins the discussion by detailing his understanding of the nature of software development. It specifies his conception of the 'plex' concept, how that concept relates to emerging thoughts about software components, and how software development can be performed by systematically composing large scale systems out of smaller components. His presentation is important for researchers interested in historical debates about software component-based design. Additionally, his explanation of the 'plex' concept highlights aspects about the specific nature of software systems.

> "Ross: … A 'plex' has three parts: Data, Structure, and Algorithm (i.e. behaviour). You need all three aspects if you are going to have a complete model of something – it is not sufficient to just talk about data structures, though this is often what people do. … The key thing about the 'plex' concept is that you are trying to capture the totality of meaning, or understanding, of some problem of concern. We want to do this in some way that will map into different mechanical forms … using different software implementations." (op. cit p. 62)

Ross explains his concepts using the example of implementing a banking system. His comments use terminology such as 'semantic packages' and 'idealized plex' that are precursors to the terminology used today in object-oriented design theory such as 'business processes', 'use-cases', and 'analysis objects'. A point that would have highlighted important differences between software systems and other engineered systems concerns the difference between the 'plex' and the 'idealized plex'. Ross states that the 'idealized plex' is "one in which the mechanical representation has been thrown away". This is equivalent to the difference between analysis level objects and design/implementation level objects in contemporary object-oriented design theories. The

difference between those objects has never been satisfactorily explained and still causes problems in present day software development (Kaindl 1999)[10]. However, before the participants could debate those differences the discussion changed to the problem of terminology.

> "van der Poel: You are using, without definition, many terms which I just don't understand." ((NATO 1976a) p. 63)

Perlis replied by defining the terms used by Ross (plex, data, structure, function, algorithm, and component) in terms of LISP – "they are all just functions." From that point, the discussion moves into different interpretations of Ross's comments.

The debate highlights the fact that software development researchers could identify differences between software and traditionally engineered systems. Those differences provide a glimpse into the fundamental nature of software but it is extremely difficult to discuss them because researchers lack a common universe of discourse or collection of cohesive concepts with which to label and discuss the issues. The universe of discourse that is currently used comes from other engineering domains yet the terms borrowed from those disciplines fail to precisely capture the meanings of the concepts used in software development.

Towards the end of the conference report, the discussion turned to the, now famous, term – 'software crisis'.

> "Editors: *Quite early in the conference statements were made by several members about the tendency for there to be a gap, sometimes a rather large gap, between what was hoped for from a complex system, and what was typically achieved*." (op. cit p. 77)

The subsequent discussion by the participants was preceded by comments by Buxton who attempted to put the debate about 'software crisis' into an objective context.

> "Buxton: In a conference of this kind, when those present are technically competent. One has a tendency to speed up communication by failing to state the obvious. Of course 99 percent of computer systems work tolerably

[10] This is discussed in more detail in chapter 5.

> satisfactorily; that is the obvious. … The matter that concerns us is the sensitive edge, which is socially desperately significant." (op. cit p. 77)

The subsequent debate, which concerned the degree to which there was a problem, was captured by the comments of Kolence and Ross.

> "Kolence: I do not like the use of the word 'crisis'. It is a very emotional word. The basic problem is that certain classes of systems are placing demands on us which are beyond our capabilities and our theories and methods of design and production at this time. … There are many areas where there is no such thing as a crisis … It is large systems that are encountering great difficulties.
>
> Ross: It makes no difference if my legs, arms, brain, and digestive tract are in fine working condition if I am at the moment suffering from a heart attack. I am still very much in a crisis." (op. cit p. 78)

These comments highlight an issue that has never been solved. What exactly is the 'software crisis'? Buxton noted that it was concerned with the "sensitive edge" of development but what is that edge? Kolence implied that development approaches, at that time, worked appropriately enough. It was the theories concerning large-scale construction issues that needed to be addressed. Alternatively, Ross implied that while 99 percent of development was performed successfully using existing theories, perhaps the problems faced at the "sensitive edge" were indicative of a fundamental misunderstanding in the research community.

The participants did attempt to identify the causes of the 'crisis', beginning with Kinslow who drew an analogy with bridge design.

> "Kinslow: … I have never seen an engineer build a bridge of unprecedented span, with brand new materials, for a kind of traffic never seen before – but that's exactly what happened on OS/360 and TSS/360." (op. cit p. 79)

For Kinslow's analogy to be valid, it must be assumed that terms such as 'unprecedented requirements' and what it means to meet them must be somewhat similar across the disciplines. A bridge designed to span a particular distance and to carry a particular load needs to ensure that the weight-bearing capacity of the materials used in a particular structural arrangement will be sufficient. That must be determined in conjunction with the effects of environmental conditions such as potential weather situations and the

supporting characteristics of the ground on which the bridge will be built. The validity of that comparison is analysed later in this chapter. However, a comment made in the working paper by Gill was included in the current debate by the editors and highlights the difficulty faced when making these analogies and using them to develop better theories about software design.

> "Gill: Software is as vital as hardware, and in many cases much more complex, but it is much less well understood. It is a new branch of engineering, in which research, development, and production are not clearly distinguished, and its vital role is often overlooked." (op. cit p. 80)

One of differences between software and other engineered systems, which may be a result of those differences and which has often been made, was then stated by Kinslow. Unfortunately, no useful explanation has been accepted by the development community to explain why this situation exists.

> "Kinslow: Personally, after 18 years in the business I would like just once, just once, to be able to do the same thing again. Just once to try an evolutionary step instead of a confounded revolutionary one." (op. cit p. 80)

Ross then made a comment that, with the benefit of 30 years of hindsight, seems quite prophetic. A software development community that is in search for solutions but that has an insufficient understanding of its problems and why they exist is vulnerable to people who claim to have 'the' solution. Perhaps he envisaged the future of proposed solutions and the inevitable marketing hype that led to their fanatical support.

> "Ross: My main worry is in fact that somebody in a position of power will recognize this crisis – it is a crisis right now, and has been for some years, and it's good that we are getting around to recognizing the fact – and believe someone who claims to have a breakthrough, an easy solution. The problem will take a lot of hard work to solve. There is no worse word than 'breakthrough' in discussing possible solutions." (op. cit p. 81)

The final topic discussed concerns the overriding belief that software development is a branch of engineering. The editors note that the majority of the comments reproduced here were made during the discussion on software engineering education that occurred towards the end of the conference.

> "David: May I add another question: What does software engineering and computing engineering have in common with engineering education as it is defined in the United States today, or in Western Europe?" (op. cit p. 82)

David goes on to answer his own question in a subsequent statement which the other participants discussed.

> "David: … Certainly Richard Hamming has stated that the essence of computing today is an engineering viewpoint. It certainly is not mathematics in the classical sense. In order to find colleagues who have a philosophy which may contribute to our own enterprises, engineering is a much more fruitful area than would be one of the sciences or mathematics, at least in my opinion. ...
>
> Software engineering and computer engineering have an extremely important and nice aspect to them, namely that people want to work on things that meet other people's needs. They are not interested in working on abstractions entirely, they want to have an impact on the world. This is the real strength of computing today, and it is the essence of engineering.
>
> Ross: I agree very strongly that our field is in the engineering domain, for the reason that our main purpose is to do something for somebody…
>
> Randell: I am worried about the term 'software engineering'. I would prefer a name indicating a wider scope, for instance 'data systems engineering'.
>
> Dijkstra: We, in the Netherlands, have the title Mathematical Engineer. Software engineering seems to be the activity of the Mathematical Engineering par excellence. This seems to fit perfectly. On the one hand, we have all the aspects of an engineering activity, in that you are making something and want to see that it really works. On the other hand, our tools are basically mathematical in nature." (op. cit p. 82)

The final comment goes to McIlroy, who despite his own use of analogies with engineering disciplines (McIlroy 1968), notes that software development is different to those disciplines and researchers must keep this in mind.

> "McIlroy: ... I am concerned about the connection between software engineering and the real world. There *is* a difference between writing

> programs and designing bridges. A program may be written with the sole purpose to help write better programs, and many of us here have spent our life writing programs from the pure software attitude. More than any other engineering field, software engineering in universities must *consciously* strive to give its students contact beyond its boundaries." ((NATO 1976a) p. 83)

3.2.2 Analysing the Analogies Used During the 1968 NATO Conference

Analysis of the 1968 NATO conference report shows that during the course of the conference a number of analogies were made between software development and other engineering disciplines. The implication is that particular characteristics of the nature of software systems and other engineered systems must be equivalent for the analogies to be valid. However, a number of observations were also made by the participants that suggest their understanding of software systems was too insufficient for them to determine the equivalence of those characteristics. Moreover, many observations were made that provided a glimpse of the fundamental nature of software systems, suggesting it is quite different to that of engineering disciplines. With those observations, and with a more detailed understanding of other engineering disciplines, those analogies are now examined. Before the presentation however, it is important to make the following note. The analyses of the comprehension exhibited by those researchers and the ensuing analyses of the analogies made by them is not intended as a criticism of the integrity of their research. The purpose of the analyses is to obtain a better understanding for the future. Indeed their comments and opinions are understandable for researchers in a discipline that was still a fledgling at the time[11].

The analysis of the analogies begins with the two comments made about aircraft design. Graham commented that we build systems like the Wright brothers built aeroplanes – "build the whole thing, push it off the cliff, let it crash, and start over again." (op. cit p. 7). Gillette then noted that the problem of building software systems to specification and schedule was analogous to the aircraft industry that had similar problems. Examining why the Wright brothers built their aircraft that way, and examining how aircraft are now designed, highlights issues that provide an insight into the nature of software systems. Graham's analogy implies that the Wright brothers' design was tested mainly through

[11] These issues are discussed in more detail in chapter six.

trial and error because they did not precisely understand the principles involved. The Henry Ford museum provides a general-purpose introduction to the Wright brothers (The Wright Brothers 1995) and shows that assumption is too simplistic. The Wright brothers were repairing and designing bicycles when they became interested in flight and decided to design and build a flying machine. Wilbur contacted the Smithsonian Institute in 1899 and began to research the known theories of aeronautics. Presumably, this would have included the theories of Bernoulli who, in the early eighteenth century, formulated the principle relating the velocity and pressure of the flow of fluid/air (e.g., (Giancoli 1988) pp. 311-312). Techniques in glider design at that time were based on the principle of wing shape so that the velocity of the air flow above the wing would be greater than the velocity of the air flow below it. Bernoulli's principle shows that this results in lower air pressure above the wing, resulting in the wing lifting. While other factors such as turbulence also play a significant part, the Wright brothers used this research to identify the important principles that needed to be addressed to design the aircraft. They were: wings to provide lift, a power source for propulsion, and a system of control. The control problem was addressed by realising that the system needed to be controlled in its three degrees of movement: pitch, yaw, and roll (figure 3-1). Wilbur devised a method of controlling the position of the wings during flight so they could be manipulated to change the direction of the plane. A prototype test kite was then built to verify the technique. To test the wings the brothers built a number of gliders, which they tested at Kittyhawk. After a number of designs were tested, the brothers used a wind tunnel to refine the shape of their wing designs and achieve the lift required. Once the control and wing designs were successfully achieved, the engine and propeller of the propulsion system were built and the system as a whole was used to achieve their first flight.

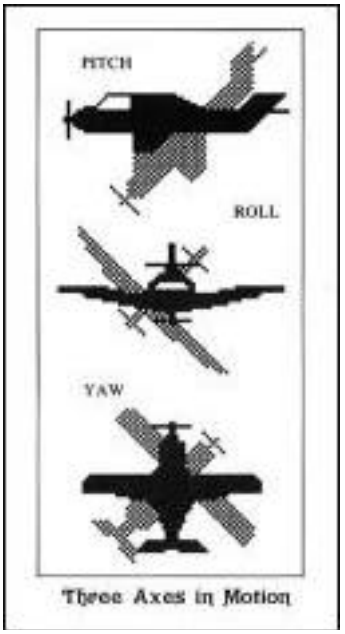


Figure 3-1: Pitch, Roll & Yaw

The design strategy of the Wright brothers was much more sophisticated than simply 'make it look like a bird, push it off a cliff, and see what happens'. It is clear they understood the basic principles of aircraft design before they began. The shape of the wings and airflow across them are the determining factors in the generation of the lift required for keeping a flying machine off the ground. Moreover, the

direction of those wings with respect to the wind controls the direction of that lift. They realised that the physical characteristics of the form of the wings, their size, shape, mass, and direction, are crucial design issues. At that time many mathematical techniques and equations existed to predict those properties. However, they were simply not detailed enough to model the designs to a level of detail required by the Wright brothers. They knew the basic principles of aeronautical design. However, because the precise nature of those principles and the precise physical characteristics of the implementations that exploited them were not known, working prototypes had to be employed to develop the design. Again, this is not a criticism of Graham's analogy. Nevertheless, a deeper analysis of that analogy can be used to develop a more profound understanding of engineering design, and, as a consequence, software design.

The Wright brothers could design systems that approximately worked as anticipated. What they could not do was determine the precision of those designs without testing them. That principle is evident in the contemporary process of aircraft design. Mathematical models now exist that define the principles of aeronautical theory and the properties and form of physical materials. That has allowed aircraft designers to design and build successful systems with far less reliance on working prototypes. For example, the Boeing 777 aircraft was the first jetliner to be 100% designed using 3-dimensional solid modelling technology. The result was the elimination of the requirement for a full-scale design model to test the assembly of the design (Boeing 1998).

> "The 777 is the first Boeing airliner 100% designed using 3-D solid modelling technology. The software used is CATIA (computer-aided, three dimensional interactive application) [and ELFINI (Finite Element Analysis System)] developed by Dassault Systems of France.... The 777 division used more than 2,200 CATIA workstations networked to an eight-mainframe computing cluster, this being the largest single CAD project anywhere, requiring 3 Tera-Bytes … of data to store the information. In addition to being a 3-D design tool, CATIA is also used as a digital pre-assembly tool. In the past, Boeing built a full-scale non-flying mockup of the complete aircraft to check fit for interference problems at a cost of 2.25 million dollars. Since the various systems were designed independently, it was necessary to make sure that a bolt did not occupy the same physical space as a hydraulic line, or that an electrical conduit did not run across the middle of a ventilation duct. The

> mockup was also used to check accessibility of all the parts for maintenance work. With a 3-D database that everyone uses simultaneously, the interference problems are eliminated, and the access question is answered by maneuvering a digital 'mechanic' in 3-D space." (Schokralla 1998)

The ability to design to specification and schedule has been significantly improved through the use of CAD technology. It has provided the capability to model the characteristics of proposed designs at a level of detail necessary for the design to be validated against the specifications, thereby eliminating the traditional requirement of developing full-scale, functional prototypes. These models consist of mathematical representations of the relevant characteristics of the designed system's physical form. Moreover, they are evaluated using mathematical representations of the principles and theories that relate those characteristics in aeronautical research.

The analogies made between aircraft design and software development describe similarities between the level of system understanding, the ability to predict the working properties of the design, and the ability to design to specification and schedule. Those analogies appear useful on the surface. However, closer inspection reveals those attributes of engineering design are based on the ability to model and predict the properties of the physical building materials from which the system is constructed. Moreover, they are validated in conjunction with mathematical representations of the theoretical principles of the discipline. It is not clear what analogous physical properties and theoretical principles exist in the discipline of software development. This is highlighted in Kinslow's comparison that claimed bridge designers do not build systems of unprecedented span with unknown materials for a kind of traffic not seen before. The ability to support a particular load is determined by the properties of the physical materials and structural patterns in which those materials are arranged. For instance, concrete has different load bearing characteristics than timber and arch bridge arrangements have different load bearing characteristics than suspension bridge arrangements. In addition, those characteristics must be considered within the context of environmental conditions such as potential weather situations and the supporting characteristics of the ground on which the bridge will be built. Again, the properties of the materials that are used by the discipline to achieve the requirements appear to have no direct analogue in software development.[12]

[12] Analogies with building design are analysed in greater detail later in this chapter.

Naur used an analogy with automobile design to highlight the importance of large-scale system partitioning. A statement such as "you can design the motor more or less independently of the wheels" is essentially true. The implication derived from that analogy was that software engineers could also enjoy significant development gains through early system partitioning. For that analogy to be valid, the factors that determine how a system can be partitioned must also be analogous. However, many dependencies exist between automobile subassemblies and an analysis of those reveals import insights concerning the nature of software development and engineering design. While little design work is performed on the wheels as such, they are considered as part of the rolling chassis, which requires a great deal of design work. It is interesting to examine the reasons for the general 'architecture' of the automobile and the nature of the dependencies between the subsystems. Is it actually designed to facilitate the separation of concerns between subsystems, as suggested in the original analogy, or do other factors exist?

The following description of automobile design is based on a discussion with the Senior Project Engineer in the Vehicle Performance Group, Holden Ltd (formerly General Motors Holden Australia) (Beltrami 1998). That discussion covered many issues raised by the analogy, including:

- The relationship, including dependencies, between wheel design and engine design.
- The conceptual architecture of the automobile. Identification of the major subsystems, how the architecture evolved in that manner, and why it has failed to change from the same basic 'shape'.
- The relationship between the engine and chassis subsystems and the design reasoning behind the chosen arrangement between those subsystems.

The only property of the 'wheel design' that affects the engine design is its physical size. Because the wheel size is a determining factor in the ground clearance of the vehicle, it affects the design of the spatial topology of the engine. Engine design is constrained by many factors, one of which is that it must occupy set physical dimensions. Another example of the effect of wheel size on engine design is, to some extent, on cars with automatic transmission. Wheel size affects engine revs at particular speeds and that is a determining factor in the design of automatic transmissions. Moreover, the fuel economy

of vehicles, which is often an important criterion of the customer, is affected by engine revs at cruising speeds, and that is affected by wheel size. For those reasons, wheel size is often one of the first design decisions made when designing, or making major modifications to the design of a vehicle.

The conceptual architecture of the automobile at Holden Ltd consists of a number of major subsystems:

- Powertrain (engine, transmission, differential, gearbox, etc).
- Chassis (suspension, brakes, wheels, floorpan, etc).
- Body (the shell. ie., panel work, bumpers).
- Electrical.
- Trim.
- Materials.

Each of these major subsystems has its own design group within the company (along with a design group to develop the safety aspects). Naur was correct in detailing the absence of major dependencies between wheel and engine design, however examining the design decisions related to the dependencies between the engine and the chassis as a whole provides a more interesting example. Does this new analogy continue to support Naur's implication that the generic architecture of the vehicle has developed to allow independent design of the chassis and engine?

A vehicle engine needs to generate enough power to pull a total weight of approximately 2300 kilos. That includes the weight of the engine itself, the rest of the vehicle, passengers, and luggage. That power is delivered via the drivetrain to rotate the wheels. The movement of those physical components generates inertial forces that significantly affect the handling of the vehicle as the forces from the engine are transferred to the chassis at the engine mounting points. The engine and chassis are joined by physical necessity but they are kept as separate as possible to reduce the effect of these inertial loads. As the engine revs, the torque produced causes the physical engine to rotate. If it was rigidly connected to the chassis the entire chassis would move causing severe handling problems for the driver. That is evident in relatively old, high-powered cars that connect the engine to the chassis using solid rubber mounts. More recent vehicles use hydraulic mounts that minimise the torque effects. Because the effects of the engine

forces on the chassis handling are so significant, the designer must make a choice. In the first option, the chassis and engine could be implemented as a single subsystem using a counter-balancing flyweight to minimise the engine forces on vehicle handling. That would provide advantages in other areas such as system simplicity and rigidity but would result in significantly extra weight and engine size. Alternatively, the engine and chassis could be designed as separate subsystems. That would minimise the dependencies between the two but would require the design of relatively complex mounting arrangements. In automobile design everything affects everything else. New subsystems must be 'tuned' to work as desired and to not affect other subsystems. For instance, even the position of the indicator stalk affects the design of the driver airbag system. The dependencies between subsystems are enormous. However, because the system's design has evolved over a long period of time, experience has reduced the effects of those dependencies. Changing to a completely different automobile configuration would require replacing all those years that have been used to minimise the effects of those dependencies. Moreover, all the investment put into design, production, and service of the existing technology would need to be reproduced.

Theories in software architecture, and the original comments by Naur, suggest large-scale software systems can be partitioned to make use of design teams and to minimise dependencies between systems. In automobile design however, the design teams are created to work on the subsystems of an architecture that has evolved over a long period. The egg and chicken are around the other way. The design of systems in traditional engineering disciplines requires the manipulation of physical materials and the combination of components constructed from those materials. The materials and components interact and exhibit mechanical properties that engineers use to provide the desired functionality. In addition, undesired properties may exist and their effects on the desired functionality need to be minimised. The important distinction is that engineers cannot necessarily 'produce' the exact functionality they desire using physical components. Rather they combine physical components in a system that produces properties that implement the desired functionality. That is not the case in software development.

The difficulty in expressing the characteristics of the systems that software developers construct is exhibited in the analogies used to express the 'types' of systems that are built and, consequently, how the general approach to design is described. For instance, Naur

drew an analogy between the "large, heterogeneous constructions" of software and those of civil engineering and architecture. Naur's analogy continued by recommending Christopher Alexander's theories in *Notes on the Synthesis of Form* (Alexander 1964) as a useful source. Alexander characterises the understanding of a 'large' and 'heterogeneous' system in the following passages.

> "Today more and more design problems are reaching insoluble levels of complexity. ... To match the growing complexity of problems, there is a growing body of information and specialist experience. This information is hard to handle; it is widespread, diffuse, unorganized. ... As a result, although ideally a form should reflect all the known facts relevant to the design, ... the technical difficulties of grasping all the information needed for the construction of such a form are out of hand – and well beyond the fingers of a single individual." (op. cit p. 3)
>
> "The reason that iron filing placed on a magnetic field exhibit a pattern ... is the field they are in is not homogeneous. If the world were totally regular and homogeneous, there would be no forces, and no forms." (op. cit p. 15)

According to Alexander, all design problems must achieve some 'fitness' between the form and its context. The context of the problem provides a set of conflicting constraints and the designer must satisfy them. For example, "The iron filings constitute a form, the magnetic field a context." (op. cit p. 20)

These design theories appear applicable to the problems faced by software developers who also face the challenge of constructing complex systems with a large and conflicting set of contextual constraints. However, are the design theories of Alexander applicable because the situations faced by the respective disciplines are truly similar? Although software development and traditional engineering disciplines utilise similar terms to label aspects of their design processes, further analysis reveals that the fundamental nature of those aspects are considerably different. Alexander states, "The ultimate object of design is form" (op. cit p. 15). However, it is not clear that Alexander's theories, whilst being extremely interesting in their own right, are directly applicable to software development. Obviously design patterns, which are based on Alexander's theories, have provided significant benefits for software developers. However, differences between the respective disciplines suggest the reason why they are useful may be different to the reason they are

beneficial in Alexander's architecture research[13]. For instance, the concept of physical 'form', which provides the foundation for Alexander's theories, has no direct analogue in software systems.

Many analogies were used during the 1968 NATO conference to develop ideas about *how* software developers should go about the process of designing and implementing software systems. However, the insights provided by the conference participants concerning the nature of software systems and the subsequent analysis of the original analogies shows the fundamental nature of *what* the respective disciplines build is significantly different. Therefore, although the analogies appear valid and much of the terminology is used across disciplines, significant questions arise concerning the appropriateness of applying the term 'engineering' to software development. The original motivation for doing so was obviously well intentioned.

> "The need for software manufacture to be used on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering." (NATO 1976a)

However, the observations made by the participants, and the subsequent analysis of the analogies made, shows the application of the analogies do not fully consider the nature of software systems. At the end of the 1968 conference, many questions existed concerning the applicability, rather than the original intent, of the term 'software engineering'.

3.2.3 The 1969 NATO Conference

The most striking aspect of reading the 1969 conference report is the significant change in attitude of the participants. Like the 1968 report, the 1969 report provides a transcript of many of the debates that occurred as well as the publication of a selection of the working papers presented. However, while the structure of the proceedings is similar, the content is markedly different. In fact, the editors of the reports take great care to detail the difference in the introduction.

> "The intent of the organizers of the Rome [1969] conference was that it should be devoted to a more detailed study of the technical problems, rather than including also the managerial problems which figured so largely at

[13] This is discussed in chapter seven.

> Garmisch [1968]. However, once again, a deliberate and successful attempt was made to attract an equally wide range of participants. The resulting conference bore little resemblance to its predecessor. The sense of urgency in the face of common problems was not so apparent as at Garmisch. Instead, a lack of communication between different sections of the participants became, in the editors' opinions at least, a dominant feature." (NATO 1976b) p. 145)

The structure of the conference report is similar to the 1968 report. The debates are grouped into discussions concerning software specification, software quality, software flexibility/portability, large system case studies, and software engineering education. Like the 1968 report, the debates contain many insightful observations that highlight aspects of the nature of software systems and the differences between them and traditionally engineered systems. For example,

> "Schwartz: In my experience on large systems, we have pictured the systems as a flow of data. On these projects we have people whose sole job is to develop and to change and re-do table specifications. Wherever possible we try to have programming languages which divorce the data definition from the actual procedures…
>
> Randell: I am reluctant to draw a definite line between the concepts of program and data. One of the nice things about SIMULA is that the concept of process definition in some sense includes, as special cases, procedure and data structure definitions." (op. cit p. 155)

This exchange highlights important observations concerning the interrelationships that exist between program and data. Moreover, many similar observations are evident in the transcript. What was noticeably absent, however, was any discussion concerning the nature of software development and its relationship with engineering disciplines. The section of the 1968 report that dealt with software engineering education concentrated on why software developers should look to those other engineering disciplines. Moreover, the previous analysis and description shows that many issues remained unsolved that undermined the assumption that the disciplines are analogous. The same section in the 1969 report however, appears to assume that the analogy is valid and the question had changed from what software engineering has to do with traditional engineering, to what software engineering has to do with computer science. Indeed, at the risk of inferring too

much from the transcript provided, the assumption that software development is an engineering discipline appears to pervade the majority of the discussions. At the end of the conference reports the reader is left with the impression that the discipline of software development entered the 1968 conference with the proposal that it would like to become an engineering discipline and needed to examine the relevant issues to see if the goal was appropriate. It then left the 1969 conference with the belief that it is an engineering discipline and that only the technical issues about how to successfully engineer software systems remained to be solved. In the introduction to the 1969 report, the editors make the following note about the previous year's conference.

> "The Garmisch conference was notable for the range of interests and experience represented amongst its participants. In fact the complete spectrum, from the inhabitants of ivory-towered academe to people who were right in the firing-line, being involved in the direction of really large-scale software projects, was well covered. The vast majority of these participants found commonality in a widespread belief as to the extent and seriousness of the problems facing the area of human endeavor which has, perhaps somewhat prematurely, been called 'software engineering'." (op. cit p. 145)

Although many of the participants still believed the label 'software engineering' had been accepted prematurely, the issue was not questioned in the 1969 conference transcript. Interestingly, the editors note that the most dominant feature of the conference was the communication gap that appeared between the participants during the conference.

> "Eventually the seriousness of this communication gap, and the realization that it was but a reflection of the situation in the real world, caused the gap itself to become a major topic of discussion. Just as the realization of the full magnitude of the software crisis was the main outcome of the meeting at Garmisch, it seems to the editors that the realization of the significance and extent of the communication gap is the most important outcome of the Rome conference." (op. cit p. 145)

The final discussion of the conference, which is presented first in the transcript, dealt with the need to "talk about, rather than just suffer from, the effects of the communication gap." (op. cit p. 147). Strachey begins by examining the differences between theory and practice.

> "Strachey: … This sort of debating point is not helpful. The truth of the matter is that we tend to look with doubt and suspicion at the other side; whichever side of that particular barrier we are. On one side we say 'Well, there's nothing we can get out of computing science: look at the rubbish that they are talking'. Or we stand on the other side and look at the very large programs and we say 'Goodness me; what rotten techniques they use and look: they all fail.'
>
> One of the most interesting things that has been shown at this conference is that these projects don't all fail. It has been shown that some of the them have been quite astonishingly successful." (op. cit p. 147)

The discussion then turns to how the theories of researchers could be demonstrated and verified so practitioners could adopt them with increased confidence. The discussion of pilot projects led to an increased distinction between theoreticians and practitioners that results in the following remarks by Dijkstra and Randell.

> "Dijkstra: I would like to comment on the distinction that has been made between practical and theoretical people. I must stress that I feel this distinction to be obsolete, worn out, and fruitless. It is no good, if you want to do anything reasonable, to think you can work with such simple notions. Its inadequacy, amongst other things, is shown by the fact that I absolutely refuse to regard myself as either impractical or not theoretical.
>
> …
>
> What is actually happening, I am afraid, is that we all tell each other and ourselves that software engineering techniques should be improved considerably, because there is a crisis. But there are a few boundary conditions which apparently have to be satisfied. I will list them for you:
>
> We may not change our thinking habits.
>
> We may not change our programming tools.
>
> We may not change our hardware.
>
> We may not change our tasks.
>
> We may not change the organisational set-up in which the work has to be done.

> Now under these five immutable boundary conditions, we have to try to improve matters. This is utterly ridiculous. Thank you. (*Applause*).
>
> Randell: … 'There's none so blind as them that won't see.' … If you have people who are completely stuck in their own ways, whether these are ways of running large projects without regard for possible new techniques, or whether these are ways of concentrating all research into areas of ever smaller relevance or importance, almost no technique that I know of is going to get these two types of people to communicate. … You have to have good will. You have to have means for people to find out that what the others talk is occasionally sense. This conference may occasionally have done a little bit of that. I wish it had done a lot more. It has indicated what a terrible gulf we so stupidly have made for ourselves. (op. cit pp. 151-152)

The philosophical gulf between theoreticians and practitioners of software development appeared suddenly between the 1968 and 1969 conferences. Interestingly its occurrence coincides with what the editors noted as the premature acceptance of the label 'software engineering'.

3.3 The Evolution of the Artefact Engineering View

From the presentation and analysis of that initial understanding of software engineering, an examination is now presented of how that artefact engineering view of software development has evolved.

McIlroy's invited talk at the 1968 NATO conference was one of the first attempts to directly describe software components in terms of engineering terminology. His *Mass Produced Software Components* (McIlroy 1968) is considered a seminal paper on software reuse (Krueger 1992) and is based on direct analogies with traditional engineering disciplines.

> "Software components (routines), to be widely applicable to different machines and users, should be available in families arranged according to precision, robustness, generality, and time-space performance. Existing sources of components – manufacturers, software houses, users' groups, and algorithms collections – lack the breadth of interest or coherence of purpose to assemble more than one or two members of such families, yet software

> production in the large would be enormously helped by the availability of spectra of high quality routines, quite as mechanical design is abetted by the existence of families of structural shapes, screws or resistors." (McIlroy 1968)

McIlroy claims that the existence of a few similar terms between the disciplines provides some validity to the analogy. However, he also notes that not all terms could be directly applied between them.

> "The idea of subassemblies carries over directly and is well exploited. The idea of interchangeable parts corresponds roughly to our term 'modularity', and is fitfully respected. Yet this fragile analogy is belied when we seek for analogues of other tangible symbols of mass production." (McIlroy 1968)

The basis of McIlroy's conviction seems to be that software components could be understood like engineering components. For example, when talking about the use of table mechanisms in compiler writing, he says, "I claim we have done enough of this to start taking such things of the shelf." His subsequent claims about developing a sub-industry of components with varying degrees of precision, robustness, time-space performance, and generality were justified using a detailed example of the "lowly sine function". That is reinforced with briefer descriptions concerning how the claims can also be applied to the application areas of numerical approximation routines, input-output conversion, two and three dimensional geometry, text processing, and storage management.

It is impossible to argue with McIlroy's desire to improve the efficiency of software development by utilising mass produced software components. Especially when the practice is perceived to be such a fundamental part of traditional engineering design.

> "What I have just asked for is simply industrialism, with programming terms substituted for some of the more mechanically oriented terms appropriate to mass production. I think there are considerable areas of software ready, if not overdue, for this approach." (McIlroy 1968)

However, McIlroy fails to address a number of issues. His claim that software catalogues could consist of components classified by "precision, robustness, time-space performance, size limits, and binding time of parameters" appears possible for the small number of application areas he discussed. The sine function and other application domains all consist of concepts that are either extremely well defined *a priori* (e.g.,

mathematics) or the relevant concepts have been suitably codified by practitioners over time (e.g., input-output functions and compiler writing). Indeed, at that time, there were a number of very successful families of components that were widely reused (Reed, 1995 in Reed 2000). It is not clear how McIlroy's ideas, no matter how well intentioned, apply to other problem domains in which the relevant components cannot be so easily defined and parameterised for classification. That was borne out in the subsequent discussion by the conference participants. Perlis began by noting the following.

> "Perlis … Specialists in every part of software have a curious vision of the world: All parts of software but his are simple and easily parameterized; his is totally variable." (McIlroy 1968)

D'Agageyeff suggests why this might be the case.

> "d'Agapeyeff: … It is extremely difficult to construct this software [file handling systems] in a way that is efficient, reliable, and convenient for all systems and where the nature of the package does not impose itself on the user. The reason is that you cannot atomize it. Where work has been successful it tends to be concerned with packages that have some structure. … But why do we need to take atoms down off the shelf? What you want is a description which you can understand, because the time it takes to code it into your own system is really very small. In that way you can insert your own nuances. The first step in your direction should be better descriptions." (McIlroy 1968)

Those comments allude to the importance of identifying the differences between software components and engineering components. Kolence brought that point back to the nature of design.

> "Kolence: We are concerned with a mass design problem. In talking about the implementation of software components, the whole concept of how one designs software is often ignored. Yet this is the key thing." (McIlroy 1968)

The comments by the conference participants show the initial ideas presented in McIlroy's call for software industrialisation are worthy of further investigation. However, issues concerning the differences between the nature of software design and engineering design must also be considered. Unfortunately, those concerns were not investigated

further. Other participants in the debate continued to assume the analogies were valid and that the applicability of the ideas to software development are obvious.

> "Naur: What I like about this is the stress on basic building principles, and on the fact that big systems are made from smaller components. … A comparison with our hardware colleagues is relevant. Why are they so much more successful than we are? I believe that one strong reason is that there is a well established field of electronic engineering, that the young people start learning about Ohm's Law at the age of fourteen of thereabouts, and that resistors and the like are known components with characteristics which have been expounded at length at the early level of education. The component principles of our systems must be sorted out in such a form that they can be put into elementary education." (McIlroy 1968)

The belief that the "comparison with our hardware colleagues is relevant" was based on the justification of a small number of perceived similarities with those disciplines. The important questions concerning the differences however, were not addressed:

> What are the component principles of our systems? and
>
> How are they different to traditional engineering disciplines?

The discussion of the previous section shows that between the 1968 and 1969 NATO conferences these questions appeared to have been either accepted without question, forgotten, or temporarily replaced by more pressing, short-term, technical problems. Unfortunately, analysis of the evolution of component ideas in software engineering shows these fundamentally important questions have rarely come back to the fore.

The 1970s saw great advances in software engineering theory. Brooks' essays on software engineering, based on his large-scale system building experience, appeared in *The Mythical Man-Month* (Brooks 1975). The introduction of the 'information-hiding' concept by Parnas appeared in *On the Criteria to be Used in Decomposing Systems into Modules* (Parnas 1972). And the emphasis on large-scale system design issues by DeRemer appeared in *Programming-in-the-Large Versus Programming-in-the-Small* (DeRemer and Kron 1976). All of these are considered canons of the discipline. During that time the belief that 'engineering' could continue to be used as a valid metaphor for software development remained. Moreover, the questions elicited from McIlroy's call for the industrialisation of software engineering were still not addressed.

Fifteen years after the NATO conferences, Wegner published his lengthy discourse, *Capital-Intensive Software Technology* (Wegner 1984). He begins with the assumption that software development could be treated analogously to traditional engineering development and then extends the engineering metaphor to the understanding of development resources and products as capital goods.

> "Striking similarities between industrial and software technology have led to considerable borrowing of the terminology of industrial technology for corresponding concepts of software technology.
>
> …
>
> Any reusable resource may be thought of as a capital good whose development cost may be recovered over its set of uses. Thus, it seems reasonable to identify the notion of capital goods with that of reusable resources and the notion of capital with that of reusability.
>
> …
>
> Capital formation in software technology is dependent on the implementation of concepts and models rather than on the construction of physical machines. Our generalized notion of capital includes both conceptual and physical capital formation because we see reusability as a key denominator." (Wegner 1984)

Wegner examines the capital-intensive aspects of software development, both in terms of the existing state of the art and from his predictions about the future. His treatise is divided into four parts:

1. Software Components: reusability of components, interfaces, function, data, and process abstractions, distributed and concurrent processes, and object-oriented concepts.
2. Programming in the Large: paradigms of software technology, paradigms of development life cycles, and reusable concepts and models.
3. Knowledge Engineering: people-oriented knowledge engineering, knowledge-support environments, and computer authoring technology.
4. Accomplishments and Deficiencies of Ada: a case study of Ada as a capital-intensive technology, and an analysis of the question of whether it is a product or a process.

The first two sections identify important issues that concern the applicability of the artefact engineering view. Wegner identifies the following types of software components:

- Function abstractions: Specified by its input-output relation. Its operation is dependent solely on the data parameters. The function implementation is hidden from the user.
- Data abstractions: Able to store an internal state and functionality that determine the precise operation of the component. Both data and function specifics are hidden from the user. These correspond with contemporary notions of non-threaded objects.
- Process abstractions: Similar to data abstractions but have an independently executing thread of control. These abstractions may operate concurrently and may be distributed across machines.

Wegner's analysis examines aspects of those components that makes them difficult to treat as capital-intensive resources in the traditional sense. The nature of component interfaces were investigated, identifying aspects of syntactic, compile-time issues and semantic, run-time issues that make the realisation of 'plug-and-socket' models of program construction from existing components difficult. For process models, the analysis highlights concurrency control issues and global data sharing as aspects that differ from conventional notions of component interaction.

Rather than examining why those differences exist and how they affect the validity of the artefact engineering view, Wegner's discussion implies they will be solved by dealing with issues in development process models. Specifically, the future application of knowledge engineering techniques to the software development process to help the software developer in the construction of those systems. His analysis of issues concerning abstraction, specialisation, pattern recognition, and reusable models, are very useful and predict some current issues in software architecture and design patterns. However, while the discussion does identify specific differences between the capital-intensive goods of software technology and those of traditional engineering, it does not examine them in enough detail to identify the underlying principles of software technology, nor does it determine if they invalidate the artefact engineering view of software development.

Wegner's conclusion states that those differences will be accounted for in the future application of expert-system approaches to the software development domain. However,

the benefit of 15 years of hindsight shows that has not occurred, despite considerable research. It may be that Wegner's principles were correct and that more research is needed. However, other comments in his paper suggest the differences might be more fundamental and the metaphor itself needs to be examined in more detailed. His analysis of reusability made the following statement.

> "At the present time, reuse of a component in successive versions of an evolving program appears to be a more important source of increased productivity than reuse of code in different applications. Components are rarely portable between applications, and even if they are, the incremental benefit of using a component in two applications is only a factor of two. But the number of versions of a systems over its lifetime can number in the hundreds or even the thousands." (Wegner 1984)

Wegner's examination of the differences between software and traditional capital-intensive components fails to explore this issue even though the ability to easily reuse components across systems is fundamental to traditional engineering. Subsequent research in reuse technology has examined those issues in more detail but has also failed to provide a software component marketplace that rivals traditional engineering components. Moreover, a precise examination of the differences between what an application is in the respective disciplines has also never been made. Once again, it may be that more research is required. However, Wegner later makes an insight about the understanding of software components that questions the metaphor for understanding. When discussing the importance of knowledge engineering in the software development process, he states,

> "Euclid's *Elements*, a magnificent piece of knowledge engineering, provided a basis for managing geometrical knowledge." (Wegner 1984).

The implication of the subsequent discussion was that the future application of knowledge engineering would identify a similar foundation for software engineering knowledge. Research has shown that Euclid's foundations are not the solid basis of geometrical knowledge that Wegner asserts. It is only one, though the most popular and 'default' one, of many axiomatic foundations of geometry. If Wegner's analogy is valid, how does the

issue of conceptual relativism affect the ability of the software engineering research community to identify its founding principles? [14]

At approximately that time, Spector and Gifford contributed to the research debate by publishing *A Computer Science Perspective on Bridge Design* (Spector and Gifford 1986). They interviewed a partner in a consulting engineering firm that specialised in bridge design and believed the results provided experience and insights that could be of use for computer systems designers – both software and hardware. Their questions covered topics from the design process, project management and organisation, tool use, reliability and failures.

Spector and Gifford draw analogies between bridge design and software development based on some obvious similarities.

> "Structural engineers decompose a bridge into a hierarchy of subcomponents, all of which are ultimately constructed from relatively simply objects like beams and plates. Programs … for example, are also hierarchically decomposed, but the primitives are instructions… Dynamically, the bridge-design process is arranged so that separate groups can address separate aspects of the design. A bridge-designer's concerns for functionality, reliability, serviceability, and even aesthetics are familiar to computer systems designers." (Spector and Gifford 1986)

They also note some general differences.

> "The most noticeable difference is that the bridge-design process is much more structured than computer systems design. Similar design decompositions and project organizations are used for each bridge. Standard specifications … further constrain designs, by mandating standardized requirements and constraints on materials." (Spector and Gifford 1986)

Specific differences were also detailed and classified in terms of attention to reliability, the use of tools, standardised bridge requirements, standardised material specifications, formal design documents, and separations of design from implementation.

[14] Chapter five discusses these issues in further detail.

The authors conclude that this mature engineering discipline might provide a "glimpse of the future of computer system design".

> "As computer science matures, there may be more standardized specifications and designs. When the design space for certain application areas becomes more constrained, it may be possible to produce clearer specifications earlier in the design phase. Reliability guarantees may assume increasing importance, and the use of tools may become more prevalent." (Spector and Gifford 1986)

Spector and Gifford's analysis finishes with caveats that suggest the differences between the disciplines may make specific predictions about the future of software engineering difficult. However, questions and answers provided in the interview can be used to highlight additional glimpses of the component principles of engineering disciplines and help determine whether they are applicable to software systems.

On the surface, the design processes appear similar. The interviewee identified bridge-design as consisting of preliminary design, main design, and construction phases. The preliminary design phase is directly analogous to the initial architecture design stages of software development. The preliminary design stage "describes the various alternative structures that were considered, estimates the costs of each alternative, and usually recommends one of the designs" (Spector and Gifford 1986). The main design stage appears similar to other parts of the analysis and design phases of software development.

> "The main design phase involves a complete structural design, making drawings, and writing specifications that describe the tests that materials must pass before they can be used, their quantities, and some of the construction techniques. In effect, the bridge is completely specified during the main design phase." (Spector and Gifford 1986)

However, detailed descriptions of how the main design phase of bridge-design is performed highlights principles of engineering components that simply do not exist in software components. The first step in the design process is to establish the design criteria, or detailed specifications, that the bridge must meet. Those criteria are specified in terms of specific stresses and loadings that the structure must cope with. Those criteria are context independent in the sense that they are applicable to all bridges. Indeed, the

American Association of State Highway and Transport Officials (AASHTO) publishes an annually revised design specification document that all bridges must meet.

> "[It] prescribes load capacities for vehicular traffic in terms of weight, number, and frequency. It gives design loads for wind and outlines procedures for obtaining seismic loads. It sets allowable stresses for steel, concrete, and other materials, and details design rules for such components as stiffeners, columns, etc. It indicates what tests are necessary for various materials before they can be approved for use. Most of the individual specifications are component specifications, although some specifications are given on a system basis." (Spector and Gifford 1986)

This provides the starting point for the design specification, however additional criteria must also be established.

> "For example, creep is the deformation over time of a material under constant stress. The formula for creep is not universal, so we specify the formula that we'll use for a particular project. Another factor of increasing importance with larger bridges is natural phenomena: If a bridge is in an area where hurricanes can occur, or where there is considerable seismic activity, we have to establish appropriate design loadings to account for these phenomena. The goal is to establish acceptable bounds in terms of the relevant probability of risk and the cost and importance of the project." (Spector and Gifford 1986).

A mathematical model of the design, which was accepted in the preliminary design phase, is then created. It specifies "where joints, pins, and other connections are to be placed – we would consider, for instance, how the bridge should be connected to the piers. We try to get a general outline of the various components of the bridge" (Spector and Gifford 1986). That mathematical model is then evaluated against the detailed specifications, which are also represented in mathematical notation, to ensure the design will meet the requirements.

> "There's the dead load of the structure itself, as well as the live load of the vehicles on the bridge. We have to determine how many situations to account for. Do we combine the live load with a full hurricane wind? The answer is 'no' because there wouldn't be vehicles on the bridge during a hurricane. … We also have various safety factors for each combination. … This level of

> analysis gives us the forces acting on all the components." (Spector and Gifford 1986)

This process is remarkably similar to the process identified in the analysis of engineering designs for the cruise control systems discussed in the previous chapter. Engineering disciplines can identify the fundamental principles of their disciple and represent them in the context-free language of mathematics. Moreover, they have also developed techniques for representing the properties of their components and systems/structures using similar notations. Those designs can be evaluated with respect to the requirements, and predictions about the success of the design can be made. That has provided a significant improvement in the design of traditional engineering systems.

> "I have to admire the courage of those pioneer engineers, trying to build long flexible bridges without the benefit of much analysis or knowledge of the dynamic effects of the wind … We've since learned how to actually calculate most of the stresses and deflections from all types of loads… Up until the 1950's we were using slide rules and desk calculators to help determine the forces on components. On reasonably large, indeterminate structures, we used approximation techniques to reduce the number of simultaneous equations that needed solving. That would leave us with a maximum of 25 simultaneous equations that needed solving… With the advent of computers, we returned to classical analysis techniques with matrix methods. This allowed us to routinely solve hundreds of simultaneous equations. I think the aeronautical industry really led the way in this area… Today we're also using finite-element methods, which allow us to combine linear components with plate elements, and even to compute stresses in solids. With these methods, we're able to calculate the response of just about any type of structure to any conceivable load, static or dynamic." (Spector and Gifford 1986)

Obviously, spectacular engineering failures have occurred – see for example (Petroski 1994). Spector and Gifford broached this subject in their interview. If specifications and designs can both be modelled and evaluated formally using mathematical techniques, why do failures occur? They report that failures generally occur when engineers extrapolate beyond their knowledge or models. Engineers still must decide what needs to modelled and to what level of precision. They also need to determine safety factors to account for variability in loads and materials, which can never been modelled exactly, as well as

errors that may occur due material fatigue. Significant failures have also occurred due to errors introduced during the connection of the components.

The ability to mathematically model the fundamental component principles of their discipline allows traditional engineers to analyse and verify their designs before construction. That point was also made by Smith and Dallen who published a comparison of software engineering design and VLSI design, also in the mid 1980s (Smith and Dallen 1984). Interestingly, that publication compared the two disciplines from the VLSI perspective rather than the other research reported here which describes the software engineer's perspective on the processes of other engineers. Their report begins by drawing analogies between the disciplines.

> "There are good reasons for drawing analogies between the VLSI and software design processes. All design methodologies, irrespective of the discipline, embody the same developmental stages. A conceptual model is transformed into a physical reality by gradually refining the implementation details. The design is tested and evaluated to verify that it meets the design objectives or requirements." (Smith and Dallen 1984)

The comparison describes the design process of the two disciplines and maps them onto a common framework for analysis. One of the main conclusions of their analysis is that both disciplines face similar problems in the area of design verification and analysis.

> "Key to the timely evaluation of a design is in early measurement against design objectives. While issues of design quality are recognized as important, no real place has been found for insuring design quality in the design process. If real progress is to be made in VLSI and software engineering, both function AND quality will have to be coped with early in the design process. This dictates the need for a better balance in the use of proofs, analysis, and simulation in support of both software and VLSI design." (Smith and Dallen 1984)

By the beginning of the 1990s, software engineering research was providing more detailed discussions about what was required of software development in order to become an engineering discipline. Those publications and debates detail many useful goals and ideals, however none of them address the fundamental questions that arose at the end of

the NATO conferences twenty years earlier – What are the underlying principles of software components and systems and how are they different to traditional engineering disciplines?

IEEE Software published a special issue in 1990 on *The Challenge of Software Development*. The guest editors, Lewis and Oman, note that the understanding of software components was the driving force for the evolution of the discipline. Yet, while other aspects of computing had made rapid advancements, software development methodology was still "little more than a black-art" (Lewis and Omen 1990). The journal gathered 15 academic and industry people to discuss the problem and they identified two broad themes:

- There is an untapped potential for productivity gains through the reuse of standard software components.
- There is a trend toward greater reliance on tools like rapid application-development and design tools.

As part of that journal issue, a number of papers were published that highlight current thinking in the area. Two of those captured the understanding of software components and how a software engineering discipline could evolve. In *Prospects for an Engineering Discipline of Software* (Shaw 1990), Shaw examines the issues to be addressed for software development to become an engineering discipline. Summarising many different definitions of the term 'engineering', Shaw abstracts out the following common principles.

> "Creating cost-effective solutions … to practical problems … by applying scientific knowledge … building things … in the service of mankind." (Shaw 1990)

The development of that scientific knowledge was examined by discussing the evolution of engineering disciplines – in particular civil engineering and chemical engineering. Shaw notes that as those fields evolved from simple crafts to professional disciplines two important processes took place. The disciplines developed the ability to capture and pass on the rationale of routine designs and they helped develop and utilise a supporting science that could explain and predict the fundamental properties of proposed designs.

Based on that analysis, Shaw proposed two tasks to assist the evolution of software development towards and engineering discipline. They were:

- pick an appropriate mix of short-term, pragmatic, possibly purely empirical contributions that help to stabilise commercial practice and
- invest in long term efforts to develop and make available basic scientific contributions. (Shaw 1990)

The endeavour to identify, collate, and disseminate designs and their rationale from known implementations proceeded rapidly from approximately that time. Shaw herself had been previously working on software systems abstractions (Shaw 1984) and at about the same time as the *Prospects …* paper, she also published one of the first specific papers on software architecture. That paper attempts to identify and classify well-known large-scale system structures (Shaw 1989). Since then, software architecture research has produced many useful theories and case studies to capture and disseminate large-scale design rationale. Similarly, research in design patterns has also sought to capture and disseminate useful rationale from existing designs. However, the second point made by Shaw has received far less attention from the research community. Shaw noted that computer science has developed some good models and theories to contribute to the supporting science of software engineering. Algorithms and data structures, programming language semantics and type systems, and compiler design theories have all been used to improve the design practices of the discipline. However, no contribution to the 'scientific foundations' of software systems has allowed the same systematic design process to be applied to all software systems as, for example, the way Newtonian mechanics provided a foundation for mechanical and civil engineering.

The other interesting paper in that IEEE Software issue was *Planning the Software Industrial Revolution* by Cox (Cox 1990). Published 20 years after the NATO conferences, its thesis is similar to McIlroy's original call for mass produced software components. Cox suggested that software engineering research had been predominately concerned with improving development processes and needed to change to a product-centric paradigm.

> "The familiar process-centric paradigm of software engineering, where progress is measured by advancement of the software-development process, ... The paradigm that may launch the Information Age is the same one that

> launched the Manufacturing Age 200 years ago. It is a product-centric paradigm in which progress is measured by the accretion of standard, interchangeable, reusable, components and only secondarily by advancing the processes used to build them." (Cox 1990).

While much of the motivation was the same as McIlroy's, Cox notes that the emergence of object-oriented technology could now provide a basis to allow people to reason about software components in a similar manner to how they reason about tangible components in other disciplines.

> "In the broadest sense, 'object-orientated' refers to an objective, not a technology for achieving it. It means wielding all the tools we can muster, from well-proven antiques like Cobol to missing ones like specification languages, to enable our consumers by letting them reason about our products via the commonsense skills we all use to understand tangible objects." (Cox 1990)

Cox identifies a number of differences between software components and traditional engineering components: complexity, nonconformity and mutability, intangibility (invisibility), single-threadedness, and ease of duplication. He argues that the issue of intangibility could be overcome with object-oriented technology. He deals with the other issues in subsequent publications (Cox 1991; Cox 1992).

Cox's arguments did not examine, however, how components are understood in the respective disciplines. He argues that software reuse could overcome the issue of intangibility. However, his arguments do not examine how the design processes of other disciplines are able to reason in terms of the underlying component principles and utilise existing components to meet design objectives which are stated in terms of those principles. The question of the underlying principles of software component principles was still not addressed.

The introductory paper by Lewis and Oman in that IEEE Software issue provided a summary of software engineering evolution. Their conclusion was that the challenge for the 1990's was to develop a sufficient understanding of the development process to automate it as much as possible. This would provide the evolutionary path for software engineering research in the 1990s.

> "1990 and beyond: This era will see the application of expert-system techniques to software engineering. The combination of software-engineering workstations, expert systems, and automated techniques for development will find widespread use in the software industry." (Lewis and Omen 1990)

That prediction is similar to the one made by Wegner many years earlier (Wegner 1984). Lowry, in *Software Engineering in the Twenty-First Century* (Lowry 1992), examines in detail the impact of knowledge-based approaches on software engineering. He summarises attempts to apply AI techniques to software engineering at the beginning of the 1990s and makes predictions about the future. His proposal is 'transformational programming' where prototyping, validation, and modifications are done at the specification level and automatic program synthesis translates specifications into efficient-code. One of the central features of transformational programming is software architecture.

> "To support software developers, software architectures will include the functional roles of major software components and their interrelationships stated in an application-oriented language; a domain theory that provides precise semantics for use in automated reasoning; libraries of prototype components with executable specifications; program synthesis capability to produce optimized code for components after a prototype system has been validated; a constraint system for reasoning about the consistency of a developing software system; and design records that link requirements to design decisions." (Lowry 1992)

According to Lowry, that revolution in software engineering would result in "a broad consensus that knowledge-based methods will lead to fundamentally new roles in the software-engineering life cycle and a revised view of software as human knowledge that is encapsulated and represented in machine manipulable form" (Lowry 1992).

That picture of the future of software engineering is quite attractive. Moreover, Lowry goes into extensive detail about how it would be achieved and the benefits it would produce. However, two important assumptions about the nature of software components were made by Lowry that were never examined. First, there was an assumption that those domain-level 'knowledge components' could be thought of and manipulated in a similar manner to traditionally engineered components. Many of the predictions made were based

on the future ability to adequately formalise those concepts and store them for potential reuse. The justification for the assumption was that existing techniques and future expectations in the area of AI show that it may be possible. However, to date that has certainly not been the case. There is no evidence to believe that 'knowledge components' can be universally defined and formalised in computer implementable terms.

The second assumption concerns how those components could be reused to construct a system. Lowry notes that the major obstacle to be overcome was the ability to control the search for potentially reusable components during the program synthesis stage. That point assumes that software components could be sought out and used to realise the domain-level components identified during the initial stages of design. It assumes that components could be found to meet the design rather than designs being generated to utilise the component-base. Moreover, the justification assumes that that was how the process occurs in traditional engineering design.

The inability to adequately describe the underlying principles or scientific knowledge of software components was addressed by an ACM Task Force on the Core of Computer Science. That report, *Computing as a Discipline* (Denning, Comer et al. 1989), identifies 3 different paradigms that pervade software development.

> "The three major paradigms, or cultural styles, by which we approach our work provide a context for our definition of the discipline of computing. The first paradigm, theory, is rooted in mathematics and consists of our steps followed in the development of a coherent, valid theory: (1) characterize objects of study (definition); (2) hypothesize possible relationships among them (theorem); (3) determine whether the relationships are true (proof); (4) interpret results. …
>
> The second paradigm, abstraction (modeling), is rooted in the experimental scientific method and consist of four stages that are followed in the investigation of a phenomenon: (1) form a hypothesis; (2) construct a model and make a prediction; (3) design an experiment and collect data; (4) analyze results. …
>
> The third paradigm, design, is rooted in engineering and consists of four steps followed in the construction of a system (or device) to solve a given problem:

> (1) state requirements; (2) state specifications; (3) design and implement the system; (4) test the system." (Denning, Comer et al. 1989)

Software developers face the unique situation where they must deal with all of those paradigms simultaneously. Moreover, the authors suggest that research efforts face the problem of having to devise explanatory theories that encompass all of the issues. Research may sound plausible when they explain development theories in terms of one of those paradigms, however all of them must be considered.

> "Many debates about the relative merits of mathematics, science, and engineering are implicitly based on an assumption that one of the three processes (theory, abstraction, or design) is the most fundamental. Closer examination, however, reveals that in computing the three processes are so intricately intertwined that it is irrational to say that one is fundamental. …
>
> The three processes are of equal – and fundamental – importance in the discipline, which is a unique blend of interaction among theory, abstraction, and design." (Denning, Comer et al. 1989)

During the early to mid 1990s, many research agendas progressed based on the artefact engineering view of software development. Research ideas in software architecture, object-oriented technology, domain modelling, and design and component reuse rely on that guiding assumption. D'Ippolito notes that one of the differences between traditional engineering and software engineering is that engineers are able to model their designs in order to compare them with the requirements and make predictions about the implementation (D'Ippolito and Plinta 1989). His research group advocates the use of domain modelling to provide similar benefits for software engineering and provide examples in the simulation of military systems (D'Ippolito and Lee 1992a). The ability to successfully model the components of real-world tangible systems is then extrapolated to suggest the approach could be applied to all software systems and provide the basis of software engineering. Indeed, they claim it puts the "engineering into software engineering" (D'Ippolito and Lee 1992b).

Similar ideas have been put forward by researchers in software composition. For instance, the Ithica (Intelligent Tools for Highly Advanced Commercial Applications) Esprit II Project proposed a component-based approach to application development based on

object-oriented technology, domain modelling, software architectures and system frameworks (Nierstrasz, Tsichritzis et al. 1991; Fugini, Nierstrasz et al. 1992; Nierstrasz, Gibbs et al. 1992; Tsichritzis, Nierstrasz et al. 1992). Despite some successful applications, Nierstrasz notes that many issues are still to be resolved. Those issues relate to component connections and composition models, and useful graphical abstractions to represent software components (Nierstrasz and Meijler 1995).

Other research issues have also been developed through analogies with engineering disciplines. Kogut developed ideas about design reuse by comparing software engineering with chemical engineering (Kogut 1994; Kogut 1995) and Leveson explained important issues in software system safety by comparing them with progress in steam engine design (Leveson 1992).

Research in software reuse also developed ideas that suggest software systems components could be developed, located, and synthesised into application systems in analogous ways to other engineering systems. For instance, (Castano and DeAntonellis 1993; D'Alessandro, Iachini et al. 1993; Fugini and Faustle 1993).

The emergent research field of software architecture has also used many comparisons with traditional engineering systems. For example, Van der Linden's research into constructing large-scale systems from 'building blocks' (Linden and Muller 1995); Perry & Wolf's foundations for software architecture (Perry and Wolfe 1992); Kructhen's explanation of the different types of software architecture in terms of the architecture views (Kruchten 1995); Inverardi's explanation of software architecture's as processes of chemical reactions (Inverardi and Wolf 1995); and Whitehead's explanation that software architecture can be used as the basis for a component marketplace (Whitehead, Robbins et al. 1995).

Despite all these publications being based on the artefact engineering view of software development, not all research agreed that engineering should be used as a metaphor for understanding software development. Once again, their disagreement identifies the underlying principles of the components and systems as the stumbling block. For example, Marco, in his book *Software Engineering: Concepts and Management*, states the following:

> "The logical nature of the product ... is the major difference between software 'engineering' and real 'engineering'. Because of this … there are few physical

> laws which can be used to model, describe, or predict the behavior of software. Obviously some mathematical 'laws' are relevant to software, but these have not yet been demonstrated to fill the role that physical laws do in other forms of development. It is because of the lack of physical laws that the software aspect of computer science is sometimes called artificial science – like political science or social science – rather than natural science like physics or chemistry." (Marco 1990)

This does not suggest that rigorous mathematics cannot be used in software development. It states that what is represented by the mathematics used is completely different in the respective disciplines and cannot be used to justify the analogies.

Moreover, researchers in fields such as software architecture, which do subscribe at least implicitly to the artefact engineering view, identify problems not found in other engineering disciplines. For instance, Garlan identifies a number of reasons why it is so difficult to build software systems out of existing parts (Garlan, Allen et al. 1995).

1. Assumptions about the nature of the components: Many software components make assumptions about supporting infrastructure that exists within those components or within other components. They make assumptions about how the thread of control is passed through a collection of components as they are executed. Finally, they make assumptions about the nature of the data that they will be manipulating.
2. Assumptions about the nature of the connectors: Software components make assumptions about the protocol or pattern of interaction that will be made between them. They also make assumptions about the nature of the data that is passed during that communication.
3. Assumptions about the global architectural structure: Software components make assumptions about the other large-scale subsystems that exist and the global architecture style that governs their means of communication and their visibility.
4. Assumptions about the construction process: Some software components, especially those concerned with the instantiation or initialisation of an application make assumptions about the order in which the application is 'constructed' or instantiated.

Those assumptions are not made by components of other engineering disciplines and they provide a glimpse into the unique nature of software components and systems. Further research suggests those differences could be addressed by considering the component

interfaces in more detail (e.g., (Shaw 1995a)) however, they do not address the fundamental, underlying principles of software components that require them to make those assumptions and why they are not required by other engineering components.

From the mid 90s onwards, a number of publications questioned the artefact engineering view explicitly or provided more detailed analysis of the fundamental nature of software systems and components. During the 1995 ICSE conference Jackson gave a keynote speech *The World and the Machine* (Jackson 1995).

> "The requirement – that is, the problem – is in the *world*; the *machine* is the solution we construct. The point is trite and obvious. But perhaps we have yet to come to terms with it, to understand it fully, and act on that understanding." (Jackson 1995)

Jackson examines the relationship between software systems and the world to which they apply and identifies a number of interesting facets.

- the *modelling* facet, where the machine simulates the world;
- the *interface* facet, where the world touches the machine physically;
- the *engineering* facet, where the machine acts as an engine of control over the behaviour of the world; and
- the *problem* facet, where the shape of the world and of the problem influences the shape of the machine and of the solution.

He notes that a number of issues make it difficult for software developers to deal with those facets of the relationship between the world and the machine and provides a number of useful principles to observe when dealing with them. The core of Jackson's observations were concerned with the how the problems in the world could be modelled successfully in the machine.

> "Traditionally, I am claiming, we pay too little attention to the world in which our problems are found." (Jackson 1995)

The issues that needed to be addressed by software engineering researchers were: it's difficult to successfully model the world, there are many different and valid views of the world, and the common language terms used to capture the descriptions of the world are inherently ambiguous.

Gilb also notes the problem of ambiguous terms, but in relation to how software engineers describe the design process. Terms like ‘system’, ‘design’, and ‘component’ have well-understood meanings in traditional engineering disciplines but they have not been defined appropriately for the discipline of software development. Therefore, before researchers continue to develop theories for “What should software engineering be?”, he suggests they should concentrate on the questions: “What is engineering?” and “What is software engineering?” (Gilb 1996).

Those two questions were indirectly addressed by other research emerging at that time. Wasserman, in *Toward a Discipline of Software Engineering* (Wasserman 1996), identifies eight fundamental concepts that have emerged and remained constant during software engineering research. Consequently, he claims those concepts provide a foundation for determining “What is software engineering?”

- Abstraction: The ability to deal with complex problems by suppressing some of the unnecessary lower-level detail. It allows developers to represent concepts and terms that are familiar in the problem and solution domains. Moreover, it is the central concept of information hiding.
- Analysis and Design Methods and Notations: Analysis methods provide a means of formalising the problem domain. Design deals with the structure of the system implementation. There is a cognitive leap to be performed from the problem to the solution and methods attempt to assist this process. However, lack of universal design notations and the interrelated nature of the two processes make this concept difficult to deal with.
- User Interface Prototyping: User Interface prototyping is essential for quickly developing and determining the requirements of the system with the client. Moreover, the UI is important for the effective use of the system. However, it is clear that good interface design skills are different from those needed in other aspects of development.
- Modularity and Architecture: Issues of large-scale and large-granularity design significantly influence the quality of systems. Architecture styles and design patterns are providing standardised or, at least, better-publicised, design exemplars.

- Life Cycle and Process: A well-defined and manageable process provides benefits for software developers. Considerable research attention has been focussed on processes, however it appears as though the issue of software process is not as fundamental to software engineering as are abstraction and modularization.
- Reuse: The long-standing notion of component reuse is essential to a discipline of software engineering. Some small-granularity reuse success has been achieved, however success beyond the level of function and well-defined class libraries has proved to be more difficult.
- Metrics: Metrics currently exist for testing, quality assurance and cost estimation. However, it is impossible to measure improvements in software engineering without a well-defined set of items to be measured and accurate measurements of current practice.
- Tools and Integrated Environments: Integrated support for the development process is essential to improve software engineering. However, the diverse range of existing environments reflects the wide range of development processes and methods currently being used.

All of those concepts have proved to be extremely useful in software development. However, they all contain aspects that are not completely understood and are the subject of ongoing research efforts. Those aspects are related to the fact that software engineering researchers have not been able to identify the fundamental principles of software components and systems – the supporting science for software engineering.

Xia examines several of those issues in *Software Engineering: a methodological analysis* (Xia 1997). He also notes that software engineering must develop its supporting science, however current research efforts produce results based on concepts that are not properly defined or universally understood. Moreover, those long-term research efforts are often overlooked because of pressing business concerns to produce short-term solutions.

Commentaries on software engineering education have suggested aspects of the software supporting science. Maibaum attempts to identify the *praxis* of software engineering so it could be formalised and taught to software engineering students.

> "Is the knowledge used by software engineers different in character from that used by engineers from the conventional disciplines? The latter are

> underpinned not just by mathematics, but also by some physical science(s) – providing models of the world in terms of which artifacts must be understood. … Software engineering may be distinguished from other engineering disciplines because the artifacts constructed by the latter are physical, whereas those constructed by the former are conceptual." (Maibaum 1997)

Moreover, Maibaum notes that the 'real world' constrains the construction of physical systems in a way that is has no analogue in the engineering of concepts and abstractions. The subsequent assertion is that logic should become the supporting science of software engineering because "logic is the mathematics of concepts and abstractions". Maibaum's justification is that philosophical logicians have been dealing with concepts and abstractions long before the existence of computers. However, he seems to ignore the work done by philosophers in epistemology and metaphysics who have also been dealing with concepts and abstractions long before the existence of computers but who do not define them in terms of logic.

Parnas has also published many articles on software engineering education, specifically on the difference between computer science and software engineering (see for example (Parnas 1997; Parnas 1999)). His argument that computer science and software engineering should be treated as different disciplines is quite valid. However, the examination of computer science as the supporting science of software engineering does not detail the exact nature of that science. He notes that "Engineers *do* use mathematics". Therefore, software engineers should use mathematics. Indeed,

> "Every programmer uses 'formal methods' because programs are formal and programming is formalisation. However, in software we use different mathematics! We need discrete mathematics and notations suited for piecewise continuous functions (tabular expressions)." (Parnas 1997)

There is no examination of how concepts and abstractions can be represented using formal methods and what are the limitations of that approach. Just because formal methods can be used to represent some aspects of software engineering does not necessarily mean it is the supporting science being searched for. The problem is nobody has proved it either way.

An emphasis on modelling can also be seen in the relatively new area of Systems Engineering. This area of software development has arisen out of traditional engineering

disciplines such as control systems, automotive engineering, and the computerisation of what were originally mechanical systems. One of the central tasks of the systems engineering process is to model the solution to clarify requirements and analyse alternative solutions (see for example). The main practitioners are in fact engineers, and in terms of this discussion, are converting existing artefacts into software models. This line of thinking traces back to the comments on D'Ippolito earlier in this chapter. However, just like those earlier comments, the modelling approach taken by systems engineers fails to consider how the ability to successfully model the components of real-world tangible systems is then extrapolated to be successfully applied to all software systems

Finally, recent research has provided renewed emphasis on component-based software engineering (CBSE). However, while research is progressing and some results are being used in practice, researchers note that "there's little agreement on what 'components' and 'component-based software engineering' are" (Kozaczynski and Booch 1998). Brown and Wallnau provide a summary of the closing discussions at the workshop on CBSE at ICSE 98 (Brown and Wallnau 1998). Those workshop participants provide a number of definitions of what a component is, however Brown and Wallnau note a number of key differences between those definitions. Aspects of component granularity, context dependence, and component autonomy differ between the various definitions. Those aspects have no direct analogues in traditional engineering components and are similar to the problems previously identified by Garlan that make it difficult to construct software systems from existing building blocks. The report notes that while research has yet to identify the fundamental principles of software components and systems, commercial utilisation of CBSE is progressing based on interface constraints imposed by the somewhat standardised, commercial component infrastructure products. Moreover, the consensus in the research community is that those research problems will be solved and that CBSE provides one of the best prospects for improving software engineering in the next century (McConnell 2000).

3.4 Conclusion

The term 'software engineering' was coined for the 1968 NATO conference. An analysis of that conference showed the term was suggested merely to provide a starting point for discussion concerned with improving software development. It appears as though the

term was implicitly accepted even though many problems with it were identified. The assumption was that the metaphor of engineering was useful for software development and that those problems would be solved by subsequent research. An analysis of that research shows one question has continued to arise but has never been thoroughly addressed. What are the fundamental principles of software components and systems? Research has suggested that identifying and developing the supporting science of software engineering can solve that fundamental issue. Moreover, the supporting science must somehow deal with a critical difference between software systems and traditionally engineered systems. Software systems are based, somehow, on the notions of concepts and abstractions whereas traditionally engineered systems are constrained by their physical tangibility and can be understood by using mathematical representations from the physical sciences. Moreover, research suggests that the supporting science of software engineering is somehow based on the mathematical representations of logic. Those issues were summed up in the paper by Baber – *Comparison of Electrical "Engineering" of Heaviside's Times and Software "Engineering" of Our Times* (Baber 1997). An analysis of that paper, however, shows our understanding of the similarities and differences between software and traditional engineering is still not sophisticated enough.

Baber's paper argues that software development is not yet an engineering discipline, at least not in the sense commonly accepted by traditional engineering disciplines. Rather, what is practised today is a pre-engineering phase of what can and should become a true engineering discipline. By examining the transition of those traditional disciplines from their pre-engineering phases to their current state, we could learn from their successes and failures, and accelerate our own transition.

By examining the history of electrical engineering during its transition period and by using supporting examples from shipbuilding and bridge design, Baber identifies three phases in the evolution of an engineering discipline.

1. The "pre-engineering" phase where the evolution of the discipline is driven by practical needs and concerns. General properties and relationships of the building materials of the discipline are identified and formulated into "rules of thumb" to assist practitioners.

2. The "consolidation" phase which marks the beginning of the transition from a practice-driven discipline to a theory-driven one. The observations and generalised

rules identified during the pre-engineering phase are codified and integrated into formal, mathematical theories which explain the interaction between the underlying "quantities of interest" of the discipline.

3. The "reformulation and reorganisation" phase in which the theoretical work is repackaged in a manner which makes it more useable for the practitioner. Because the theoretical work, which explains the interaction of the "quantities of interest", usually consists of very complex mathematics, it needs to be made more useable and to be shown as beneficial to the developer. This step allows the practitioners of the discipline to move to a theory-driven approach.

During the practice-driven approach to development, the practitioner is only able to represent the physical form of the system design. However, once developers begin to utilise knowledge of the theoretical foundations of the discipline it becomes possible to represent particular properties of the proposed system. That enabled the designer to model the design with a representation that provides the ability to predict the suitability of the design to serve a particular purpose or to meet some predetermined requirements.

Although the use of theoretical foundations provides significant leverage in system design, Baber's historical analysis shows there was considerable opposition to the utilisation of that theory-driven development. That opposition came from the established members of the discipline who held positions of considerable authority and influence. Consequently, a division occurred between the proponents of the respective approaches. They each developed their own conflicting theories and resulting predictions of observable phenomena. Whilst the theory-driven practitioners were able to prove the validity of their own theories while the practice-driven theories could not, it was not enough to convert all of the practitioners away from the traditional approaches.

Baber presents a detailed timeline of the events that marked the transition period of electrical engineering. Those events feature specific examples of engineering design problems caused by the practice-driven approach, examples of the rift that occurred as members of the discipline moved towards theory-driven development, and the specific examples which mark the three phases of the transition process.

1. Pre-Engineering Phase: Examples of the design errors caused by the practice-driven mentality of the discipline. For instance, in 1856 the chief electrician of the Atlantic Telegraph Company, Whitehouse, ruined the first transatlantic cable by applying an

input voltage that theoretical analysis had predicted would be too great. Whitehouse was aware of that theoretical analysis and had chosen to ignore it.

2. Consolidation Phase: In 1873, Maxwell published twenty equations which, by building on the general theories of others, succeeded in explaining the relationships between the fundamental quantities of the discipline. The fundamental quantities of interest in electricity were: the electric field, the magnetic field, and the electric charge. The secondary quantities of interest included: current, time rate of change of electric field, time rate of change of magnetic field, and voltage. However, due to the complexity of the mathematics, the ability to successfully use his theories was beyond the reach of almost all electrical practitioners.

3. Reformulation and Reorganisation: Oliver Heaviside saw a copy of Maxwell's theories when they were published and immediately realised their potential for improving the design of electrical systems. However, it took Heaviside many years to fully understand and apply those theories in practical usage. For instance, in 1877 he "explained theoretically why the maximum working speed of an undersea telegraph circuit was different in the two directions" (Baber 1997). In 1887, in co-operation with his brother, he wrote a paper that included for the first time the condition for distortionless transmission. Although they were theoretically valid, both those examples, and others, faced opposition from proponents of the practice-driven approach because they contradicted existing theories. In fact, it was not until around 1890 that the "balance of power and influence" began to change from the practice-driven practitioners to the theory-driven ones.

Baber's understanding of the evolution of engineering disciplines is similar to the view taken by the researchers already presented in this chapter. From that understanding, he draws an analogy with software engineering by presenting a selection of events, errors, and failures from the history of software development. Baber asserts that the fundamental cause of the failures were the same as those in the field of electrical engineering – the "lack of a scientific, mathematical foundation or failure to apply whatever such basis may exist" (Baber 1997). He proposes that,

> "The solutions to our problems today, while different in detail, will be fundamentally and essentially the same as the solutions to those problems a 100 years ago: developing a scientific, mathematical basis for the work of the

> engineer and structuring and organizing it to facilitate its regular practical application." (Baber 1997)

The software system failures, according to Baber, are evidence that software development is in the pre-engineering phase of evolutionary progress. However, he suggests research ideas exist that mark the beginning of the other phases. For Baber, the beginning of the Consolidation Phase occurs in 1967 when Robert Floyd presented the paper, *Assigning Meanings to Programs*, in the Symposium of Applied Mathematics. It presents a complex method for analysing a computer program to determine whether its execution would fulfil certain execution criteria. The beginning of the Reformulation and Reorganisation Phase is subsequently marked by the work of Hoare (Hoare 1969) and others, who summarised Floyd's work and applied those concepts to practical software development problems. However, while Baber identifies these events, which he claims mark the beginning of the phases, he notes the phases are not yet complete so it is impossible to determine if they in fact correspond to those in electrical engineering.

Those "formal methods" techniques have since become the subject of much software engineering research. However, they have had little impact on software development in practice. Baber's conviction in formal methods leads him to suggest that software engineering's reluctance to utilise formal methods is similar to the resistance to theory-based development experienced in electrical engineering. He concludes with the following remarks:

> "The analogy between the traditional engineering disciplines and software development suggests that software development also will undergo a transition to an engineering field in the current sense of the term. If the analogy continues to hold, we can expect software engineering tomorrow to be characterized by the regular use of predictive models based on a mathematical, scientific, and theoretical foundation." (Baber 1997)

But how far does the analogy continue to hold? Based on the work of Floyd and Hoare, software developers are able to determine, mathematically, whether the execution of a computer program fulfils certain criteria, such as paths of execution and values of variables. According to Baber, that is analogous to the ability of electrical engineers to represent, mathematically, the quantities of interest of their discipline.

At first glance, the paper by Hoare, and the subsequent work by other formal methods researchers appears to be analogous to the use of mathematics by engineers. Formal methods provide a rigorous technique for proving what a program does. Therefore, it can be used to determine whether the implemented system meets its objectives.

> "One of the most important properties of a program is whether or not it carries out its intended function. The intended function of a program, or part of a program, can be specified by making general assertions about the values which the relevant variables will take after execution of the program." (Hoare 1969)

That is analogous to how engineers use mathematical techniques to prove their designs meet the required objectives. However, engineers also use mathematical techniques to model and analyse their requirements, and model the required properties of their major components. That is in addition to proving the correctness of the designs. The previous chapter that analyses the cruise control designs detailed the use of modelling in the engineering designs. During the analysis and design stages, the engineers developed mathematical models of system and component properties, models of vehicle motion as a whole, models of environmental influences, and models of human driving behaviour. Those were in addition to the models used to prove the correctness of the system during the implementation and testing stages.

The application of formal methods in software engineering does not occur during the analysis and design stages of software development in an analogous way. They differ in terms of what is modelled. The engineering analysis section of the Cruise Control chapter noted that the models used by engineers are not models of abstract functionality. They are models of how the underlying properties of the discipline, which engineering components exhibit and manipulate, are used by engineers to realise the required functionality. This crucial difference between engineering components and software components becomes clearer by looking briefly at the evolution of engineering components.

In the history of electronic engineering, the notion of an 'electric' force was identified in natural phenomena and used by people to produce useful devices. For example, as far back as 50 BC, electric eels were used to treat arthritic conditions even though nobody know how or why it worked (Hill 1975).

Science, in its attempt to understand nature, conducted experiments on that natural force and discovered that different materials produced interesting effects. Those properties and effects were labelled voltage, current, resistance, capacitance, and inductance. Many experiments were performed and mathematical relationships were determined to explain those effects on the underlying properties of the discipline. Those mathematical relationships, such as Ohm's law, represent idealised effects on the properties of the discipline. Many more experiments had to be performed to determine how physical materials could be guaranteed to meet those idealised effects. Take resistance as an example. Scientists discovered that some physical materials restricted the flow of current and that could be used to produce useful effects. However, they could not guarantee or predict the amount of resistance it would give. Ohm was able to explain, mathematically, that the relationship between the voltage applied across the terminals of a resisting device and the current passing through it should remain constant at a constant temperature. That is the device's 'resistance'. His analysis was based on many experiments and considerable mathematical analysis using analogies with the known laws of fluid dynamics (Jungnickel and McCormmach 1986). However, physical materials did not naturally exhibit that definition of an idealised resistance. Considerable experimentation was performed to identify how well physical materials could be made to match that idealised definition of resistance (Marsten 1962). For example Barrett published the results of experiments of over 100 different materials to determine their resistance (Barrett, Brown et al. 1902). Moreover, Dummer provides a bibliography of approximately 350 publications on the nature of resistance in different materials which he claims is only a starting point for a more complete catalogue (Dummer 1956). Those experiments were also hampered by the fact that the devices used to measure the concept of resistance could not be guaranteed to work predictably enough to use Ohm's law. It was not until Wheatstone developed a technique of measuring resistance that was immune to variations in the other components of the system that the resistance of physical materials could be determined (Powers 1976). Finally, it was discovered that physical resistors only approximated the idealised concept of resistance over particular temperature ranges, frequency ranges, and particular geometries of physical materials. Similar progress was made on the other idealised electronic components – for example capacitance (Podolsky 1962).

Therefore, there exists no physical material that precisely exhibits the concept of resistance as specified by Ohm's Law. However, research and experimentation has shown that certain materials, constructed in particular physical arrangements, and used in a constrained operating environment, can be made to approximate that specification of resistance for the purposes of using that law to engineer predictable systems.

The result is that mathematical models were not discovered to explain the effects of physical materials on the underlying properties of the discipline. Rather, it was discovered that the effects of physical materials on those underlying properties could be made to conform to idealised mathematical models. Those mathematical models could then be used to predict the effects of components within particular environmental parameters. Using those idealised concepts, circuit theories were devised to explain how they could be predictably combined. Those circuit theories were formulated using the known physical laws of conservation of charge and conservation of energy (Gray 1969). System design could then proceed based on those mathematical idealisations of components and systems. In fact, circuit theory is concerned solely with mathematical idealisations of circuit elements and not with physical components (Belevitch 1962). Further analysis of the history electronic engineering reveals design soon became constrained so that functionality was thought of solely in terms of those idealised components and their combinations (Darnell 1958; Brothers 1962; Darnell 1962). Readers may also be interested in Susskind's detailed examination of the early history of electronics (Susskind 1968a; Susskind 1968b; Susskind 1969a; Susskind 1969b; Susskind 1970a; Susskind 1970b).

Engineers are constrained to thinking in terms of functionality that can be achieved using the underling materials and methods of their discipline. In the case of electronic engineering, the most basic functionality was identified during the discipline's evolution and was able to be explained using mathematical idealisations. When the physical materials could be constrained to meet those mathematical idealisations, designers were able to use mathematical techniques to represent the requirements of their designs. The same mathematical techniques could also be used to represent the proposed functionality of their systems. The design process then proceeds by using techniques to solve the mathematical equations relating the requirements and the possible functionality.

This is not how formal methods are used in software engineering. In electronic engineering, the requirements are represented in terms of functionality performed on the

underlying properties of the discipline. That functionality can then be specified using mathematical techniques and the components of the discipline can be used to realize that mathematical specification. There are no analogous underlying properties in the discipline of software development. The requirements are represented using concepts and abstractions. The implementation medium of software development is used to refine and implement those concepts and abstractions. It is not until those concepts and abstractions are refined to a very low level of granularity that formal methods can be used.

It may be argued that the discipline of software engineering is simply waiting for its 'Newton' to come along (Gallagher 1997) and identify the appropriate underlying principles of the discipline. Moreover, it may be possible to develop a discipline of software engineering by constraining the component base to a subset of axiomatic 'idealisations' that will allow us to develop systems analogously to traditional engineers. Those suggestions cannot be properly evaluated without a thorough investigation of what software developers deal with – concepts and abstractions. That is the subject of Chapter 5 of this thesis. Before that however, the thesis turns to an analysis of the artefact engineering view of software development when applied to a specific aspect of software engineering research – software architecture.

4. An Example of Understanding Based on the Artefact Engineering View – Software Architecture

4.1 Introduction

Software engineers have been discussing the architecture of their systems since the late 1960s and software architecture research has been a separate field of study since the late 1980s. While the discipline is still quite new and the ideas are still solidifying, confusion exists concerning the nature and meaning of software architecture and that confusion is restricting the progress of software architecture research and the adoption of its ideas in practice. For instance, the call for papers for a recent IFIP conference on software architecture (Perry 1998) details the need to address the following questions:

1. What are the most difficult tasks performed by practising software architects and what is available from research to help solve them?
2. Where are the gaps between business needs and research results, and what can be done to bridge those gaps?
3. What are the important problems being addressed in research and why are they (or why are they not) relevant to practice?

Mobray (Mobray 1998) recognises the importance of architecture research ideas and discusses why they are so hard to put into practice. He notes that software architecture ideas differ because of confused terminology, the lack of complete models, and disagreement about which views of the system are necessary. One reason for the differences is the lack of a universally agreed definition or even understanding of what software architecture is or should be. Bennett (Bennett 1997) captures the result of that confused understanding by noting that the research community is almost unanimous in its conviction that software architecture describes something about the structure of a system and that it plays a vital role in determining the systems emergent properties. However, they are much less unanimous on the questions of which elements should be included in the architecture, how to co-ordinate different collections of those elements (views), and how to evaluate the architecture against the external requirements. The problem is not that

there are no answers to these problems, rather, the difficulty arises from the fact that there have been so many different answers given.

The confusion exists because the understanding of the term architecture is based on analogies with traditional engineering or building disciplines. That way of understanding is a consequence of the artefact engineering view of software development and is evident in the philosophy of the self-proclaimed 'World-wide Institute of Software Architects' (WWISA 1999):

> "There is a compelling analogy between building and software construction. It is not new, but it has never taken root and bloomed. The analogy is not just convenient or superficial. It is truly profound. It not only raises the right questions, it has the answer to what has been called 'The Software Crisis.'
>
> Software architecture is now at a point identical to where building architecture was in the mid-1800's as it faced the inventive momentum of the industrial revolution. Now, as then, people with very different skills and roles can – and do – call themselves architects. In 1998, they refer to themselves as software architects despite training as engineers or programmers, not architects. However, it is no longer adequate for a software craftsman with a flair for design to build the huge, complex infrastructures of the information revolution. …
>
> Software systems are being built in a manner akin to erecting an office building without an architect and without clear roles." (WWISA 1999)

This chapter examines our understanding of software architecture by presenting an architecture-centric case study of a software system and then a chronological review of the theoretical understanding of software architecture. The case study provides practical examples of architecture issues by tracing the large-scale system structures used during the design, implementation and maintenance of the HyperEdit system. It also highlights architecture issues that are not easily explainable within the current understanding of software architecture. The final section traces the history of architecture ideas in software engineering research and shows the current understanding of architecture is based on analogies with traditional engineering disciplines. Further analysis however, shows two fundamental differences exist between the types of systems built by software developers and traditional engineers or architects, and those differences undermine the validity of

those analogies. The result is that the confusion surrounding software architecture can only be removed if a better understanding of software development as a whole can be achieved.

4.2 HyperEdit: A Case Study in Software Architecture

This section presents a case study in software architecture. The system under study is the diagram meta-editor system, HyperEdit, which is one of a collection of co-operating tools designed to support software engineering design activities. The development of that tool suite formed the HyperCase project (Cybulski and Reed 1992) of the Amdahl Australian Intelligent Tools Program (AAITP), a co-operative research effort between Amdahl Australia and La Trobe University.

This case study was originally intended to provide a tangible basis for a discussion of research ideas in software architecture. However, it soon became apparent that significant differences exist between theory and practice and the case study was expanded to investigate those differences. That investigation considered the design rationale used by the development team and identified software development issues that affected the architecture decision making process. The concluding remarks apply resulting insights to architecture research as a whole.

The HyperEdit application was chosen as the subject for the study because it was the first application to be developed within the project and it had a six-year period of evolution. That evolution was the result of revisions in its core functional requirements and to its responsibilities as a member of the co-operating tools within the evolving HyperCase project. That is, changes to the architecture of the larger system within which it operated. The term ‘large-scale application’ is obviously as relative one. The HyperEdit system contained requirements that resulted in subsystems that are common in many ‘large-scale applications’ – regardless of their size. They include the ability to communicate with other applications, the ability to manipulate large amounts of data that is stored in a remote repository, and complex GUI manipulation capabilities.

The study begins with a brief description of the global HyperCase environment and its evolution. Subsequent sections detail the evolution of the HyperEdit architecture. They include descriptions used in the original design documents and representations developed during maintenance procedures. Because developers other than the original implementers performed that maintenance process, the documentation has enabled identification of

system representations that were beneficial in acquiring the knowledge necessary to perform system modifications. The concluding sections discuss factors that influenced the architecture decision making of the design team and, finally, the results are compared with existing theories in software architecture research.

4.2.1 The Global HyperCase Architecture

The HyperCase environment is a loosely coupled collection of design support tools used to develop the documents produced during the software engineering process. In addition, all of the documents produced can be interconnected using a hypertext connection mechanism. In the original conception (Cybulski and Reed 1992) the tool suite consisted of a number of front-end authoring tools, a number of design support tools (such as configuration management and design reasoning recording), and management support tools (such as a project tracking (Cleary and Reed 1993)). Figure 4-1 depicts the architecture of the original system concept. The authoring tools, including HyperEdit, would communicate to the document repository (HyperBase) using the communication mechanism (EventTalk). The tools advise the HyperBase of all user-instigated changes to the documents; those changes are applied; and the front-end tools are informed to update their displays appropriately. Relevant support tools are then invoked, and finally, the HyperBase modifies the persistent data in the physical repository (HyperDict).

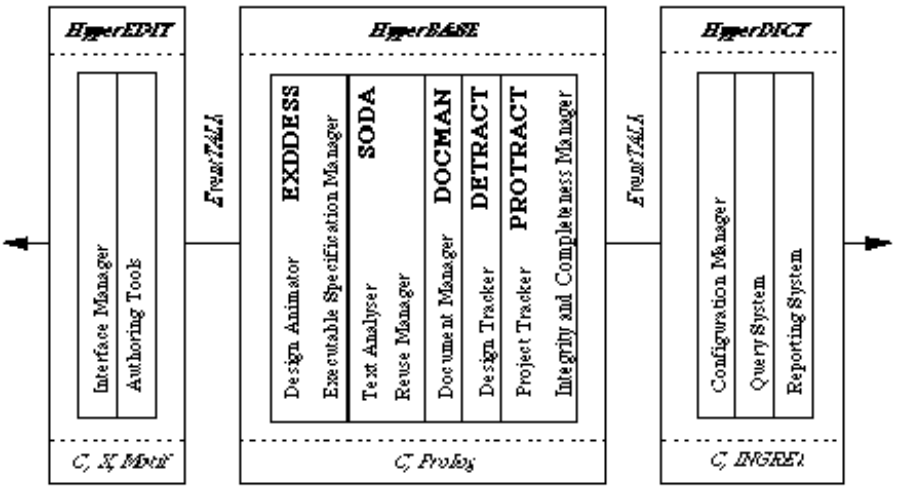


Figure 4-1: Original HyperCase Conceptual Architecture

The final architecture of the implemented HyperCase system turned out to be quite different to the original conception. It consists of a central message server, which co-ordinates communication between the clients of the system, a central repository that stores the majority of the persistent data in the system, and the series of application tools. Moreover, each user is supported by a link server that co-ordinates and creates the

hypertext links, and a hypertext server, which launches new applications, navigates through the system, and follows the hypertext links (figure 4-2).

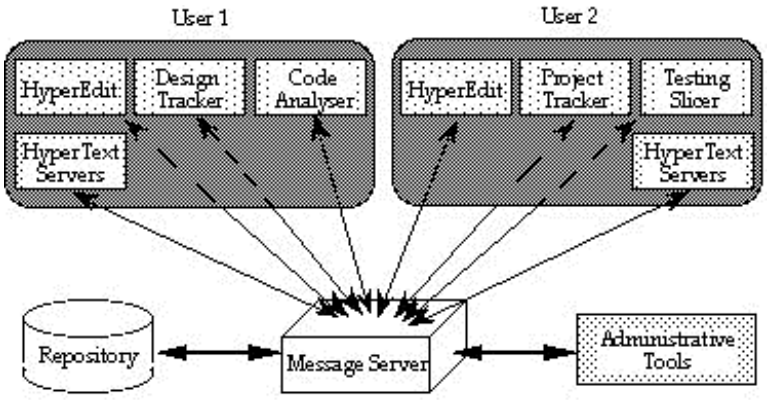


Figure 4-2: HyperCase Implemented Architecture

The message server, a subscribe/dispatch system, is responsible for all inter-client and client-repository communications. Client applications subscribe to particular classes of messages and all client-generated messages are addressed to either a particular message class or a particular application. That mechanism facilitates point-to-point, broadcast, and multicast methods of inter-client communication. An additional advantage of the message-server architecture is that individual tools are well insulated from changes made to other tools.

The core functional requirements of the original HyperCase concept were realised in the initial implementation. However, the gross structural arrangement is quite different from the original conceptual architecture. The implemented HyperCase system differed from its original conception for two reasons. These are worth noting because they relate to the HyperEdit case study.

- First, many of the changes were due to increased knowledge of implementation alternatives and system partitioning and intercommunication alternatives as the project proceeded.
- Second, the HyperCase project comprised a number of individual tools that were the product of independent Ph.D. research projects. The implementation of those tools proceeded at varying speeds and their designs were continually modified as the relevant research ideas evolved.

4.2.2 The HyperEdit System

In presenting the software architectures that were devised or utilised during the evolution of HyperEdit, the following phases are considered:

- The original system concept and requirements.

- Initial implementation.
- Reverse engineering and maintenance process.

The discussion is based on the textual descriptions and high-level graphical representations used in the documents produced during the various phases of the project lifecycle. In addition, individual project member's recollections of the architectures discussed during design meetings that were not recorded in project documentation were also utilised.

4.2.2.1 Original System Concept

HyperEdit was part of a research project whose aim was to produce proof-of-concept tools. Accordingly, the original requirements were stated as a loosely defined collection of ideas that evolved over time, rather than a rigorously defined specification document. The central requirement, as proposed by Jacob Cybulski, was to produce a graphical editor editor. That is, an application that would allow the graphical creation of the tools required to produce the software engineering diagrams developed during a project's lifecycle. The process of defining the specific editors was to be purely graphical. The concept became known in the project as the 'graphical definition paradigm' and was a major intellectual challenge.

Conceptually, the system would be started in a meta-editor mode that would allow the definition of a particular type of diagram editor. That definition process would consist of the design and construction of the graphical objects to be manipulated during the creation of a particular diagram style. For example, the creation of a state-transition diagram editor would require the identification of the objects that exist in that style of diagram, the states and transitions, and their particular attributes. In addition, it would require the identification of the operations that could be performed on those objects by the editor. For example, the ability to change the visual appearance of the

> *"HyperEdit, a graphical editor construction tool, ... incorporates windows, menus, object-oriented graphics, and mouse control. Unlike other tools, its customisation is fully interactive and totally user-driven, there is no need for HyperEdit or the application re-programming, recompilation, nor relinking. Such extendibility is achieved by the availability of a HyperEdit meta-editor in which users can define tailor-made editors, their window and page layout, the type and look of editable graphical objects, finally the editor buttons, controls, menus and their behaviour."* (Cybulski and Proestakis 1991)

objects, such as text annotations, and the ability to connect the objects together.

The definition of specific editor objects required the ability to define the following functionality through graphical manipulation:

- The definition of the visual appearance of the objects.
- The definition of object customisation. E.g., which attributes will be modifiable in the eventual specific editor (text fields, line colours, fill styles, etc).
- The definition of connection rules. E.g., the implementation of syntax-directed editing through the specification of how objects can connect to other objects.

The definition of specific editor functionality included:

- The definition of the Graphical User Interface (GUI) layout.
- The definition of the functionality available to the user, specified through possible menu items.

After defining the objects, connection rules, and available functionality of a particular diagram editor, HyperEdit could be started as that specific editor.

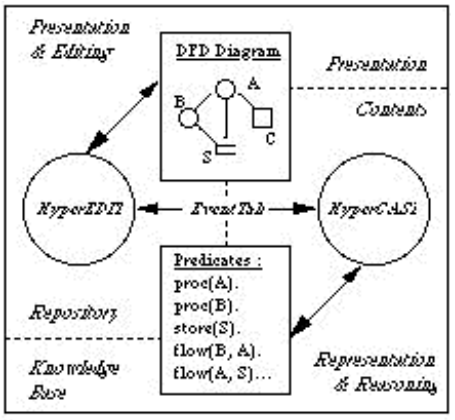


Figure 4-3: HyperEdit Conceptual Architecture

The original design goal included a requirement to provide a clear delineation between the visual representation of a diagram and its semantic content. It would then be possible to utilise HyperEdit to represent the same structured, software design document in terms of different software diagramming techniques (figure 4-3). For instance, a textual representation of object definitions would exist in the repository and they could be represented using particular flavours of object-oriented analysis methods (e.g., Booch, Rumbaugh, or UML). Moreover, it was a requirement to dynamically update the visual presentation and diagram editor definitions without needing to shut down and 're-compile' the system. For example, if the connection rules of a particular diagram style were modified or a new component was added to the palette of a particular editor, those changes would be propagated and incorporated into currently executing HyperEdits. Because multiple instances of a specific HyperEdit editor could be running simultaneously, the system was required to communicate with other HyperEdit

instances, in addition to the repository, to update object, diagram and editor definitions while the application was executing. Furthermore, the multiple executing HyperEdits could be distributed across many networked machines.

The document describing the original HyperEdit concept (Cybulski and Proestakis 1991) contained a section describing the 'system architecture'. It details the major functional components of the system. Unfortunately, it does not include a graphical representation.

> "Three major sub-systems may be identified in the HyperEdit architecture, ISDUIMS, SAT and EventTalk. Their brief description follows:
>
> - SDUIMS (ISD User Interface Management System): The core of HyperEdit presentation layer consists of a number of text and graphic primitives. The primitives are of sufficiently high level to facilitate functional expression, ease of use, and flexibility in the creation of windows, dialogue boxes, menus, palettes, buttons, text and graphics, all to be mouse and keyboard controlled. …
> - SAT (Systems Analysis Tools): To assist with quick acceptance and efficient cross-over to HyperEdit, ... the system is equipped with a number of standard text and graphics editors, which could be used in the construction of system analysis and design documents. E.g., data flow diagrams, entity-relationship diagrams, … etc.
> - EventTalk: HyperEdit allows full control over text and graphics editors from some other user program via a specially devised communication protocol – EventTalk. The main objective of EventTalk is to advise the controlling program of all user-instigated changes to the document contents associated with the creation, deletion and editing of its components … so that it could perform validation of user actions." (Cybulski and Proestakis 1991)

4.2.2.2 Initial HyperEdit Implementation

The initial implementation of HyperEdit differed from its original conception in terms of both functionality and gross structural architecture. The core requirements, the ability to define, produce, and utilise specific software diagram editors, and support the graphical object definition of the components contained within those editors, was achieved.

However, the ability to customise the functionality of specific editors was not implemented. Moreover, the clear separation between the visual presentation and the semantic content of diagrams was not attained. Nevertheless, the primary goal, the ability to graphically create and utilise different diagram editors, was realised.

Figure 4-4 depicts the utilisation of the implemented HyperEdit system. The diagram on the left shows HyperEdit in its object-editor mode being use to create a new 'documentation object', which will be used in entity-relationship diagrams. Its visual appearance is synthesised from a collection of graphic primitives and its specific behaviour is enabled by providing menu interfaces to standard graphical object functionality. The object is subsequently utilised in the entity-relationship diagram, which shows the documentation object having the ability to modify the value of its text and the ability to use its visual attributes to depict some user-defined status. Those abilities were specified during the object's creation.

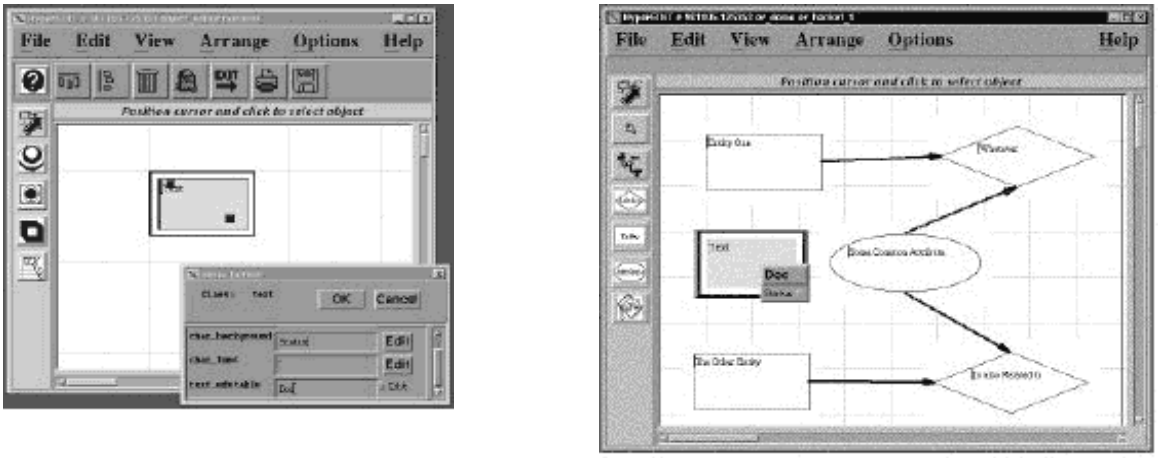


Figure 4-4: HyperEdit Object Editor and Entity Relationship Editor

In the project documentation the architecture of the implemented system was depicted using two different graphical representations. The first was a layered system that described the gross structure of a single executing HyperEdit with well-defined interfaces between the major components (figure 4-5) (Proestakis 1993).

- The HyperEdit layer provides the collection of diagram manipulation functions.
- The ET (EventTalk) library is the protocol level of communication between the particular HyperEdit and the other executing HyperEdits and the repository.
- The Message Server layer is the physical, distributed communication mechanism, which was implemented in a Blackboard style.

- The Database Server provides the functionality necessary to interact with the physical repository.

Figure 4-5: HyperEdit Layered Architecture

A second architecture was required to represent the operation of multiple HyperEdit instances executing on distributed machines (figure 4-6). That architecture depicts the distributed nature and communication requirements of the system, which are hidden in the abstraction model of the layered architecture.

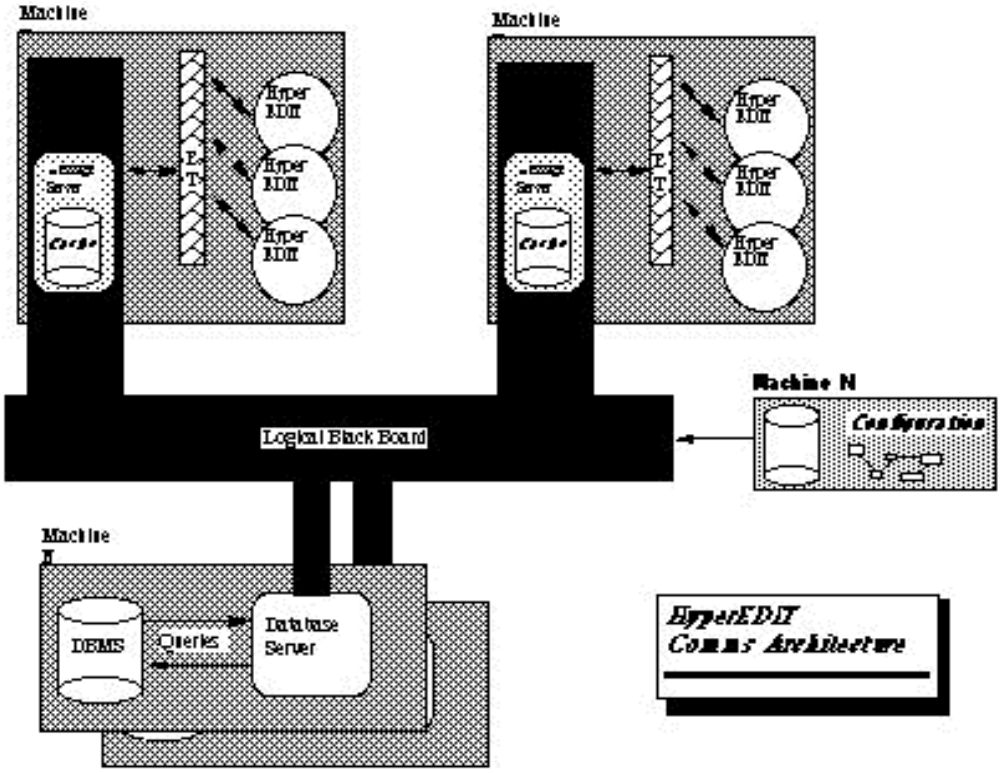


Figure 4-6: HyperEdit Distributed Communications Architecture

The need for different representations of the system highlights two interesting architectural issues. The first concerns why those two architectures were required and the differences between what they represent. The second concerns the types of connections that exist between the components of software systems.

The layered architecture uses individual layers to represent the major functional components that exist in any single instantiation of the system. The complexity of the communication detail is hidden in the abstraction of the 'ET library' and 'Blackboard' layers, while that detail is explicitly depicted in the distributed blackboard architecture.

This use of abstraction is a well-known property of particular architecture views and is a consequence of information hiding in good software engineering design.

However, unlike the layered architecture, the distributed blackboard architecture shows the HyperEdit system can consist of multiple instances of itself. Those multiple instances can run simultaneously on distributed machines, communicating with each other as well as with a common data repository. This feature of software architecture – the ability of a system to be connected with multiple executing instances of itself – does not exist in any other discipline. An opposing argument could cite two electronics chips of the same type connected together. However, in the HyperEdit example, and for software systems in general, two or more separate software instances can be executed and communicate with each other from a sole, implemented, executable program file[15].

The other interesting architectural issue concerns the types of connections that exist between the components of the software system and their visibility in the source code implementation. In the layered architecture, there are differences between how the individual layers are realised and how they interface with each other. At the highest level, the HyperEdit engine and EventTalk (ET) routines exist in the same executing HyperEdit process. The program modules that explicitly implement those layers are evident in an

"The logical black board system is implemented through a network of local message servers. There is one message server on every node running one or more HyperEdit editors. The messages are distributed through the database server, which acts as the governing executive / message distributor for the black board system. The message server therefore has several functions. Firstly, it acts as the interface mechanism between the database or future knowledge base and the executing HyperEdits; ... Secondly, it provides a local storage mechanism, ... Thirdly, the message server acts as the local controller for hypertext operation."

"The message server is implemented using remote procedure calls, as is the database server. ... At this point, there are only two remote functions, ETBBSendMessage and ETMessageReceived. ... On receiving a remote call, the server interrogates the type of message and acts accordingly. … The HyperEDIT editors then process the message and send a reply to the black board executive (database server). … The database server then collates the replies, determines if any further actions are required and sends control messages to the message servers." (Proestakis 1993)

[15] This unique feature of software systems is discussed in more detail later in the chapter.

examination of the source code and the connections between components are procedural invocations in that source code.

In contrast, the blackboard system operates as a separately executing process and is used by HyperEdit to facilitate the communication between individual HyperEdit applications and the repository. The blackboard system has an instantiation on each node of the distributed system. Unlike the HyperEdit engine and ET routines, the source code that implements the blackboard functionality is not found in the HyperEdit source code. However, the interface calls to the blackboard mechanism are evident. To utilise the blackboard 'layer' the HyperEdit application must incorporate a library of routines that implement the interface to that layer at link time. This is due to the actual implementation of the blackboard system. The concept of link-time incorporation of library routines in software implementation and execution is such an established part of software development that it does not appear noteworthy. However, because the concept does not exist in other disciplines, and the design and construction processes of those disciplines serve, at least partly, as the basis for developing architecture theories for software engineering, it needs to be considered.

The implementation of the message servers, which exist within the blackboard communication mechanism, are realised using operating system pipes and remote procedure calls. Consequently, the HyperEdit layer and the Blackboard layer operate as two distinct system processes that communicate through operating system, rather than source code, connections. The interface connections between the layers are evident in the procedure calls to the appropriate routines. However, the implementation of the Blackboard layer exists in a separately executing process. Again, this is not a feature of the design and construction of corporeal artefacts.

The source code modules that implement the final layer of the architecture, the Database Server, similarly do not exist in the HyperEdit source code. However, unlike the blackboard mechanism that becomes a layer of the application during the linking stage, the invocation of the database interface routines is not evident anywhere in the HyperEdit code or Blackboard layer code.

The database server is a separately executing process that may exist anywhere on the network. It interfaces to the blackboard layer by linking in the appropriate library of routines. The communication between HyperEdit and the appropriate database routine is

> *"The final component [the DBMS] is responsible for the information storage and retrieval requirements of the HyperEdit system. This layer also introduces a further layer of abstraction, as its implementation is not database or file management method specific. That is, the server's interface is static, but the implementation of the data access routines is flexible, in terms of interfacing with the user's site storage mechanisms." (Proestakis 1993)*

made possible by sending a message through the blackboard interface, which is then interpreted by the database server. The appropriate database routine is not invoked by a direct procedure call from the preceding layer. It is invoked by interpreting attributes of an abstract data type that is read from the blackboard. That is, the routine to invoke is determined by analysing the value of the data that is passed through the connection. There is no explicit path of procedural invocation that can be traced from the HyperEdit engine routines, through the ET interface, to the blackboard interface, and finally, to the database server. This is evident in the text-box description of the Database Server (DBMS) implementation.

To summarise, three types of software component connection were identified:

1. Source component to an internal source component: Those procedural invocations exist between source modules that both exist in the implemented source code. Furthermore, large-scale abstractions of component connections are explicitly evident in the source code.

2. Source component to an explicit, external source component: Those procedural invocations connect source code modules with other modules that have been implemented in external libraries. Those libraries do not become part of the system until they are linked. Moreover, they may be third party packages for which there is no available source code implementation. Therefore, the architecture abstractions are based on components that are known about in the source code but the details of those components may only be evident in the machine code of the system.

3. Source component to an implicit, external source component: A module invocation exists, as in the previous two cases, however it is used to pass on a data structure to an auxiliary software system. The auxiliary software system uses the value of the data structure to determine which internal module should be invoked. In this situation, architecture connections are based on components that may not be known about by

the source system and are only evident in the executing machine code of external processes, possibly running on other machines.

This range of connection types highlights the way in which software architectures abstract away unnecessary implementation detail. However, it is worth noting that these types of component connections are not evident in other engineering disciplines. Moreover, it highlights the difficulty in identifying the boundaries between communicating software systems.

4.2.2.3 Architectures used during System Maintenance

A large amount of maintenance has been performed on HyperEdit since its initial implementation. That was due to the prototype nature of the system, changes to its operating environment, and changes to some of its core functionality. To facilitate the maintenance process, high-level system depictions of both its implementation and operation were developed. Developers other than those responsible for the original implementation did the majority of the maintenance performed on HyperEdit. Those developers had a thorough understanding of the system's requirements and functionality, however they did not possess a full working knowledge of the system's implementation or internal operation. Therefore, a process of reverse engineering was required to develop system representations that would assist in deriving the knowledge necessary to make the required modifications. The resulting system depictions are those derived during that process and were actually used to assist the maintainers rather than being created solely for the purposes of satisfying documentation requirements.

During the maintenance process, the developers needed to perform two basic activities to gain the required understanding of the implementation. First, they needed to identify where functional concepts were implemented in the source code. For example, a bug existed in the code that implemented the resizing of components on the screen. The maintainers, familiar with the system functionality, possessed knowledge of what concepts were involved and the instances of operation that would cause the bug to manifest itself. However, locating where those concepts were implemented in hundreds of thousands of lines of source code, which had been separated into different procedures, modules, files, and directories by someone else, was not a trivial exercise. The second type of activity required was the ability to trace how the operation of the application

caused the flow of control to move throughout the implemented source modules. Due to the event-based nature of the programming environment, that also was a nontrivial task.

To support those maintenance activities, two large-scale structural representations of the application were created. The first allowed the maintainers to identify the location at which modifications in the source code were required. The HyperEdit system was implemented in a Unix environment using C and X/Motif as the implementation medium and a relational database as the repository. The architectural depiction generated by the maintainers consists of a hierarchy of informal block diagrams detailing the module calls throughout the application. The highest-level diagram contained blocks for each of the C files and X libraries with the connections representing module interactions across the files. Each file has its own refinement diagram to represent the source code modules that are defined in each file and interconnections depicting module interactions. The cross-reference and function definition information was generated using standard Unix code analysis tools (*ctags* & *cxref*). The call-graph architecture diagrams were automatically generated using HyperEdit itself. Furthermore, a WAIS (wide-area information server) search facility was used to search the source code. The maintainers were able to search for specific concepts in the source code and locate routines in which those concepts were defined or utilised. The static, source code call-graph architecture was then used to locate the appropriate file and source code modules to modify. Moreover, its immediate dependencies could also be viewed. That representation of the system's static, source code implementation showed the cross-reference information required to perform many of the maintenance tasks. However, only interconnections visible as explicit procedure calls were visible. It was not possible, for example, to follow the event-based nature of the X/Motif GUI environment because the information necessary was hidden in the precompiled libraries of that GUI environment.

The second maintenance architecture was used to represent the control flow through the application. The HyperEdit application was graphically intensive and required the implementation of many low-level graphic manipulation routines. The decision to use X/Motif as the GUI toolkit constrained the developers to its

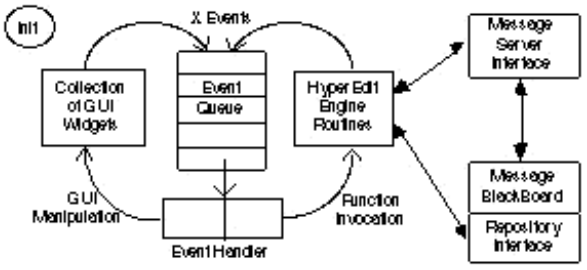


Figure 4-7: Event-Based Operating Architecture

event-based programming model. That model hides many of the direct procedural invocations that are explicit in procedural programming languages and that made it difficult to trace the connection between GUI events and source code invocation. An X/Motif system begins operation by invoking a sequence of initialisation routines to create the infrastructure for an event-based execution environment. That infrastructure includes the creation of the initial graphical user interface and the instantiation of the event loop system. Once the application sets up its execution infrastructure it goes into a loop in which the 'Event Handler' provides the two-way causal relationship between GUI manipulation events and procedure invocations in the user-defined source code (the HyperEdit engine). The architecture depicted in figure 4-7 was utilised to represent that event-based operation.

In that architecture four different areas are identified:

- Initialisation Process: The collection of routines required to create the operating infrastructure needed for the GUI environment.
- Graphical User Interface: The collection of widgets on the display that are manipulated by the user input to generate events. They include the routines present in the X/Motif libraries that perform the low-level graphical manipulation that result in visible changes on the display.
- Event Loop System: Consists of the 'Event Queue' and 'Event Handler'. Events are sent to the queue from the GUI and the application 'Engine'. The 'Event Handler', based on the information defined during the Initialisation process, processes them appropriately. The concept of 'callbacks' allows the translation of GUI events to application Engine procedure invocations. That is why it can be so difficult to debug X/Motif applications. The abstractions make it extremely difficult to follow causal links between code invocations and tangible manifestations on the display and vice-versa.
- The Application Engine: The collection of routines that perform the processing required when a particular piece of functionality is invoked from the user interface. That includes data modification and transactions with the repository, interfacing with the message passing system, and generating appropriate X/Motif events to realise the runtime GUI dynamics.

Both the dynamic operation architecture and the static, call-graph implementation architecture were useful during the system maintenance process. The call-graph representation of the system used large scale abstractions based on the building constructs provided by the implementation medium – source code statements, functions, and C files. In contrast, the dynamic operation architecture used abstractions based on the run-time, functional groupings of the routines rather than where they exist in the source code implementation. Both are valid representations of the system architecture, yet they graphically depict completely different collections of concepts.

4.2.3 Maintenance That Affected the System Architecture

Three major modifications to the HyperEdit system and their effect on the system architecture are detailed. The first modification, a change in the application's communication mechanism, resulted in a major modification to the system architecture. The second modification required the addition of an application interface that would allow HyperEdit to be controlled by a remote manipulation tool. That modification resulted in a redesign of the HyperEdit architecture, however due to resource constraints the necessary changes to the large-scale structure were not made. Consequently, the redesign was compromised to make the modifications fit the existing architecture. The final modification was the extraction of hypertext functionality to exist as a separate tool. Whilst it was considered to be a major modification, there was no change evident in the system architecture. Those modifications are summarised and their consequences for software architecture in general are discussed.

4.2.3.1 Changing the System Communication Mechanism

As the project evolved the original blackboard style architecture was found to be inadequate and was replaced with a more robust, flexible, and efficient message passing system. The HyperEdit application required two types of communication. First, the ability to communicate with the repository to load and save data and second, the ability to communicate with the other executing HyperEdits in the overall environment.

The initial choice of communication mechanism was the blackboard-style configuration described previously (figure 4-6). However, during actual utilisation of the system, the blackboard architecture did not exhibit satisfactory performance. The problems associated with the practical use of the system were caused by the prototype nature of the tools using the connection mechanism and the number of messages sent by those clients.

The blackboard server had the responsibility of keeping track of all the currently executing HyperEdit processes and ensuring that the current blackboard message remained available until all processes had polled the blackboard for that message. Moreover, the blackboard server had to ensure each process accessed the current message only once. In theory, the blackboard architecture should have sufficed in a conventional and fully operational environment. However, because the project was of a prototype nature, with tools whose stability was initially unreliable, keeping track of the currently executing HyperEdits was a difficult job. In addition, ensuring that each process read all messages only once was difficult when each process polled the server at different intervals and could disappear at random times.

The communication mechanism not only had to meet the needs of the HyperEdit application but also the requirements of the other applications in the HyperCase tool suite. It became apparent as those tools evolved that the aforementioned problems with the blackboard style architecture would only be exacerbated. Moreover, as the research, design, and implementation of those other tools proceeded it became apparent that the global connection mechanism would need to support point-to-point, multi-cast, and broadcast communication styles between the tools. With the magnitude of the number of messages envisaged, the utilisation of a single storage location, which all tools would be required to access, would be inefficient from a performance perspective.

The blackboard architecture was replaced with the current subscribe/dispatch message server topology (Baragry, Cleary et al. 1994) (figure 4-2). In that arrangement, all messages in the system are categorised into particular classes and each tool in the HyperCase tool suite subscribes to the classes they are interested in. The messages generated by the individual tools are assigned a particular message class before being sent to the server and the server subsequently dispatches those messages to the message queues of the tools subscribed to that class.

The message server topology had a number of advantages over the original blackboard style. Most importantly, it satisfied the range of communication styles required by the individual tools. In addition, because the responsibility for storing the messages was shifted to the individual tool's message queue, rather than the single storage location of the blackboard arrangement, the storage and processing requirements of the central message server were significantly reduced. The result for the message server, which was

the bottleneck of the system, was improved communication performance and insulation from the occasionally unreliable nature of some of the evolving tools.

The modification of the communication mechanism also solved a number design principles and conceptual integrity issues that had been compromised during the original implementation process. HyperEdit was the first tool to exist in the tool suite and its implementation and the implementation of the communication mechanism occurred concurrently. As that implementation proceeded, the clear delineation between the communication responsibilities of HyperEdit, the repository, and the blackboard system became blurred. The replacement of the communication mechanism provided the opportunity to ensure its conceptual integrity by completely separating it from the HyperEdit-specific functionality and ensuring it existed as a separate process within the global HyperCase environment. That enabled future modifications to occur to the message server without requiring the downtime of HyperEdit.

4.2.3.2 The Addition of a Remote Manipulation Interface

A major modification was made to HyperEdit to create an interface that would allow the remote generation, manipulation, and analysis of HyperEdit documents. An example of that functionality was the creation of a visualisation tool for Amdahl Corporation's ObjectStar rule-based development environment (Amdahl 1998). A HyperEdit editor was defined to create diagrams to depict the high-level structure of ObjectStar applications. The visualisation tool was used to analyse an ObjectStar application's source code and subsequently send messages to an executing HyperEdit to automatically construct the high-level graphical representation of that system. Another HyperEdit editor was defined to represent the fine-grained rule language of ObjectStar modules. Each module in the high-level system representation was then automatically linked, using hypertext, to a visual depiction of the source code for that module. Finally, by using the message server functionality, the visualisation tool was also able to receive events from the HyperEdit editors when user-instigated modifications to the visual representation of the system were made. Those modifications could then be passed to another tool to automatically replicate the graphical changes in the physical source code of the system. The user was subsequently able to graphically navigate through and modify an ObjectStar application. The ability to remotely manipulate and analyse HyperEdit diagrams provided the ability to develop an ObjectStar application using both textual and graphical representations with changes made in one medium automatically replicated in the other. Moreover, because

the graphical representation and manipulation tool, HyperEdit, was completely separate from the tool that analyzed a particular programming environment, similar systems could be developed for any language or development environment without requiring any modifications to the HyperEdit system.

The implementation of the remote interface required two architecture-level analyses of the HyperEdit system. First, a means of communicating between an executing HyperEdit and its remote manipulation tool had to be provided. Second, once the HyperEdit process had received a particular message, a means of invoking the appropriate diagram manipulation routines had to be implemented. The replacement of the blackboard communication mechanism with the message passing style architecture satisfied the communication requirements between executing HyperEdits and remote manipulation tools. However, identifying the appropriate source code routines to invoke for diagram manipulations was not as simple because of the high number of dependencies between the X/Motif GUI code and the internal data manipulation code within the HyperEdit Engine. Therefore, the implementation of the remote manipulation interface required a re-design of the large-scale system arrangement of the HyperEdit engine routines to provide a clear interface between the GUI manipulation code and the internal diagram manipulation functions.

The conceptual arrangement of the new system would make it possible to invoke any piece of functionality from the remote interface that was available from the existing GUI. Although the diagram itself was stored as an easily identifiable data structure in the source code, the routines that manipulated that data structure were a mixture of user-defined code and pre-defined functionality provided by the X/Motif GUI environment. The existence of that mixture, and the high number of dependencies caused by it, was a result of designing the system to utilize the X/Motif GUI system. To provide a clear interface between the GUI/X environment and the user defined source code, the block entitled “HyperEdit Engine Routines” in the dynamic operation architecture of HyperEdit (figure 4-7) was separated into two distinct parts. The first, the GUI Callbacks module, was responsible for dealing with the reception of the X events that represent the user’s manipulation of

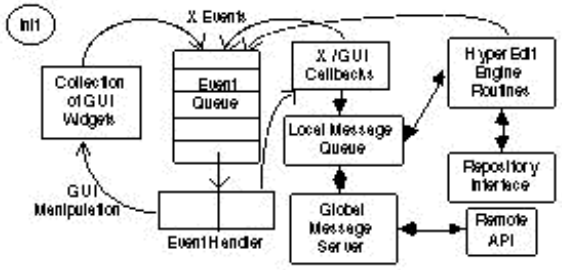


Figure 4-8: Redesigned Event-Based Operation

the system at the GUI. The second, the 'new' Engine, contains only user-defined source code routines that make changes to the internal diagram representation and appropriate updates to the GUI. The new engine would receive its invocations solely from the local message queue. With that conceptual arrangement, the source of the message sent to the engine could either be from the system's GUI callbacks or from any other tool that could generate messages compliant with the HyperCase environment's message server. The new structural arrangement is depicted in figure 4-8.

The implementation of that conceptual architecture did not eventuate. From a technical point of view, the design appeared to meet the requirements. However, from a management perspective, the human and financial resources did not exist to allow such major modifications to the system. Nevertheless, the functionality to remotely manipulate the tool was important and a compromise was reached. The amount of available functionality was reduced to meet the capabilities of the existing architecture.

The message passing system was utilised to send remote calls to the appropriate HyperEdit, however the translation of those messages into appropriate function invocations did not occur through a distinct, abstract interface. Some remote requests had direct functional implementations in the HyperEdit engine. For instance, a request to add an object to the current document at a particular location was quite simple to satisfy. An equivalent procedure already existed in the collection of engine routines. When a message of that type was received, the equivalent engine function was invoked with the message attributes used as the appropriate parameters. In contrast, a message requesting the addition of a joining flow or edge between two existing objects on the screen was not as easy to achieve. The code to implement object connections was written in terms of the many X events that are generated throughout the joining process in the GUI. Each mouse click or drag performed in the joining process on the GUI would generate events. Those events resulted in procedure invocations in the engine that stored information as the process proceeded. The joining process in the engine was an aggregation of all those generated events and resulting procedure calls. No single function existed that accepted two particular component identifiers and the relevant connection attributes to produce the appropriate flow. That original arrangement occurred as a result of designing the system to efficiently utilise the implementation constructs provided by the GUI environment. To change that arrangement, the remote manipulation routine was required to simulate the generation of X events that occur during the GUI-based joining of components. That was

achieved by breaking the message into its constituent parts and invoking a number of low-level engine routines.

In the end, the overriding resource constraints forced the conceptual integrity and design principles to be compromised. It was not conceptually pure, but it worked. While that compromise was acceptable because of the prototype nature of the system, the consequence is code that is harder to understand and modify. Finally, from an architecture perspective, the compromise resulted in modifications that occurred at a level which had no impact on the gross structural arrangement. Therefore, the new functionality is not evident in the representation of the implemented architecture (figure 4-7).

4.2.3.3 Extraction of the Hypertext Mechanism

The final major modification to be described was the extraction of functionality that implements the hypertext mechanism. This modification was performed to change the hypertext functionality from something of limited capability and whose implementation was tightly coupled to the application and its data, to something that existed as a separate tool and provided far greater functionality. The motivations, design alternatives, and ramifications for the software architecture are discussed.

A requirement of the global HyperCase system was that all components of the documents produced during a software development lifecycle should have the potential to be the source or destination of a hypertext link. That would enable the developers, maintainers, and project managers to traverse the logical relationships through the document base regardless of the document types. Links would connect graphical design documents, source code modules, requirement statements, design rationale, and project management schedules.

The initial implementation of the hypertext functionality in HyperEdit was quite primitive compared to the current version. A conscious design decision was made initially to keep the hypertext functionality as simple as possible until ongoing research within the project had fully completed the hypertext requirements of HyperEdit/HyperCase. The hypertext information was initially realised by keeping it closely coupled to the HyperEdit document components. The implementation of each generic diagram component contained an attribute to store link information. That information provided details of connections either to other HyperEdit diagrams or to other documents and the tools required for displaying them. The HyperEdit application contained the appropriate

functionality to modify that attribute information and to process the information when the link was followed. While that implementation was adequate for dealing with the HyperEdit system, it was understood that it would not satisfy the hypertext requirements of the other emerging tools in the HyperCase tool suite. The design rationale of the initial implementation however, ensured the functionality was easy to replace when the full requirements of the system were developed.

Two high-level issues had to be resolved to implement a more versatile hypertext system. First, how should the conceptual data model be organised to store the hypertext information? Should hypertext links be stored as attributes of the source component or should those relationships between components be stored as first-class components themselves with the source, destination, and other useful information recorded as attributes of the link. A good example of the differences between those two alternatives is evident in the implementation of global information systems. The most popular, the World Wide Web (WWW), uses a very simplistic hypertext model, where the link information is stored within the data, i.e., as tags in the HTML document. Alternatively, another global, hypermedia-based information system, Hyper-G, stores its hypertext information separately from the data shown on each page (see for example (Flohr 1995)). The Hyper-G client is responsible for associating the data and hypertext information in a seamless manner that allows the user to navigate the information base. The Hyper-G hypertext model provides greater flexibility and functionality than the WWW model. However, the trade-off is that Hyper-G systems require more intelligent clients and servers to store and display the information.

The second design issue to be resolved was where the processing capability required to generate, modify, and follow hypertext links should be implemented in the conceptual architecture. Should all of the tools replicate the hypertext functionality, or should a single tool be developed to handle all hypertext operations? Those issues were solved through research in hypertext-based, information system design that was performed within the project at the time (Cooper 1996). In addition, internal design meetings solved problems relating to the application of the concepts developed in that research to the specific problems faced.

To resolve the first issue, a decision was made to have a data model in which the hypertext link information was kept separate from the software document components. That allowed for far greater flexibility when dealing with the hypertext information. The

hypertext information became 'first-class' objects rather than simply attributes of other data. Architecturally, it is interesting that the result of that high-level decision, which influenced many aspects of design, and set constraints on the hypertext functionality of the system, is not evident in the conventional architecture representations. The implemented architectures of HyperEdit remain identical regardless of which data model had been chosen. Whilst data models are used extensively in software engineering design, it is interesting that they are not considered part of the system architecture, even though the functionality of the system is often devised with respect to the data model envisaged.

The resolution of the second design issue, assigning the responsibility for the hypertext functionality, occurred with the decision to construct a single tool to deal with the creation, deletion, modification, and traversal of hypertext links. The primary design constraint on the decision was flexibility. The production of a single hypertext server resulted in the majority of the functionality residing in a single location, rather than being replicated in all of the tools. Consequently, new tools added to the environment would only require relatively minor functional enhancements to make them compliant with the hypertext system. Moreover, future modifications made to the hypertext requirements would be implemented in a singe tool rather than in all HyperCase tools. Again, it is interesting to see the effect of that system modification on the HyperEdit architecture. The source code to realise the hypertext functionality was removed from the 'HyperEdit Engine Routines' (figures 4-5 & 4-7) and now exists as a separate tool in the HyperCase tool suite. The execution of the tool suite requires each user to have a 'HyperText Server', which is responsible for the hypertext operations of all the tools used by that user. The HyperText Server is represented as a high-level module in the global HyperCase architecture (figure 4-2). However, representations of the individual HyperEdit architecture (figures 5 & 7) fail to show that module, even though its existence is required for the successful operation of HyperEdit. The situation in which a major component of the system is seemingly invisible in the large-scale representation causes us to repeat the often-asked question, "What constitutes software architecture?"

4.2.4 Factors That Influenced Architecture Decisions

With the benefit of hindsight, and with the knowledge gained from improved understanding in software architecture research, it has been possible to identify factors that played an influential part in how architecture decisions were made in the evolution of

HyperEdit. Those factors were not necessarily explicit in the design reasoning at the time, however under the spotlight of the case study their importance has now been recognised. The factors discussed are:

- The effect of changing requirements on the architecture as the design progresses.
- Knowledge of available architecture alternatives and their practical consequences.
- The effect of the implementation environment on the designer's ability to choose different architectures.

The subsequent discussion details the degree to which those factors are specific to the project under investigation and those which are applicable to software development in general.

4.2.4.1 Changing Requirements

The effect on the architecture of changing system requirements poses some well-known but largely unanswered questions. The modification that added the remote manipulation capability to HyperEdit is an example that highlights the reasons why some changes require architecture modification and others do not. As mentioned in the previous section, the modifications that implemented the remote manipulation capability can be divided into two parts. The first was the ability to communicate with HyperEdit and the second was the ability to invoke the appropriate internal functionality.

The message passing style communication architecture of HyperEdit has proved to be quite resilient to changing requirements. To implement the remote manipulation interface the infrastructure required to communicate with HyperEdit was already present because HyperEdit was already able to communicate with other executing HyperEdits and with the repository. To realise the remote manipulation communication all that was required was a new communication protocol and the ability for HyperEdit to distinguish between HyperEdit/repository messages and the remote manipulation messages. The required changes took place at a single level of design abstraction and were implemented without architecture modification. The conceptual design of HyperEdit contained a single subsystem, the message server, which dealt with the inter-tool communication protocol. Analysis of the flow of control through that subsystem shows clear interfaces between it and the rest of the HyperCase/HyperEdit system. Implementing the modifications required for remote manipulation communication did not affect how the flow of control

moved around the global system. Similarly, new tools could, and have, been added to the global environment without modifications to the global architecture. Moreover, other inter-tool communication protocols have been specified without architecture modification. By looking at the conceptual design of the system, modifications that only affect a single level of design abstraction and do not affect the flow of control through that conceptual model were implemented without changes to the system architecture.

In contrast, the modifications needed to process and implement the remote manipulation messages required changes to the interaction between the dedicated HyperEdit code and the X/Motif environment. Because of the way those components were initially implemented, the required modifications resulted in design changes to the flow of control within the subsystems that comprise those architecture components. A brief discussion of the interaction between the user-defined code and the X/Motif environment is needed to explain the issue. The utilisation of the X/Motif GUI package required the developers to conform to the event-based programming model provided by the GUI system. In that model, manipulations at the GUI front-end of the system result in procedural invocations in the source code via the X event handler, and vice versa (figure 4-7). The application contained a great deal of user-defined source code that was independent of the GUI specific code. For example, code for message and data processing. In addition, the complexity of the graphical manipulation requirements of the application resulted in the use of many high-level and low-level graphical manipulation routines provided by the GUI toolkit. Consequently, the interaction of the user defined source code with the GUI environment was spread across many levels of conceptual granularity. Furthermore, because the flow of control through the system is subject to the event-based model of the GUI environment, the point at which control was passed to the source code from the GUI occurred at many different levels of conceptual granularity. The changes required to implement the remote manipulation capability would require changes to the point at which the flow of control was passed through to the user-defined code. The reason architecture modifications were required to realise that part of the remote manipulation capability was because changes would need to be made at multiple levels of conceptual design abstraction.

Making an architecture resilient to requirements change that affect many levels of design abstraction appears to be considerably harder than making a design malleable when the modifications only affect a single level of abstraction. A characteristic of mature domains

is that they consist of well-understood collections of concepts and their relations. When determining the potential for future changes to result in modifications to the system architecture it is possible for the designer to envisage how those interconnected concepts will cope. However, this is a lot harder to do in domains that do not have well-understood collections of concepts.

4.2.4.2 Knowledge of Architecture Alternatives

One of the major modifications to HyperEdit was the replacement of the blackboard style communication mechanism with the subscribe/dispatch message-passing arrangement. During the course of this case study it was interesting to look back at the design meetings and documents to understand why the blackboard architecture was selected when hindsight shows the decision was inappropriate. The result has been the discovery of how knowledge of possible architecture alternatives influenced the designers' ability to choose the initial architecture. Moreover, it highlights how difficult it is to determine how a proposed architecture will perform before it has been implemented.

As is often the case in large team projects, many aspects of the design become matters for debate and opinion. The design of the communication mechanism for HyperEdit, and the surrounding HyperCase environment, was such a case. The need to allow for flexibility was recognised at an early stage. The requirements stated that tools would have the ability to communicate explicitly with each other and that specific inter-tool communication should be able to be monitored without the sending or receiving tools being aware of it. For instance, as a HyperEdit tool communicated with the repository, the project tracking tool would be able to 'eavesdrop' on the ensuing communication and automatically track the developer's progress, updating the relevant project management charts and reports appropriately (Cleary and Reed 1993). Two high-level alternatives were considered to satisfy those requirements: the blackboard architecture that was eventually chosen, and a dispatch-messaging approach, similar to the one that eventually replaced the blackboard system.

At that early stage in the development process, both high-level design alternatives appeared to satisfy the functional requirements. The blackboard arrangement was selected over the message passing infrastructure because some members of the team had had direct experience in implementing a blackboard system in the past. The developers knew how to implement the high level concepts of the blackboard arrangement while the

implementation of a message passing arrangement would require more design work to determine how it could be implemented.

The lack of knowledge of different architecture alternatives was also a significant factor. The blackboard style was not selected because it was the only known communication mechanism, however it appeared to be the most appropriate choice amongst the limited number of known alternatives. Contributing to this lack of knowledge of other alternatives was the level of experience of the developers. They simply had not encountered enough large-scale system designs to develop an extensive knowledge base of architecture arrangements from which an appropriate solution could be selected. That does not excuse the team for its decision, however it is certainly the case that the level of knowledge of different architecture styles is now more widely understood than during the time frame of this project. Furthermore, while the general design issues have been known for some time, the level of knowledge of architecture issues across the entire software engineering community has improved as a result of case-studies of the types reported here. In addition, analysis of implemented architectures, taxonomies of popular styles, publications of design patterns, and workshops to collaboratively develop new architectures and styles have improved the level of knowledge of large-scale structure issue in software development (e.g., (Wolf 1997)). The level of knowledge of different architecture styles possessed by the software engineer is no longer based solely on direct personal experience.

A related factor was the lack of knowledge about the consequences of choosing a particular architecture. The concept of a blackboard communication mechanism appeared to be appropriate during the design phase of the project. However, it was not until the system was implemented that all project members finally accepted the impracticality of the structure to meet the needs of the project and execution environment. Issues such as message throughput and system stability could not be determined until it was implemented. That was a result of the failure of the developers to evaluate the abstract concept of the blackboard style with respect to the practical execution environment of HyperEdit / HyperCase rather than the generic system requirements.

Software engineering researchers and managers would like to think that developers make their decisions about system architecture completely objectively. However, in this case study at least, the design decisions of the developers were affected by experiential bias. The number of design alternatives was limited to those of the designers' previous

experience. In addition, the alternative that the designers were most familiar with received additional weighting in the ensuing comparison. Finally, the developers were also affected by their lack of knowledge concerning the way in which the actual implementation of those high-level concepts influence the non-functional attributes of design.

4.2.4.3 Influence of the Implementation Medium on Architecture Decisions

The case study has identified a number of areas where decisions about the architecture of the system were influenced by the implementation medium in which it was to be realised. Those influences were:

- The selection of a particular architecture because it was known how to implement it in the chosen operating system and programming languages.
- The development of a particular architecture because part of its components already existed in the chosen operating system and programming languages.
- The restriction of architecture alternatives to meet the particular operating and connection requirements of a previously selected software component.
- The restriction of architecture alternatives because of the execution model of the virtual machine that executed a chosen programming language.

One of the contributing factors in the selection of the blackboard style was the fact that it was known to be realisable in the chosen implementation medium. The developers had previous experience with the blackboard concept and had enough knowledge of the chosen implementation medium to immediately see how the concepts of the blackboard style could be realised using pipes, filters, text files, and appropriate server functionality. Moreover, prior knowledge of the remote procedure call support provided by the operating system made it easy to see how the inter-machine communication could be handled. Whilst the ability to implement a particular architecture does not make it the most appropriate, it was recognisably a contributing factor in the architecture selection process.

This case study also uncovered evidence of architecture alternatives being devised to utilise the known infrastructure or services of the implementation environment. That is, starting with knowledge of the smaller granularity building blocks or services provided by the operating system, an architecture was created specifically to utilise them. That was

evident in the decision to replace the blackboard system with a message-passing style. After the decision was made to replace the blackboard mechanism, a review by David Cleary of the high-level inter-process communication mechanisms of the chosen operating system (SystemV Unix) was performed. That revealed three alternatives: message queues, semaphores, and shared memory. The existence of the message queue infrastructure led to the investigation of a generic message passing approach, based on those queues, for solving the problem. The generic style was then tailored to the subscribe/dispatch approach to meet the specific functional requirements of the project (Baragry, Cleary et al. 1994). While the subsequent message passing implementation has satisfied the flexible communication requirements of the HyperCase project, a significant factor in its selection was not just its technical attributes but also the fact that the operating system provided mechanisms to directly support its implementation.

Interestingly, during the project's maintenance, work was performed on the message server to improve its performance and expand the available functionality. During that process the operating system supplied infrastructure of the architecture (message queues) were abandoned and replaced by retro-fitting the architecture on top of the socket level functionality of the operating system. The 'retro-fitting' required additional code to realise the concepts of the message passing style in terms of socket level communication, which is at a lower level of abstraction than message queues but which provide additional and more flexible functionality. However, in terms of the executing applications, the socket-based message passing continued to work in the same manner as the message-queue based architecture. Such developments are not uncommon. They are consistent with the basic consequences of information hiding and prototyping in which functions are implemented as quickly as possible and then replaced with an improved version at a later stage (Reed, 1994 in Reed 2000). The point is that the architecture was devised to utilise a set of design abstractions provided by the operating system and continued to work as devised when that underlying infrastructure was replaced.

The underlying implementation environment supported architecture creativity by providing direct support for some architecture styles, however in other instances it served to constrain the developers. At the beginning of the HyperEdit design, a GUI support package was required to provide the low and high-level graphic manipulation. At that time, the only alternative available within the financial constraints of an academic research group, which would operate in our Unix environment, was X/Motif. As

discussed previously, an X-based application begins by setting up an event loop and generating the initial user interface. As the GUI is manipulated, it generates events that, as they are processed, invoke the appropriate routines in the user-defined code. That choice of GUI support package forced the developers to use the event-loop based architecture of X/Motif for the implementation of HyperEdit. However, the original conceptual design specified a layered style architecture. The resulting system could be represented by both styles. The source code implemented by the developers was structured in a manner consistent with the layered approach. However, because the procedure invocation was dependent on the X event handler, the dynamic operation of that architecture was event-based. Structurally, the code represented a layered system but behaviourally it operated as an event-based system. In contrast, as the global HyperCase project evolved, additional applications were developed utilising Tcl/Tk as the GUI support package. That provided similar high-level GUI capabilities as X/Motif without constraining the developers to an event-loop architecture. Because HyperEdit required low-level graphic manipulation capabilities not provided by Tcl/Tk it was not possible to replace the X/Motif GUI environment.

The choice of programming language provided another example of architecture constraints imposed by the implementation environment. Although no specific example exists in the HyperEdit development, a number of other tools in the HyperCase environment had components of their systems implemented using Prolog. The conceptual design phase of ProTract, the project tracking tool (Cleary and Reed 1993), identified the need for some inferencing capabilities. As mentioned earlier, the project tracker would intercept messages produced by the development tools and attempt to infer how the status of the project was progressing with respect to the project plans. The rule-based language, Prolog, was best suited to implement that capability. However, Prolog is an interpreted language that evaluates the rules using a backward chaining inference engine. During the subsequent design of the project tracking tool, the developer was forced to design the system architecture and individual components within the constraints imposed by the backward chaining inference engine that forms the basis of Prolog's execution (Cleary 1997). In that particular situation, the implementation language was chosen based on the requirements and the architecture of that system was subsequently designed to operate within the conceptual limitations imposed by that language.

In this case study, a team of researchers were implementing a large and complex system. Some of the decisions clearly flowed from a lack of experience. However, even among more experienced designers, the nature of the development and run-time domains, i.e., the overall implementation environment; will often drive the choices made at higher levels of abstraction, including the highest level, the system architecture.

4.2.5 Discussion

Research literature strongly suggests that the system architecture should be set as early as possible in the design process and that doing so can provide significant benefits. This is especially true for product-line commercial software as well as the research based prototype project discussed in this case study. However, the HyperEdit study identified a number of issues that made the selection of the architecture at the beginning of the design process extremely difficult. The first concerns the factors that influenced the decision-making ability of the developers. They were: the designer's lack of knowledge of different architectures and their properties, the influence of the implementation medium on architecture decisions, and the ability to cope with changing requirements during the project lifecycle. The second issue concerns what the designers should be specifying in that initial architecture. There were a number of different architecture representations used during the development of HyperEdit. Which one(s) were the starting architecture or architectures? What elements should be included in the system architecture? Finally, where was the boundary between the HyperEdit architecture and the architecture of the global HyperCase environment? Those issues are discussed keeping mind the research based nature of the HyperCase project.

4.2.5.1 Deciding On the Initial Architecture

The degree to which requirements can change during the course of a project is obviously project dependent. While some projects can have their requirements firmly set before beginning design, that was not the case in the HyperEdit/HyperCase project. The goal of the project was to produce tools that test and evaluate concepts that support software development based on ongoing software engineering research. Therefore, the system requirements of the individual tools, including HyperEdit, were subject to change more than those of traditional software development projects. That played a significant part in the ability to determine a correct architecture at the beginning of the project. However, as the majority of software development involves maintenance, it is possible to argue that

the changing requirements experienced in this case study simply occurred a lot earlier in the application's lifetime compared with other projects. Indeed, some researchers argue that software development is a continual maintenance process (Gallagher 1997).

Traditional literature on system design has always stressed the need to design for malleability or modifiability, yet little research currently exists that compares the malleability of recognised architecture styles. Kazman (Kazman, Bass et al. 1994) has described a method of analysing system architectures to determine their ability to meet future modifications. However, the method requires knowledge of the future changes and can only be applied to mature domains in which a canonical functional partitioning has been performed. For instance, Kazman's paper used the example of user interface toolkits that have a well-codified set of concepts. A great deal of research is still required to achieve similar techniques in domains that do not have the same degree of well-defined functionality.

Doble (Doble 1997) has provided three patterns for change resilience in software architecture. He suggests that because of the business environment, developer knowledge, and performance/schedule constraints, it may not be possible to design to cater for all possible changes. His three patterns are:

1. The "Design for Now" Approach: Due to tight schedules, low developer experience and turbulent business environment, it may be best to worry about the current requirements and let the future pay for itself.
2. The "Laser-Guided Bombing" Approach: The development team might have enough experience and enough development time to develop a list of anticipated changes that could be catered for, based on the business needs. This approach hopes to cater for a subset of all possible changes.
3. The "Saturation Bombing" Approach: With enough developer expertise, suitable project constraints, and well-defined business domain, it may be possible to design to anticipate all imaginable changes.

Observations from the HyperEdit case study also identified the effects of the developer's lack of knowledge of known architectures and the ramifications of choosing a particular architecture on the decision making process. This situation frequently occurs in project situations where new domains and applications are being undertaken. The current state of understanding in software architecture and design patterns would certainly have improved

the development of HyperEdit. However, many issues still require attention. For example, one observation noted that while it was possible to reason about architectures at an abstract level, it was not until they were actually being implemented that some issues became apparent. It was not until the process of implementation that the developer obtained the level of detail necessary to effectively rationalise about the architecture concepts. Those issues have parallels in the area of general decision making theory.

Much of the research in software architecture is premised on a development process that matches what is termed the rational model of decision making.

> "The individual has objectives and a payoff function that permits the ranking of all possible alternative actions to those goals. The actor is presented with and understands the alternative courses of action. The actor chooses the alternative (and consequences) that contribute most to the ultimate goal. In a rigorous model of rational action the actor can accurately rank all alternatives and consequences and can perceive all alternatives and consequences … this assumption has been at the heart of consumer behaviour theories and microeconomics, political philosophy, and social theory." (Laudon and Laudon 1996)

However, Laudon and Laudon's summary of models of decision making (Laudon and Laudon 1996) details three criticisms of the rational model. First, the number of comparisons required to evaluate all alternatives is computationally impossible in a human time frame. Decision makers simply do not have time to compare all possible alternatives. Second, because of conflicting goals, it is not possible to rank all alternatives and consequences realistically. Third, in a real life situation it is impossible to specify a finite set of alternatives and consequences. Those criticisms are as applicable to architecture decision making as they are to general-purpose decision making. Moreover, each of those criticisms is compounded by the fact that software architecture is such a new field which deals with abstract concepts that have yet to be defined and have no direct, tangible, manifestation that can facilitate a common understanding.

Laudon and Laudon reviewed several alternative models of decision making that address the deficiencies with the rational model. Whilst those models do not provide any specific model for software architecture they can help explain how software architect's arrive at the decisions they do.

- Satisficing: Choosing the first available alternative in order to move closer to the ultimate goal instead of searching for all the alternatives and consequences.
- Bounded rationality: Idea that people will avoid new uncertain alternatives and stick with tried and true rules and procedures.
- Muddling through: Because of the existence of conflicting goals, this method involves successive limited comparisons where the test of a good decision is whether the majority of people agree with it.
- Psychological influences: The effect of underlying personality dispositions towards the treatment of information, selection of alternatives, and evaluation of consequences.

Those alternative models of decision making explain many of the observations made about issues that affected the architecture decision making performed during the HyperEdit lifecycle. However, those issues are not generally discussed in software architecture research.

The final issue that had an effect on the designers' ability to set the initial architecture was the influence of the implementation medium. Those issues were: the ease with which an architecture alternative could be implemented in the chosen implementation medium; the selection of an architecture to utilise known building blocks of the computing environment; and the constraints placed on possible architectures by the execution model of the implementation environment. Those observations contradict the traditional view of software development in which the high-level design is determined before any low-level decisions are made. The observations are, however, supported by design research in other disciplines that recognise the effects of lower level issues on the current level of design abstraction. Those disciplines include electronic and mechanical design theory, traditional architecture and conceptual theory building. For example, Alberts' thesis on design theory discusses general design issues in electronic and mechanical design:

> "In practice however, design will never be of a completely top-down nature. ... Knowledge about which functions can be realised given specific physical properties of the realisation material is propagated upwards. Without such knowledge about the feasibility of alternatives, design would result in a 'blind search'." (Alberts, Wognum et al. 1991; Alberts 1993)

Darnell quoted the keynote speech by Morton at the 1959 Electronic Components conference, which described how this principle had reached an extreme in electronic circuit design. In that domain, the ability of designers to develop useful solutions was restricted to those achievable using known components.[16]

> "With this viewpoint, the system designer has translated his overall system requirements to those of components, thinking only in terms of classical inductance, capacitance, resistance, tubes, and transistors. The component designer, adopting this viewpoint, therefore has been limited in his permissible solutions only to finding new techniques and materials for the classical elements." (Darnell 1962)

Similarly, Lawson described those effects in the discipline of traditional architecture and how an architect's education deliberately utilises that feature.

> "… there is no meaningful division to be found between analysis and synthesis but rather a simultaneous learning about the nature of the problem and the range of possible solutions … [And] architects need to know, for example, about the structural properties of wood but this does not mean they could become furniture designers." (Lawson 1980)

Finally, this issue is not just something which affects system design but is also well understood with regard to conceptual theory building. The ability of the human mind to develop systematic theories is influenced by the knowledge of the underlying concepts that already exist in human language. Our language is unavoidably permeated by concepts and theories. Common nouns are used to represent our concepts and not the things we perceive. Therefore, it would be impossible to assume conceptualisation begins with a 'clean slate' with which we can develop our conceptual objects and models. The objects in the world are already delineated to some extent by the classifications embodied in socially inherited language. In fact, learning a language essentially means learning to grasp objective thought concepts (attributed to Frege in (Popper 1979f))[17].

The waterfall model of design is no longer regarded as the best approach to software development, however research still suggests that the best way to approach development

---

[16] This was discussed in chapter 3.

[17] This is discussed in more detail in the next chapter.

is to perform the architecture level of design first. The system architecture is important and needs to be determined in order to set the pattern of design. However, little research exists which incorporates the observations found in this case study. In fact, the evidence suggests it may be exceedingly hard to determine the correct architecture at the beginning of the project and that indeed it may not always be necessary to do so (Reed 1987; Perrochon and Mann 1999; Reed 2000). In software architecture research, the recognition of those principles is starting to become more prevalent. For example, Cockburn (Cockburn 1996) details the affects of social issues on software architecture. Those social issues included the psychological bias of the developers, the knowledge of language, and software development skills. Furthermore, Cockburn suggests a number of design principles/patterns for dealing with those effects. However, software architecture research has a long way to go in determining how the architecture level of design should be performed in a manner that incorporates those observations that serve to contradict the rational model of decision making that underlies the dominant view of software engineering.

4.2.5.2 What Constitutes the Software Architecture?

The case study highlighted three factors that make it difficult to specify the large-scale structures of a software system. First, there were many architectures represented during the HyperEdit lifecycle. Which one, or ones, should be identified as the most appropriate? Second, what components should be specified in the system architectures? Third, what connections should be specified in the system architectures?

There are many definitions of the term software architecture (see (SEI 1997)), however using the simplest definition of the term, the high level structure of the software system, a number of different, yet valid, architectures were identified. They can be categorised into three broad groups:

- Conceptual Architectures: The representations used during the conceptual design phase of development that depict what the designer believes should be implemented. In the case study, conceptual architectures were used to convey the initial ideas of both HyperEdit (figure 4-3) and its global environment – HyperCase (figure 4-1).
- Static Implementation Architectures: The representations that depict the source code modules and the relationships between them. Examples in the case study are

the layered architecture of HyperEdit's initial implementation (figure 4-5) and the call-graph structure used during maintenance.

- Dynamic Operation Architectures: The architectures that depict how the system executes in terms of functional abstractions of the implemented system and execution abstractions of the computing environment (e.g., processes, distributed machines). The distributed communications architecture (figure 4-6) and the event-based operation architectures (figures 4-7 and 4-8) are examples.

Other researchers have identified multiple high level representations for software systems, for example (Kazman, Bass et al. 1994; Kruchten 1995; Soni, Nord et al. 1995). Those different representations may use different labels, however overall, they identify similar collections of structures. They all contain representations of the developers conceptual or logical view, representations of how it is implemented in source code, and representations of how it operates in a computing environment[18]. Furthermore, definitions of software architecture are now recognising the existence of multiple views of the system (e.g., (Bass, Clements et al. 1998)). However, there is still no general agreement about which views should be specified to represent the system architecture.

The understanding of those different views is based on analogies with traditional engineering development whose high-level system design may consist of different diagrams for different stakeholders in the development process. For example, a building design may have a different representation for the architect, interior decorator, landscape gardener and the electrician. Using those analogies as the basis for our understanding has been evident from the earliest software architecture research publications (e.g., (Perry and Wolfe 1992)) to the most recent (e.g., (Bass, Clements et al. 1998))[19]. However, the case study identified a number of observations that are not easily explainable using analogies with traditional engineering development. The first issue was the different types of component connections found in the HyperEdit architecture. Those different types, which are not found in other engineering disciplines, make it difficult to determine what components should be represented in the architecture and how the connections between the components should be represented. The second issue was the lack of a data model in the system architecture. None of the different architecture views suggested by research

[18] These issues are discussed in detail later in this chapter.

contain a representation of the system data model. The data model is extremely important to the system yet it is not considered part of the architecture. Interestingly, no other discipline produces systems that pass complex data items between components or build functionality that manipulates complex data types. Finally, the conceptual and implemented architectures of both HyperCase and HyperEdit were different to each other. Moreover, when maintenance was performed and system architectures were extracted from the implemented system, they were different to the original conception. It could be argued that was because of the research nature of the HyperCase project, the changing requirements, and the lack of developer knowledge. However, another case study (Bowman, Holt et al. 1999) has shown the difference between the conceptual and concrete architectures of the Linux operating system. Furthermore, it is not until the architecture was extracted from the implemented system that it could be compared with the conceptual or original design architecture. That is not the case with views of traditional building systems because the architecture can be viewed directly in the implemented system/artefact.

Despite the fact that there is much confusion about many aspects of software architecture, which can not be resolved using analogies with traditional engineering disciplines, those analogies continue to be used as the basis for our understanding. An investigation of those analogues is now presented.

4.3 Software Architecture Theory: An example of understanding based on the artefact engineering view

4.3.1 The Origins of Software Architecture Understanding

The first papers to specify the large-scale structures of software systems appeared in the late 1960s. In 1968 Dijkstra detailed the large-scale structure of the 'THE-Multiprogramming System' (Dijkstra 1968). His discussion discussed the advantages of partitioning the operating system into layers like 'onion-rings'. At the NATO conference in 1969, Sharp made the following lengthy comment that could be viewed as leading the way to contemporary software architecture research.

---

[19] This is discussed in the next section.

"I think that we have something in addition to software engineering: something that we have talked about in small ways but which should be brought out into the open and have attention focussed on it. This is the subject of software architecture. Architecture is different from engineering.

As an example of what I mean, take a look at OS/360. Parts of OS/360 are extremely well coded. Parts of OS, if you go into it in detail, have used all the techniques and all the ideas which we have agreed are good programming practice. The reason that OS is an amorphous lump of program is that it had no architect. Its design was delegated to a series of groups of engineers, each of whom had to invent their own architecture. And when these lumps were nailed together they did not produce a smooth and beautiful piece of software.

I believe that a lot of what we construe as being theory and practice is in fact architecture and engineering; you can have theoretical or practical architects: and you can have theoretical and practical engineers. I don't believe for instance that the majority of what Dijkstra does is theory – I believe that in time we will probably refer to the 'Dijkstra School of Architecture'.

What happens is that specifications of software are regarded as functional specifications. We only talk about what it is we want the program to do. It is my belief that anybody who is responsible for the implementation of a piece of software must specify more than this. He must specify the design, the form; and within that framework programmers or engineers must create something. No engineer or programmer, no programming tools, are going to help us, or help the software business, to make up for a lousy design.

Probably a lot of people have experience of seeing good software, an individual piece of software which is good. And if you examine why it is good, you will probably find that the designer, who may or may not have been the implementer as well, fully understood what he wanted to do and he created the shape. Some of the people who can create shape can't implement and the reverse is equally true. The trouble is that in industry, particularly in the large manufacturing empires, little or no regard is being paid to architecture." (NATO 1976b) (p. 150)

A few years later, Spooner developed his own "Software Architecture for the 1970s" (Spooner 1971), contrasting it with Dijkstra's large-scale system structure. As the 70s progressed, practitioners began detailing the advantages of theorising about those system-level structures and the consequences of decisions made at those higher levels of design. Parnas described how the effectiveness of modularization is dependent upon the criteria used in dividing the system into modules (Parnas 1972). In addition, Brooks wrote his essays on software engineering (Brooks 1975) in which chapter four, *Aristocracy, Democracy, and System Design*, stressed the importance of the conceptual design phase and how it affects subsequent development. Those examples show software developers were able to identify and reason about high-level structures of their software systems and recognised the importance of decisions made at that level of design. Moreover, it shows that the term 'architecture' was well established as the word for designating those structures.

Interestingly, that period of time saw a more distinct partitioning of software and hardware design as separate activities, which had until then been more closely entwined (Weinberg 2000). Indeed, Brooks, who was the originator of many software architecture ideas, also published articles on the architecture of computer hardware (Brooks 1962). Given this, and the extent to which Brooks draws on analogies with hardware development paradigms in *The Mythical Man-Month* (Brooks 1975), it could be argued that many of the concepts Brooks used for understanding the large-scale partitioning of software systems would have evolved from his understanding of the concepts involved in computer architecture[20].

Despite those, and many other examples of software developers reasoning about the large-scale structures of their systems, it was Mary Shaw's 1989 paper, *Larger Scale Systems Require Higher Level Abstractions* (Shaw 1989) that led to the separate area of research that is today referred to as software architecture. In that paper, Shaw recognised the existence of high-level system representations that are used during the development process and which could be recorded and passed onto other designers. Shaw had been working on abstraction techniques previously (Shaw 1984) and noted the use of those abstractions in the development process could result in a "software architecture level of

[20] The influence of experience on the ability to understanding new phenomena is discussed in the next chapter.

design." Shaw's work identified and labelled a number of different styles of architecture that are still used as examples today. For example, 'layered' and 'pipe & filter'. While Shaw's paper discussed the importance of higher-level system abstractions, it merely identified concepts which others began to theorise about.

Perry and Wolf's paper (Perry and Wolfe 1992), as its title suggests, laid the foundations for many architecture research ideas. It also contained the first attempt to define architecture, or at least, the concepts of software architecture. They stated that a model of architecture consists of three components: elements, form, and rationale. The elements are either processing, data, or connecting elements; form is defined in terms of properties and relationships among the elements (the constraints); and rationale provides the underlying basis for the architecture in terms of system constraints. Much of the understanding in that paper was derived through analogies with other disciplines that highlighted similarities and differences. For example, computer hardware, network architecture, and traditional building architecture. One of those analogies compared the different representations of software system the multiple views of a traditional building design that are used by the various stakeholders in the development process. That specific analogy is discussed in detail in a later section.

From those research foundations, many definitions of software architecture have emerged. Of the early definitions, the one by Garlan and Shaw was the most often cited:

> "Beyond the algorithms and data structures of the computation; designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives." (Garlan and Shaw 1993)

Garlan and Perry, in an introduction to a special issue on software architecture in the IEEE Transaction on Software Engineering, provided a simpler, all-encompassing definition:

> "The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time." (Garlan and Perry 1995)

However, neither of these, nor any other definition, has become an accepted standard. The Software Engineering Institute web site houses many of the definitions that have been published in software architecture literature (SEI 1997). The most recent definitions differ from the originals by catering for issues that emerged out of published experience reports – the existence of multiple views of software architecture.

The HyperEdit case study presented a number of architecture types that were uncovered during that system's development. They are now compared with other work on software architecture views. A number of software architecture case studies and theories based on practical experience have been published suggesting the need for multiple large-scale representations to capture the architecture of a software system. Soni (Soni, Nord et al. 1995), as a result of surveying many software systems used in industrial applications, identified four different large-scale structural depictions used throughout the development process:

- Conceptual architecture: describes the system in terms of the major design elements and the relationships among them.
- Module interconnection architecture: encompasses functional decomposition and system layers.
- Execution architecture: describes the dynamic structure of the system.
- Code architecture: describes how the source code, binaries, and libraries are organised in the development environment.

Kazman (Kazman, Bass et al. 1994), while discussing the analysis of quality attributes of system architecture, asserted that the architecture can be described from (at least) three different perspectives:

- Functional: Partitions the overall behaviour into a collection of functions that are individually simple enough to conceptualise.
- Structural: The collections of components that represent the computational entities and the connections and control relationships between them.
- Allocation: Depicts how the domain functionality is realised in the software structure.

Finally, Kruchten presented his collection of system representations that had been successfully used to capture the architecture information in several large projects (Kruchten 1995):

- Logical view: Where the required system is decomposed into a set of key abstractions, taken (mostly) from the problem domain.
- Process view: Depicts how the main, functional abstractions map onto executing processes and threads of control.
- Physical view: Reflects distributed aspects by showing how the software maps onto the hardware.
- Development view: Focuses on the actual software module organisation in the development environment.

Those four views are depicted with a fifth view that illustrates them with a few use-cases or scenarios.

From those experience reports, the use of multiple views to represent the system architecture has become accepted in the discipline and has become part of the most recent definition of software architecture by Bass et al:

> "The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the external visible properties of those components, and the relationships among them." (Bass, Clements et al. 1998)

By externally visible properties the authors of the definition mean the "assumptions components can make of a component, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on." (Bass, Clements et al. 1998). The intent of the definition is that "a software architecture must abstract away some information from the system … and yet provide enough information to be a basis for analysis, decision making, and hence risk reduction." (Bass, Clements et al. 1998). The authors also note that "the definition makes clear that systems can comprise more than one structure, and that no one structure holds the irrefutable claim to being the system architecture." (Bass, Clements et al. 1998)

The prevailing consensus in software architecture research is that those representations are different views of the system architecture, where each view provides a different

abstraction of the underlying implementation detail. Therefore, each view is a subset of the detail that exists in the implementation. That way of understanding the nature of software architecture views can be traced back to the foundations paper of Perry and Wolf (Perry and Wolfe 1992). From their analogies with traditional building architecture they noted:

> "… a building architect works with the customer by means of a number of different views in which some particular aspect of the building is emphasized. ... For the builder, the architect provides the ... floor plans plus additional structural views that provide an immense amount of detail about various explicit design considerations such as electrical wiring, plumbing, heating, and air-conditioning. ... Analogously, the software architect needs a number of different views of the software architecture for the various uses and users. At present, we make do with only one view: the implementation." (Perry and Wolfe 1992)

The same analogy was used by Bass et al to explain their definition of architecture. They claim the multiple representations are analogous to the different building representations used by the architect, the interior decorator, the landscaper, and the electrician. They summarise the most useful representations or views used by software developers as: module structure, conceptual or logical structure, process structure or co-ordination structure, physical structure, uses structure, calls structure, data flow, control flow, and class structure. (Bass, Clements et al. 1998)

The IEEE draft recommended practice for architectural description (IEEE and Committee 1998) also reflects this understanding of large-scale software structures. The standard recognises the growing importance of software architecture in system development, however it is more cautious about how it refers to architecture and architecture views. The term architecture is defined very generally in the standard as "the highest-level conception of a system in its environment". Similarly, the concept of an architecture view is defined as "a representation of a whole system from the perspective of a related set of concerns".

Those definitions recognise the existence of the concepts but deliberately take care not to relate them to any preconceived structural meaning. The result however, is that the definitions are so general they could apply to many different things. Finally, the standard,

while recognising the existence of many different system views and viewpoints does not specify which ones should be used in a project.

Despite those definitions, confusion still exists concerning the exact nature of the representations, why they are necessary, and which ones should or should not be included in the description of the system architecture. Other researchers have offered explanations for this.

Clements, in his overview of the field (Clements 1996), suggested five reasons why the community has failed to reach a consensus on what exactly we mean by software architecture.

1. Advocates bring their own methodological biases with them. While most definitions of the term agree at the core, they differ seriously at the fringes. Those differences are attributable to the motivation each researcher has for examining the structural issues in the first place.
2. The study is following practice, not leading it. Research still involves observing the design principles and actions used whilst developing real systems and abstracting the commonalties.
3. The field is still quite new.
4. The foundations have been imprecise. The field contains a remarkable number of undefined and ambiguous terms. In addition to the textual terms, diagrammatic representations of architectural structures also suffer from ambiguity in interpretation.
5. The term is over-utilised and its meaning as it relates to software engineering is becoming diluted.

That confusion concerning the meaning of software architecture was observed by Bass et al (Bass, Clements et al. 1998) who noted that definitions of system structures include the following:

- Architecture is high level design.
- Architecture is the overall structure of the system.
- Architecture is the structure of the components of a program or system, their interrelationships, and principles and guidelines governing their design and evolution over time.

- Architecture is components and connectors; architecture is components, connectors, and constraints.

They continued by suggesting the lack of a well-accepted definition is not as troubling as it appears because the concept of software architecture can still be successfully used while a discipline-wide consensus evolves. Their argument uses the notion of an 'object' as a similar example. The exact definition of an 'object' is still debated by object-oriented programming researchers and practitioners, yet the apparently ill defined concept has resulted in a full-fledged paradigm shift in software development. (Bass, Clements et al. 1998)

To summarise the current understanding of software architecture:

- Software developers have been able to identify and theorise about the large-scale structures of software systems since early in the discipline.
- Those large-scale structures are considered the 'architecture' of the software system. That understanding is based on analogies with traditional engineering disciplines whose built systems exhibit large-scale structures that are termed the 'architecture'.
- Research has successfully sought to improve the development process at the software architecture level of design.
- Experience suggests many system representations are required to depict the architecture of a software system.
- Those representations are considered analogous to the multiple representations of traditionally built artefacts.
- Confusion still exists about the exact nature of software architecture and the views used to represent it.

In analysing the current understanding of software architecture, a brief summary of the traditional notion of architecture is presented to see if it can clarify the confusion.

4.3.2 Traditional Notions of Architecture

The architecture of a built thing, in general parlance, refers to its "unifying or coherent form or structure" (Miriam-Webster Dictionary 1997). That generic concept is easy to understand when dealing with our vast range of physical artefacts. People without

specific training in the respective fields can perceive building architecture, computer architecture, naval architecture, etc. However, difficulties arise when you apply the same concept to elicit the architecture of a system whose only tangible manifestation of the construction is the source code implementation (Bennett 1997). Despite this, the generic term 'architecture' appears to be appropriate when referring to the large-scale structure or form of software systems. Therefore, because the field has yet to agree on precisely how the term should apply to software systems it is worth looking at its historical usage in an attempt to gain some insight.

Interestingly enough, many reference books in the field of architecture itself fail to define the term (for example (Pevsner, Fleming et al. 1975; Standen 1981)). Moreover, those that do, describe something quite ethereal that fails to assist in the application of the term to software. For example:

> "The art of designing and building according to rules and proportions regulated by nature and taste, so that the resultant edifices arouse a response by virtue of their qualities of beauty, geometry, emotional power, picturesque, intellectual content, or sublime essence, is called *Architecture*, a term which suggests something far more significant, sophisticated, and intellectually complex than a mere building, although it must also involve sound construction, convenient planning, and durable materials. ... Architecture implies a sense of order, an organisation, a geometry, and an aesthetic experience of a far higher degree than that in a mere building." (Curl 1993)

An often quoted definition by Sir Henry Wotton from his 1624 book, *The Elements of Architecture*, states that it must fulfil three conditions: 'Commodite', 'Firmness', and 'Delight'.

> "To constitute architecture, a building must not only be conveniently planned for its purpose ('commodity'), and be soundly built of good materials ('firmness'), but must also give pleasure to the eye of the discriminating beholder ('delight'). It is this third quality, added to the other two essentials, that differentiates 'architecture' from mere 'building." (Briggs 1959).

A chronological comparison of the different definitions of traditional architecture reveals how the term has become more encompassing over time. Briggs, commenting on Sir Henry Wotton's definition, detailed how primitive buildings such as huts and even the

Egyptian pyramids could be counted as architecture. Moreover, "in modern times, bridges and other structures which are now commonly regarded as 'civil engineering' rather than architecture or even building." (Briggs 1959). In spite of this historical meaning of the term, current usage and definitions of 'architecture' certainly do include those items.

While the discipline of architecture itself has proceeded without a formal definition of the term – at least not in the sense that we seek, there is a long history of architects formally discussing the nature of their discipline. It is generally accepted that Vitruvius' treatise, *Ten Books On Architecture* (Vitruvius 1931) in the first century BC, was not the first systematic account of architecture, however his works are the most ancient which have survived to this day. In addition, there exists a vast number of books that detail specific architectures, the history of the discipline, and theories explaining particular aspects of the discipline (e.g., (Watson 1990; Kruft 1994; Gelernter 1995)).

> "Another version of this theory looks not to whole forms, but to general principles of form which are even more abstract and universally applicable than types. For many centuries these principles were thought to be embodied in the five Orders of architecture (Tuscan, Doric, Ionic, Corinthian, Composite), each of which set out specific rules for the proportions of columns and the spaces between them, the proportions of entablatures relative to the columns and the spaces between them, embellishment and ornament of the complete ensemble." (Gelernter 1995)

Not all architects agree on the most appropriate solution for a particular problem's requirements or even on the best architectural design theory. However, the discipline does have a common understanding of what it means to be an architect and what the goal of architectural design is:

> "That is what architects are, conceivers of buildings. What they do is to design, that is, supply concrete images for a new structure so that it can be put up. The primary task for the architect, then as now, is to communicate what proposed buildings should be and look like." (Kostof 1986)

That common understanding has coalesced over a long period of time through the publication of architectural theories and education of architects in apprenticeships, guilds, schools, and universities. The exact definition of what architecture is may vary, however

there exists a common understanding of what the architect does – the architect designs representations of physical structures so they can be built.

4.3.3 Issues That Undermine the Existing Understanding of Software Architecture

The logical progression from the recognition of large-scale structures in software systems; to Shaw's call for an architecture level of design; through to Perry and Wolf's foundations for the discipline; and finally to the explanation of the multiple, high-level representations required to depict a software system as different views of the implementation detail appears valid. However, a more thorough comparison of the systems built by the respective disciplines shows it is quite specious. It is based on the implicit assumption that the software development process is analogous to those 'construction' disciplines in which the completed artefacts or systems exhibit a unique representational abstraction, fixed during the early stages of design, which we describe as 'the architecture'. The problem of obtaining an acceptable definition of software architecture or a set of common architecture views is due to the assumption that software systems have an analogous, unique design abstraction, determinable at the early stages of the design. That understanding of architecture and the use of architecture views follows from Perry and Wolf's statement,

> "… there are a number of interesting architectural points in building architecture that are suggestive for software architecture."

However it ignores the statement that began that sentence,

> "While the subject matter of the two is quite different..." (Perry and Wolfe 1992).

The subject matter of the two is quite different and any attempt to use analogies between the disciplines can only be done by ensuring that conjectures extrapolated from those analogies are not invalidated by those differences.

A comparison of the disciplines shows that two important differences exist between the artefacts that software developers produce and those produced by the more established engineering disciplines. The first is the concept of form and the other is the concept of system execution. Those differences between the fundamental natures of the respective systems have a significant impact on the way we use the notions of architecture and architecture views in the development process.

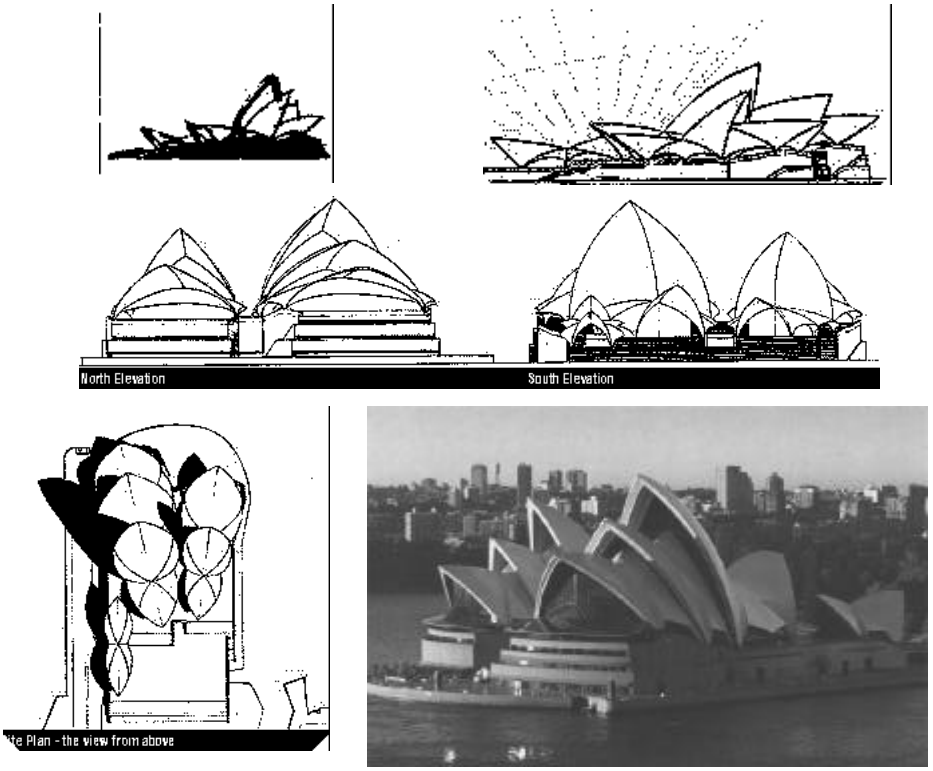


Figure 4-9: Architecture Diagrams and Physical Representation of the Sydney Opera House

Systems produced by traditional engineering disciplines are corporeal. They have a physical form, a tangibility that allows the viewer to perceive its large-scale structure – its architecture. That architecture can be viewed in the original design documents, traced throughout the design process and viewed in the physical realisation of the system. Australia's most famous piece of architecture, the Sydney Opera House, provides a good example. Figure 4-9 depicts the large-scale system design developed by the architect and a picture of its physical appearance (Sydney Opera House 1999). You can see the architecture in the design and in the realisation.

The analogous concept of form does not exist for software systems. Figure 4-10 depicts one of the software architectures of the HyperEdit system. It also depicts the only tangible

aspect of that system, its source code. You cannot see the architecture of a software system by looking at the thousands of lines of source code. It simply does not exist in the same fashion. The difference is so obvious it can easily be missed. Others have claimed the user interface can be thought of as a tangible aspect of a software system. However, that does not invalidate the claim that there is a fundamental difference between the forms of the systems produced by the respective disciplines.

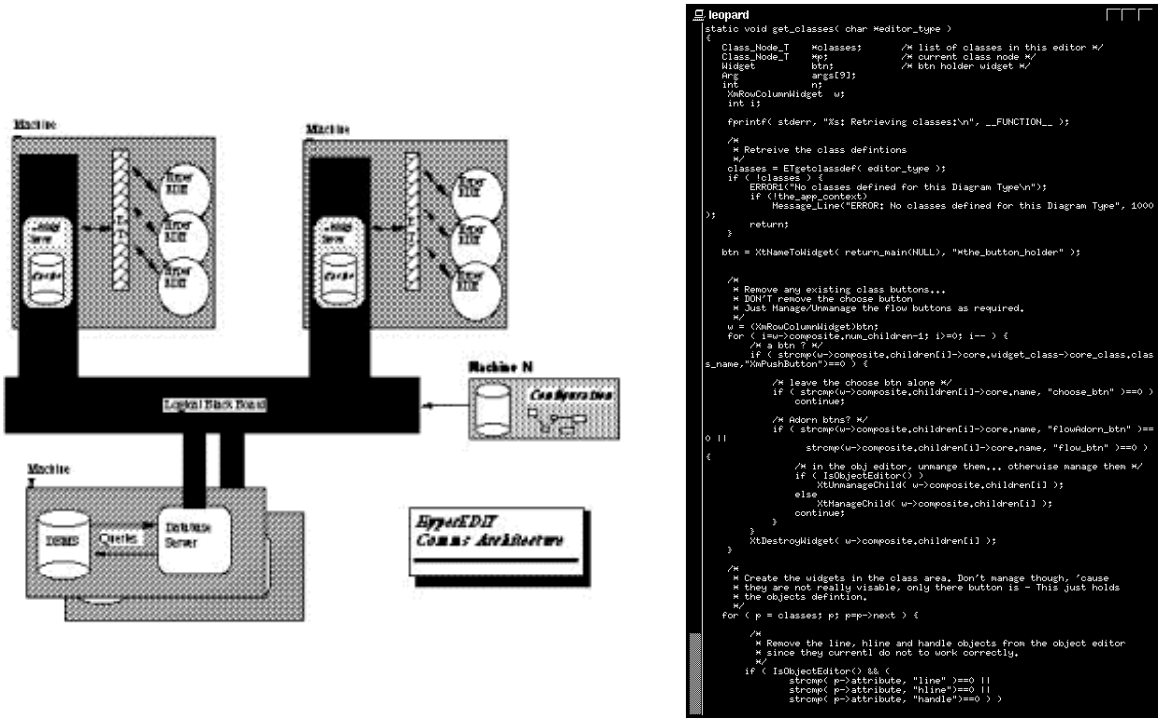


Figure 4-10: Architecture Diagram and Physical Implementation of HyperEdit System

The difference between the concept of system form in the respective disciplines affects how the notions of architecture and architecture views are used in the development processes. The architecture of a physical artefact describes its physical form. To repeat the previously stated quote:

> "That is what architects are, conceivers of buildings. What they do is to design, that is, supply concrete images for a new structure so that it can be put up. The primary task for the architect, then as now, is to communicate what proposed buildings should be and look like." (Kostof 1986).

Architects represent the geometric properties of the building materials and/or components. The physical magnitudes and relations of those components and how they are juxtaposed in space. That is the case in traditional architecture, civil engineering, and

mechanical engineering. Those architectures depict the physical form of the system or the components that comprise the system. System functionality is inferred from those components[21].

Traditional building disciplines produce many different representations of their system architecture. Those views are constructed by removing some of the implementation detail and leaving a subset of the devised form. They are understood in the context of the global structure using the understanding of the physical form or features of the entire system. For example, how the wiring moves throughout the spatial arrangement of the automotive vehicle, or how the plumbing system is laid out within the spatial arrangement of the building. Those high-level representations can be developed both before the system is realised or as documentation after the system is completed. They depict a view of what the physical system is or will be. Not how the system will operate, but how the system will exist as a corporeal artefact.

Software systems have no analogous physical form. They are not tangible systems. The high-level, abstract, design representations must be different to those produced by the peer level of design in other engineering disciplines. Empirical research has shown that software developers produce multiple, high-level abstractions to represent their systems and the evolution of research ideas has assumed that they can be devised and used in an analogous manner to those architecture views of other disciplines. It may indeed be possible, however the current understanding of software architecture views is based on an assumption that has never been validated. During software development, large-scale design representations are created in the conceptual design phase, the implementation stage, the maintenance stage, and all other stages in between. Do they have any relation to each other? Is it possible to derive them all from the source code? Are they immutable in the same sense as traditionally built architectures? Software engineering researchers answer "Of course!" to these questions and use further analogies with other engineering disciplines as justification. Those justifications however, fail to consider the differences between the disciplines and the lack of tangibility of software is one difference that makes the use of those analogies hard to justify. To determine whether those multiple representations of software architecture are views in an analogous sense to other disciplines the following question needs to be answered. What is it about the nature of our

[21] Electronic engineering generally does not have that property and is discussed separately.

discipline, rather than other disciplines, which makes it so? That question will be addressed in the next chapter.

The other important difference between software systems and traditionally engineered artefacts concerns the concept of system execution. Software has a distinction between the implemented system, the collection of source code, and the executing system, that is, the way the source code is executed by the implementation environment to realise the required system. That distinction does not exist in any other discipline. A software system is nothing more than a collection of source code statements until it is compiled and executed, statement by statement, by the 'virtual machine' implied by the semantics of the programming language. It is not until that stage that the system realises the desired result. A fact that is taught to all computer science students and perhaps forgotten not long after.

Some researchers contest the uniqueness of the distinction between system implementation and system execution. Counter arguments make analogies with other disciplines such as, "What about the flow of movement through a building?" or "What about the execution of a motor vehicle or electronic device?" To refute those claims, a distinction is made between the operation and the execution of a system. This distinction is critical to realising the differences between software systems and traditionally built artefacts and, therefore, warrants a few examples. Users can operate a software system through its user interface but that operation cannot occur until the system is being realised through its execution by the computer. The HyperEdit case study showed that the structure of the static, source code implementation of a software system represents something completely different to its dynamic operation. They are not different abstractions of the complex, underlying detail.

Motor vehicles and electronic devices certainly operate but they are not executed in the same manner. The construction of a motor vehicle results in the existence of a constant mechanical linkage between the physical components. As the driver is operating the vehicle, the gross structure of its dynamic operation is exactly the same as the gross structure that was the result of its construction. Similarly, computer architecture remains the same whether the machine is being used or not. A user can operate mechanical and electronic devises but they have no need of an external system to provide its execution. They may require power through electricity or combustible fuel for the components of the system to operate and exhibit the required properties. However, once supplied that power

they continue to execute independently and have no need of concepts such as a 'thread of control'.

4.3.4 Examining the Fundamental Nature of Software Systems to Understand the Representations Used to Depict Them

A large amount of invaluable research and empirical study has been undertaken in the field of software architecture. However, this thesis has identified two important differences between software development and traditional engineering disciplines that serve to undermine the understanding we have of software architecture and architecture views. Software systems have no large-scale, visible, gross structural form that is analogous to traditionally engineered systems. In addition, software systems have a distinction between the physical source code implementation and how that code is executed by the computer to realise the required system. Those differences invalidate the assumption that software architectures are analogous to those used by traditional engineering disciplines and that software architecture views are different abstractions of the underlying implementation detail of software systems. However, by using traditional engineering architectures as a contrast, those differences provide an insight into the fundamental nature of software systems. That fundamental nature and the high-level abstractions used to represent it explain why those representations must necessarily be multiple, independent architectures.

The architectures used to represent the only 'tangible' part of the system that exists, the source code implementation, are fundamentally different to those used to represent the executing system. Representations of the source code implementation depict how the system is implemented using the building blocks provided by the implementation language(s). Those building blocks include files, procedures, functions, rules, object definitions, etc. That is the only system representation that can be directly perceived by us, yet it does not contain all the implementation detail necessary to understand what the system does or how the system executes to realise the requirements[22]. It is missing services provided by the operating system; services provided by other software systems, both those provided at compile time by linking in additional libraries and those provided at run-time by communicating processes; and it is missing information that affects the

[22] Again, some may argue that the user interface constitutes a tangible aspect of the system. That debate is not considered here because it does not alter the subsequent conjectures.

operation of the system because it is hidden in data values rather than being explicit in procedural invocation. The source code is the lowest level of system granularity, the detail from which larger-scale abstractions are generated. However, it is missing the detail necessary for the system to execute. That additional detail is available only at run-time after the source code has been compiled and is being executed. The missing information is depicted in the abstract concepts evident in the architecture representations of the dynamic execution of the system. Those representations detail the operating system processes, the inter-process communication abstractions, and the other services that become part of the system at compile or runtime. The representations we have to depict the static implementation of the system and those which represent the dynamic execution of that system are different. One is not merely a subset or more abstract 'view' of the other. They are different, and the reason they are different is because of the differences that exist between the discipline of software development and those from which we draw the concepts of architecture and architecture views. Our systems have no tangible form and our systems have a distinction between system implementation and system execution.

The difference between system implementation and system execution also highlights the fact that no software system representation, from lowest level of detail, through to most abstract architecture contains the information that explains how the system is executed. It is not immediately obvious because few, if any, other disciplines require it in their system representations. In other disciplines you look at the architecture of a system and infer how it works. That is because those systems are not executed by another machine. Software systems are executed and knowledge of the operation of that execution engine, the virtual machine implied by the language, is necessary to understand how the system is executed.

The majority of systems are implemented in procedural or object-oriented languages and developers can conceptualise the operation of those by implicitly following the procedural invocations as the thread of control moves through the system components. Object-oriented terms like 'message passing' are still, at the code level, procedure invocations. Designers viewing system representations automatically apply that knowledge of how that model of abstraction operates to solve a problem, often without explicitly realising it. It becomes evident however, when attempting to understand a system representation that has been implemented in a language that utilises its own virtual machine rather than traditional procedural invocation. For example, understanding how a system implemented in Prolog operates must be done with the knowledge of how a

backward-chaining inference engine works. In addition, systems implemented in functional languages such as Scheme or Lisp must be evaluated with respect to how the computer executes that language. The dynamic execution architectures of a realised system are not generated by abstracting away detail from the large and complex implementation because those details do not exist in the implementation. Again, we have an architecture representation that is not a subset or abstraction of some other, more complex, representation. It is different to the implementation because of the fundamental nature of software systems.

The other noted difference between the respective disciplines, the concept of system form, also leads to the necessity of multiple independent architectures to represent a software system. Shaw's original article on software architecture noted the existence of higher level abstractions for software systems that could lead to an architecture level of system design. Traditional building disciplines develop the architecture, the gross structural form of the system, during the initial design stages of the development process. It can then be tracked and modified through the design process, and subsequently viewed in the realised system. This cannot be achieved in software design. The current understanding of software architecture views is that they are different abstractions of the complex, underlying implementation detail. The existence of large-scale software representations developed during system design is often noted, but research does not explain their relationship with the representations that are generated after the system has been implemented. That is due to the nature of the elements that are contained in those representations. They are not representations of corporeal components in an analogous manner to traditional system architecture.

The concepts represented in the design level depictions of software architecture contain abstract domain level concepts. They are mentally conceived entities that have no tangible manifestation. They may attempt to model or mimic tangible things, but they themselves have no form. The realisation process of a software system as an executing computer program occurs by implementing those mentally conceived, domain level concepts using the constructs provided by the programming language and operating system, and subsequently executing them in a machine. Those mentally conceived notions may be similar to implementation level concepts, however they do not have to be. Indeed the essence of software development is the process of implementing those domain level concepts of our minds using the constructs provided by whatever implementation

environment is at our disposal[23]. This is not generally the case in any other engineering discipline.

Progress in software design research is concerned with reducing the cognitive distance between the concepts that exist in our minds and those that are realisable in the implementation medium of our discipline. Programming language improvements, such as object-oriented languages and FGLs, attempt to bring the implementation level closer to the mentally conceived components. Alternatively, design methods and patterns attempt to provide techniques that help to develop mental level components, and their interactions, that are more easily, and predictably, realisable in our implementation medium(s).

High-level software design representations consist of abstract concepts that depict domain level functionality and/or behaviour. In contrast, large-scale representations of the implementation can consist of abstract concepts provided by the implementation medium. For instance, language constructs (e.g., functions, rules), virtual machines, inferencing engines, files, operating system processes, etc. They are different collections of concepts.

A conceptual architecture can be realised by many implementation architectures and an implementation architecture can be represented by many conceptual architectures. The difference between the two can be explained through a better understanding of a word that is often used in software architecture research – 'abstraction'. The existence of different architectures for a software system has been explained as different abstractions of the complex implementation detail. The definition of the word abstraction is often quoted from Shaw's work as a simplified description of a system that emphasises some of the system's details or properties while suppressing others (Shaw 1984). That definition matches the one in a standard English dictionary. It also matches how views are assumed to be generated in traditional built architecture, where each view is a subset of the system as a whole. However, that is not the situation with software architectures. The design and implementation architectures contain different collections of concepts. They are not different subsets of the underlying system. They match the definition of abstraction discussed in philosophy and psychological – for example (Corsini 1984). In those fields, abstraction is the technique by which higher order concepts are used to further intellectual reasoning by representing distinct, yet similar, particular instances. For example, apples

[23] This is discussed in more detail in the next chapter.

and bananas can be represented by a single concept, fruit. That is how abstraction is used in software architecture. The collection of particular implementation concepts, such as objects, message queues, etc are represented by a different concept such as a blackboard. A blackboard does not exist in the software system. What 'exists' is a collection of programming objects or procedures, in conjunction with operating system message queues. We simply choose to refer to that collection by the single concept 'blackboard'. Similarly, there is no particular instance of 'fruit'. There are apples, bananas, oranges, etc. We simply choose to refer to them collectively as 'fruit'[24].

Software architecture views are not developed by merely removing the unwanted detail. They involve the generation of higher level, abstract concepts to represent the underlying detail. Moreover, many higher level concepts can be used to represent the same particular instances. That is why many architectures can be used to describe the high level structure of a software system.

A counter argument is often made that engineering disciplines, such as electronic and chemical, design system architectures that do not represent the physical form of their components. Their high-level designs also represent components of functionality or behaviour rather than the physical form of the system. That would appear to be similar to design-level software architectures. However, differences exist which serve to invalidate attempts to use direct analogies from those disciplines. The functionality depicted in high-level electronic engineering architectures exists in terms of componentised aggregations of a small set of physical properties on which the discipline is based. They are: current, voltage, capacitance, inductance, and resistance – this is even simplified to binary operations in digital logic design[25]. The concepts realised by the functional components of those disciplines may be large and complex but they are all aggregations of those basic elements. Moreover, the functionality is ultimately constrained by those properties (Darnell 1962). The nature of the components used in those representations are fundamentally different to the nature of the functional components depicted in high-level software design representations because functionality in software is not constrained in any analogous way.

---

[24] Again, these issues are detailed in the next chapter.

[25] This was discussed in the previous two chapters.

It is because of the differences between software development and traditional engineering disciplines that software architectures developed at the beginning of the design process are not views of the complex implementation detail in an analogous manner to traditional engineering disciplines. They consist of completely different collections of concepts. It is true that some representations, for example high-level object diagrams, have a smaller cognitive distance between the design level concepts and the implementation level concepts. However, that is not true of all high-level software architectures developed early in the design process. Because of the nature of software development they cannot be. Those architectures, developed during the early stages of the design process rather than derived from the implementation, are created at the 'architecture level of design'. They are different from the architectures developed during the same stage of other disciplines and are not different views of the implementation complexity.

4.3.5 Discussion

The current understanding of software architecture views as different abstractions of a complex, underlying, implementation is based on an extrapolation from the assumption that software development is analogous to traditional engineering disciplines. However, that notion of architecture views fails to consider differences between the respective disciplines: the concept of visible system form and system realisation through execution. Those differences provide a contrast that gives an insight into the fundamental nature of software systems and the processes required to develop them. Moreover, it is that fundamental nature which necessitates the existence of multiple, independent representations. While the 'views' suggested by other researchers are similar to those representations suggested by this theory of software systems, they are based on specious analogies with other disciplines and lack the understanding required to explain why they are necessary[26].

Software architecture research is undoubtedly producing results that are benefiting the development community (e.g., (Bass, Clements et al. 1998; Bass and Kazman 1999)). However, the lack of a complete understanding of software architecture is causing confusion regarding what is considered the architecture of a software system and how those representations can be used to improve the development process. A better

understanding of the fundamental nature of software systems and their development is a necessity. Answers are needed for the questions that are often posed in commentary-style journal articles (e.g., (Gilb 1996)) and in informal conference discussions and keynote addresses (e.g., (Reed 1987; Xia 1998)). "What do we build and how do we build them?" "What does software engineering really mean?" These are not easy questions to answer. They will not present quantitative results that are easily testable or easily publishable. What is required is work on the philosophical foundation of the discipline. Without a good understanding of the nature of our own discipline we will continue to grasp at analogies and attempt fit the square-pegs of other disciplines into the round-holes of our own problems. The foundation of that understanding is presented in the next chapter.

[26] A paper summarising this argument was submitted for publication in 2000 (prior to submission of this Thesis) and subsequently appeared in WICSA 2001.

5. Uncovering a Foundation for Software Engineering

5.1 Introduction

The artefact engineering view of software development, which pervades software engineering research, is based on perceived analogies with traditional engineering disciplines. However, a detailed study of the systems produced by the respective disciplines, the approaches they use to develop them, and an analysis of the analogies used, shows software engineering research needs to develop a better understanding of the underlying principles of both disciplines in order to determine the applicability of that view.

Traditional engineers build artefacts. In contrast, software engineers build models of reality. Nevertheless, software engineers would like to build systems using techniques similar to those used by other engineers.

The design approaches used by those other engineers have evolved, in part, due to the nature of the systems they build and the components and materials used to build them. Therefore, for the design process of software engineering to be thought of as analogous, the nature of what the respective disciplines build must also be analogous. That is, it must be possible to understand the nature of software components and systems in a similar way to the nature of engineering components and systems. For that to be possible, the fundamental nature of models of reality must be analogous to the fundamental nature of traditionally built artefacts. To determine the validity of that assertion it is necessary to turn to other disciplines that have examined the nature of models of reality. Those disciplines are the fields of philosophy, specifically in the areas of metaphysics and epistemology, and the discipline of psychology, which has developed theories to explain concept development, utilisation, and evolution. Examining the theories developed by those disciplines uncovers a foundation for the understanding of software systems and the processes used to produce them. The conclusion is that software systems cannot be understood in a similar manner to traditionally engineered systems. In order to develop an engineering discipline of software development, far more attention must be paid to the fundamental nature of the systems that are built.

The chapter begins by briefly explaining the process of software development in terms of the models it builds. In the research literature, there are many references to the fact that software developers build conceptual models. This chapter examines the use of models in the development process and identifies the developer's formation of conceptual constructs as one of the major sources of difficulty both for software engineering research and for software development in general. Existing research clearly acknowledges the existence of conceptual modelling issues in software development. The problem appears to be that very few people have fully considered the impact of the relevant philosophical and psychological issues on the ability to 'engineer' those models as software systems.

A description is then provided of the assumptions that must hold if the development of the conceptual construct can be performed using an 'engineering' approach. The relevant issues from the fields of philosophy and psychology are then detailed and discussed to determine the validity of those assumptions. It should be noted however, that it would be impossible to explain all of the relevant issues from those disciplines in a single chapter. Such an undertaking would require an entire book in itself. The major theories from those disciplines are presented to argue that 'software engineering' is an attempt to 'engineer' conceptual processes. The software development process is then described a second time noting the influence of the relevant issues that become apparent from the more detailed understanding of the human conceptual apparatus.

The goal of the chapter is not to provide an explanation of how those foundational issues specify how software engineering should be performed. It is simply to show that the issues have significant ramifications for how we currently think about the development process. Moreover, it may be that by viewing software development in that new light, current problems in software engineering research may be better understood.

5.2 The Conceptual Construct

The process of software development can be partitioned into the following phases[27]:

- Requirements Elicitation: where a description of the problem is obtained from the client.

[27] The exactness of the partitioning is the source of some conjecture in software engineering research. The classification of activities into the development phases shown is necessary to allow the subsequent

- Analysis: where the requirement specifications are analysed to identify ambiguities, contradictions, etc, and the developers acquire a thorough understanding of the problem to be solved.
- Design: where a software-based solution to the specified requirements is devised.
- Implementation and Testing: where the proposed solution is implemented and tested to ensure it meets the client's requirements.

Different design methods and programming languages influence the analysis and design phases as the evolution of the software solution proceeds towards a structure that can be implemented in the chosen programming language. The most well known programming approaches are (see Design in (Marciniak 1994)):

- Structured: The problem is broken into a hierarchical collection of subproblems that represent the functions required to implement the system. The top-level function provides the complete solution, calling appropriate subprograms as necessary. Many data structures store the required information, which are then passed to the appropriate routines as they are called. This approach is directly supported by procedural programming languages and has historically been the most popular approach in software development.
- Data-structured: Like the Structured approach, the data-structured approach partitions the system in terms of data and functionality. However, it is aimed at database intensive systems that have a large, common, data store rather than many smaller data structures. The data is defined first and in detail. The procedures of functionality are then designed with respect to the data model. This approach is directly supported by database manipulation languages.
- Modular: Based on the notion of 'information hiding' (Parnas 1972), this approach advocates decomposing a problem into modules such that each module contains some specific functionality identified during the design. The advantage of this approach is that changes made to the system, such as a modified data model, have little effect on other areas of the system. The problem of change dependency is a problem with the Structured and Data-structured approaches. The

---

discussion to proceed. However, the arguments presented no not rely on the exactness of that classification.

goal of Modular design is to encapsulate the appropriate data structures with its manipulating procedures. This approach can be implemented using traditional procedural programming languages.

- Object-oriented: Beginning as an extension to Modular design that treated the information-hiding modules as 'objects', object-oriented design has now become a completely new paradigm of design that deals with objects, classes, and inheritance between classes. The key concept is to encapsulate data and functionality together to minimise change dependency. However, the theory is more detailed than the original Modular approach. Object-oriented design is quickly becoming the most popular form of design and object-oriented programming languages have been designed to more explicitly support the relevant design concepts than traditional procedural languages.

In addition to the different design methods, different software process models specify the order in which the particular phases should be carried out. Example process models include chaotic, waterfall, multiple builds, evolutionary, spiral, rapid prototyping, and disciplined evolutionary (see Design in (Marciniak 1994)).

Software engineering theories have devised improved process models for ordering the phases and improved paradigms for designing and implementing systems. However, the gross structure of software development has remained unchanged. That is, obtain the requirements from the client, develop a sufficient understanding of the problem to produce a design, devise a solution to the problem that can be implemented in software, and then implement and test that solution. To determine if those phases of software development can be 'engineered' it is necessary to look at the artefacts produced by them. The work presented here looks specifically at the most recent theories in software development, object-oriented design, though the issues presented are equally applicable to other design approaches. It should also be noted that the description of the software design process is necessarily simplified to concentrate solely on the artefacts produced. Many theoretical aspects of the process are omitted for brevity.

The most recent theories in object-oriented software engineering (Larman 1997; Jacobson, Booch et al. 1998; Bruegge and Dutoit 1999; Oestereich 1999; Pooley and Stevens 1999) all suggest similar approaches to those development phases and the artefacts produced by them. During requirements elicitation, use-cases are used to

Withdraw Money Use Case:

1. The Back Customer identifies himself or herself.
2. The Back Customer chooses from which account to withdraw money and specifies how much to withdraw.
3. The system deducts the amount from the account and dispenses the money.

Figure 5-1: Withdraw Money Use-Case

develop and represent the system requirements in a manner that can be understood by the customers, users, and developers. Use-cases represent the behaviour of the system from the user's point of view, where the 'user' is anything external to the system that interacts with it. For example, a user might be a person, another information system or a hardware device (Pooley and Stevens 1999) (p. 99). The behaviour is represented as chunks of functionality that the system offers to add a result of value to the users of the system. Those users are referred to as 'actors' in the individual use-cases (Jacobson, Booch et al. 1998) (pp. 134-135). Jacobson et al use the example of a complex banking system. An individual use case for a (very simple) customer ATM withdrawal is depicted in Figure 5-1.

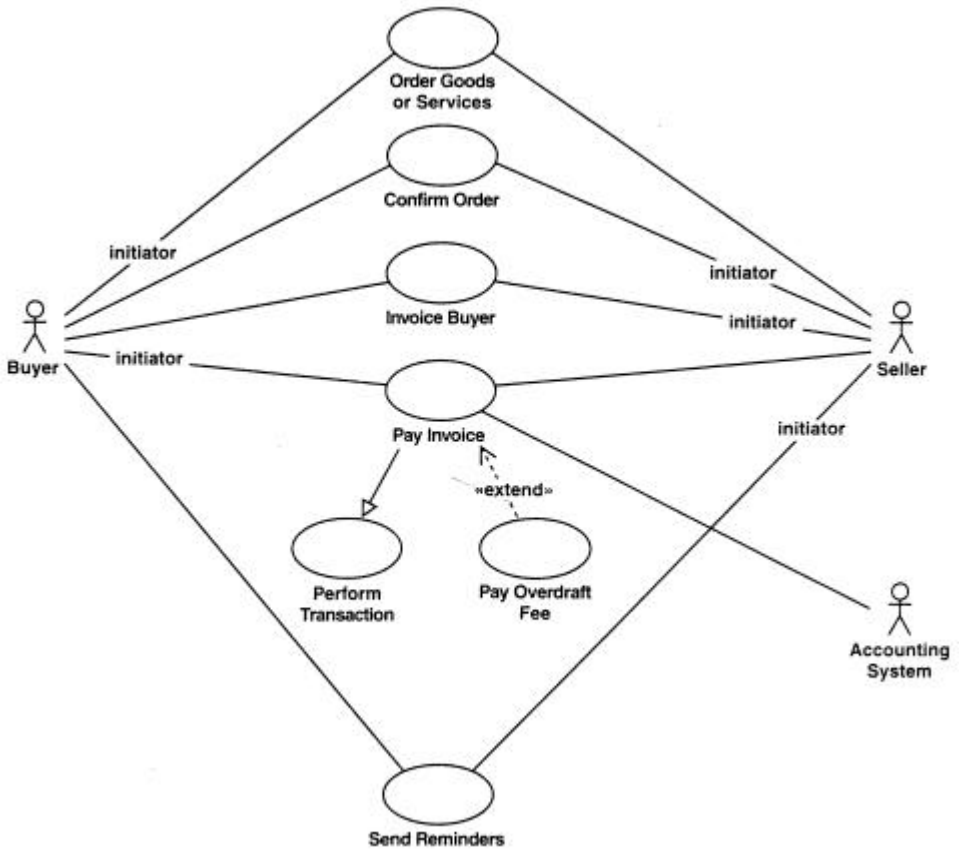


Figure 5-2: Use-Case Model Diagram

The use-case model for an entire system or subsystem is the combination of the individual use-cases. Additionally, that model can be represented in graphical form. For example, figure 5-2 (from (Jacobson, Booch et al. 1998)) depicts the use-case model for a 'Billing and Payment' subsystem. The structure of use-cases is not rigidly defined and the requirements elicitation process may require further refinement of the initially presented use-cases. Their main purpose is to provide a simple means of communication between the interested parties and to provide a basis for analysis and design. Moreover, their use, in combination with domain models and business process models, allow the developers to obtain a thorough understanding of the problem space.

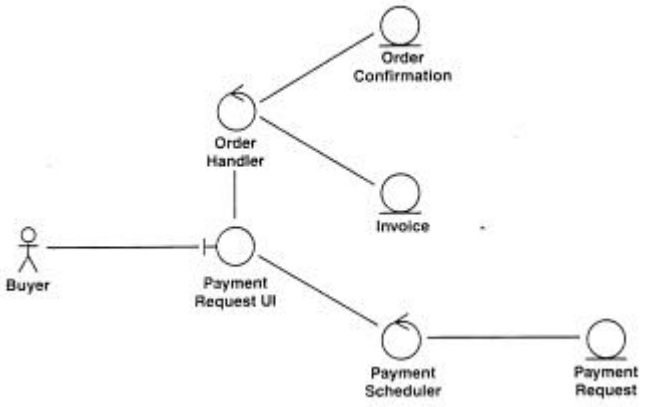


Figure 5-3: Analysis of Pay Invoice

During the analysis phase, the requirements specified in the collection of use-cases are refined by identifying and reusing similar concepts (using terminology from both the developer's and the customer's vocabulary), specifying the functionality with greater precision, and removing ambiguities (Jacobson, Booch et al. 1998) (p. 174). Consequently, the developers also become more familiar with the problem domain. The result of the analysis phase is the generation of an analysis model that identifies the significant concepts of the problem and the way they need to interact to provide a satisfactory solution. For example, a diagram of the concepts resulting from an analysis of the 'Withdraw Money' use-case is depicted in figure 5-3 and the analysis of a 'Pay Invoice' use-case is depicted in Figure 5-4 (both diagrams from (Jacobson, Booch et al. 1998)).

As use-cases are refined during the analysis phase, the system moves closer to design because the identified concepts and their interactions become more formal. Indeed, some object-oriented researchers argue that

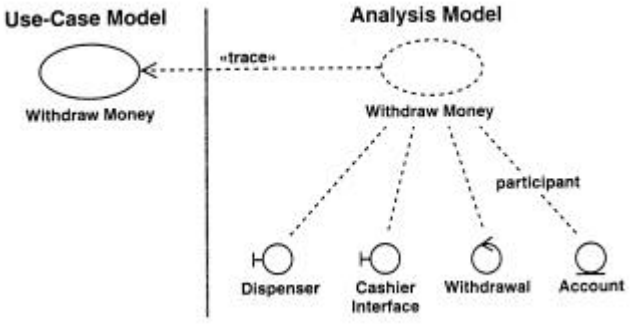


Figure 5-4: Analysis of the Withdraw Use-Case

the product of the analysis phase, the analysis model, can be viewed as an initial version of the design model (Jacobson, Booch et al. 1998) (p. 178).

The boundary between analysis and design is not clearly defined. The goal of analysis is to specify exactly what the system is to do without necessarily how it is supposed to do it. However, constructs used to represent concepts during the analysis phase often have direct analogues in the design phase. Therefore, it may not be clear where the analysis concept ends and the design concept begins.

Despite this blurred boundary, the goal of system design is to transform the analysis model into a design model. The design model is comprised of constructs that are directly realisable in the chosen implementation medium, that is, the combination of hardware environment and software programming language(s). The analysis model consists of the identified concepts from the problem domain and the interactions between them that are required to realise a solution to the problem. However to implement the system, those concepts need to be realised using the constructs of the chosen programming languages (in this case, 'objects'). For example, figure 5-5 (from (Jacobson, Booch et al. 1998)) depicts a more detailed design representation of the Invoice class than the one identified during the requirements and analysis stage. The statechart diagram depicts how the Invoice object changes state as its internal functionality (submit, schedule, time out, and close) is executed. The notions of 'state' and 'functionality of objects' are constructs provided by the implementation environment and are applied to the concept, 'Invoice', to allow its implementation in a software system[28].

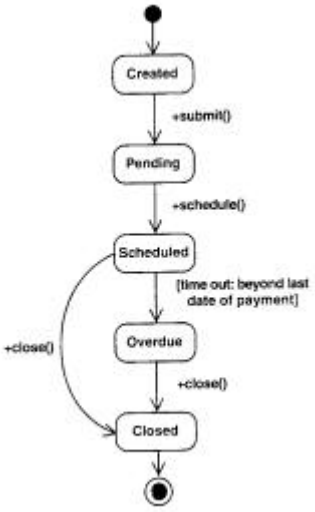


Figure 5-5: Invoice Statechart

To depict how the entire software design may be partitioned into high-level constructs of software engineering theory, a design may also include a software architecture and a deployment model. For example, figure 5-6 (from (Jacobson, Booch et al. 1998)) depicts how the total Banking System may be distributed across the many machines linking the buyer and the client.

[28] This point is discussed in more detail later in the chapter.

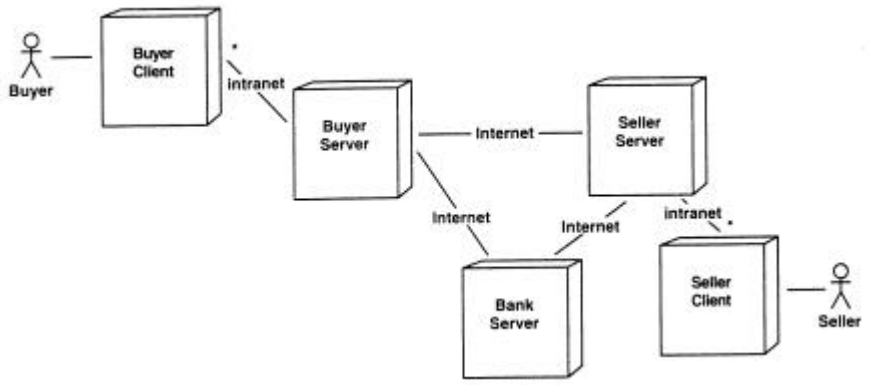


Figure 5-6: Design Deployment Model

Many factors outside of the original functional requirements influence the transformation from analysis model to design model. Non-functional requirements and constraints such as system performance, response time, reliability, modifiability, etc all impact on the how the analysis model is shaped into a collection of constructs that are executed by the machine. The desire to reuse existing software components also influences the transformation. Software architecture styles, product-line architectures and design patterns all provide reusable large-scale structures that are known to realise successful solutions. Similarly, component libraries and design frameworks provide previously implemented smaller-scale components that can be used to reduce development time and improve system quality.

The artefacts produced during the transition from requirements elicitation to the design representation are often talked about as models. The object-oriented references cited have

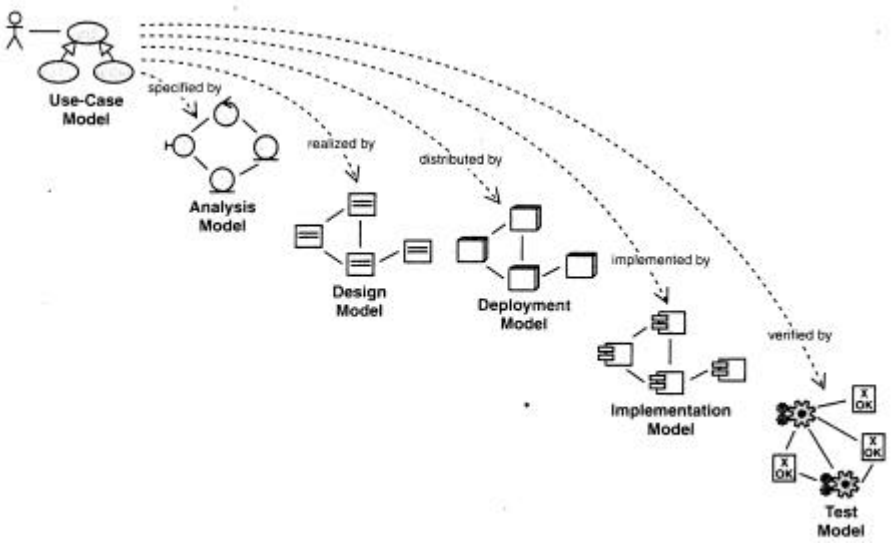


Figure 5-7: Relationship Between Models

talked about use-case models, analysis models, design models and their relationships (figure 5-7). Dillon and Tan describe the models produced during the development process, and the transformations between them, in more general terms (Dillon and Tan 1993) (Figure 5-8).

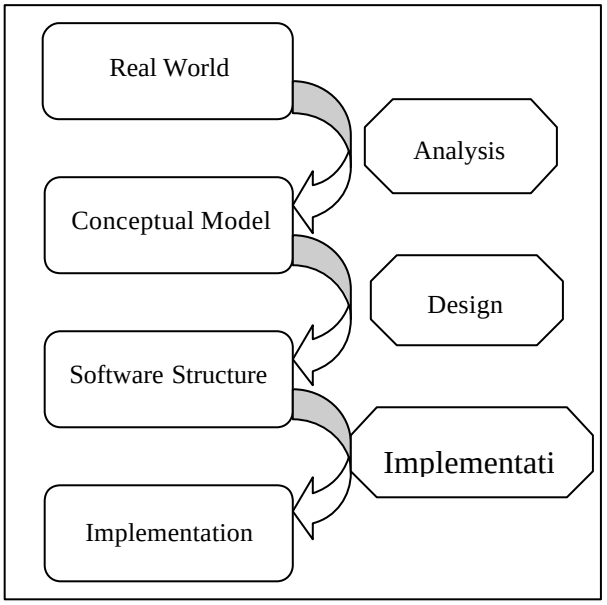


Figure 5-8: Development of Models

> "The conceptual model consists of the model of the real world of interest… It is a representation of the essential characteristics of the real world that are important for the problems that the software system is meant to address. This model is arrived at by a process of analysis or knowledge acquisition. No assumptions are made about the nature of the software structures that will be used to encode the structure of the software system.
>
> Once the conceptual model has been defined and verified, the process of its transformation into the software structure begins. Since the software structure defines the basis of the software implementation, sufficient attention has to be paid to the classes of structures that are available in the implementation medium when defining the software structure. In the traditional software engineering, the software structure is the program structure…
>
> During this process of transformation, the conceptual structures are transformed into the set of acceptable software structures. This process is referred to as design. The software structure model of the system should provide the data structures, the knowledge structures as appropriate, the functions, procedures and methods, the methods of control and inference if necessary, and the modules in the system." (Dillon and Tan 1993) (pp. 24-26).

Texts and papers describing software development methodologies constantly refer to terms such as conceptual models, use-case models, analysis models, design models, process models, architectures, architecture styles, design patterns, programming

paradigms, design paradigms, implementation mediums, and programming constructs. However, the use of those terms is certainly not consistent. The understanding of their place in the development process can be traced back to the origins of the discipline. For example, the notion of 'Concept' was discussed during the 1968 NATO conference and presented in chapter 3. Brooks, in his famous *No Silver Bullet* paper, noted the following,

> "The essence of a software entity is a construct of interlocking concepts: data sets, relationships among data items, algorithms, and invocations of functions. This essence is abstract in that such a conceptual construct is the same under many different representations. It is nonetheless highly precise and richly detailed. *I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor or representing it and testing the fidelity of the representation.* We still make syntax errors, to be sure; but they are fuzz compared with the conceptual errors in most systems." (Brooks 1987) [Brooks' italics].

His earlier book, *The Mythical Man-Month* (Brooks 1975), dedicated a chapter to the importance of the conceptual construct and the importance of conceptual integrity in system design.

Harel also discussed the nature of models in his follow-up to Brooks' article, *Biting the Silver Bullet: Toward A Brighter Future For System Development* (Harel 1992). He described the need to depict the system analysis and design representations using strata of conceptual models that tame the complexity of the solution. That should be done "by allowing the designer to capture the system's inherent conceptual structure in a natural way." He goes onto say,

> "We will first conceptualize, using the 'proper' entities and relationships, and then formulate and reformulate our conceptions as a series of increasingly more comprehensive models represented in an appropriate combination of visual languages. A combination it must be, since system models have several facets, each of which conjures up different kinds of mental images." (Harel 1992)

Harel believed the conceptual model should capture the result of problem analysis by consisting of a functional model of the system and a behavioural model that depicts how the functional model will be executed. A structural model is then used to represent the

result of the design phase by depicting the implementation responsibility of the various parts of the conceptual model to constructs that can be implemented in software.

While many different methods have been proposed, the purpose of system analysis and design has remained unchanged. During system analysis, a model is developed that depicts the problem domain concepts and their interactions in a manner that completely meets the needs of the requirements. The goal of system design is then to transform that model into a collection of concepts and interactions that can be directly realised using the implementation of software developers – programming language constructs, operating system constructs, and hardware constructs. The result of the design phase is what Brooks labelled the conceptual construct.

5.3 Engineering the Conceptual Construct

One of the aims of software engineering research is to improve the process of designing and implementing the conceptual construct. It is founded on the belief that the utilisation of an 'engineering' approach will result in software systems that exhibit reduced development costs and improved product quality. Aspects of this 'engineering' approach that have been identified in traditional engineering disciplines and have become goals for software engineering research include:

- Design Reuse: Software designs that solve one problem can be used to solve other problems.
- Component Reuse: Software code fragments written to implement one design can be used in other software implementations. That includes individual components of the implementation and also the gross, structural form or architecture of the solution.
- Predictable or Repeatable Design Processes: Standardised processes of design and implementation that can be successfully analysed to improve estimation of development time and product quality.
- Formal Methods: The ability to represent the artefacts used in the development process in a rigorous mathematical formalism that can then be analysed and manipulated using known mathematical techniques.
- Standard Engineering Methods: Standard techniques to deal with the complexity of large-scale software design problems.

- Graphical Notations: The ability to communicate software designs and solutions between developers using standard graphical representations.

Those aspects are all related and can be thought of as stemming from the issue of software design and component reuse.

> "Software development cannot possibly become an engineering discipline so long as it has not perfected a technology for developing products from reusable assets in a routine manner, on an industrial scale. Software reuse cannot, in turn, achieve this status unless we make the following provisions: a sound scientific foundation that encompasses relevant design principles, widely accepted engineering standards that compile these principles into working practical solutions, and coherent managerial standards that enable the deployment of these solutions under acceptable conditions of product quality and process maturity." (Mili, Addy et al. 1999)

Attempts to incorporate significant levels of reuse in the development process have ranged from the elicitation of the client's requirements all the way to the implementation of the source code (e.g., (Cybulski, Neal et al. 1998)). Some of them have been very successful. However, as many researchers have commented, the promises of software reuse remain for the most part unfulfilled. Few researchers (in the literature) have ever questioned whether the goal is even possible (e.g., (Glass 1998a)), while others, of course, may have questioned it in more informal debates. The reason no one knows whether the goal is possible is because software engineering does not have a fundamental understanding of the types of systems it builds or the intellectual processes required to build them. The history of the artefact engineering view of software development (Chapter 3) showed that researchers simply assume software systems are analogous to traditional engineering disciplines, therefore, the goal is possible.

The belief that conceptual structures can be designed and implemented using an 'engineering' approach that incorporates significant amounts of reuse requires one significant assumption on the part of the software development community. That belief is stated in the following assertion:

> The identification of items that can be reused from previous applications, from the requirements analysis stage to the implementation stage, assumes that different clients and developers experience the same reality and can

> model it using similar collections of distinct concepts and concept relationships. Moreover, those concepts and relationships can be specifically defined in terms of essential features and represented the same way in two different applications using the implementation medium of software development – hardware and software constructs.

Occasionally, the distinction between physical artefacts and human thought constructs are highlighted by software engineering researchers. For example,

> "Ultimately, a program is a fiction, not made of matter that wears an tears; it is closer to encapsulated human thought than to physical artifacts…" (Belady 1989) (p. viii)

> "There is no consensus about what technical approaches are best for various kinds of reuse problems and little understanding of the nature of reuse opportunities, let alone the constraints, difficulties, and short comings of reuse." (Biggerstaff and Perlis 1989) (p. x)

However, those highlighted issues rarely lead to a questioning of the assumption. It is simply believed that the reason we have not achieved the required levels of reuse is because we haven't yet mastered the technical difficulties. To determine the validity of that assumption, what is required is a review of other disciplines that already research those issues – philosophy and psychology. In a nutshell, we need to examine the question, when we build software, are we actually doing what we think we're doing?

5.4 Is the Assumption Valid? A Summary of the Relevant Research in Philosophy and Psychology

Karl Popper's 1979 article, *Two Faces Of Common Sense* (Popper 1979g), began by apologising for being concerned with philosophy as most philosophers seemed to have lost touch with reality. Furthermore, he felt there was a widespread feeling of anti-intellectualism in the community. Regardless of whether or not a similar feeling of anti-intellectualism exists in the software engineering community, it is clear that philosophical issues are rarely discussed systematically in the research literature. Nevertheless, Popper continued,

> "... we all have our philosophies, whether or not we are aware of this fact, and our philosophies are not worth very much. But the impact of our philosophies

> upon our actions and our lives is often devastating. This makes it necessary to try to improve our philosophies." (Popper 1979g)

Those philosophies guide our understanding of both the global agendas of software engineering research, which are discussed in the next chapter, and they explain our understanding of the more specific, underlying principles of software components and systems. Those underlying principles concern our understanding of concepts, theories, models, and abstractions, which are topics philosophers have been debating for over two thousand years.

However, the application of those philosophical debates to the specific context of software engineering is not a simple task. Bechtel discusses the issue while presenting philosophical theories in the context of cognitive science.

> "The fact that philosophical claims lie so far removed from empirical inquiry poses a challenge to anyone turning to philosophical investigations from training in empirical research. In order to evaluate a philosophical claim you must follow the often complicated chain of reasoning offered in support of the claim. This, however, is not meant to deter outsiders from entering the philosophical arena. … All that is required for the nonphilosopher to get involved with philosophy of mind is to begin to confront the issues. This means becoming an active participant in the debates by offering arguments for or against different positions. It is not enough simply to turn to philosophers as authorities and cite what a particular philosopher has said as an answer to one of these foundational questions. … Rather than simply accepting an authority, it is necessary to explore the issues and to evaluate the arguments advanced for competing claims." (Bechtel 1988a)

A few software engineering researchers have used philosophical positions in support of particular research ideas, even though they may not continue to be supported within the discipline of philosophy itself.

> "This proclivity to borrow positions from philosophy is rather common but poses serious dangers because what might be controversial in philosophy may be accepted by a particular scientist or group of scientists without recognizing its controversial character." (Bechtel 1988b)

The definition of objects in terms of 'intension' and 'extension' is one such example in software engineering research and it is discussed later in the chapter.

To ensure this treatise does not fall into the same trap, the chapter begins with a detailed account of the relevant philosophical theories that relate to the underlying principles of software engineering. Discussions with senior researchers in both philosophy and psychology have also been used to ensure important areas have not been missed or misunderstood (Cumming 2000; Fox 2000). The presentation begins with the ideas of classical greek philosophy and traces the debates through to the analysis of language by Wittgenstein. Obviously, the presentation could not include every debate. It focuses on a number of traditions within the history of philosophy, each of which have offered a general perspective on important issues relevant to our understanding of the underlying principles of software engineering. Other philosophical traditions, such as the hermeneutics of Heidegger, could have been included (for example, see Heidegger in (Urmson and Ree 1989)). However, the material presented is enough to raise the issues required to challenge the artefact engineering view of software development and suggest the benefits of a model building view, which is the goal of the thesis. Future research could then make use of Heidegger's theories in more specific applications of that view.

The resulting presentation compresses a long history of debate into to a short summary and, consequently, it may be difficult to follow – especially for an audience of software engineering researchers who are not primarily trained in philosophy. However, it was necessary to present that amount of detail within the size constraints of this thesis to develop an appreciation of the relevant issues for the understanding of the underlying principles of software engineering.

The presentation of those philosophical traditions is followed by a presentation of the relevant research in the field of psychology. Those theories of concept development, utilisation, and evolution are based on a more empirical approach and the results may be more accessible to the reader. However, they lack the depth of explanatory argument contained in the discipline of philosophy. Importantly, the relevant theories from psychology research parallel those from philosophy for the purposes of this treatise.

5.4.1 Western Philosophy: Metaphysics and Epistemology

5.4.1.1 The Definition of Concepts in Classical Greek Thought

The person generally credited with first discussing the problems we have when defining concepts is Socrates. Living around 400 BC, he did not write any of his own thoughts down, however they were recorded in the works of Xenophon, Aristotle, and, most copiously, in the dialogues of Plato. The goal for Socrates was to discover universally true definitions of our concepts. At that time, education was offered by the Sophists, who were itinerant professors that travelled between cities giving lectures. The sophists mainly taught the art of rhetoric and argued that precise definitions were impossible because words meant different things in different contexts. Consequently, they taught logical tricks to manipulate those meanings for the purposes of winning an argument. Socrates argued that the search for universal truth, rather than mere victory in debate, required the definition of concepts and maintained that we could not have correct knowledge in any field until those definitions were discovered.

Socrates attempted to discover that higher knowledge of concepts by developing 'Socratic definitions'. Using dialectic discussion with others, he attempted to identify the set of essential characteristics of a concept that all instances of that concept possessed. Socrates' dialectic discussions would begin by asking his interlocutor for the definition of a term, for example, "What is courage?" The response would be subsequently questioned until Socrates could gather enough evidence to refute the original definition. For instance, Socrates would provide counter examples which people agreed represented the original concept but would fail to completely satisfy the definition offered. The respondent would then be asked for another definition that would be similarly picked apart. The result would inevitably be that no satisfactory definition could be attained.

Socrates' quest for absolute knowledge led, in part, to his execution. Plato, however, continued the philosophical tradition, which included the search for the understanding of universal definitions. He argued that it was impossible to identify the essential characteristics of concepts in the instances that we perceive. Through a long line of arguments Plato submits that because we can identify concepts such as 'tableness' in the physical instances that we perceive, then they must exist. However, because we cannot successfully develop definitions that account for all the observed examples, there must be differences between the concept and the instances that instatiate it. For example, because

we know about concepts such as 'courage' and 'knowledge', and can recognise examples of them, those concepts must exist and somehow we must have knowledge of them. However, the instances we perceive are only incomplete reproductions of the pure notions of 'Courage' and 'Knowledge'.

Plato believed those absolute concepts, which he termed 'Forms' or 'Ideas', do not exist in the physical world. They exist in some other world which our souls have access to before we are born. During our lifetime, we perceive objects that remind us of these pure Forms and the goal of philosophy is the pursuit of understanding them. The consequence is that all possible concepts exist in the world of Forms and we understand the sensory world around us as objects that faintly copy these innate and subconsciously known ideal Forms. Moreover, our knowledge is about these abstract Forms.

Aristotle was a student in Plato's academy who sought to develop a complete taxonomy of the natural world based on a rigorous logical analysis of worldly experience. His subsequent analysis and theories of abstraction resulted in a different understanding of the problem of concept definitions. Aristotle also believed in a notion of Forms but those 'Forms' do not exist in a different, non-physical world as Plato had suggested. Aristotle argued that if Plato was correct then it would be impossible to account for the occurrence of change. That is, Plato's philosophies could not account for the generation of new substances. To solve the dilemma, Aristotle suggested it was necessary to differentiate between matter and form. For example,

> "A table is wood and glue put together in a certain way. Aristotle distinguishes as separate aspects of the table its matter (the wood and glue) and its form (how it is put together, its structure). … Form is *immanent*: the form of a table exists only as the form of this table or that table, that is, as the form of certain matter. There is no separately existing transcendental Platonic Form of Table." (see Aristotle in (Urmson and Ree 1989)).

Moreover, the form or structure of an object is normally determined by its function. For example,

> "It is because of what it has to do that a table has a flat top and four legs. Form may in fact be identified with function: to say what a table *is* is to say what it does or is for." (see Aristotle in (Urmson and Ree 1989)).

According to Aristotle, matter could subsequently be broken down into further elements such as earth, air, fire, and water. He then argued that all human thought is concerned with manipulating these forms using the principles of categorical logic.

> "Thought is the more active process of engaging in the manipulation of forms without any contact with external objects at all. Thus, thinking is potentially independent of the objects of thought, from which it abstracts the form alone. Even the imagination, according to Aristotle, involves the operation of the common sense without stimulation by the sensory organs of the body. Hence, although all knowledge must begin with information acquired through the senses, its results are achieved by rational means. Transcending the sensory preoccupation with particulars, the soul employs the formal methods of logic to cognize the relationships among abstract forms." (Kemerling 1997a)

Aristotle also attempted to provide a means of systematically defining all aspects of reality. He believed that true knowledge could be represented in categorical logic and that thought and language provides a true account of reality. His categorical logic consisted of assertions of subject-predicate form and that all predicates could be defined in terms of the essential features or properties they exhibited. Those features were divided into ten particular categories. The most important category is substance, which describes a thing in terms of what it most truly is. The substance then acts as the subject for which the other categories can be attached as predicates. Those other categories are: quantity, quality, relative, where, when, being in a position, having, acting on, and being affected by. Used in conjunction, these categories provide a comprehensive account of what any individual thing is. For example,

> "Chloë is a dog who weighs forty pounds, is reddish-brown, and was one of a litter of seven. She is in my apartment at 7:44 a.m. on June 3, 1997, lying on the sofa, wearing her blue collar, barking at a squirrel, and being petted." (Kemerling 1997a).

Furthermore, Aristotle believed his system could explain both specific instances, such as Chloe the dog, and the more general species or genera, such as dogs or animals. Therefore, for Aristotle, to recognise an object was to identify the appropriate Form within it. Moreover, the essence of that Form could defined in terms of its essential properties according to Aristotle's categories (Bechtel 1988a).

Plato and Aristotle's philosophies, of which the issue of concept definition were only one aspect, served as the basis of science and philosophy for many centuries. Indeed, it is argued that people are either Platonist or Aristotelian in their philosophy of the world.

> "That is, each of us is inclined either toward the abstract, speculative, intellectual apprehension of reality, as Plato was, or toward the concrete, practical, sensory appreciation of reality, as Aristotle was. The differences between the two approaches may be too fundamental for argumentation or debate, but the coordination or synthesis of the two together is extremely difficult, so choice may be required." (Kemerling 1997a).

5.4.1.2 How We Have Knowledge Of Concepts: Rationalism, Empiricism, and the Kantian Revolution

During the ensuing centuries, the progress of philosophy was significantly influenced by the power of the Christian church. While its progression continued, it did so only within the bounds of accepted theology. With the advent of the renaissance and the growth of science based on empirical studies, philosophy began to distinguish itself from accepted Christian dogma by providing alternative solutions to problems of the mind and of the natural world. Of particular interest to software developers is the notion of our understanding of the world and how it can be modelled in a rigorous manner. Those issues are examined in the areas of metaphysics and epistemology.

- Metaphysics seeks to understand what sorts of things ultimately compose our reality and how they relate to one another.
- Epistemology seeks to understand how people can have knowledge of that reality and how that knowledge can be represented.

Aristotelian philosophy provided a basis with which scientists could describe and classify natural phenomena. However, neither his metaphysics, nor the work of philosophers after him, could explain the dynamic nature of phenomena or explain the rapidly developing field of natural sciences (Bechtel 1988a). Two different avenues of thought emerged to explain these: either all of our thoughts about the world are inferences from sensory experience (empiricism) or the world exhibited a natural order that could be derived by rational analysis (rationalism).

Around the seventeenth century, philosophers in continental Europe, especially Descartes, Spinoza, and Leibniz, led the rationalist approach. They believed that because our

experiences could be error-prone or illusionary, the only way to develop a precise understanding of the world is to use pure reason. In order to provide that certainty, the rationalists, like Plato before them, took their model of knowledge from mathematicians who derive theorems from axioms they took to be indubitable (Bechtel 1988a). For example, Spinoza claimed to deduce the entire system of thought from a restricted set of definitions and self-evident axioms (Kemerling 1997a). The senses had a role to play but they were secondary to that of reason.

> "We can get beyond guesswork and fallible opinion to knowledge by operating as geometricians and arithmeticians operate, namely by pure thought, not vitiated by the deliverance of our senses. Where we can calculate and demonstrate we can know. No set of sense-impressions can yield knowledge. Where we can only observe and demonstrate we cannot know. No set of sense-impressions can yield knowledge. Only by exercises of pure thought can we ascertain truths." (see Epistemology in (Urmson and Ree 1989)).

The metaphysical belief required to follow the rationalist approach is that the world consists of a number of ready made parts and the relation between those parts has been constructed in a logically sensible manner. Therefore, the rigorous use of logic can successfully define it. However, there were a number of problems with this approach to epistemology. The use of pure reason guarantees the indubitability of our knowledge but leaves serious questions about its practical content. By ignoring the effect of our sense impressions on knowledge, we can only deal in abstract truths. For example,

> "Pure geometry cannot tell us the positions or dimensions of actual things in the world, but only, for example, that *if* there is something in the world possessing certain dimensions, *then* it has certain other dimensions. Geography could get nowhere without geometry, but geometry by itself cannot establish the position or even the existence of a single hill or island. Truths of reason win the prize for certainty only at the cost of being silent about what, if anything, actually exists or happens." (see Epistemology in (Urmson and Ree 1989)).

While Rationalism was developing in continental Europe, philosophers in Britain were developing an opposing epistemological view, Empiricism, where reason plays a less

central role to the importance of sensory perception. The philosophers Locke, Berkeley, and Hume had noticed the success that scientists such as Bacon, Newton, and Ryle had achieved using a program of rigorous observation and experimentation. Consequently, they sought to provide a theory of knowledge that was compatible with a carefully conducted study of nature.

The empiricists tried to show that knowledge is developed by observing naturally occurring phenomena and then developing concepts and theories from it using the principles of induction and association. For example, Locke maintained that every thing we know or believe is made up of ideas that come from our receiving sense impressions and then reflecting on them. To explain our knowledge of concepts that we had not encountered physical instances of, he suggested that complex ideas could be developed by conjoining simpler components.

> "My idea of 'unicorn,' for example, may be compounded from the ideas of 'horse' and 'single spiral horn,' and these ideas in turn are compounded from less complex elements. What Locke held was that every complex idea can be analyzed into component parts and that the final elements of any complete analysis must be simple ideas, each of which is derived directly from experience." (Kemerling 1997a)

For Locke, those elements could be refined until an idea was defined in terms of its primary and secondary qualities. The primary qualities are its quantifiable attributes such as size, weight, microscopic structure, etc. The secondary qualities are subjective views of the element. For example, smell, taste, and colour. From these qualities, we then define an object as a particular class or species by identifying its *nominal essences* – the most important features of that object.

The empiricists also believed that the world consisted of identifiable objects and that observation and abstraction could infer their important properties and interrelationships. However, this system of epistemology also encountered problems when subject to rigorous analysis. For example, there is always the possibility of mistaken perceptions. Empiricists hold that the foundations of our knowledge are sense-impressions, but what guarantee is there that those impressions are correct? How do we know those sense-impressions are unadulterated by any assumptions, guesses, or expectations?

Furthermore, assuming the initial sense-impressions are correct, what guarantee is there that our inferences from that sensory stimulus to the complex ideas are correct?

> "Our knowledge of the world around us, together with our mere beliefs and conjectures about this world, are all conglomerations of interlocking conclusions inferred, sometimes legitimately, sometimes riskily and sometimes illegitimately from our impressions. Knowledge, unlike belief and conjecture, would be the product solely of legitimate and riskless inferences. But then what, if anything, can guarantee our inferences themselves against being mistaken? … We discover the ways in which things always or sometimes happen only by finding them happening and then collating our findings; and even then the laws and regularities that at any particular time we claim to have ascertained are always subject to subsequent correction." (see Epistemology in (Urmson and Ree 1989)).

Both the empiricists and the rationalists had assumed that the objects of knowledge exist independent of us and we must determine how we can know them. That assumption dates back to Aristotelian philosophy which assumed that man is a 'blank tablet' that can be filled with knowledge either through inferences from sensory experience or from rational analysis. However, neither account could provide a systematic explanation for our knowledge of the world.

> "The rationalists had tried to show that we can understand the world by careful use of reason; this guarantees the indubitability of our knowledge but leaves serious questions about its practical content. The empiricists, on the other hand, had argued that all of our knowledge must be firmly grounded in experience; practical content is thus secured, but it turns out that we can be certain of very little. Both approaches have failed." (Kemerling 1997a)

In the late seventeenth century, Kant suggested that the only way of resolving these problems was to revolutionise our way of thinking about knowledge. He considered it analogous to that of Copernicus who resolved many issues in astronomy by changing the fundamental assumption of astronomical theories from an earth-centred solar system to a sun-centred one. Kant proposed that rather than assume the objects of our knowledge exist independent of our minds, it was actually the process of conceptualisation that partly created the things we experience. The categories we contain in our cognitive apparatus

are applied to the sensory input we receive to create the world we conceive. Kant's approach attempted to synthesise the approaches of the rationalists and the empiricists and resolve the dilemmas that each one faced. Knowledge is created from the sensory inputs we receive, but it is the abstract truths of our reason that provide the organising principles of that sensory input.

Kant classified our statements about the world using a twofold process. They are either analytic or synthetic and they are either *a priori* or *a posteriori*. An analytic statement is something that is true by virtue of the meaning of the words in it. For example, the statement "a bachelor is unmarried" is true because of the definition of the word 'bachelor'. A statement, therefore, is analytic if its negation results in a logical absurdity. All other statements are then synthetic. For example, the statement, "the car is red", maybe false and is therefore synthetic. Only these synthetic statements make useful claims about the world. A statement is *a priori* if it is "independent of all experience and even of all impressions of the senses" (see Kant (Urmson and Ree 1989)), that is, it relies solely on pure reason. All analytic statements are then *a priori* because they can be verified by virtue of the definition of their terms and without resorting to experience. In contrast, *a posteriori* statements must be grounded in experience and may be true or false in particular cases. Combining the two, three types of statements are then possible (analytic *a posteriori* being impossible by definition):

- **Analytic *a priori*:** All analytic statements as already discussed.
- **Synthetic *a posteriori*:** Matters of fact that we know through our experience.
- **Synthetic *a priori*:** Kant identifies statements from mathematics and science in this category. For example, "every event has a cause". "This can be denied without logical absurdity and yet, in its complete generality, is something neither confirmable nor falsifiable by sense experience." (see Kant (Urmson and Ree 1989))

Kant, in contrast to previous philosophers such as Hume, suggested that we have synthetic *a priori* knowledge about the world. For example,

> "Our knowledge that two plus three is equal to five and that the interior angles of any triangle add up to a [180 degrees]. These (and similar) truths of mathematics are synthetic judgements, Kant held, since they contribute significantly to our knowledge of the world; the sum of the interior angles is

> not contained in the concept of a triangle. Yet, clearly, such truths are known a priori, since they apply with strict and universal necessity to all of the objects of our experience, without having been derived from that experience itself. … The question is, how do we come to have such knowledge? If experience does not supply the required connection between the concepts involved, what does?" (Kemerling 1997a).

Kant argued that it is the synthetic *a priori* notions that allow us to impose order on the sensory world we experience. For example, the statement "x causes y" is used by all people to understand the relationship between phenomena.

> "It [causality] is certainly not abstracted from any *perceived* necessary connexion, since all that we ever perceive is successions of occurrences. That we do not abstract the relation of causal necessity *from* perception had already been shown by Hume … yet we do apply this concept *to* perception." (see Kant (Urmson and Ree 1989)).

Kant labeled the synthetic *a priori* concepts that we apply to perception, 'Categories', and set about identifying all of those logical structures.

The result of Kant's conjectures is that there is no single independent world that we can all know because what we experience is influenced by the 'categories' we subconsciously impose on the world in an attempt to understand it. Our knowledge is based on the world we construct in our attempt to find meaning and purpose. These ideas repudiate the metaphysical quest to identify everything that exists. The goal now became not to understand the structure of the world but the structure of how we understand the world.

It is necessary to detail two philosophical terms which now become important – idealism, and its opposite, realism.

- Idealism: is the theory that physical objects have no existence apart from a mind that is conscious of them.
- Realism: asserts that physical objects exist essentially independently of the mind of any perceiver.

Kant's theories are therefore a type of idealism.

5.4.1.3 Pragmatism, Analytic Philosophy, and Logical Positivism

After the introduction of Kant's revolutionary epistemology, philosophers in America continued the investigation of how our conceptual apparatus influences our perception of the world. The work of Peirce, James, and others led to the philosophical principle of pragmatism. Peirce's pragmatism (in the late nineteenth century) was based on the belief that humans have an unshakeable desire to understand the world around them.

> "Feeling keenly dissatisfied by any suspension in judgment, we invariably seek to eliminate it by forming a belief, to which we then cling firmly even in the face of evidence to the contrary. So powerful is this urge to believe something in every circumstance that many people (as Bacon had noted centuries before) adopt beliefs upon whatever seems ready-to-hand, including individual interest, appeals to authority, or the dictates of a priori reasoning." (Kemerling 1997a).

From this line of reasoning, Peirce concluded that humans develop concepts and theories to help them understand the world around them. As those concepts and theories are employed, they build expectations about the world that may fail. When those expectations fail, the conceiver modifies those concepts and theories to provide a better match with the experienced reality. Moreover, he suggested that as people investigate reality, they will converge, in the long run, on the same conception of the world, the one that most clearly corresponds with reality.

Peirce also wondered how we could define those concepts. The result was his pragmatic maxim:

> "Consider what effects, which might conceivably have practical bearings, we conceive the object of our conception to have. Then, our conception of these effects is the whole of our conception of the object." (see Peirce in (Urmson and Ree 1989)).

A concept, then, is defined in terms of the effects it has on our senses and we know we have true knowledge of an object when we can predict those effects when we come to deal with it.

An interesting aside that parallels the problem that software engineering research has with terminology concerns the label applied to Peirce's philosophy. As other philosophers

extended and modified his line of thought, the concept of 'pragmatism' became more general than Peirce had originally intended. To save his original meaning Peirce wrote,

> "To serve the precise purpose of expressing the original definition, he begs to announce the birth of the word 'pragmaticism', which is ugly enough to be safe from kidnappers." (see Peirce in (Urmson and Ree 1989))

Philosophy, especially in English speaking countries, then moved towards an analysis of language to explain how we derive and can be sure about the meaning of concepts and propositions. At the beginning of the twentieth century, Frege attempted to construct a foundation for the meaning of mathematical terms and expressions by defining them using purely logical concepts – in the process creating the first predicate calculus. That process led Frege to focus more closely on the meaning of terms and symbols in language in general. Using puzzles about concept identity, Frege developed his theory about sense and reference (or denotation). For example, he considered the statement "The morning star is the evening star". In the existing theory of language, terms were simply thought of as words that referred to existing objects of experience. The terms 'morning star' and 'evening star' both refer to the planet 'Venus'. Subsequently, the statement "the morning star is the evening star" is the same, in terms of our knowledge of things, as the tautology "Venus is Venus". However, this is obviously not the case. The first statement is more informative than the second is. To solve this problem, Frege distinguished between the 'sense' and the 'reference' of a term. The 'sense' of a word, phrase or symbol is the concept it expresses, while its 'reference' is the real world object it represents.

> "The expressions '4' and '8/2' have the same denotation [reference] but express different senses, different ways of conceiving the same number. The descriptions 'the morning star' and 'the evening star' denote the same planet, namely Venus, but express different ways of conceiving of Venus and so have different senses. The name 'Pegasus' and the description 'the most powerful Greek god' both have a sense (and their senses are distinct), but neither has a denotation. However, even though the names 'Mark Twain' and 'Samuel Clemens' denote the same individual, they express different senses." (Frege in (Zalta 1999))

Analytic philosophy follows Frege's logical analysis of language and attempts to solve philosophical problems by restating them in precise logical terminology. Two of the

earliest analytic philosophers were G. E. Moore and Bertrand Russell. Like Frege, Russell began by attempting to resolve the philosophical problems of mathematics using logic. He did this by analysing the fundamental terms of mathematics in terms of purely logical concepts and then developing a symbolic logic to allow the deduction of all mathematical propositions. After publishing this work, with Whitehead, in *Principia Mathematica*, he then attempted to use logic to solve philosophical problems in metaphysics, epistemology, ethics, political theory, and the history of philosophy. Russell believed that philosophers should use the 'scientific method' of evaluating all hypotheses through the weighing of evidence and they should use logic to exhibit the 'logical form' of natural language statements. Because the logical form of a statement could be significantly different to its grammatical from, "a statement's logical form, in turn, would help the philosopher resolve problems of reference associated with the ambiguity and vagueness of natural language." (Russell in (Zalta 1999)). Russell's analytic approach was able to solve many philosophical problems, such as the use of descriptions and proper names, and as a result, it prepared the way for the development of the philosophical movement called logical positivism.

At the end of the First World War, the Vienna Circle, a group of philosophers, mathematicians, physicists and social scientists, began the movement of logical positivism. Noting scientific advances, such as Einstein's theory of relativity, and Frege and Russell's use of logical analysis to analyse and solve philosophical problems, they proposed to reduce the philosophy of human knowledge by using only logical and scientific foundations. One of their central doctrines was the 'Verifiability Principle' which asserts that a statement is meaningful only if it can, in principle, be true or false.

> "Hence the class of meaningful propositions is exhaustively divisible intro those whose truth-or-falsity can be established on formal grounds (i.e. logic and mathematics), and those in which it is, or could be, confirmed by verification (or falsification) through sense-experience." (see Logical Positivism in (Urmson and Ree 1989)).

A consequence was that because many theories in metaphysics and, to a lesser extent, epistemology could not be verified by sensory experience, they were considered meaningless. Furthermore, they believed that the natural language used to express philosophical problems could be restricted to remove ambiguities. That language would

then match the logical structure of the world and the logical structure of human knowledge of the world.

> "Experience (it was held) can be resolved into its ultimate constituents, namely the immediate and incorrigible sensory observations of which the observer's world consists. The structure so presented is reflected in language; more precisely, it can be shown by logical analysis that the propositions in which knowledge is expressed are similarly reducible to elementary propositions, corresponding one-to-one with actual or possible items of sense-experience." (see Logical Positivism in (Urmson and Ree 1989))

One of the most influential philosophers on the logical positivists was a student of Russell's, Ludwig Wittgenstein. Wittgenstein began from the metaphysical premise that the world consists entirely of simple facts, none of which are dependent on any other. Human beings then have thoughts about these worldly facts that are our way of picturing the way things really are. Moreover, our thoughts can be represented in language. Because human language represents and communicates the facts in logical propositions, human language must have a structural similarity to the way things really are. The entire world could, in principle, be adequately represented as a long list of atomic sentences.

> "An informative statement will be a picture of some possible state of affairs in the same way as a sketch-map can picture a battle or the arrangement of the furniture in a room." (see Wittgenstein in (Urmson and Ree 1989)).

5.4.1.4 Human Understanding and Conceptual Relativism

Wittgenstein left the field of philosophy for 10 years and returned with a completely different theory of language and attempts to define meaning from it. His original theory was that language represented the logically connected facts of the world and that a restricted, ideal language could be constructed that perfectly described that world of facts. However, he realised that this theory, and the work of the logical positivists, required too much precision in the use and definition of words and in the representation of logical structure. In practice, people's use of language does not conform to such a rigid structure. Rather than simply state facts about the logical structure of the world, people use language in a variety of different ways, which he refered to as different language games. Examples of games include giving orders, asking, thanking, cursing, greeting and praying (Wittgenstein 1968). These different language games are learned through childhood and

while a few of them may take precedence in philosophical analysis, they do not account for all language usage.

By analysing the use of words in different language games, Wittgenstein developed a new theory for the meaning of words. The accepted doctrine was that names apply to particular objects when those objects possess certain features or properties. That theory goes back to Socrates quest for the defining properties of concepts and Aristotelian essentialism. Wittgenstein showed that many real world entities that are referred to using the same term cannot be defined using the same set of essential features. Wittgenstein uses the example of the term 'game'.

> "Consider for example the proceedings that we call 'games'. I mean board-games, card-games, ball-games, Olympic games, and so on. What is common to them all? – don't say: "There must be something common, or they would not be called 'games'" – but look and see whether there is anything common to all. – For if you look at them you will not see something that is common to all, but similarities, relationships, and a whole series of them at that. To repeat: don't think, but look! – Look for example at board games, with their multifarious relationships. Now pass to card-games; here you will find many correspondences with the first group, but many common features drop out, and others appear. When we pass next to ball-games, much that is common is retained, but much is lost. – Are they all 'amusing'? Compare chess with noughts and crosses [tic-tac-toe]. Or is there always wining and losing, or competition between players? Think of patience. In ball games there is winning and losing; but when a child throws his ball at the wall and catches it again, this feature has disappeared. Look at the parts played by skill and luck; and at the difference between skill in chess and skill in tennis. Think now of games like ring-a-ring-a-roses; here is the element of amusement, but how many other characteristic features have disappeared! And we can go through the many, many other groups of games in the same way; can see how similarities crop up and disappear…
>
> And the result of this examination is: we see a complicated network of similarities overlapping and criss-crossing: sometimes overall similarities, sometimes similarities of detail." ((Wittgenstein 1968) par 66)

Wittgenstein suggested that although these different examples of the same concept do not have a shared set of defining features, they do have something in common that cannot be easily identified as particular properties.

> "I can think of no better expression to characterize these similarities than 'family resemblances'; for the various resemblances between members of a family overlap and criss-cross in the same way." (Wittgenstein 1968) par 66).

Wittgenstein concluded that words and sentences can not have precisely defined meanings or logically defined constructions. The purpose of language is not to represent the structure of a logically defined reality but to allow communication between people. Therefore, the meaning of words and sentences can vary as groups of people apply them.

> "The members of any community—cost accountants, college students, or rap musicians, for example—develop ways of speaking that serve their needs as a group, and these constitute the language-game … they employ. Human beings at large constitute a greater community within which similar, though more widely-shared, language-games get played." (Kemerling 1997a).

Wittgenstein then concluded that the existence of many philosophical mistakes was due to the inability to identify the nature of the particular language games and the rules that govern them.

The theory that the precise meaning of words is relative to its use within a context can be traced back to Frege's analysis of language (Reck 1997). Furthermore, another recent philosopher, Willard Quine, analysed the notions of logical positivism by taking the consequences of the their theories to their logical extremes. By doing so, he also arrived at the conclusion that the precise meanings of concepts are relative to the observer. Quine however went further than Wittgenstein and suggested that all concepts, even those of scientific observation, are culturally dependent[29]. He argued,

> "… words do not have specific meanings, but only meanings in the context of a whole network of other words to which they are connected in the sentences we take as true." (Bechtel 1988a).

---

[29] The relativism of scientific observation is discussed in the next chapter – specifically in the theories of Feyerabend.

In fact, Quine argued that every theory contains its own ontology and we should abandon the idea that words can have specific meanings.

> "Statements, apart from an occasional collector's item for epistemologists, are connected only deviously with experience, [so that] there is many a slip twixt objective cup and subjective lip." (see Quine in (Urmson and Ree 1989)).

The basic core of this conceptual relativism, that the precise definition and meaning of concepts is dependent on context, is now generally accepted as standard in contemporary philosophy. The conceptual schemes (contexts) that govern the precise meaning of a concept can operate over an entire culture or period; or it may be conceived more narrowly as the theoretical framework of a particular community: for example, quantum physicists, or software engineering researchers (see Cognitive Relativism (Fieser 1999)). However, debate still occurs concerning particular aspects of conceptual relativism. For example, the exact nature of the relationship between the meaning of a concept and its conceptual scheme, the relationship between different conceptual schemes, and issues concerning whether particular conceptual schemes take precedence over others. Some philosophers argue an extreme form of conceptual relativism in that no conceptual scheme has precedence over any other and that conceptual relativism implies a form of relative idealism. In contrast, other philosophers argue that the success of science shows that some form of conceptual schemes must be applicable to all people. Still others have argued that relative idealism must be tempered by some sort of common-sense realism (Smith 1995).

This conceptual relativism can be traced back to the Sophists, particularly Protagoras, who noted,

> "Man is the measure of all things – of things that are, that they are, of things that are not, that they are not." (see Cognitive Relativism in (Fieser 1999)).

However, the success of Socrates, Plato, and Aristotle in belittling the philosophy of the Sophists has meant that conceptual relativism had few supporters through the ages. Nevertheless, the thorough analysis of the ensuing philosophic traditions has shown that the Socratic definition of concepts is still impossible.

5.4.1.5 Definition and Meaning

To reduce the ambiguity and vagueness of words in language, philosophers and logicians have sought techniques to agree on the definition of terms. Swartz (Swartz 1997) and Kemerling (Kemerling 1997b) explain the issues and analyse the effectiveness of attempts to develop precise definitions of concepts. They detail the following types of definitions:

- Stipulative Definitions: Specify how a term is to be used. It assigns meaning to a new term or restricts the meaning of a term in a particular context. The use of stipulative definitions for new terms is always correct because there are no existing standards with which to compare it. However, when stipulating the definition of existing concepts, especially those with a well-entrenched usage, it is often difficult to stay within the newly imposed boundaries.
- Lexical Definitions: Provide a description of how a term is already used within a community. For example, dictionaries provide lexical definitions. The correctness of the lexical definition is the degree to which it accurately captures the common usage. Therefore, they do not define words, they merely report the standard usage.
- Precising Definitions: Reduce the vagueness of a term in a particular context. They often combine the previous types of definitions by beginning with the lexical definition and them stipulating the new restrictions on its usage. Examples include the attempt by legislators to define the meanings of commonly used terms in a legal context.
- Theoretical Definitions: Are examples of stipulative definitions within a scientific or intellectual context. Often these definitions imply the acceptance of other definitions within a larger, encompassing theoretical framework. For example, Newton's theoretical definitions of 'mass' and 'inertia' "carried with them a commitment to (at least part of) his theories about the motion of physical objects." (Kemerling 1997b)
- Persuasive Definitions: Used to attach some emotive meaning to a term for the purposes of winning an argument. Persuasive definitions generally bastardise existing lexical definitions and have no legitimacy. For example, "'Logic' is by definition 'the study of the means by which to win an argument'. (Swartz 1997)

Philosophers have for quite some time defined the precise meaning of a term using the concepts: 'extension' and 'intension'. The extension of a term is the collection of things to which that term is correctly applied. That application can refer to things in the past, present or future tenses. For example, "the extension of the word 'chair' includes every chair that is (or ever has been or ever will be) in the world." (Kemerling 1997b). The intension of a term, on the other hand, is the set of defining or essential characteristics that are shared by everything to which it applies. For example, the intension of the word 'chair' may be "a piece of furniture designed to be sat upon by one person at a time." (Kemerling 1997b). Similarly, the intension of the word 'triangle' may be "(the properties of) being closed, having three straight sides, and lying in a plane." (Swartz 1997).

There is a reciprocating relationship between the intension and extension of a term. As the intension of a term becomes greater or more precise, the extension, the set of things to which it applies, decreases. For example,

> "The term 'black' denotes a certain class. Adding the term 'round' to 'black', viz. increasing the intension, creates a new expression whose extension is a new, smaller, class, a proper subclass of the former." (Swartz 1997)

Terms can be defined using the intension or extension separately. Definition by intension is most obvious in the disciplines of science and mathematics. For example, "x is a circle" means "x is a locus of a point in a plane lying equidistant from a given point" (Swartz 1997). However in general situations and especially when teaching children, terms are often defined by extension – pointing out or describing example objects. The meaning of the term is defined by pointing out and noting particular examples of a category and also noting examples of objects that are definitely not in the category. These definitions by extension can include pointing out particular objects, not only visually, but by using any of the senses or by describing objects that the person has been in contact with before. A restricted form of definition by extension, which is solely concerned with visually pointing out a particular example, is definition by ostension.

For a long time it was thought that definition by extension leads to the subject developing a mental definition by intension. That is, as objects are pointed out as either part or not part of a particular category, the subject would develop theories about which were the essential features that determined an object's membership in the category. However, that theory has come under attack in recent times in both the disciplines of philosophy and

psychology (Swartz 1997). The thrust of the attack concerns the identification of the essential attributes. Because there are many features of an object to identify, there exists a large number of attributes that all members could have in common. None the less, people do develop meanings of concepts successfully using definition by extension. Swartz claims that the ability to do so is innate in our conceptual apparatus.

> "How is it, then, that human beings can ever use this method, and indeed frequently do so with such success? Here one must offer a scientific theory, a theory to the effect that we human beings are physically sufficiently like one another that we will often, after only a few tries, 'come up with' the same sorts of linguistic hypotheses as those of our fellow human beings. In short, the explanation is that we have a built-in (hard-wired perhaps) predisposition to frame similar sorts of linguistic posits as other human beings." (Swartz 1997).

Despite the fact that people can successfully define objects by extension, that definition can never guarantee the creation of the exact intension. Terms with the same extension may have different intensions. In contrast, the definition of the term by intension does specify the exact extension. However, while an intensional definition does specify an exact extension, it does not guarantee that the intension specifies the correct set of objects that was intended. While we would like a definition to 'fit' the intended extension as closely as possible, often a particular set of defining characteristics omits certain objects that were intended or includes additional objects that were not. To find an intension that fits the required extension exactly is very difficult, and in some cases, impossible.

The Classical Theory of Definition, as described by Swartz, has two principal tenets,

> "(1) that a 'proper' intensional definition states in the definiens the logically necessary and sufficient conditions for the application of the definiendum; and (2) that there are intensional definitions for each of the class terms (e.g. 'horse', 'house, 'musical instrument', 'educated person', etc.) which we use." (Swartz 1997).

This theory does not claim that a concept has an innate definition, or set of essential features that can be identified. However, it does claim that for most concepts, definitions can be constructed, based on identifiable features, to suit that purpose. Proponents of this Classical theory often use examples from mathematics and geometry to illustrate the

process. However, Swartz notes that the theory cannot automatically be extrapolated to other concepts.

> "Somewhat uncritically, on the basis of this model, people believed that comparable definitions could be constructed in more ordinary contexts, that definitions of ordinary class terms – not just those in mathematics and science – ought to specify the necessary and sufficient conditions for the application of the definiendum." (Swartz 1997)

As has been pointed out, the more recent philosophers (e.g., Frege, Wittgenstein, Quine, and their academic descendants) have shown that this is not the case. The precise meaning of a term is dependent on the context and culture in which it is used. Apart from some extreme examples, there can be no single, universal intensional definition.

Swartz and Kemerling discuss two different alternatives to deal with the problems facing intensional definitions – the 'cluster-concept' (Swartz 1997) and the 'genus and differentia' (Kemerling 1997b) definitions. To explain the idea of a cluster concept Swartz uses the definition of the concept 'lemon'. Lemons exhibit the following characteristics: are yellow, sour, ovoid, grow on trees, as big as a ten year old's fist, are juicy, have internal seeds, have a peculiar aroma, have a thick skin, internally segmented, pulpy, have a pocked surface, green prior to maturation, grow in a semitropical environment, have a waxy skin, contain vitamin C, and are edible. If all of these features were used to specify the concept's intension it may over specify the term 'lemon'. For example, if an object were found that did not meet one of these characteristics but did meet all the others, it would probably still be classified as a lemon. However, it is difficult, if not impossible, to identify which ones are necessary for an object to meet in order to be classified as a lemon. As long as it meets most of them, then it would be a lemon. A 'cluster-concept' is a collection of features, none of which is individually necessary, but the majority of which, are.

> "Thus in one sense we do not know the definition of 'lemon': that is, we cannot give a classical intensional definition for it. Yet it would be absurd to say that we do not know what 'lemon' means. Of course we do. The concept, lemon, is a cluster concept, and we know the conditions in the cluster and we are fairly well agreed on their relative importance." (Swartz 1997).

In addition to the idea of the 'cluster-concept', logicians have proposed the use of 'genus and differentia'. The process begins by identifying a more general or broader category that includes all of the intended members. Because this will include items that are not intended, the genus is supplemented with a differentia, which identifies the defining characteristics of the intended items from the others that exist in this general category. For example, to define the concept 'chair' the general genus to which it belongs may be "piece of furniture" and the differentia required to identify all chairs from the other pieces of furniture may be "designed to be sat upon by one person at a time." (Kemerling 1997b)

Swartz concludes that although the classical theory that concept definition can be achieved using intension and extension is still quite pervasive, and does work for a small number of domains, it should not be relied upon.

> "In short, very often we know the extension of a term very well, we can even 'go on' reasonably well, yet we are unable to specify the intension, and moreover ought on many occasions to resist the demand that we try to give an intensional definition for the term." (Swartz 1997)

5.4.1.6 Using Theories to Understand Phenomena: The Philosophy of Science

The theories concerning the formation and definition of concepts have significant ramifications for software engineering researchers, specifically in the areas of component design and reuse. Those issues are examined later. However, they do not explain how concepts are used to construct complex explanatory theories or models of reality. Those issues are examined in philosophy of science and they have significant ramifications for software engineering, specifically in the areas of design reuse, software architecture, and design patterns.

The 'philosophy of science' is reasonably recent in the discipline of philosophy, however its origins can be traced all the way back to Aristotle. Indeed, until the scientific revolution of the seventeenth century it was difficult to distinguish between philosophy and science – there was no sharp line between physics and metaphysics. However, since the nineteenth century, philosophers of science have been theorising about how science progresses and how people can know the knowledge it develops is valid. Those theories deal with issues that include the methods used by science, the relationships between experiment and theory, and whether the theories developed by science match the reality of the world. All of these have ramifications for how software engineers develop their

designs to automate real world processes, how software engineers can be sure those designs are valid, and whether or not those designs can be reused for other problems and in other contexts. To detail these issues, it is necessary to explain the theories of the logical positivists and their opponents, specifically those of Karl Popper. Philosophers of science have also developed many theories to explain how scientific and non-scientific disciplines have evolved. Those issues are discussed in the next chapter, which examines how software engineering research can progress.

The logical positivists have been the most influential group on the philosophy of science in the twentieth century. As has been discussed, the logical positivists originated in the 1920s and were predominantly influenced by the philosophical ideas of positivism, the successful use of logic by Russell and Frege to explain mathematics and language, and the radical advances in physics – especially those of quantum mechanics and relativity theory. They proposed to determine what made science a reliable source of knowledge.

Their epistemological view was that the world exhibited a logical structure that was captured in the language used to express and communicate it. This positivism led them to the Verifiability Principle, which asserts that the only significant theories of science are those than can be verified as being either true or false by testing them against experience. Their explanation of science was that scientists propose hypotheses which were accepted if sufficient evidence was found to support it. The logical positivists assumed the method of discovery was a type of induction where the scientist observes phenomena and recognises regularities of cause. Those regularities are captured in scientific hypotheses that can be verified using experimentation. If the hypothesis was successfully verified, then it must capture the essential structure of reality and could become a scientific law. The logical positivists believed the issues involved in discovering the hypothesis could only be explained by psychologists and, instead, concentrated on developing detailed explanations for how the hypotheses could be justified using the tools of logic. As an example, Rudolph Carnap attempted a logical, axiomatic account of all space and time. In his *Logical Structure of the World* he details a rigorously formal version of empiricism (see (Fieser 1999) on Carnap).

Some philosophers prior to the logical positivists had discussed the issues concerning the discovery of hypotheses. Comte, who introduced the ideas of positivism, believed that the concepts involved in hypotheses, specifically those of interaction such as ‘cause’, were intellectual constructs developed by scientists. That is, there is no natural concept of

cause. It is something invented by us to explain the regularities observed in nature and to allow us to make predictions about future events. A contemporary of Comte's, Whewell, believed these concepts did exist but they could not be seen directly. He believed Kant's notion that we apply our innate notions of cause on the reality that we experience. The correct notion of cause could only be discovered by successfully developing theories that explain reality (see Philosophy of Science in (Urmson and Ree 1989)).

The theories of the logical positivists became the dominant explanation for the structure of science. However, there were many problems with them. One of those was that scientific thinking does not necessarily follow the strict canons of logic. Moreover, others had argued many years before against the assumption that hypotheses, which are developed using induction by abstracting general theories from repeated observation, could be successfully verified using empirical testing. The empiricist Hume had argued that inductive evidence could never establish definitively, or even render probable, the truth of any general claim. It is always possible that there might be counter-evidence to a general claim that simply had not been discovered yet. Similarly, the mathematician Poincaré argued against the basic positivist assumption that reality had a logical structure that could be captured in all-encompassing scientific laws. Poincaré began to study the philosophy of his discipline when the successful introduction of non-Euclidean geometries challenged the assumption that geometry had a single logical structure. His research showed that scientific theories may start with experience but they do not exactly match it.

> "For example, look at the problem of finding a mathematical law that describes a given series of observations. In this case, representative points are plotted in a graph, and then a simple curve is interpolated. The curve chosen will depend both on the experience which determines the representative points and on the desired smoothness of the curve even though the smoother the curve the more that some points will miss the curve. Therefore, the interpolated curve – and thus the tentative law – is not a direct generalization of the experience, for it 'corrects' the experience. The discrepancy between observed and calculated values is thus not regarded as a falsification of the law, but as a correction that the law imposes on our observations. In this sense, there is always a necessary difference between facts and theories, and

> therefore a scientific theory is not directly falsifiable by the experience" (see Poincaré in (Fieser 1999)).

Poincaré's conclusion was that scientific theories are not immutable truths. Rather, they are explanations, chosen by convention, that help people explain reality and make useful predictions about the future. Therefore, neither Euclidean geometry, nor any other such as Reimann geometry, is the 'true' geometry. One may simply be more convenient than the other for the purpose we wish to use it for.

These issues, which contradict the theories of science proposed by the logical positivists, were constructed into formal theories by Karl Popper. Popper was a contemporary of the logical positivists and was part of many of the meetings of the Vienna Circle. However, he did not agree with their positivist approach, in particular, with the Verification Principle. His attack began on the belief that people devise theories using induction. Like Hume, Popper showed this belief must be false because evidence could be found to support virtually any theory (Popper 1979b). He argued this belief was part of the prevailing influence of Aristotelian essentialism in that "intuitive knowledge consists in grasping the 'indivisible form' or essence or essential nature of a thing" (Popper 1983). If all things could be defined in terms of their essential features then the things that comprise those defining features would also have to be defined. Similarly, those defining features would also have to be defined and an infinite regress of definition would ensue. Moreover, this belief assumes the empiricist 'bucket-theory' of knowledge, in which people's minds begin as an empty bucket that is filled by observing and conceptualising the objects of experience. However, because there are so many facts to be observed, and because so many features can be identified for each thing, how can the essential ones be determined? Popper's conclusion was that people do not observe things and seek to somehow define them in terms of important features, as the logical positivists had assumed. Rather, they observe defining or important features and characteristics and then seek a label, a concept name or theory, to represent them in their knowledge structure (Popper 1983).

Popper believed people had a purpose in developing concepts and theories and this purpose directed how concepts and theories were identified. He explained his theory based on his '3 worlds' model of human knowledge (Popper 1979g; Popper 1979c):

- World One is the physical world.

- World Two is our conscious experience or perceptions of world one.
- World Three is the collection of theories that we use to explain our understanding of world one. These include theoretical situations, problems, problem situations, critical arguments, the state of discussion on the state of critical arguments; and the contents of books, journals, etc.

The link between the worlds is our mind. Humans can devise theories to explain the world. As the mind (world 2) experiences reality (world 1) it can devise explanatory concepts and theories (world 3) that can be communicated, written down, and shared with others. However, the link also works the other way. As humans perceive the world (world 1) in their minds (world 2) they automatically apply the concepts and theories that are inherent in culture and language (world 3). All our actions in the first world are influenced by our second world grasp of the third world. Popper quotes the philosopher Myhill, "our formalisations correct our intuitions while our intuitions shape our formalisations." (Popper 1979c). Concepts are partly a means of formulating theories and partly a means of summing up theories. In this sense, Popper has provided similar arguments to those of Kant when he proposed a solution to synthesise the epistemological theories of empiricism and rationalism. For Popper, people do not share an objective reality. However, they do develop and share an objective knowledge (world 3) of it.

With this model of knowledge, Popper showed that different concepts and theories (w3) could not only be used by the mind (w2) to explain our experience of nature (w1), but that it was unavoidable. All observation is theory-laden. For Popper, theories are not developed by simply observing nature. Concepts, theories, and therefore knowledge, are developed within the context of solving a problem. Therefore, it is the overriding context of problem solving that governs which features of reality are observed. In an attempt to understand nature, different theories are implicitly applied and tested to see how well they match reality. A scientist's or anyone else's observations are selectively designed to test the extent to which a given theory functions as a satisfactory solution to a given problem. As Charles Darwin noted, "all observation must be for or against some view." (Popper 1979d)

Following these arguments, Popper concluded that the only true test of a theory was not whether it could be verified but whether it could be falsified. All knowledge is provisional, conjectural, and hypothetical – scientific theories can never be finally proved.

Observation can merely provisionally confirm or conclusively refute them; hence at any given time we have to choose between the potentially infinite number of theories that will explain the set of phenomena under investigation. Therefore, 'true' theories are only those that are yet to be falsified, and the only scientifically useful theories are those that can be falsified. They only 'laws of nature' are those hypotheses that are yet to be refuted.

Popper claims the development of knowledge is a continuous process of "conjectures and refutations". A scientist begins by making a conjecture about how the world is and then seeks to refute that hypothesis by testing the theory with attempts to falsify it. If the hypothesis is disproved, then it should be discarded and replaced with a different conjecture. For example, consider the different theories of gravitation. Galileo's explanation that gravity acts as a constant acceleration is an adequate theory to explain falling objects on earth. However, it fails to provide an adequate explanation for the motion of celestial bodies. Consequently, Newton conjectured a new theory of gravitation as a force that is not a constant acceleration but one that is proportional to the mass of the interacting bodies and the inverse square of the distance between them. When dealing with the phenomena of falling objects on earth, the mass of the falling object and its distance above the ground are negligible compared to the mass and radius of the earth. Therefore, Galileo's theory is a good approximation to Newton's in some contexts. Finally, Newton's theory was later superseded by Einstein who developed a theory of gravitation outside the confines of Newtonian mechanics. His theory explains gravitation as curvature in the entire fabric of space & time.

According to Popper, the development of newer, more complex theories is due to imaginative leaps of understanding. Moreover, he claims that it may not be until we have been working on a problem for a considerable length of time that we really begin to understand it to the extent required to see the full ramifications of the solution we have devised. When we first encounter a problem, we don't know much about it. We always start with an inadequate solution and then criticise it. Only then can we develop a better theory, which will also be criticised. This process gradually leads to useful knowledge.

> "To understand a problem means to understand its difficulties; and to understand its difficulties means to understand why it is not easily soluble – why the more obvious solutions do not work. We produce the obvious solution and then criticize them, in order to find out why they do not work. In this way, we become acquainted with the problem, and may proceed from bad

> solutions to better ones – provided always that we have the creative ability to produce new guesses, and more new guesses. … If we have been working on a problem long enough, and intensively enough, we begin to know it, to understand it, in the sense that we know what kind of guess or conjecture or hypothesis will not do at all, because it simply misses the point of the problem, and what kind of requirements would have to be met by any serious attempt to solve it. We begin to see the ramifications of the problem, its subproblems, and its connections with other problems." (Popper 1979d)

The theories of Popper, and other philosophers of science, oppose the claims of the logical positivists. However, their original conviction in positivism, logic, and verifiability still have a strong influence on how people, especially non-philosophers, view the evolution of science. Popper notes,

> "Human observation showed an immensely powerful 'need for regularity' … which makes them sometimes experience regularities even when there are none; which makes them cling to their expectations dogmatically; and which makes them unhappy and may drive them to despair and to the verge of madness if certain assumed regularities breakdown." (Popper 1979b).

Indeed, Popper shows that a single counter-instance is never enough to refute the belief in an existing theory and they may, in fact, be retained even though considerable evidence conflicts with them.

> "There are fashions in science, and some scientists climb on the bandwagon almost as readily as do some painters and musicians." (Popper 1979e)

Other philosophers of science, especially Kuhn and Feyerabend, have argued that Popper did not fully considered the effects of our theories on the concepts we identify. They disagree with Popper that our collection of theories (Popper's world 3) is objective and accumulative. That is, that everyone has access to the same collection of explanatory theories about the world and that those theories simply grow in a linear, progressive fashion as newer, more complex theories are discovered to replace older, simpler ones. Kuhn, being the first philosopher to systematically study the history of science as well as the philosophical theories about it, argued that science passes through stages. In each stage the researchers of a particular discipline operate within what Kuhn called a paradigm (and later a disciplinary matrix). These paradigms consist of the concepts,

theories, and example problems that researchers use to conceptualise the world they observe. Therefore, the concepts and theories developed by researchers in a discipline are contextually dependent on the encompassing paradigm they are operating within. In this sense, Kuhn is similar to Quine, who argued on epistemological grounds that all concepts are context-dependent. The consequence of Kuhn's theory is that the very notion of 'objective facts' that can be viewed by all researchers is called into question. All observations are tinctured by theory.

> "How we capture behavior may depend upon the theory we are using to try to understand the behavior. Theory-ladenness does not entail that we can see whatever we want to. Given the way we have been trained to see, what we see is determined by what there is to see." (Bechtel 1988b)

Kuhn argued that as a scientific discipline progresses, it passes between different paradigms. During those transitions a revolution occurs, where the concepts used to understand and theorise about a set of phenomena are completely replaced by a new set of concepts and theories. For example, the shift in physics from Newtonian mechanics to quantum mechanics. The result of those revolutions is a new paradigm that consists of new concepts and theories that can not be objectively compared with the previously existing paradigm because of their paradigm dependence. For example, the different theories of gravitation developed by Newton and Einstein exist within the different paradigms of Newtonian mechanics and quantum mechanics. Any attempt to use these concepts of gravitation outside the paradigm-dependent way of conceptualising reality simply does not make sense. Moreover, these paradigms govern how new observations of phenomena are conceptualised and those concepts and theories are also paradigm-dependent.

Feyerabend, a student of Popper's, took the notion that concepts and theories are relative to a particular paradigm to an extreme. He proposed a deliberately controversial theory that the meaning of all terms is relative to a particular view of the world and that those views are not neatly packaged in successively utilised paradigms as Kuhn had suggested. Therefore, science can progress with a plurality of encompassing views. Moreover, it may be possible that one theory can develop 'facts' that refute a competing theory and that a newer theory may reveal 'facts' that were not possible with the older one. The progress of science, therefore, does not occur with a traditional, sanitised, concrete methodology and researchers should abandon the belief that it does so. Feyerabend believed scientists

become stuck in a conservative view of their discipline and develop theories that simply seek to perpetuate each other without taking the risk of developing anything truly new and exciting (Feyerabend in (Zalta 1999))[30].

5.4.1.7 Consistency and Coherence in Theory Creation

The idea that concepts and theories are relative to some paradigm or encompassing viewpoint does not mean it is possible to "see whatever we want to see". We all interact with the same reality, but the way that reality is understood as concepts and theories is influenced by previously established ways of understanding. Moreover, freedom of construction of those concepts and theories can be exercised only within limits. Within theories concepts must be consistent with each other and constitute a coherent and consistent system.

> "Consistency is the controlling logic relation. Although one cannot always systematically prove consistency when one has it ... one can recognize inconsistency." (Lee 1973).

One set of concepts is no closer to a 'true' reality than any other. It is only more useful than another in that it yields conceptual interpretations that better suit the apparent demands of a given situation or problem. As the process of understanding proceeds, it is possible that concepts can be created in terms of other concepts and eventually their relation to concrete, real-world experiences gets left behind. For example, logic and pure mathematics are two areas where conceptual manipulation of symbols occurs without those symbols requiring any direct, perceptual referents.

> "With increasing powers of symbolisation, the mind grasps abstract relations and veridical perception is established. An orderly external world composed of facts is systematically organised by further application of logical principles, and veridical percepts are placed in relation to this more inclusive order. Finally, the principles of order themselves can be critically examined in a system of pure logic." (Lee 1973).

Abstraction is used to devise concepts and theories that encompass a wider range of applicability than the original theories. For example, the role of mathematical modelling

[30] The views of Kuhn and Feyerabend are discussed in more detail in the next chapter, along with other philosophers of science such as Lakatos and Laudan.

in social science deals with abstract idealisations of real world entities and not with the direct concepts.

> "We have learned that pure mathematics is neutral and, when applied, it is applied to our ideas about some matter of fact, and not the facts themselves. What gets mathematized is not a chunk of reality but some of our ideas about it." (Bunge 1973).

One of the most influential tools in the generalisation of concepts and theories to encompass a wider area of applicability is the use of analogy. For example, as mentioned in chapter 3, Ohm devised the law of basic current flow by studying the well-established theories of hydrodynamic systems (Jungnickel and McCormmach 1986). Ohm was able to show that both disciplines contained the concepts of substance flow through a conductive element, the concept of force to generate the flow, and the concept of resistance to the flow in that conductive element. He was then able to devise a mathematical relationship for current flow based on the established mathematical relationships that described hydrodynamic systems. By developing a more abstract theory that describes two or more systems, if only in a sketchy way, it is possible to identify analogies and thereby generate theories to explain phenomena in new fields or generate encompassing theories across a range of fields (Bunge 1973). However, although an abstract theory can be used to explain the phenomena in two distinct situations, it is only because of an identified similarity. It does not mean the two situations are identical or that other abstract theories in one of those areas can be used to explain phenomena in the other.

Analogy is a very powerful tool in developing new theories. However, it is only able to suggest equivalence without being able to establish it. Therefore, the reckless use of analogy has also been misleading in scientific research. Bunge provides many good examples of this in a range of fields including quantum physics, information and entropy, and social evolution. Identity implies equality, equality implies equivalence, and equivalence implies similarity, however the converse is not true. "Analogy is undoubtedly prolific, but it gives birth to as many monsters as healthy babies. In either case its products ... are just that: newborns that must be reared, if at all, rather than worshipped" (Bunge 1973).. Bunge continues by quoting Gerard, "Analogical thinking is ... in our view not so much a source of answers on the nature of phenomena as a source of challenging questions". (Bunge 1973).

Many issues of epistemology and philosophy of science that relate to software engineering have been detailed here, though the discussion has certainly not all covered all of them. Before detailing the effects they have on software engineering research, it is necessary also to detail relevant theories that psychologists have devised to explain the role of concepts and theories in our understanding of real-world processes.

5.4.2 The Psychology of Cognition

Ultimately, the development of a software system requires that we can conceptualise the real world and convert our understanding into an executable description. But the fact that we can understand and interact with the world at all is quite an astonishing feat in itself! Furthermore, the fact that we can do so without a conscious understanding of how our mind performs this task belies the complexity of the processes involved. Psychologists have been researching the field of cognition for many decades and have devised theories that explain how we develop our conceptual apparatus and how that apparatus is used to allow us to function in the world. Steven Pinker begins his book, *How the Mind Works*, with the disclaimer, "we don't understand how the mind works – not nearly as well as we understand how the body works" (Pinker 1997). Despite the fact those complex operations have not been fully understood, psychologists have devised many experiments and illusions that provide glimpses into how the mind operates and they have developed theories that explain both cognition and concept development and utilisation.

We must be able to conceptualise in order to function in the world the way that we do. The human brain gets its visual information about the world from the splashes of light that pass through the eyes and onto the retina. The light from the 3-dimensional world forms a 2-dimensional image on the retina which is somehow perceived as a collection of 3-dimensional shapes, objects, surfaces, etc with different depth, texture, and colours. Without the cognitive apparatus required to reconstruct the apparent 3-dimensional structure of the world, everything that we see would just be a constant stream of visual psychedelia. The ability to do so seems to be largely innate in us. Research shows that the limited visual experience of three to four month old infants is enough to perceive the visual milieu as a collection of cohesive, solid, objects that follow natural laws of movement and contact interaction (Kellman 1996; Spelke and Hermer 1996).

From the ability to visualise the world as a collection of objects, the mind has evolved the ability to think about those objects as concepts or ideas – the ability to generate

knowledge. Evolutionary theories of natural selection suggest that the ability to generate knowledge and reason about those objects has helped us to deal with and survive in the world (Pinker 1997). From the sensory experience obtained through interaction with real world objects, concepts and categories are developed along with the ability to infer rules concerning their interaction. No two physical situations are exactly alike and the ability to infer in terms of categories has ensured that we do not have to treat every situation as completely new. Consequently, our ability to survive in the world has improved.

> "An intelligent being … has to put objects in categories so that it may apply its hard-won knowledge about similar objects, encountered in the past, to the object at hand." (Pinker 1997)

The fact that we have conceptual apparatus that enables us to perceive the world in a manner that allows us to develop knowledge about it is interesting for software development. More important though are the theories that explain how that apparatus works and how those concepts and categories are identified. They have significant ramifications for the assumptions that underlie software engineering research.

5.4.2.1 The Classical Theory of Categories

The classical theory of categories holds that something is a member of a particular category because it satisfies the set of necessary and sufficient features or attributes that constitute the category's defining properties, functions, and uses (McCauley 1987). As people interact with the world they become acquainted with the important properties of a particular category as they deal with the individual objects that instantiate them. Perception then, is a decisive process where the person interacts with objects and utilises certain attributes of that object to infer what sort of concept it is. Therefore, categories can be treated as specifications (Bruner 1958).

Using this classical theory of categories, learning is a bottom-up process where people start with the simplest objects and categorise them in terms of the essential attributes. Objects that are more complex are then identified by combining the previously defined simple concepts. Moreover, the psychology of cognition can be understood in terms of set theory where objects belong to a particular set based on their defining features and simpler or more complex objects can be understood in terms of set operations such as subsets, intersections, unions, etc (McCauley 1987).

That classical theory became quite popular, however attempts to use it to formally define particular concepts ran into anomalies. For instance, Pinker uses the example of the concept 'bachelor'.

> "A bachelor, of course, is simply an adult male who has never been married. But now imagine that a friend asks you to invite some bachelors to her party. What would happen if you used the definition to decide which of the following people to invite?
>
> Arthur has been living happily with Alice for the last five years. They have a two-year-old daughter and have never officially married.
>
> Bruce was going to be drafted, so he arranged with his friend Barbara to have a justice of the peace marry them so he would be exempt. They have never lived together. He dates a number of women, and plans to have the marriage annulled as soon as he finds someone he wants to marry.
>
> Charlie is 17 years old. He lives at home with his parents and is in high school.
>
> David is 17 years old. He left home at 13, started a small business, and is now a successful young entrepreneur leading a playboy's lifestyle in his penthouse apartment.
>
> Eli and Edgar are homosexual lovers who have been living together for many years.
>
> Faisal is allowed by the law of his native Abu Dhabi to have three wives. He currently has two and is interested in meeting another potential fiancée.
>
> Father Gregory is the bishop of the Catholic cathedral at Groton upon Thames.
>
> The list … shows that the straightforward definition 'bachelor' does not capture our intuitions about who fits the category. Knowing who is a bachelor is just common sense, but there's nothing common about common sense. Somehow it just finds its way into human … brains." ((Pinker 1997) p. 13)

As an extension to Pinker's challenge to use this simple definition to determine whom to invite, imagine writing the piece of software that defined all of the terms and then

automatically chose bachelors based on the situations presented. The simple definition becomes a complex collection of rules and data specifications.

5.4.2.2 The Prototype Theory of Concept Identification

In the mid 70s Eleanor Rosch and her colleagues conducted a number of experiments in cognitive development that the classical theory of categorisation was unable to explain, see for example (Rosch 1978). They found that people do not categorise objects in terms of defining attributes. In addition they discovered that people's categories do not have clear-cut boundaries and that complex objects are not categorised in terms of features identified in simpler concepts and then abstracted to higher-level concepts.

Rosch suggested people conceptualise objects as belonging to a particular category by developing prototypes – stereotypical examples that a person believes correctly exemplify their understanding of that category. As new objects are perceived they are compared with the prototypes to determine which category they should belong to. Objects are not defined in terms of their essential attributes, they are categorised based upon some typicality rule which compares them to a previously identified exemplar. As people learn about a particular environment, the prototype objects are classified before the more marginal objects can be dealt with.

Like the classical theory, the prototype theory establishes that people's conceptual apparatus is constructed as a hierarchic collection of categories. However, unlike the classical theory, they found it is not generated in a bottom-up fashion. Rosch and her colleagues found that people initially identify what she terms *basic-level* categories – for example, the category 'chair'. As experience grows, both more specific and more generic categories are devised to classify objects. For the 'chair' example, a more specific, or subordinate category, would be 'high-chair' or 'stool' while a more generic, or superordinate category, would be 'furniture'. The basic-level categories tend to be the easiest to identify and correspond to the objects most often perceived. In addition, members of each basic-level category usually have a family-resemblance and contain many of the same attributes. Furthermore, people tend to have similar ways of interacting with them. However, the superordinate level categories may not possess similar attributes. More detailed descriptions of the classical theory and the results of Rosch's work can be found in (Rosch 1978; Keil 1987; McCauley 1987; Neisser 1987a; Neisser 1987b; Gelman 1996; Pinker 1997).

5.4.2.3 The Role of Theories in the Understanding of Concepts

Since the publication of Rosch's experimental results, psychologists have been attempting to devise theories of cognition that can successfully explain them. Current theories examine the role that intuitive theories about the world play in our means of conceiving the structure of that world. The classical theory of categories assumed that the world as we see it simply exists and that as we move through it we understand what is going on by abstracting concepts and their interactions from the sensory cues we experience. That is, we are somehow separate from the world and the human mind simply perceives the important properties of particular objects and moves inferentially to concepts and their relationships. In contrast, recent theories suggest that people's relationship with the world is more interactive. As we act in and with the world, we understand it, partly, by imposing on it our expectations of what concepts and interactions we believe exist.

Research has found that rather than being defined in terms of essential characteristics, people categorise objects in terms of they roles the play within intuitive theories about how the world operates. Whereas concepts were traditionally treated as isolated, atomic units, it is now recognised that they are interrelated and influenced by larger knowledge systems of theories. Pinker again uses the ever-popular chair example.

> "An artifact is an object suitable for attaining some end that a person intends to be used for attaining that end. The mixture of mechanics and psychology makes artifacts a strange category. Artifacts can't be defined by their shape or their constitution, only by what they can do and by what someone, somewhere, wants them to do. Probably somewhere in the forests of the world there is a knot of branches that uncannily resembles a chair. But like the proverbial falling tree that makes no sound, it is not a chair until someone decides to treat it as one." (Pinker 1997).

There is evidence that suggests children begin to form a set of concepts and tacit theories in their first year of life (Keil 1987; Gelman 1996). As the child encounters and interacts with events in the world, the objects involved tend to be grouped together and form an expectation about how objects will interact in future encounters. Rather than as an analysis of discrete objects in the world, categories are formed by analysing and somehow storing the structure of those events (Fivush 1987).

Researchers propose different theories to explain the exact nature of the interaction between concepts and theories (for example see (Lakoff 1987; McCauley 1987; Meddin and Wattenmaker 1987; Neisser 1987b)). However, they all agree that categories are somehow understood in terms of theories rather than defining features. The term 'theory' is often used interchangeably with the term 'idealised cognitive model'. McCauley explains the differences as follows.

> "… 'theory' in contrast to 'idealized cognitive model', connotes constructs that systematically characterize certain aspects of the world, but also a degree of formality, which probably does not apply to all the cognitive structures in question. … In contrast, 'idealized cognitive model' includes less systematic constructs that may not adequately describe the more developed cognitive frameworks that structure large areas of human experience. … Idealized cognitive models are simplified mental constructs that organize various domains of human experience, both practical and theoretical. Theories should, perhaps, be construed simply as the more elaborate and complex of our idealized cognitive models." (McCauley 1987).

The collection of a person's knowledge then is not simply a hierarchy of categories abstracted from sensory experience. It is the sum of all these cognitive models or theories. These are then used to plan behaviour and develop new knowledge by mentally playing out combinatorial interactions among them in the mind's eye (Pinker 1997).

Theories not only capture our understanding of concepts, but it is those theories that allow people to conceptualise phenomena as they operate within the constant stream of sensory experience. The world can be conceived as an infinite variety of concepts and properties, people's innate theories of the world impose an order on this endless amount of detail to allow us to function in it. They form an idealised representation of reality that under-emphasises or ignores a huge number of possible features by implicitly assuming their relative lack of importance.

> "They specify a set of cues in our environment that serve to define the situation and therefore establish expectations about probable changes in the environment and appropriate responses to them." (McCauley 1987).

For instance, McCauley uses Kant's explanation of the concept 'triangle' to highlight the fact it would be impossible for people to develop their idealised concepts solely by abstracting from experienced instances.

> "No amount of instances of, for example, a triangle … could ever be adequate to the concept of a triangle in general. It would never attain the universality of the concept which renders it valid of all triangles, whether right-angled, obtuse-angled, or acute angle; it would always be limited to a part only of this sphere. The schema of the triangle can exist nowhere but in thought … Still less is an object of experience or its image ever adequate to the empirical concept…" (McCauley 1987).

The consequence, as McCauley continues, is that "the *world-in-itself* is forever inaccessible" (McCauley 1987). The world we conceive has already been filtered by the conceptual apparatus that allows us to cope with that huge amount of detail.

The influence of our innate theories on our conceptualisation of reality is highlighted in experimental results that were performed many years before these theories of cognitive development were devised. For example Carmichael showed that concepts identified in language affect how people perceive different shapes (Carmichael, Hogan et al. 1932). Similarly, Wertheimer showed how people automatically attempt to group disparate visual information into clusters so that they can be understood (Wertheimer 1958).

5.4.2.4 Human Understanding and Conceptual Relativism

At the basic-level, people identify similar collections of concepts because they are based on similarities in appearance and function (Rosch 1978). The superordinate and subordinate categories however, are developed through cultural convention and are learned and passed on through language use (McCauley 1987; Neisser 1987a; Pinker 1997). Therefore, as people learn a language they also learn about a culture's concepts and theories of the world. All documented cultures have words for the elements of space, time, motion, speed, mental states, tools, flora, fauna, and weather, and logical connectives (not, and, same, opposite, part-whole, and general-particular) (Pinker 1997). However, the meaning of words and the means of conceptualising the world is culturally dependent. This has been shown in various studies in sociology and anthropology, see for example (Levi-Strauss 1962; Levi-Strauss 1986; Knudtson and Suzuki 1992; Lee and Karmiloff-Smith 1996).

In addition to cultural dependence, the level of expertise in a domain can affect the conceptualisation of phenomena. There is evidence to suggest that as mastery of a domain occurs the basic-level of the conceiver changes. As a domain is mastered, larger and more complex cognitive models are developed and the new basic-level becomes the next level in the hierarchy of categories that contains the greatest level of detail (McCauley 1987). The expert is able to categorise objects more efficiently than a novice based on these more sophisticated models of the domain.

There is more than one model that can explain a particular situation and it is possible to entertain these models simultaneously. The ability to do so depends on imaginative capacity and different aims and purposes (Meddin and Wattenmaker 1987). Those different models or theories can have different levels of completeness, they may not be fully consistent, and they can provide different starting points from which further knowledge can be inferred. They also represent to different levels of veridicality the world we are trying to conceive.

> "Our various cognitive models offer alternative descriptions of the world. Everyone recognizes from time to time that certain descriptions are not only less helpful than others (given the problem at hand), but also that some are *for all intents and purposes* false." (McCauley 1987).

The basic level concepts and theories identify some of what McCauley terms, the "major joints of the world". However, as "we undertake steps of increasing sophistication … we rely increasingly on the developed, abstract theories that we consciously entertain." (McCauley 1987). There is no guarantee that these abstract theories and concepts provide definitive reflections of the world. They can only be relied upon based on their perspicuity rather than proven representational accuracy.

The theories of cognition and human perception/conception are more complex than the classical theories suggest. However, researchers note that the classical theory of mind still pervades many theories of science, suggesting it is a carry over from Aristotle's essentialism (Gelman 1996; Pinker 1997). Nevertheless, it has been superseded with theories that define concepts and categories, not as a collection of essential attributes, but as things that exist within a encompassing theory or model of observed phenomena. Pinker summarises with the following extract.

> "Buckminster Fuller once wrote: "Everything you've learned … as obvious becomes less and less obvious as you begin to study the universe. For example, there are no solids in the universe. There's not even a suggestion of a solid. There are no absolute continuums. There are no surfaces. There are no straight lines." In another sense, of course, the world does have surfaces and chairs and rabbits and minds. They are knots and patterns and vortices of matter and energy that obey their own laws and ripple through the sector of space-time in which we spend our days. They are not social constructions, not the bits of undigested beef that Scrooge blamed for his vision of Marley's ghost. But to a mind unequipped to find them, they might as well not exist at all. As the psychologist George Miller has put it, "The crowning intellectual accomplishment of the brain is the real world … [A]ll [the] fundamental aspects of the real world of our experience are adaptive interpretations of the really real world of physics." (Pinker 1997)

5.5 Understanding the Foundations of Software Engineering

The theories from philosophy and psychology identify issues that have a tremendous significance for software engineering research. Although it is impossible to comprehensively capture all of the theories from these disciplines in such a small number of pages, it is clear the epistemological and metaphysical assumptions that underlie current thinking in software engineering research are, at best, too simplistic – at worst they are fundamentally wrong.

The epistemological assumption required to 'engineer' the conceptual construct in software development was stated at the beginning of this chapter as follows:

> The identification of items that can be reused from previous applications, from the requirements analysis stage to the implementation stage, assumes that different clients and developers see the same reality and can model it using similar collections of distinct concepts and concept relationships. Moreover, those concepts and relationships can be specifically defined in terms of essential features and represented the same way in two different applications using the implementation medium of software development – hardware and software constructs.

Two aspects of the research shown highlights the inherent difficulty in 'engineering' conceptual constructs. They are the nature of concepts and they way the human mind uses conceptual models to understand reality. Both philosophical analysis and psychological experimentation have shown that the human conceptual apparatus does not specify universally applicable definitions of concepts in terms of essential characteristics or attributes. Nor does it identify concepts simply by abstracting them from the observed reality. However, the belief that concepts are defined in such a way is still prevalent outside the areas of philosophy and psychology. That includes the area of software engineering. Many philosophers and psychologists have argued this belief is a product of the still prevailing influence of philosophers who set the foundations for western thinking – Socrates' quest for true knowledge through definition and Aristotle's attempts to define all knowledge through essential characteristics (see for example (Popper 1979a; Lakoff 1987; Bechtel 1988a; Gelman 1996; Pinker 1997)).

In software engineering, the influence of this implicit philosophical assumption is evident in the justifications used for particular design paradigms. The most recent of these is object-orientation. The relevant literature argues that object-orientation has benefits over previous development approaches because it allows the developer to directly implement the concepts identified in the problem domain. Those design methods claim that requirements are elicited by identifying the phenomena that needs to be automated. Analysis techniques are then used to represent that phenomena as a collection of concepts and relationships. Those concepts can then be specified as objects and classes by identifying the essential properties and functionality. Some researchers even refer to the previously discussed issues from philosophy and psychology as further justification for that belief. For instance, the following references (Sowa 1983; Dillon and Tan 1993; Martin and Odell 1995; Bruegge and Dutoit 1999) all appeal to the concepts of intension and extension as justification for the assumption that concepts can be defined in terms of essential attributes. Consequently, object-orientation allows software developers to successfully implement our models of reality. However, that belief is based on the classical theory of understanding concepts and not on a thorough analysis of the relevant, contemporary explanations provided by those disciplines.

The discussion that began this chapter showed that the most recent theories of software development state that requirements should be captured in use-cases. Those use-cases specify snippets of functionality and are extracted from the different perspectives of the

many stakeholders in the development process. The use-cases are not developed by abstracting concepts from the observed reality, they are a combination of the observed reality and the existing conceptual apparatus of the stakeholder that is applied to that reality in order to understand it. Indeed, the use-cases presented by the stakeholders are subsets of the larger, encompassing theories that are used by that particular stakeholder to understand the entire phenomena that needs to be automated in software. Those encompassing theories or cognitive models could be different for different stakeholders.

The level of 'expertise' of the clients and customers of the problem domain may also be different to those of the analysts and software developers. Therefore, the concepts and theories developed to understand the same phenomena might be different. Although they may have the same label, the precise meaning of the concepts contained within those use-cases are dependent on their encompassing theories. Moreover, the use-cases are represented in natural language, which philosophers and psychologists have shown is already theory-laden.

Finally, the different people who participant in the requirements elicitation process can utilise different collections of concepts and theories to understand the same phenomena. Therefore, there is no guarantee the theories, and the smaller use-cases, used by the respective stakeholders to understand the phenomena are consistent with each other. Indeed, there is no guarantee that those explanatory theories used by the respective stakeholders are commensurable.

Software developers face the dilemma that they have to analyse the requirements presented by the different stakeholders, however, they can only analyse what those different stakeholders have described in the use-cases. They can only analyse what has been said or written in natural language and not necessarily what the stakeholders exactly meant. A number of issues become apparent:

- Different stakeholders can represent the same phenomena (a snippet of functionality that needs to be automated) using different use-cases.
- The same concept represented in different use-cases can have slightly different meanings in each context.
- Different concepts in different use-cases can refer to exactly the same phenomena.

- It may be exceedingly difficult to compare the precise meanings of different use-cases even though the concepts and relationships they describe appear to be similar.

These issues must be overcome during the analysis and design stages of the development process. Analysis seeks to amalgamate the use-cases into a single cohesive and consistent theory of the phenomena that needs to be automated in software. That theory is referred to as the analysis model or conceptual software architecture. The philosophical and psychological issues presented in this chapter provide insights into the fundamental nature of that analysis model, the factors that influence its creation and evolution, and the way the designer evaluates the effectiveness of the model as it is developed. It is noted that some of the issues discussed here may be construed as design issues rather than analysis issues. Furthermore, the following discussion on the design phase may also contain issues that some researchers may categorise as implementation issues. The aim is not to provide a hard distinction between analysis and design or design and implementation; rather it is to highlight the important issues and not how they should be categorised.

The analysis model must consist of a collection of concepts and relationships that, when implemented, realises the aspects of phenomena specified in the system requirements. In general, the understanding of the community outside the disciplines of philosophy and psychology assume those concepts and relationships can be inferred from observed phenomena and that, necessarily, they must also be evident in the natural language used to capture that phenomena in the requirements. However, contemporary theories in philosophy and psychology contradict this form of positivism and have shown that reality does not consist of easily identifiable parts. Rather, there may be many different but equally valid sets of concepts identified to understand the same phenomena. Those concepts and theories used to represent that phenomena are imposed by our intellects onto reality as a means of understanding the infinitely partitionable sensory experience. Moreover, the concepts and theories devised are not simply abstractions of that observed phenomena but are implicitly applied to sensory experience specifically to help understand the phenomena within the context of a particular problem solving activity. Therefore, many different analysis models can successfully represent the elicited requirements. Furthermore, those models can consist of many different collections of

concepts and relationships, and those concepts and relationships can exist at many different levels of generality.

The analysis of the different object-oriented designs for the cruise control systems detailed in chapter 2 of this thesis highlights the situation. The table comparing the identified 'objects' in those designs is reproduced here. Each of the seven object-oriented designs identifies a different collection of concepts used to develop a model that represents the problem to be solved. Moreover, that collection does not even consider the other software designs that utilised different design paradigms. Each of the models represents an understanding of the cruise control problem, however the precise meaning of the concepts used in each one is specific to the context of the model they appear in.

Design Example	Objects Identified.
Booch	Driver, Brake, Engine, Clock, Wheel, Current speed, Desired speed, Throttle, Accelerator. (9)
Yin & Tanik	Driver, Brake, Engine, Clock, Wheel, Cruise control system, Throttle, Accelerator. (8)
Birchenough	Driver, Wheels, Accelerator. (3)
Gomaa (JSD)	Cruise control, Calibration, Drive shaft. (3)
Wasserman	Cruise controller, Engine monitor, Cruise monitor, Brake pedal monitor, Engine events, Cruise events, Brake events, Speed, Throttle actuator, Drive shaft sensor. (10)
Appelbe & Abowd	Driver, Brake, Engine, Clock, Wheel, Cruise controller, Throttle. (7)
Gomaa (Booch OO)	Brake, Engine, Cruise control input, Cruise control, Desired speed, Throttle, Current speed, Distance, Calibration input, Calibration constant, Shaft, Shaft Count. (12)

Table 5-1: Cruise Control 'Objects' (from Chapter 2)

The ability to develop models at different levels of generality is also highlighted in the other cruise control examples. In addition to the object-oriented designs, three researchers developed models using the notion of feedback control systems to identify the appropriate concepts and relationships. The concepts identified by Higgins and Shaw are represented

in the following table (Table 5-2). Both developers initially created a model of the system that consisted of generic, feedback control concepts. Those concepts were then replaced by concepts that are specific to the particular problem domain – cruise control systems.

Design Example	Objects Identified.
Higgins (generic)	Actuating entity, Reference input, Summing point, Control action, Controller, Control signal, Disturbing entity, Disturbance, Controlled system, Controlled output, Feedback elements, Primary feedback signal.
Higgins (specific)	Driver, New desired speed, Set speed Summing point, Desired speed, Set throttle pressure summing point, Throttle pressure, Throttle, Power, Environment, Speed gain/loss, Car, Speed, Speed sensor, Measured speed
Higgins (complex)	Car on summing point, Car on signal, Cruise control on summing pt., Cruise control on signal, CC active summing pt., CC active signal, Set speed summing pt., Desired speed, Set throttle pressure summing pt., Throttle pressure, Throttle, Power, Environment, Speed gain/loss, Car, Driver, Press brake, Press accelerator, Speed, speed sensor, Measured speed, New desired speed, Brake/Accelerator sensor.
Shaw (generic)	Set point, Controller, Input variable, Process, Change to manipulated variable, Controlled variable.
Shaw (specific)	Activate/Inactivate switch, Controller, Desired speed, Throttle setting, Engine, Wheel rotation, Pulses from wheel.

Table 5-2: Generic and Specific Cruise Control Concepts

Both of those designs represent the same problem, however, they identify different concepts to the other object-oriented designs. Higgins goes further and provides a more complex design based on a more sophisticated generic model of feedback systems. Again, this model is a valid representation of the problem – the model is just more complex. The original, generic feedback designs also provide valid models of the cruise control problem. They represent the same reality as the specific models, they are just at a different level of abstraction. To use the terminology provided by Rosch, the object-

oriented designs represent the basic-level categories while the generic feedback models consist of superordinate-level categories. While some people may conceive of the cruise control problem in terms of basic-level categories, it is equally valid that someone with expertise in the domain of feedback control will conceptualise the same reality in terms of the superordinate categories. In Chapter 2, the design reasoning of Jones, who was trained as an electronic engineer, illustrates that different way of understanding.

The preceding philosophical and psychological foundations show that none of these different models is a better match with reality than the others. The only characteristic that can be used to differentiate between them is the 'usefulness' of each model for solving the exact requirements of the problem.

The foundational issues identified also state that the collection of concepts and relationships that constitute the analysis models must be constrained by logical consistency and coherence. The field of human anatomy provides an interesting illustrative example because the human body is a large and incredibly complex system, and it is something we all possess an example of. The two most prominent ways of modelling the gross organisational structure of the anatomical system is in terms of regional topography and functional systems. Table 5-3 contrasts the structural arrangement between two popular texts on anatomy and physiology.

The editorial board of *Gray's Anatomy* detail the rationale for the choice of organisational structure:

> "*Gray's Anatomy* was founded on the principle that to understand the body's construction it is necessary to analyze it in terms of its component systems as well as its regional topography. … Of course, this arrangement is to some extent an artificial separation of what in the body are intimately interdependent components, both during development and in the mature body. It is obvious that whilst there are indeed many clinical conditions where dysfunction of a particular system occurs, there are many others in which topographical nearness of different systems is the prime consideration. ... Clearly what is needed is both a systematic account and a regional, topographical one. … This would require much more than a single volume." (Gray et al 1995)

Anatomy, regional and applied (Last 1978)	*Gray's Anatomy* (Gray et al 1995)
Discusses the smallest 'components' larger than cells. Skin, muscles, tendons, bones, joints, mucous membranes, serous membranes, blood vessels, lymphatics. It also discusses the nervous system. It then partitions the body into: Upper Limb, Lower Limb, Thorax, Abdomen, Head & Neck, and Central Nervous System.	Partitions the body into the following major components: Cells & Tissues, Integumental System[31], Skeletal System, Muscle, Nervous System, Haemolymphoid System[32], Cardiovascular System, Respiratory System, Alimentary System[33], Urinary System, Reproductive System, Endochrine System[34], and Surface Anatomy.

Table 5-3: Contrasting Anatomical Models

If the human anatomy was to be implemented as a software system, both of these structural arrangements would result in different analysis models or conceptual architectures in which the concepts constitute a coherent and consistent system. As Gray states, the most appropriate conceptual model would be dependent on its intended function. In contrast, a conceptual model that consisted of lower limbs, thorax, respiratory system, skeletal system, and alimentary system would not be logically consistent. The different large-scale concepts overlap in function due to varying levels of generality. Moreover, it can immediately be seen that many functions would be impossible to implement due to our intimate knowledge of what bodies do. For instance, which concept contains the implementation of the femur (thigh bone), the 'lower limb' or the 'skeletal system'? How could this body 'see' anything without any concept implementing a pair of eyes? The consistency of the conceptual model of this fictitious body is obviously flawed; our detailed knowledge of the body's functionality and small-scale componentry make it obvious. However, how is it possible to detect analogous, logical, inconsistencies in the conceptual models of systems in domains in which we do not possess such intimate knowledge? The only method of assurance is a constant process of validation of the

[31] Skin and its derivatives: hair, nails, glands, etc.

[32] Blood and its derivatives: red blood cells, bone marrow, hemoglobin, etc.

[33] Food consumption and processing.

model as its design proceeds. As Popper noted, the process consists of a continuous application of conjectures and refutations until a model is developed that can not be falsified. The knowledge required to identify inconsistencies in a model is dependent on the purpose of the model. However, that knowledge is not always immediately obvious. To repeat the quote used earlier,

> "To understand a problem means to understand its difficulties; and to understand its difficulties means to understand why it is not easily soluble – why the more obvious solutions do not work. We produce the obvious solution and then criticize them, in order to find out why they do not work. In this way, we become acquainted with the problem, and may proceed from bad solutions to better ones – provided always that we have the creative ability to produce new guesses, and more new guesses. … If we have been working on a problem long enough, and intensively enough, we begin to know it, to understand it, in the sense that we know what kind of guess or conjecture or hypothesis will not do at all, because it simply misses the point of the problem, and what kind of requirements would have to be met by any serious attempt to solve it. We begin to see the ramifications of the problem, its subproblems, and its connections with other problems." (Popper 1979d)

This issue highlights immediate questions for software engineering research. For instance, the analysis model of the system can also be referred to as the system's logical or conceptual architecture. Research suggests that model, or architecture, should be created relatively early in the development process and it then sets the path for subsequent steps in that process. However, philosophy of science suggests that while developers may possess the knowledge required to validate that model early on in the development process, they may not possess the knowledge required to successfully falsify it. Moreover, it may not be until the development process is well into the design, or even the implementation stage, that that knowledge is generated by the developers. This would appear to supply some credence to software engineering researchers who claim the architecture of a software system cannot always be determined in the earliest stages of the design process as theory suggests, and that there are cases where it need not be. (Reed 1987).

---

[34] Regulation of internal functions.

A second issue that makes it difficult for developers to refute the proposed analysis model concerns whose knowledge has been used to develop the analysis model. Conceptual relativism suggests that while there is some common sense realism in that we all experience the same reality as sensory inputs, the conceptual arrangements we use to understand that sense data depends on our previous experiences and level of expertise. The precise meaning of the concepts and theories developed to understand and solve a particular problem are subjective to the person understanding it. However, the analysis model is derived from the requirements use-cases and they are generated by a number of different stakeholders in the development process. In many situations, the clients and users of the system, who help to generate the use-cases, have a greater level of expertise or knowledge in the particular problem domain than the system analysts and developers. Therefore, the precise meanings of the concepts and relationships specified by the clients in the development of use-cases would be different to those of the developers. The use-cases represent snippets of the client's model of how the problem domain operates, while the analysis model is a product of the developer's understanding of the client's model. As Popper states, it may not be until the developers have worked on the problem for a very long time, perhaps until the system implementation, that they fully 'understand' what the clients had intended. The fact that use-cases are represented in natural language serves to exacerbate the problem. Following from the previous arguments of Swartz about intension and extension, the majority of software development is performed in problem domains that consist of concepts that can not be precisely defined. Therefore, it would be easy for clients and software developers to have different understandings of the same labelled concept because of the spectrum of its ambiguity.

The process of design aims to transform the conceptual model developed during the analysis stage into a collection of concepts and relationships that are implementable in software. The theories presented from philosophy and psychology uncover foundational issues that concern differences between the concepts used in analysis and those used in design. They also uncover issues concerning the influence of design criteria on the preceding analysis process, and how the evolving design model can be evaluated.

To implement the concepts and relationships present in the analysis model, the designers can only utilise the constructs provided by the implementation medium of software development. That implementation medium consists of programming language constructs, operating system services, hardware execution constructs, external 'off-the-

shelf' software components, and the virtual machine that executes the resulting software/hardware implementation to realise the solution to the problem. Different programming languages offer a different variety of implementation constructs. Northrop (Northrop and Richardson 1991) has classified them into the following design categories: function-orientated design, data flow-orientated design, data structure-orientated design, object-based design, and object-orientated design.

Object-oriented languages are claimed to have advantages over other programming languages because they allow the implementation of the 'concepts we perceive' by encapsulating data and functionality into a single implementation construct. Object-oriented design proceeds by identifying and specifying the properties and functionality of the concepts identified in the analysis model. However, the 'refinement' of previously identified concepts occurs within the influence of other constraints placed on the developers. Those constraints include:

- The partitioning of the solution into major components and their means of communication – the system architecture.
- The stipulation of interfaces for those components to specify the exact nature of component interaction.
- The specification of control flow to stipulate how the system will be executed by the machine to realise the solution.
- The consideration of non-functional requirements such as system performance, maintainability, and modifiability.
- The desire to utilise previously existing software components.

Jacobson et al (Jacobson, Booch et al. 1998) provide a comparison of the differences between the analysis and design models:

Analysis Model	Design Model
Conceptual model, because it is an abstraction of the system and avoids implementation issues	Physical model, because it is a blueprint of the implementation
Design-generic (applicable to several designs)	Not generic, but specific for an implementation
Three (conceptual) stereotypes on classes: <<control>>, <<entity>>, and <<boundary>>	Any number of physical stereotypes on classes, depending on the implementation language
Less formal	More formal
Less expensive to develop (1:5 ratio to design)	More expensive to develop (5:1 ratio to analysis)
Few layers	Many layers
Dynamic (but not much focus on sequence)	Dynamic (much focus on sequence)
Outlines the design of the system, including its architecture	Manifests the design of the system, including its architecture (one of its views)
Primarily created by "leg work", in workshops and the like	Primarily created by "visual programming" in round-trip engineering environments; the design model is "round-trip engineered" with the implementation model
May not be maintained throughout the complete software lifecycle	Should be maintained throughout the complete software lifecycle
Defines a structure that is an essential input to shaping the system – including creating the design model	Shapes the system while trying to preserve the structure defined by the analysis model as much as possible.

Table 5-4: Comparison of the Analysis Model and the Design Model (from (Jacobson, Booch et al. 1998) p. 219)

While there are specific differences between the two it is assumed that the design model is based, in part, on a refinement of what exists in the conceptual model. However, the issues of philosophy and psychology show there are more significant differences than those presented in conventional object-oriented design literature.

Analysis Model	Design Model
Concepts and relationships can not be precisely defined by intension.	Concepts and relationships must be defined by essential features and specific functionality
The precise meaning of concepts and relationships is dependent on the context of the theory in which they are contained	The precise meaning of concepts and relationships, their definitions, are independent of the system in which they are implemented.
Concepts and relationships are constrained only by the previous experience and imaginative ability of the stakeholders in the development process	Concepts and relationships are constrained by the constructs provided by the implementation medium and the execution model of the virtual machine that executes it.

Table 5-5: Comparison of the Analysis Model and Design Model based on the Philosophical and Psychological Foundations

One of the claimed advantages of object-oriented development is that developers can use objects in a uniform modelling approach throughout the development process (Kaindl 1999). That belief is based on the classical theory of conceptual understanding, which states that all concepts can be specified in terms of essential features or intensional definitions. As philosophers and psychologists have noted, the classical theory of categorisation has proved too simplistic to explain the human thought process and has now been superseded by more sophisticated theories. The foundations show that the components of the analysis and design models, though the same label may be used to refer to them, represent inherently different things. This explains why the transition from object-oriented analysis to design is not as easy as suggested by object-oriented design methods. Those methods suggests the transition is smooth and easy, in practice it has been shown to be quite difficult (Kaindl 1999). This also explains why researchers are beginning to question the assumption that object-orientated development is advantageous

because it allows developers to more easily implement their model of reality (see for example (Hatton 1998)).

As stated previously, the problem with the classical theory of definition is that concepts are defined in terms of attributes, which themselves have to be defined. The result is an infinite regress of definitions. That problem is not faced by design model concepts in software development because they have been built by aggregating, encapsulating, and abstracting the constructs provided by the implementation medium. Analysis model concepts can not be defined because there exists no axiomatic level of definition in our conceptual apparatus. In contrast, the concepts used in the design and implementation models of software are built on top of the axiomatic definitions of the Von Neumann computer architecture. Progress in software development has produced abstractions that allow developers to design and implement above that axiomatic level. Moreover, the progress from machine code languages to assembler level languages, and then data flow-orientated design, data structure-orientated design, function-orientated design, object-based design, and object-orientated design has made it appear as though developers can now analyse, design, and implement systems using notations that closely match our models of reality. That justification for progress in software design methods and languages has been based on the classical theory of concept definition. Research in the fields of philosophy and psychology has shown that view is too simplistic.

Dreyfus has noted this same phenomenon in his critique of artificial intelligence research (Dreyfus 1992). Artificial intelligence attempts to formalise intelligent activity by transforming it into a set of computer instructions. He shows this is based on the ontological assumption that explicit facts exist in the world and they can be formalised in the context-free environment of computer software. That assumption is similar to the one implicitly made by software engineers and, as has been suggested by philosophers and psychologists, is also made by most communities, both scientific and non-scientific, who seek to understand human thought processes. Dreyfus quotes Chomsky (from *Language and Mind*) to note the predisposition of researchers to use simplistic examples to justify the belief in the classical theory of understanding[35].

[35] Other software engineering researches that exemplify these foundational issues are presented in the next chapter.

> "There has been a natural but unfortunate tendency to 'extrapolate', from a thimbleful of knowledge that has been obtained in careful experimental work and rigorous data-processing, to issues of much wider significance and of greater social concern." ((Dreyfus 1992) p. 79)

The previous discussion of the analysis model considered the effects of the philosophical and psychological foundations on how that model is evaluated during development. Those effects have even more ramifications during the design and implementation stage. If the developers don't develop the knowledge required to falsify the model until the design or implementation phases, and a falsifying example then appears, does the model need to be replaced? It may be that the originally conceived collection of concepts and relationships appeared adequate to satisfy the required properties in the implementation. However, a new situation may present itself during the implementation stage that was not previously considered. Similarly, the requirements may be modified to consider a new situation that was not previously required. Does the model need to be substantially modified or can the required properties be implemented within the previously existing concepts? Can the conceptual integrity of the model be 'fudged' to ensure the designed model continues to satisfy the requirements? These are all questions for future research.

5.6 Conclusion

Software engineering research would like to improve the development of software systems by using approaches similar to those of traditional engineering disciplines. The evolution of those other engineering approaches has been based, at least in part, on the underlying principles of the systems those disciplines design and build, and the materials and components used to build them. The task for software engineering research is to identify the underlying properties of software systems and determine if they can lead to an analogous engineering approach for software development. Those underlying principles are based on the notions of concepts, abstractions, theories, and models. The disciplines of both philosophy and psychology have a long history of studying those principles and this chapter has examined the relevant research in those areas to identify the foundational principles of software engineering. Unfortunately, there are no pre-packaged collection of theories in the history of philosophy or psychology that explain all of the issues involved when developing conceptual models in the manner required in

software engineering. Not only does a single answer not exist but there are many potential answers available in the literature of those fields.

This study has identified two broad ways of understanding the underlying principles of software systems that appear to exist in both philosophy and psychology. The first, the classical way of understanding, assumes a positivist approach where people are separate from the world of sensory experience and all people experience the same reality. As people operate in the world, they generate knowledge by identifying concepts and categorising them in terms of the essential attributes they exhibit. Moreover, because the world exists separate from our knowledge of it and all people experience the same reality, that knowledge captures the world as it really is. With the classical way of understanding, software components – concepts and theories – can be understood in an analogous way as traditionally engineered components. Those traditional engineering components are described in terms of particular properties, and mathematical idealisations of their behaviour can be developed based on those underlying properties. Therefore, the classical way of understanding would provide a philosophical foundation for software engineering research based on the artefact engineering view of software development.

That classical way of understanding has been the dominant way of understanding the underlying principles of software systems until very recently. Furthermore, it has been noted that the view still dominates the guiding philosophy of most people outside the relevant areas of research in philosophy and psychology. Nevertheless, contemporary research has identified many anomalies with that classical view and has developed an alternative, more sophisticated way of understanding based on the subjective nature of our interaction with the world. In contrast with the positivist approach, it is suggested that people cannot consider their knowledge of the world as separate from the world itself. People do not observe the pre-existing parts of the world and categorise them based on essential attributes. Instead, as people interact with the world, they apply explanatory theories that help them understand and solve the problems at hand. Those theories are automatically and subconsciously applied to reality so that what is conceptualised is determined, not only by what is there, but by how we have been trained to understand it. Consequently, all observation is theory-laden. Those theories are passed on through language and cultural conventions and are modified to become more sophisticated as our experience in a particular domain grows. Concepts do not identify the pre-existing parts of the world and they cannot be universally defined in terms of essential attributes.

Concepts play roles in our innate theories of understanding and their meanings are specific to the theory in which they play that role and our level of experience in using those concepts and theories.

This new way of understanding the underlying principles of software systems contradicts the classical way, and consequently, contradicts the artefact engineering view of software development that pervades software engineering research. Alternatively, that new way of understanding provides a philosophical foundation for software engineering research that supports a model building view of software development.

That foundation does not dismiss the goal of developing an engineering approach to software development. What is required is research that explores engineering techniques based on the philosophical foundations of model building rather than artefact engineering. Moreover, this chapter has not attempted to provide a definitive summary of all the relevant issues. That would be a separate thesis in itself. Rather, it has tried to show that the issues are important and significantly affect our understanding of what software engineering is all about. Hopefully, it will provide enough evidence to start the debate that, over a period of time and perhaps in conjunction with researchers in those other fields, will determine the theories necessary to develop an engineering approach to software development.

6. Evaluating Software Engineering Research

6.1 Introduction

Despite 30 years of research, there exists a clear dichotomy between the practice of software development and the approaches suggested by software engineering theory. Our discipline is littered with examples of development ideas that appeared to offer great hope and dominated research agendas but soon fell by the wayside. The problem is that because there has been no understanding of the underlying principles of software systems and their development, there has been no basis for evaluating those software engineering ideas. Although empirical studies have increasingly become part of the field in the last few years, software engineering research can be described as the science of non-reproduced results[36].

While it is difficult to label any single, encompassing belief as the 'conventional view' of software engineering, the use of analogies with traditional engineering disciplines is a long and established tradition that promotes the artefact engineering view as the most popular means of driving research agendas. In the absence of a widely accepted understanding of software based on underlying principles, understanding has had to rely on analogies with 'built artefact' disciplines. Therefore, when software engineering ideas are proposed to improve development practices, they appear to be plausible because our understanding is already implicitly based upon them. Moreover, as a result of the demand for more and more applications, we are now desperately in search of more efficient ways of building systems. Ross predicted the inherent danger of this situation at the original NATO conference.

> "Ross: My main worry is in fact that somebody in a position of power will recognize this crisis … and believe someone who claims to have a breakthrough, an easy solution. The problem will take a lot of hard work to solve. There is no worse word than 'breakthrough' in discussing possible solutions." (NATO 1976a) (p. 81)

[36] Reed made this observation with respect to Computer Science in 1991 (Reed 1991) though, historically, it applies equally well to software engineering.

The history of our discipline shows that Ross was quite perspicacious. There has been a virtual myriad of 'silver bullets', each touted as "…a breakthrough, an easy solution". Nevertheless, despite their apparent plausibility, those ideas either do not work in practice or they are simply not adopted. The reason suggested by this thesis is that our existing understanding of software engineering – the artefact engineering view – does not match the way people actually build software.

Chapters 2, 3, and 4 presented the history and application of that understanding and analysed it in detail revealing many anomalies. Chapter 5 then provided a foundation for software engineering based on the underlying principles of software systems and the cognitive processes required to develop them – the model building view of software development. That view provides the framework with which it becomes possible to evaluate and justify past, present, and future software engineering research ideas. A primary goal of that research should now be to continue the exploration of those underlying principles so that ideas for the improvement of software development can be evaluated without relying solely on the analogies that have dominated our field so far.

Amongst the barriers to real progress in our field is the lack of understanding of the way research-based disciplines progress. One of the arguments put to justify the current state of software engineering is that it is young, and that other bodies of science and engineering have existed for hundreds, and in some cases, thousands of years. Moreover, the history of science has revealed the many mistakes and periods of slow progress made by those disciplines. Therefore, some argue why should we be concerned at our current relative position on the path of progress? However, if we can understand the way in which those other (scientific and engineering) disciplines have evolved, and can identify the nature and causes of those things which prevented progress, then perhaps we can avoid them and accelerate our own improvement.

That understanding is developed in this chapter by presenting the relevant work from the history and philosophy of science. That work details the way in which research in a particular discipline, including the formulation and evaluation of empirical studies, is guided by underlying assumptions that can change over time. Those changes however, lead to vastly different collections of theories to explain the phenomena under investigation. Unfortunately, it is difficult to compare alternative theories that are based on different guiding assumptions. Moreover, the ideas presented show that when research

based on a new set of guiding assumptions is proposed, it is often difficult for researchers to change from their established way of thinking.

The conjecture of this thesis is that this is what is happening in software engineering research. The artefact engineering view has been the established and dominant guiding assumption in software engineering research. However, a better understanding of the underlying principles of software systems and their development can lead to an improved way of understanding software engineering, that is, the model building view.

In addition, examples are presented to show that the model building view has already provided useful contributions to software engineering research. The foundations elucidated in the previous chapter and the philosophy of science ideas of this chapter have been used to build new ways of understanding important aspects of software development. They range from Ambler et al's approach to understanding the effects of different programming paradigms; to more general descriptions of how these issues impact the entire software lifecycle (e.g., Dahlbom); and finally to specific theories of software engineering provided by Naur and Blum. These researches present interesting and progressive ways of thinking about software development but they certainly have not led to mainstream research and design practices. However, they can now be re-evaluated in a new light.

6.2 The Progression of Research-Based Disciplines

The history and philosophy of science has developed many theories that explain the progress of research-based disciplines. The most popular of those is the work of Thomas Kuhn, and his ideas are often cited in software engineering research. However, his ideas are certainly not the only ones, nor are they the most recent. This section aims to present a brief, though comprehensive, account of those explanations to develop an understanding of how research in general is justified and evaluated and, hence, how software engineering research can be improved. The exposition begins with the relevant aspects of the accounts provided by logical positivism and Karl Popper. The work of Kuhn is then presented in detail and that is followed by the alternate explanations provided by Feyerabend, Lakatos, and Laudan. While that amount of detail may appear excessive, researchers often cite some of the more popularly known though controversial aspects of those theories. Therefore, an attempt has been made to provide enough information to avoid that problem, which has been previously identified by Bechtel.

> "Most scientists, however, simply adopt a philosophy of science that is popular, or that suits their purposes, and cite it as authority. This proclivity to borrow positions from philosophy is rather common but poses serious dangers because what might be controversial in philosophy may be accepted by a particular scientist or group of scientists without recognizing its controversial character." (Bechtel 1988b)

Logical positivism, as detailed in the previous chapter, became the dominant explanation for the progression of scientific disciplines in the 20th century. One of its main tenets was that nature had an underlying logical order that could be captured in rigorously specified explanatory theories. Hypotheses devised to explain real world events and phenomena were verified by deducing them from existing, axiomatic 'laws of nature', using known facts as initial conditions, and through experimentation to ensure they successfully predicted natural phenomena. Because those theories, or laws, captured the logical nature of reality, there could be no alternative paradigms for explanation. Therefore, according to the logical positivists, disciplines progressed in a cumulative manner by discovering the laws of the objective reality that was experienced by all. Although the central tenets of logical positivism have been subsequently refuted and alternative explanations for the progress of scientific disciplines have been proposed, researchers note that they still exert a powerful influence on the way people, especially those not familiar with the history and philosophy of science, believe disciplines progress (Bechtel 1988b).

Popper provides the first major objection to the positivist model of scientific progress. As the previous chapter detailed, his attacks began with the beliefs that induction could be used to develop hypotheses and that experimentation could be used to verify them as laws of nature. His theories about 'conjectures and refutations' show it is impossible to prove that scientific theories match a logical order of reality. The best that can be achieved are theories that provide useful explanations of the observed phenomena. The goal of science is to develop theories and then attempt to falsify them. Theories are only valid in the sense that they were yet to be falsified. As detailed in the previous chapter, one of the important features of Popper's theories, and earlier commentators such as Poincaré, is that observation is theory-laden. Popper attempts to account for the theory-ladeness of observation with his 3-world model of scientific knowledge. His conclusion is that theories are implicitly applied to the reality we experience in an attempt to understand it. Therefore, scientific progress is not the linear accumulation of theories that capture the

logical structure of the objective reality we all share. Rather, it is the development of improved theories, through conjectures and refutations, which continually home in on the truth. Progress is concerned with the development of theories that provide better explanations of reality. Moreover, while we cannot prove that we experience the same objective reality, Popper's model results in a world 3 that allows people to share the explanatory theories of that reality – that is, they share an objective knowledge of reality.

The previous chapter noted that philosophers after Popper argued he did not fully consider the effects of the theory-laden nature of observation. Their theories differed from Popper's by giving alternate descriptions for the way in which people identify concepts and theories and how they are used to explain phenomena. Those issues were then developed into more detailed models of the progression of scientific disciplines. Moreover, their models include a more thorough analysis of historical activity in those disciplines than previously considered.

Kuhn's *The Structure of Scientific Revolutions* (Kuhn 1962) provides a far different account of progress in a discipline than previously considered. Based on his review of historical accounts, Kuhn identifies five different stages in the progress of a scientific discipline: (1) immature science, (2) normal science, (3) crisis, (4) revolution, and (5) resolution. An immature science develops into a normal science as its initial theoretical foundations are laid and a set of theories is devised to explain and predict observed phenomena (Kuhn 1977c). It is the description of the remaining phases that set Kuhn apart from previous philosophies of science.

Normal science operates within a dominant paradigm that provides the underlying guiding assumptions that govern the way practitioners in a discipline understand the phenomena they seek to explain and provides a common conceptual framework that allows researchers to work on problems together. Philosophers still debate the exact definition of a Kuhnian paradigm. However, for the purposes of this discussion it can be explained as a collection of theories, methods, and standard example problems. Because of the theory-laden nature of observation, the paradigm sets the contextual framework that constrains a practitioner's view of the world to the concepts that fit within the framework. Moreover, researchers are often unaware of the guiding influence of those assumptions. For example, the physics paradigm of Newtonian mechanics provides a contextual framework that constrains the physicist to conceptualising the world in terms

of forces and masses. Theories devised within that paradigm can only be constructed in terms of the concepts allowed by the paradigm.

Within the stage of normal science, practitioners are indoctrinated into a paradigm during their education. Students are taught using textbooks that feature the paradigm-defining concepts, theories, and example problems. They are rarely exposed to the sort of research problems that may question the paradigm until they are well established in the discipline. Moreover, those textbook examples merely reinforce the relevant theories and omit the information about how that knowledge was acquired and about why it was accepted by the profession (Kuhn 1977b).

As practitioners work during a period of normal science they continue to do what they learned to do as students – imitate the exemplars they learned in school in new contexts (Bechtel 1988b). The examples and models provide analogies that practitioners can apply to other problems (Kuhn 1977d). The theories developed by practitioners are not the imaginative leaps of understanding suggested by Popper, which are conjectured and subsequently tested by refutation. Practitioners devise their theories based on a constraining framework and attempt to fit those theories to nature in order to explain it. However, those theories seldom fit nature precisely. Practitioners don't look for exact matches between experiment and theory. Instead, they look for reasonable agreement. During normal science those discrepancies between theoretical predictions and empirical observations are not taken to falsify the theory, but rather as creating further problems that scientists must solve. (Kuhn 1977b).

> "Closely examined, whether historically or in the contemporary laboratory, [normal science] seems an attempt to force nature into the preformed and relatively inflexible box that the paradigm supplies. No part of the aim of normal science is to call forth new phenomena; indeed those that will not fit in the box are often not seen at all. Nor do scientists normally aim to invent new theories, and they are often intolerant of those invented by others. Instead, normal-scientific research is directed to the articulation of those phenomena and theories that the paradigm already supplies." (Kuhn 1962) (p. 24).

The prevailing paradigm facilitates theoretical successes at the outset, however, Kuhn argues that eventually anomalies accrue and scientific discoveries of natural phenomena

are made that conflict with the established way of understanding. Eventually the discipline reaches a stage of 'crisis' and new fundamental theories are required. To explain these anomalies, alternative theories are suggested that go outside the contextual framework of the guiding paradigm but which offer their own promise of a new problem-solving tradition. As these new theories gather supporters, they compete to become the new guiding paradigm of the discipline. However, it is not easy to systematically compare new paradigms with each other or with the established paradigm. According to Kuhn, because of the theory-laden nature of observation, all observations, including the results of experiments, are reported in a theory or paradigm dependent manner. The problem is commonly referred to as the incommensurability of theories. Proponents of competing paradigms have to resort to non-rational or non-quantifiable means for advancing their claims, for example thought experiments (Kuhn 1977a). Their arguments are necessarily circular in that they can only use the concepts and theories allowed by the paradigm in that paradigm's defence (Kuhn 1962) (p. 94). Eventually a new paradigm emerges as the dominant one. However, because of the incommensurability of theories, practitioners cannot simply add the new paradigm to the existing one. A revolution is required in which the concepts and theories of the existing paradigm are replaced by the new one.

> "The transition from a paradigm in crisis to a new one from which a new tradition of normal science can emerge is far from a cumulative process, one achieved by an articulation or extension of the old paradigm. Rather it is a reconstruction of the field from new fundamentals, a reconstruction that changes some of the field's most elementary theoretical generalizations as well as many of its paradigm methods and applications." (Kuhn 1962) (p. 84).

The revolution does not guarantee that all practitioners will switch to the new way of conceptualising the research problems of the discipline. There exists an enormous inertia to switching between paradigms. Indeed, that is why Kuhn argues that paradigm revolutions cannot occur until the discipline has reached a stage of crisis that makes the practitioners open to new ideas.

> "As Kuhn describes the history of scientific development, it becomes quite apparent that scientists are on the whole pretty much like the rest of us, and are inclined to defending their preconceptions and commitments than to leaping off into the dark." (Abel 1981)

While many practitioners will eventually be coerced towards the new paradigm as a way of solving the anomalies of the prevailing crisis, Kuhn argues that some will continue to cling to their established ways of thinking.

> "The transfer of allegiance from paradigm to paradigm is a conversion experience that cannot be forced. Lifelong resistance, particularly from those whose productive careers have committed them to an older tradition of normal science, is not a violation of scientific research standards but an index to the nature of scientific research itself. The source of resistance is the assertion that the older paradigm will ultimately solve all its problems, that nature can be shoved into the box the paradigm provides… Conversions will occur a few at a time until, after the last holdouts have died, the whole profession will again be practising under a single, but now different paradigm." (Kuhn 1962) (pp. 151-152)

When the new paradigm gains ascendancy, the final stage, resolution, is achieved, and a new period of normal science occurs. The cycle is then repeated.

Kuhn's theories are quite popular and are often cited in software engineering research literature, especially as justification for some new idea or as reference to the often-used term – 'paradigm'. However, other philosophers of science have provided different or more sophisticated explanations for the progression of professional disciplines. The most radical of these is Feyerabend. His theories, also based on significant historical analysis, take the theory-laden nature of observation and the incommensurability of theories to more extreme consequences. He argues that because of the influence of the prevailing wisdom on the way practitioners of a discipline conceptualise the phenomena under investigation, it unduly biases their attempts to falsify it. The underlying governing assumptions help determine what is labelled as significant facts or data. Consequently, it may not be until a theory from an opposing paradigm or set of guiding assumptions is considered that data comes to light that can falsify an established theory.

Feyerabend believes a particular paradigm should never be allowed to dominate a discipline and that there should be no 'normal science' as Kuhn described it. Because practitioners become indoctrinated into a paradigm-specific view of their discipline, they unnecessarily reject alternative approaches. He argues that practitioners should entertain these alternatives before the crisis stage is reached. This is especially difficult as the

newer alternatives will not be as developed as the established paradigm and may contain errors and problems that make it easy for ardent supporters of the established paradigm to criticise them.

> "Science is a complex and heterogeneous *historical process* which contains vague and incoherent anticipations of future ideologies side by side with highly sophisticated theoretical systems and ancient and petrified forms of thought. Some of its elements are available in the form of neatly written statements while others are submerged and become known only by contrast, by comparison with new and unusual views." (Feyerabend 1979) (p. 146)

Feyerabend's views are often unfairly dismissed as being extreme. His most popularly cited book, *Against Method* (Feyerabend 1979), was deliberately written from a provocative perspective so that it could be contrasted with a companion volume to be written by his friend and fellow contemporary philosopher of science, Imre Lakatos. Unfortunately, Lakatos died before it could be written. Lakatos' work offers an account of scientific progress that provides the large-scale structure missing in Popper's theories while allowing more flexibility for theory enhancement than Kuhn's revolutions and without resorting to the methodological anarchy proposed by Feryerabend. Unlike Kuhn, Lakatos proposes that disciplines consist of competing paradigms rather than there being a single dominant one and that progress comes from within those paradigms rather than completely replacing one with another. He introduces the notion of a 'research programme', which contains the guiding assumptions for the discipline. A research programme consists of a 'hard core' set of assumptions and theories that must be accepted by the practitioners and cannot be modified. In addition, a 'protective belt' of auxiliary assumptions and theories surrounds the hard core. Progress consists of developing new theories in the protective belt to accommodate evidence that either has accumulated or is developed in the course of the research (Bechtel 1988b). Those developments are made in both the early stages of the discipline, to deal with unrealistic assumptions that need to be corrected, and in subsequent stages of the discipline to deal with anomalies detected by practitioners during their research (see 'Rationality, Historicist theories of' in (Zalta 1999)).

Lakatos argues that changes to the protective belt could either be 'progressive' or 'degenerative'. If the modifications provide explanations for the anomalies detected, and they continue to explain everything that the previous theories explained, and they allow

the research programme to make new predictions, then the change is progressive. In contrast, if the modification simply rearranges the existing assumptions to deal with the anomalies, but without resulting in any new predictions, then it is degenerative. Lakatos argues that disciplines could progress, go through extended periods of degeneration, and then progress again. As Bechtel points out, the nurturing of new and diverse paradigms is essential for the development of any discipline, even when they may, at first sight, be thought to be 'degenerative'. (Bechtel 1988b)

Laudan's explanation of scientific progress attempts to deal with problems he identified in the theories of Kuhn and Lakatos. Laudan's 'research tradition' is similar to the 'research programme' of Lakatos, however it is less rigid and does not consist of an immutable set of hard core theories. The content of his research tradition is a collection of common ontological assumptions about the nature of the world and methodological principles about how to revise theories and generate new ones. Laudan details two types of problems that must be overcome by a research tradition. The first are empirical problems such as experimental and observed anomalies that must be explained using theories within the common ontological assumptions. The second are conceptual problems that may manifest themselves as logical inconsistencies between two theories or accepted viewpoints within the research tradition.

Laudan also differs from Lakatos in how competing research traditions result in progress for the discipline. He argues that an established research tradition, which has solved the most problems, should be accepted as the most useful. However, an alternative tradition, which is currently solving problems at the fastest rate, should also be pursued. Unlike Lakatos, Laudan does not believe alternative traditions should be cumulative. It may be that an alternative tradition provides progress in an otherwise unsolved area at the expense of not being able explain previously solved problems. For Laudan, scientific progress is not about the research tradition that comes closest to the truth. Progress in a discipline can only be measured by the continual solving of problems.

These descriptions provide a glimpse of the many different theories proposed to explain the progress of scientific disciplines and research-based disciplines in general. At present, there is no single, universally agreed model that explains the progress of disciplines. This presents a problem for using these ideas to explain the progress of software engineering. Indeed, some researchers, as will be shown, have used this problem to argue that we should not use philosophy of science ideas at all in our own research. However, Laudan et

al (Laudan, Donovan et al. 1986) provide a comprehensive survey of the different philosophical models of theory change and note areas of substantial agreement between them. Those areas are summarised so they can be used to understand how software engineering research can be evaluated and justified.

The areas of agreement include:

1. The most important units of understanding scientific change are large-scale, relatively long-lived conceptual structures which different modellers refer to as 'paradigms', 'global theories', 'research programmes', or 'research traditions', and which, for neutrality, we term 'guiding assumptions'.
2. Guiding assumptions, once accepted, are rarely if ever abandoned simply because they face empirical difficulties. They tend to endure in spite of negative experimental or observational tests. In short, negative evidence is less important in the assessment of large-scale theories than is commonly thought…
3. Data do not fully conform to theory choice, i.e., observations and experiments do not provide a sufficient base for unambiguous choices between sets of guiding assumptions or between rival theories.
4. Metaphysical, theological, and other non-scientific factors play an important role in the assessment of scientific theories and guiding assumptions. Assessment is more than just a matter of the relationship between the guiding assumptions or theory and the evidence.
5. Assessments of guiding assumptions depend as much upon judgements about their potential as on their record of performance, and the former is not reducible to the latter.
6. Scientists do not make absolute judgements about the merits or demerits of a particular set of assumptions or a particular theory, but comparative judgements about extant rivals.
7. There are no neutral observations in science; rather they are all theory-laden, although not necessarily laden with the theories whose competition that arbitrate.
8. The generation of new, and the modification of existing scientific theories is not a random process; rather in most cases it takes place with respect to a heuristic or set of guidelines.

9. Guiding assumptions are never abandoned unless there is a new set available to replace them.
10. The coexistence of rival sets of guiding assumptions in a science is the rule rather than the exception. Debate about rival sets of assumptions does not alternate with periods of universal assent to one set, but occur constantly.
11. New sets of guiding assumptions are not judged by the same yardstick as well-established sets.
12. A later set of guiding assumptions seldom accommodates all the explanatory successes of its predecessors. There are losses as well as gains in the replacement process.
13. The technical machinery of confirmation theory and inductive logic has little if any light to shed on theory appraisal.
14. The assessment of low-level scientific theories is based in part on the success of the guiding assumptions with which they are associated.
15. The solutions given to problems by a scientific theory are often recognised as approximate only when that theory has been replaced by a new theory.

Laudan et al (Laudan, Donovan et al. 1986) also summarise the areas where the major philosophies of scientific progress disagreed though they are not included here.

6.3 New Guiding Assumptions for Software Engineering: The Model Building View

The use of traditional engineering disciplines as a source of ideas for the improvement of software development has provided a set of guiding assumptions that has allowed software engineering researchers to produce many useful theories since the NATO conferences of the late 60s. However, previous chapters have explored in detail the nature of the artefact engineering view of software development and identified many anomalies in it. Subsequent chapters explored the nature of concepts, theories, and abstractions using research from epistemology, metaphysics, psychology and the history and philosophy of science. That analysis suggests theories based on a model or theory building view are another useful source of guiding assumptions for improving software development. Research ideas based on the model building view already have a history in software engineering research, though they are hardly ‘dominant’. Much of that research

incorporates these ideas in specific aspects of software development. However, other work, specifically that of Naur and Blum, uses it to detail a new way of thinking about software engineering in general. This section details some of those ideas and suggests they should be pursued more explicitly in future research.

It should be noted however, that this is not meant as a thorough analysis of their work. Nor does it suggest that particular aspects of their theories are entirely correct or that they are first to make them. For example, Blum's definition of 'software engineering' is hardly the most thorough or useful, nor has he been the first person to criticise the quest for a 'silver bullet'. What the presentation seeks to achieve is to make evident the fact that software engineering research based on a different set of guiding assumptions to the conventional, artefact engineering view already exists. Moreover, that research may need to be re-evaluated with respect to a more sophisticated way of understanding the way in which research-based disciplines evaluate their work, and therefore, progress as a whole. The detailed analysis of that research would be the subject of future work.

6.3.1 Applying the Model Building View to Specific Aspects of Software Development

6.3.1.1 The Influence of Programming Language Paradigms

Ambler, Burnett, and Zimmerman detail the way different programming language paradigms guide the problem solving process of software development by providing a framework of conceptual structures for expressing the solution (Ambler, Burnett et al. 1992). While they do not explain paradigms in terms of Kuhn or other philosophers, their descriptions show their concept of programming paradigms is certainly equivalent to the way they are used by philosophers.

> "A programming paradigm is a collection of conceptual patterns that together mould the design process and ultimately determine a program's structure. Such conceptual patterns structure thought in that they determine the form of valid programs. They control how we think about and formulate solutions, and even whether we arrive at solutions at all.
>
> Once we can visualize a solution via a paradigm's conceptual patterns, we must express it within a programming language. For this process to be effective, the language's features must adequately reflect the paradigm's conceptual patterns… In practice, a language that supports a paradigm well is

> often hard to distinguish from the paradigm itself." (Ambler, Burnett et al. 1992)

The philosophical theories presented in the previous chapter noted that all observation is theory-laden and that there are many different, yet useful ways to conceptualise the phenomena under investigation. Ambler et al observe that different programming language paradigms help to constrain the types of concepts and relationships the developer implicitly applies to the problem domain to those that can be implemented in a particular style of programming language. That is, while many design methodologies are programming language independent, it is often easier to implement the results of a particular design method using a particular programming language.

They classify programming languages into three categories that together capture the continuum of approaches to software systems implementation.

- Operational Paradigms: The operational approach encompasses languages that explicitly define the sequence of step-by-step instructions required to construct a solution. This paradigm captures many different types of languages. These include imperative or procedural languages that capture, in an abstract model, the structure and operations of a Von Neumann-style machine architecture. These languages allow the specification of data variables and how they should be manipulated, step-by-step, during system execution. The operational paradigm also includes object-oriented languages, which capture the same model but encapsulate the functionality and data within a single structure. Also included are functional languages, such as lisp and scheme, which specifies a systematic approach based on a mathematical model of functional composition. These languages need to specify the step-by-step approach to implementation but using an abstract model of a virtual machine implemented above the traditional Von Neumann architecture.

- Definitional Paradigms: In the definitional paradigm there is no step-by-step description of how to execute the solution. The programming languages construct solutions by stating facts, rules, constraints, equations, transformations, and other properties about the solution value set. From this information, the system must derive a scheme, including an evaluation ordering, for computing a solution. These languages include rule-based languages that rely on inference engines,

transformational approaches, logic-based languages, and constraint-based programming languages.

- Demonstrational Paradigms: Programming by demonstration or by example, neither specifies operationally how to compute a value nor set constrains in the solution value set. Rather, they demonstrate solutions to specific instances of similar problems and let the system generalise an operational solution from the demonstrations. These languages include many visual or iconic programming languages.

6.3.1.2 The Philosophy of the Software System

Lawson also exemplifies some of the aspects of the model building view in his *Philosophies for Engineering Computer-Based Systems* (Lawson 1990). The initial description of his thesis is similar to the issues discussed by Ambler et al.

> "Software engineering methods and tools are important, but they should be the result of a well-developed philosophy for solving the application problem.
>
> By philosophy I mean a unifying common view of how a problem or a class of problems shall 'in principle' by treated. The view, which is based on concepts, must be commonly held by all project team members and all other parties with vested interests. It involves the development of a strategy from which decisions (large and small) emanate." (Lawson 1990)

As an example, Lawson details the design rationale behind the development of the Simula programming language. He notes that it's philosophy was aimed at solving a particular class of problems and its users found it useful for a much wider class of programming problems. During his subsequent analysis of 'philosophies' Lawson discusses 'philosophical decay', citing the OS/360 operating system project as an example. During that project, the initial philosophy of the solution degenerated as new features were added, time pressures intensified, and additional people were added to the project. Lawson's conclusion was that "philosophies, once established, must be nurtured and treated with respect; otherwise, they deteriorate" (Lawson 1990). This is consistent with what Brooks concludes from his experiences in the OS/360 project when he discussed the integrity of the conceptual model of the proposed design. Brooks notes that the conceptual model dictates the eventual solution and for it to remain cohesive, it should

reflect a single philosophy and flow from as few minds as possible (see chapter 4 in (Brooks 1975)).

Lawson uses the term 'philosophy for software engineering' in two different contexts. In the first context, he talks about the philosophy of the programming language. This is equivalent to how Ambler et al describe the influence of programming language paradigms on software development – they constrains the framework from which concepts and relationships can be used to synthesise solutions that can be implemented. However, in the second context, he discusses the integrity of the 'philosophy' as the ability to maintain those concepts and relationships of the conceptual model as they are refined and implemented. This use of 'philosophy' refers to the logical consistency and cohesiveness of the conceptual model devised to solve the problem. As the previous chapter noted, during design that model is transformed from a collection of analysis level concepts to a set of design level concepts. That process is more than just a refinement, those collections of concepts are fundamentally different. Therefore, when the transformation takes place the logical consistency and cohesiveness of the initial model of the solution may be lost or compromised. Moreover, as problems are identified during the system design and implementation, fixes may be introduced that degrade the consistency and coherence of that model. That is what Lawson means by retaining the integrity of the system philosophy.

Both of Lawson's uses of the term 'philosophy' are valid, however his argument, in light of the description of philosophical ideas presented in previous chapters, appears confused. Nevertheless, his paper highlights the usefulness of the philosophical ideas to software engineering. It also highlights the difficulty of using them without a thorough understanding of how they relate to each other and how they relate to software development.

The application of the philosophical theories to software development is also present in the work of Lehman, this time from the perspective of improving the enormous amount of time, money, and effort spent on maintaining software systems (Lehman 1980). From that perspective he details issues in software evolution and explains how research has progressed by developing the view that a software system can be best understood as a model.

> "Any program *is a model of a model within a theory of a model of an abstraction of some portion of the world or some universe of discourse*." (Lehman 1980) [Lehman's emphasis].

According to Lehman, software systems can then be classified into one of three classes:

- S-Programs: Programs whose function is formally defined by and derivable from a specification. Examples include programs to solve problems based on mathematical algorithms, such as the 8-queens problems, or based on concepts that are inherently well defined, such as depicting geometric shapes on the computer screen. The specification can be formally expressed and captures exactly the problem to be solved. Therefore, the solution can be precisely evaluated with respect to the specification and does not need to match any real-world process.
- P-Programs: These include programs whose specification can be captured unambiguously but concepts within it cannot be implemented in a software system without a degree of approximation. Lehman uses the example of chess playing and weather prediction. In these problem domains, the requirements of a computer system can be defined precisely – the rules of chess and the set of non-linear equations required to model global weather patterns. However, for system implementation, the solutions cannot precisely implement these requirements in a useable system. The procedures for analysing the state of a chess game and determine the next possible moves cannot be implemented completely – they can only be approximated. Furthermore, non-linear equations for weather modelling cannot be precisely implemented – they can only be approximated using simpler sets of equations. Both the problem statement and its solution approximate the real-world situation.
- E-Programs: These attempt to implement processes of human or social activity and result in a greater degree of approximation than the P-programs. Moreover, the people operating within that problem domain will use that software solution in their work. Therefore, they in turn become part of the problem they are attempting to provide a solution for. "The program has become a part of the world it models, it is embedded in it. Conceptually at least, the program as a model contains elements that model itself, the consequence of its execution." (Lehman 1980).

> Operating systems, air traffic control systems, and inventory-stock control are all examples of these systems. The pressure for system modification is immense. "As [the users] become more familiar with a system whose design and attributes depend at least in part on user attitudes and practice before system installation, users will modify their behavior to minimize effort or maximize effectiveness. Inevitably this leads to pressure for system change." (Lehman 1980)

Lehman notes that P and E programs are closely related and they differ from S-programs because they represent a computer *application* in the real world. He refers to them collectively as A-type programs. Because these programs are models of real world processes that, for various reasons, are under constant pressure to change, Lehman argues that the traditional life cycle of stages from requirements to implementation and testing is not easily applicable. Indeed, as noted earlier, Gallagher takes this point to an extreme and argues all software development should be considered as a process of maintenance rather than as a traditional process of artefact creation (Gallagher 1997).

6.3.1.3 Paradigms of Software Design Methodologies

Hirschheim and Klein also exemplify the differences between the artefact engineering and model building views of software development in their *Four Paradigms of Information Systems Development* (Hirschheim and Klein 1989). They recognise the growing importance of making explicit the guiding assumptions on the research of a professional community and identify four different paradigms that influence software systems development.

> "As developers must conduct inquiry as part of systems design and have to intervene into the social world as part of systems implementation, it is natural to distinguish between two types of related assumptions: those associated with the way in which systems developers acquire the knowledge needed to design the system (epistemological assumptions), and those that relate to their view of the social and technical world (ontological assumptions)." (Hirschheim and Klein 1989)

They identify two dimensions of both the epistemological and ontological issues: the subjectivist-objectivist dimension and the order-conflict dimension. In the first dimension, the objectivist applies models and methods derived from the natural sciences to the study of human affairs. In contrast, the subjectivist refutes this approach and seeks

to understand the basis of human life by exploring the subjective experience of individuals. In the other dimension, the ordered view assumes the world is characterised by order, stability, and functional co-ordination, while the conflict view stresses change, conflict and coercion. The result is four paradigms for systems development: functionalism (objective-order), social relativism (subjective-order), radical structuralism (objective-conflict), and neohumanism (subjective-conflict).

Those four paradigms are then explained through generic story forms, identifying the key actors, the narrative of how systems development proceeds, the major plot, and the assumptions that guide the whole process. The first two, functionalism and social relativism, are the most applicable or recognisable in software development. The functionalism paradigm closely associates with the established artefact engineering view of systems development. They label their story as the 'developer-as-systems-expert', who seeks to identify the underlying order of the domain and capture it as the rules, data, and functionality of a software implementation. In contrast, the social relativism approach identifies the developer as a facilitator, trying to interpret some structure from a reality that it inherently unstructured. It favours an approach to development that facilitates the learning of all that are involved.

6.3.1.4 The Influence on the Model Building View of the Development Process as a Whole

The philosophical foundations for the model building have also been used by Winograd and Flores in the field of artificial intelligence (Winograd and Flores 1985). Their research attacks the guiding assumptions used in the understanding of cognition and attempts to represent it in computer systems. The thrust of their argument against the established way of understanding is similar to that of Dreyfus (Dreyfus 1992), which was briefly discussed in the previous chapter, however their conclusions also include sections that are applicable to software development in general.

Winograd and Flores begin by identifying the 'rationalistic' tradition as the dominant set of guiding assumptions in the understanding of cognition and systems development. Their explanation of the 'rationalistic' tradition is similar to the classical theory of concepts and meanings.

> "Problem solving requires the representation of a situation in terms of identifiable objects with well-defined properties, and the logical application of general rules to situations so presented." (Winograd and Flores 1985) (p. 15).
>
> "Meaning can be analyzed in terms of correspondence between sentences in a natural language, and interpretations in a formal language for which the rules of reasoning are well defined." (Winograd and Flores 1985) (p. 17)

They subsequently attack that rationalistic tradition by using the philosophy of hermeneutics, specifically the work of Heidegger, and the neurobiology research of Maturana. Heidegger's philosophy builds on the notion that concepts cannot be defined independent of context. He explores the theory that concepts of reality can only be understood with respect to the person interpreting the world around them. Heidegger's 'being-in-the-world' operates by constantly interpreting its material and social environment. Rather than having knowledge of reality, his theories describe a constant process of being, which is characterised by unconsciously applying theories to reality in an attempt to understand and operate within in it. Those theories become articulated and improved when a 'breakdown' occurs that challenges the accepted way of understanding (for example, see Heidegger in (Urmson and Ree 1989)). Maturana's research in neurobiology also challenges the accepted understanding of knowledge acquisition. Rather than explaining knowledge as the interpretation and generation of concepts from sensory input, his research shows that neurological processes are triggered, not by all sensory input, but by expected patterns of sensory input. Maturana explores these results to develop his theory of understanding that is similar to the description of conceptual relativism and the application of theories in the understanding of reality detailed in the previous chapter.

Winograd and Flores apply these counter claims to the rationalistic tradition in the justification of research in artificial intelligence systems and finally to the development of software systems in general. Their conclusions explore areas that are similar to Lehman's discussion of A-type software systems and Hirschheim and Klein's social relativism paradigm.

Dahlbom and Mathiassen discuss the model building issues in all phases of the software development lifecycle (Dahlbom and Mathiassen 1993). They begin by exploring two different ways of understanding reality: the mechanistic view, which is similar to the

classical theory of meaning or the rationalistic tradition, and the romantic view, which relies on the subjective interpretation of the observer rather than on an objective, observable reality. These views are then explored in the context of software development to show how they lead to completely different ways of approaching the software design process. For example, the constructionist approach matches the artefact engineering view of development. In contrast, the evolution approach explains software development as a constant process of evolution as the users, managers, and programmers all develop more sophisticated models of the problem domain and the role of the software system in that domain. That evolution occurs over successive versions as the use of that software facilitates that evolution of understanding.

Their analysis then proceeds to a more managerial level and uses the different paradigms of understanding to explore the quality of software systems in general. That is, how the system could be evaluated as solving the original problem in the context of the problem setting.

Through an analysis of the evolution of many disciplines, Dahlbom and Mathiassen note that a change of guiding assumptions occurs in all professional disciplines as the relevant researchers seek to develop a better understanding of that discipline. Their conclusion is that these ideas can be expected to increase in the research agendas of software development.

> "If we look at the history of modern science, it all begins with the natural sciences and positivism. Hermeneutics and an interest in the humanities come later, and partly as a reaction to a dominating mechanistic perspective. This order of events seems somehow natural, and we find it almost everywhere we look. Sooner or later in the history of a practice it will turn to science for advice, passing through a stage of positivism only to enter a more chaotic period of attempts to develop hermeneutic alternatives. We have seen this happen in education, medicine, and social work. And we are seeing it happen in systems development. We began by taking an interest in computers, only later to realize that there were people involved too." (Dahlbom and Mathiassen 1993) (p. 208)

6.3.2 Improving Software Engineering Research using the Model Building View

6.3.2.1 The Research of Peter Naur

Peter Naur provides one of the most detailed explorations of the application of the model building view to software development. Naur was a major contributor at the NATO conferences and was co-editor of the published transcripts (NATO 1976a; NATO 1976b). His contributions reveal a mixed set of guiding assumptions about the nature of software engineering. As detailed in chapter 3, those comments used analogies with the complexity of civil engineering design, the architecture theories of Alexander, and the large-scale partitioning of automotive designs to inspire ideas that are based on the guiding assumptions of the artefact engineering view of software development. However, he also made a number of insightful comments that highlight aspects of software development that conflict with that way of understanding. For example, his remark during the debate following McIlroy's *Mass Produced Software Components* (McIlroy 1968) paper from the 1968 conference highlights his struggle to understand software development using the engineering analogies and still come to grips with the essential nature of software systems.

> "Naur: What I like about this is the stress on basic building principles, and on the fact that big systems are made from smaller components. … A comparison with our hardware colleagues is relevant. Why are they so much more successful than we are? I believe that one strong reason is that there is a well established field of electronic engineering, that the young people start learning about Ohm's Law at the age of fourteen of thereabouts, and that resistors and the like are known components with characteristics which have been expounded at length at the early level of education. The component principles of our systems must be sorted out in such a form that they can be put into elementary education." (NATO 1976a)

Naur's research, both prior to and following the NATO conferences, details the transition he made from understanding software systems using an artefact engineering point of view to a model building one. His examination of the issues is driven by attempts to identify the "component principles of our systems" and subsequent publications have been collected in a single volume of work (Naur 1992a). A few years before the NATO

conferences, Naur published his preliminary ideas on the important principles of software – the concepts of data, datalogy, and datamatics.

> "By datalogy I will understand the discipline of data, their nature, and use. An important part of datalogy is datamatics, the processing of data by automated means." (Naur 1992d)

His initial research examines the nature of data and compares its use in software with the underlying concepts of data in mathematics and linguistics. By the mid 70s, Naur was examining these issues in more detail (Naur 1992c). His analysis examines the relationship between data, words, concepts, and their philosophical understanding. Drawing on the work of philosophers such as Wittegstein, Naur began to develop theories for applying those issues to the specific context of software development.

> "Data science is the science of dealing with data, once they have been established, while the relation of data to what they represent is delegated to other fields and sciences." (Naur 1992c)

The mid 80s saw Naur publish research that examined in more detail the relationship between data and what it represents. His explanation of *Intuition in Software Development* (Naur 1992f) questions the emphasis of software engineering research on design methods. Based on his reading of the philosophical issues, he suggests that intuition, the way the designer understands the problem, is the driving factor in determining the integrity of the developed system. He notes that our intuition is not perfect, nor is it well understood. The role of design methods is to "guard us against the occasional errors of intuition" (Naur 1992f). According to Naur, to improve software development, research should be directed at understanding our attempts to relate the world, our knowledge of it, and our use of texts to capture that knowledge.

In 84, Naur summarises many of those developing ideas in the keynote address entitled, *Programming as Theory Building* (Naur 1985), delivered at the Euromicro 84 conference.

> "Some views on programming, taken in a wide sense and regarded as a human activity, are presented… it is concluded that the proper, primary aim of programming is, not to produce programs, but to have the programmers build theories of the manner in which the problems at hand are solved by program execution. …

> A more general understanding of the presentation is a conviction that it is important to have an appropriate understanding of what programming is… What I am concerned with is the activity of matching some significant part and aspect of an activity in the real world to the formal symbol manipulation that can be done by a program running on a computer." (Naur 1985)

His understanding of theory development is explained through the philosophy of Ryle, who explored the notion of concepts, meaning, and language (see Ryle in (Kemerling 1997)). For Naur, the conclusion for software developers is that understanding these philosophical issues can lead to three important advantages.

1. The programmer having the theory of the program can explain how the solution relates to the affairs of the world that it helps to handle.
2. The programmer having a theory of the program can explain why each part of the program is what it is, in other words is able to support the actual program text with a justification of some sort.
3. The programmer having a theory of the program is able to respond constructively to any demand for modification of the program so as to support the affairs of the world in a new manner. The design of how a modification is best incorporated into an established program depends on the perception of the similarity of the new demand with the operational facilities already built into the program. (Naur 1985)

Naur notes though, that the difficulty of this view of software development is that it is extremely difficult to capture the essence, that is, the theory of the program in an objective way that can be shared with others.

> "The main claim of the Theory Building View of programming is that an essential part of any program, the theory of it, is something that could not conceivably be expressed, but is inextricably bound to human beings." (Naur 1985)

Naur's later publications continue to explore the relationship between human thoughts, languages in general, and the structures available in programming languages (Naur 1992g; Naur 1992h; Naur 1992e). That research led him to examine why such a different view of software development could exist. His later work explores ideas similar to the history and philosophy of science sources presented earlier in this chapter (Naur 1992b). They reject the myth that science, including computer science and the sought after science

of software engineering, can be based on a rationalistic approach or on a logical foundation of objective truth. His view of software development as theory or model building concludes with three different aspects that must be kept in mind by developers.

1. The aspect of the world that is being described or pictured by the model, the modelee for short.
2. The model, being a program in execution by a computer.
3. The elements of which the model has been built, typically items of computer hardware and the mechanisms of a certain programming language.

6.3.2.2 The Research of Bruce Blum

The final and most extensive examination and application of the philosophical issues to software engineering has been performed by Bruce Blum. His work can be described as the quest to explain the software paradox (as stated by Stucki).

> "The software community has done an excellent job of attempting to automate everyone's job *except* their own!" (Blum 1985)

Blum identifies three reasons why this is the case.

1. We really don't understand what software is.
2. We have not performed a systems analysis of the software development process.
3. The implementation of new software development paradigms is as much a social problem as a technical issue. (Blum 1985).

Blum's research is summed up in his culminating work, *Beyond Programming: To a New Era of Design* (Blum 1996). Blum believes the existence of the software paradox is due to a faulty model of understanding the nature of software development and reality. That understanding developed during the embryonic stages of the software engineering discipline and remains to this day. Therefore, to improve the nature of software engineering, nothing short of a Kuhnian paradigm-shift is required.

Blum begins with his definition of software engineering and the subsequent issues raised by it.

> "The application of tools, methods, and disciplines to produce and maintain an automated solution to a real-world problem.

> Much of the software process is devoted to characterizing the real-world problem so that an automated solution can be constructed. Because the problem interacts with its solution, the software product must evolve. Moreover, because the software operates in a computer, it must be expressed as a formal model of a computation." (Blum 1996) (pp. 4-5)

To develop his new paradigm, Blum presents a detailed examination of these issues from, as he describes them, 'first principles'. Those principles are detailed during the exploration of three broad topics. First, to examine the relationship between computer science and software engineering, he explores the nature of science, the relationship between science and technology, and the concepts of truth and knowledge. Second, because software design is a form of problem solving, he explores the innate mechanisms of human problem solving, the social context of achieving solutions, and the nature of design in general. Finally, the third section applies those issues to the software engineering process. In that section, the history of software design is presented and the nature of design methods are examined in terms of the previously explored philosophical and design reasoning issues. His book culminates with an examination of how those issues result in a new paradigm for software engineering and presents the results of a fourteen year case study of the application of that paradigm to Blum's specific area of medical information systems. His conclusion is that while Naur suggests moving towards programming as theory building, he suggests software engineering should move beyond programming altogether.

The introduction to Blum's examination of science and technology critiques Mary Shaw's article *Prospects for an Engineering Discipline of Software* (Shaw 1990), which was also discussed in chapter 3 of this thesis. Noting her conclusion, that a discipline of software engineering requires a supporting science, Blum surmises that it is difficult to fault the steps Shaw proposes to develop this software science,

> "Yet it is equally difficult to understand what is meant by science and engineering in the context of computer technology. In the popular view, science uncovers the truths around which we build our technologies. Is this a valid statement? What are the truths of computer science? Are there alternative computer sciences … Is there an envelope around what can be guided by computer science, and, if yes, how do we address problems outside the envelope?" (Blum 1996) (p. 21).

Blum's subsequent analysis of science in general presents a summary of the relevant issues from the history and philosophy of science, which are similar to the sources presented at the beginning of this chapter. To determine how those issues affect software engineering, Blum examines the relationship between science and technology. He begins with Shaw's representation of the traditional view, which states that science drives technology.

> "In Shaw's model of the growth of an engineering discipline, science was given a direction by the technology; once mature, the science served to drive the technology. That is, as a technology matured, its ad hoc solutions were extracted and embedded in a body of scientific knowledge, which in time would serve as a forcing function for that technology." (Blum 1996) (p. 54).

However, his examination of the relevant research and case studies in the area of science and technology reveals that is not the case.

> "Science may support technological improvement or the identification of presumptive anomalies, but technology's problem solving mission resists the acceptance of new scientific knowledge as a solution mechanism in search of a problem; that is, technology exploits scientific knowledge, but it is not driven by it." (Blum 1996) (p. 62)

Blum continues to examine the nature of science. He asserts, an admittedly limited view, that the goal of science is to produce models of reality. He then proceeds to examine the ability of science to produce faithful models of reality. His description is based on the 3 world epistemological model of Popper and the epistemological consequences of Kuhn's theories of science. The result is similar to the conclusions of the previous chapter of this thesis, which examined different theories of epistemology in greater detail. There is no objective reality that all people share. The best that can be achieved are progressively more detailed models of it. Blum continues by examining the issues involved in modelling that reality. His analysis results in a model of

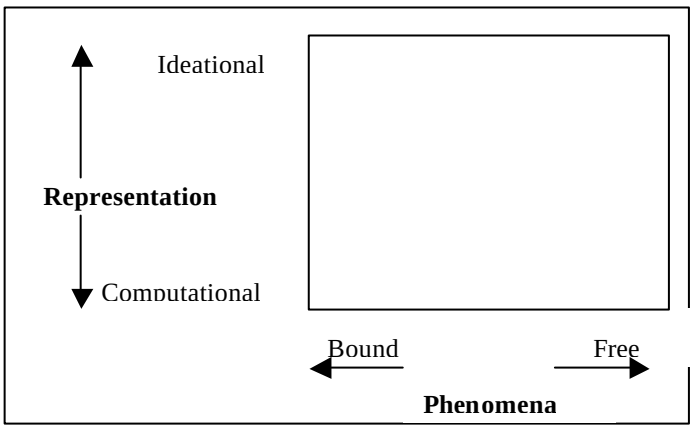


Figure 6-1: Models-of-reality space

models-of-reality (figure 6-1: from (Blum 1996) p. 72). The vertical dimension depicts representation. The computational representations are the well-defined notations of, for example, mathematics and logic. The word 'ideational' is used to capture the other extreme of ambiguous, subjective expression. The horizontal dimension represents the phenomena to be modelled. Bound phenomena represent aspects of reality that depict repeatable, observable processes such as the physical laws of motion. Models of bound phenomena can be used to describe and predict those aspects of reality. In contrast, free phenomena are processes that can be described but are not necessarily bound to behaving the same way in the future.

Traditionally, science, which develops computational models of bound phenomena, and art, which explores ideational representations of free processes, are placed at opposite corners of the model. However, Blum's analysis of the theory of models argues that a clear distinction is not possible.

> "Thus, design – and by extension, all creative activities – merge art and science/technology within the (holistic) context of a larger system. We cannot isolate 'science' and the 'art' components of our technology…" (Blum 1996) (p. 75).

Computational models of bound phenomena are possible, however Blum points out that software engineering does not always deal with bound phenomena. This leads to a central tension in software engineering.

> "The science of computer technology is that of a free phenomena. It is the study of the transformation of ideas into operations…
>
> Our goal is to reduce the central tension in the software process: the fact that we start with a need that often is poorly defined and difficult to represent and we must end up with a formal model that executes within a computer. Traditionally, computer science has chosen to concentrate on the formal aspects of the process, but more is needed." (Blum 1996) (pp. 85-86).

Blum's analysis then considers design more specifically. His begins with the traditional understanding of design, technological design, in which designers employ technological knowledge to construct an artefact that satisfies the stated requirements. Through a study of relevant areas he develops a more sophisticated model (ecological design) that incorporates the human environment in which the design is initiated, conducted, and

evaluated. That study begins by laying foundations that show the traditional understanding of design, technological, is a consequence of the rationalistic or positivist understanding of reality. Blum shows, through the work of many researchers in design theory, that disciplines are slowly moving away from that model of understanding and towards a more humanistic and subjective understanding of reality. To understand design from a more subjective understanding, Blum details theories from the areas of problem solving in cognitive science, the nature of expertise, the nature of complex problems, and the role of reflection and context in the problem solving process. The result is a collection of theories that challenge the traditional understanding of design.

> "If … 'learning, thinking, and knowing are relations among people engaged in activity *in, with, and arising from the socially and culturally constructed world*', then 'changing existing situations into preferred ones' will impact what we think and know. Thus, paradoxically, we find that change is the invariant, not knowledge.
>
> These are just some of the implications for a new era of design." (Blum 1996) (pp. 158-160).

With these foundations, Blum analyses design theories from different disciplines. They include architecture, industrial design, engineering and systems design. Finally, Blum includes theories that specify how these design processes are influenced by the different stakeholders who participate in the design process. The result of Blum's lengthy and detailed analysis of many issues is a set of foundations for understanding his definition of software engineering.

However, these foundations do not provide a 'silver bullet' for the problems of software engineering. Indeed, they show such a thing is impossible.

> "Our objective as software designers is to employ software technology to improve the human environment. We seek instruments to guide us in this endeavor, but we also recognize that such instruments may not exist independently of their use; that is, these instruments cannot be discovered, they must be designed. We are constrained by software's central tension: the environment we wish to modify exists in-the-world and may not be subject to formal descriptions, whereas the software we create exists in-the-computer and must be represented formally – as models of the real world and as models

> of computations. We might take solace from our studies of the foundation if they produced formal models of the world that could be represented in the computer or if they demonstrated the existence of an integrating force that would synthesize the individual models. But this does not seem to be the case. There is no independent 'design process' that can be discovered and dissected. We must accept the fact that the design process is an ephemeral artifact, a residue of the process's conduct. It provides clues as to what was done, it offers insights into what should not have been done, but it never establishes what ought to be done." (Blum 1996) (p. 242)

Blum's solution to this dilemma is adaptive design.

> "We should move from our historic interest in modeling the *product* in favor of a concern for modeling solutions to the *problem*. This implies a migration from build-to-specifications (together with their realizations as programs) to the use of as-built specifications that describe evolving responses to problem-space needs. That is, we should go from the characterization of solutions in-the-computer to solutions in-the-world. Clearly, such a shift goes *beyond programming*. Clearly, too, such a shift will not be easy and cannot be phased in." (Blum 1996) (p. 263)

During adaptive design, the design centres on the building of a formally expressed, conceptual model for a problem solution. Implementation is the automatically generated realisation of a correct and valid solution (Blum 1993). The concepts that comprise those conceptual models exist as fragments in the repository of the development environment and surrogates are used to capture their expression from many different perspectives and many different levels of granularity. To construct a solution to a problem, the developer generates a conceptual model using the fragments that exist in the knowledge base. That conceptual model captures the concepts of primary interest in a scheme that permits automatic transformation to an executable form and allows automated reasoning about its correctness. The execution of the application is performed by the encompassing environment that keeps track of the fragments to be executed for a particular application. In Blum's environment, the design becomes the product. The concept fragments and conceptual models that group them into applications evolve as both the developers' and users' understanding of the problem domain becomes more sophisticated (Blum 1996) (pp. 304-308).

A detailed explanation of the approach is not presented here. Blum provides many references to the description of his approach to design and a case study of its use in the development environment, TEDIUM. Clearly Blum's description of adaptive design is also evident in other development environments, for example Amdahl's ObjectStar environment (Amdahl 1998). However, for this thesis, the important aspect of Blum's approach is the philosophical basis for it and not the end product.

Finally, Blum notes that the implementation of the knowledge required to realise this system is possible only because there is a well-understood application domain that is supported by a mature technology (Blum 1993). Although Blum's detailed analysis of the foundational issues and its application in a new design paradigm is interesting, it is not clear how it can be applied to more general purpose software development.

6.4 Conclusion: Evaluating Software Engineering Research

This chapter began with a description of the different theories in the history and philosophy of science. One of the conclusions of those different theories is that research performed in particular disciplines is directed and validated by underlying guiding assumptions that determine how the phenomena under investigation is understood. Moreover, those guiding assumptions usually change as a discipline progresses and develops more sophisticated models for understanding that phenomena. In the context of software engineering, the implication of this thesis is that the artefact engineering view of software development has provided a dominant set of guiding assumptions. However, a more detailed examination of the phenomena under investigation (the underlying principles of software and software systems) shows that a model building view of software development holds the potential for a more beneficial set of guiding assumptions.

To highlight the potential of the model building view, a selection of existing research in the software engineering literature was presented that is based either implicitly or explicitly on that way of understanding software development. However, that selection was certainly not exhaustive. Because the issues presented in this thesis aim at providing a philosophical foundation for software engineering research, arguments could be made that those foundations are exemplified in many other published researches and commentaries. For example, a recent IEEE Software issue was dedicated to architectural

design with the guest editor suggesting we should be *Reevaluating the Architectural Metaphor* (Coplien 1999b). Moreover, in that issue Perrochon and Mann question the appropriateness of the belief that an architecture should always be specified before the implementation is commenced (Perrochon and Mann 1999). Other examples include popular columns in software engineering journals by authors such as Glass and Jackson that often highlight anomalies in the established way of understanding software engineering. However, there has been no established foundation to explain why those anomalies exist (see for example (Glass 1994; Glass 1998b; Jackson 1998b; Jackson 1998a)). Kumagai has suggested that perspectives on software engineering based on an eastern rather than western view of the world may be an interesting source of research ideas (Kumagai 1998). There are strong parallels between the theories of conceptual relativism presented from the disciplines of philosophy and psychology and the eastern philosophies discussed by Kumagai. Those parallels have been a source of interest for this author, however they were not used in this thesis – though they may be used in future research (see for example (Capra 1983)). Additionally, cognitive studies in software engineering research also highlight issues that can be explained using a model building view of software development rather than an artefact engineering perspective – see for example (Silverman 1983; Adelson and Soloway 1985; Silverman 1985; Curtis 1989; Curtis, Krasner et al. 1991; Dumas and Parsons 1995; Stacy and Macmillian 1995; Winograd 1995).

Despite the argument presented in this thesis, that the model building provides the philosophical foundations necessary to explain and justify many issues in software engineering research, the theories in the history and philosophy of science suggest considerable resistance will always meet any transition between sets of guiding assumptions. Both Naur and Blum have commented on the resistance to their research ideas and the sense of frustration it has caused.

> Naur: "Several of the writings of the section are unusual by their sharply critical tone of voice. I am aware that thereby they can hardly avoid being painful to certain persons involved. They are in fact quite painful to me, being manifestations of the fact that the field in which I have spent a good part of my professional life gives strong support to pretentious ignorance and misunderstanding on a large scale." (Naur 1992a) (p. 479).

> Blum: "I believe that my work has never been accepted within the mainstream because it employs an alternative paradigm and therefore is outside the mainstream. True, there are many other approaches that also exist within alternative paradigms, but that work is undertaken by communities of researchers who share their work and mutually promote their findings. Given the volume of new reports produced each year, it is not surprising that researchers marshall their time, read selectively, and focus their energies on the topics of principle interest to them. This is not sour grapes; it is simply an acceptance of reality." (Blum 1996) (pp. 302-303)

This does not suggest that problems cannot be identified in either Naur's or Blum's work. Nor does it suggest that those ideas should be accepted simply because they are based on an alternate set of guiding assumptions. However, the theories presented at the beginning of this chapter suggest that researchers evaluating these new ideas need to remain aware of the influence of guiding assumptions on the understanding of the discipline. For example, in ACM Computing Reviews, Teplitzky (Teplitzky 1994) reviews Blum's analysis of software design methods based on these philosophical foundations (Blum 1994). Teplitzky's analysis contains some interesting points, however his editorial commentary accuses Blum of suffering from "the fog of academia" and criticises his approach as being deliberately elitist when in fact Blum had simply appealed to established lines of argument in philosophy. Blum's work receives similar treatment from other reviewers – see for example (Mitchell 1996). There is nothing secretive or elitist about the dialect used by Blum. If software engineering researchers are to understand the underlying principles of software systems then they must confront these philosophical issues. While that work should clearly be criticised for errors in argument, evaluators need to ensure it is not merely being criticised for being different.

Similar debates about the appropriateness of the philosophical issues also exist in other disciplines. For example, researchers of design in general have also turned to the philosophy of science for ideas (Jacques 1981). Braha and Maimon found parallels between the theories in the history and philosophy of science and traditional engineering design (Braha and Maimon 1997). In contrast, Cross argues that design theories should not be compared with theories of science (Cross, Naughton et al. 1981). His argument centres on two principle tenets. First, he notes that design is inherently different enough from science to make the analogies invalid. Second, the theories in the philosophy of

science are currently in dispute and therefore should not be applied to the field of design in general.

> "Attempts to equate 'design' with 'science' must logically be predicated upon a concept of science that is epistemologically coherent and historically valid. The history of the twentieth-century debate in the philosophy of science suggests that such a concept does not yet exist. It would therefore seem prudent for writers on design method to back away from this particular line of argument, at least for the time being." (Cross, Naughton et al. 1981)

However, the arguments of Cross do not translate to the discipline of design in software engineering. The differences he identifies between the fundamental natures of 'design' and 'science' are based on the nature of built forms, which does not exist in an analogous way in software engineering[37]. Moreover, the fact that some disputes exist between the different philosophies of science ignores the commonalties identified by Laudan et al, which were presented earlier in this chapter.

Similar arguments against the application of philosophical foundations to software development can be found in the software engineering literature. For example, Meyer (Meyer 1997) discusses the philosophical nature of abstract data types by noting that the exact meaning of object definitions is relative to the person using them.

> "If I am thirsty, an orange is something I can squeeze; if I am a painter, it is a color which might inspire my palette; if I am a farmer, it is produce that I can sell at the market; if I am an architect, it is slices that tell me how to design my new opera house, overlooking the harbor; but if I am none of these, and have no other use for the orange, then I should not talk about it, as the concept of orange does not for me even exist." (Meyer 1997) (p. 147)

However, rather than using this observation as inspiration for examining the disciplines that have been studying the principles of conceptual relativism and then determining their implications for software engineering, Meyer argues the opposite.

> "Over the years many articles and talks have claimed to examine how software engineers could benefit from studying philosophy, general systems theory, 'cognitive science', psychology. But to a practicing software

> developer the results are disappointing. If we exclude from the discussion the generally applicable laws of rational investigation, which enlightened minds have known for centuries … and which of course apply to software science as to anything else, it sometimes seems that experts in the disciplines mentioned may have more to learn from experts in software than the reverse." (Meyer 1997) (p. 148)

Meyer's claim is not based on an examination of the philosophical issues nor any critique of the theories those disciplines have proposed. We are led to believe that because software developers have built some large and complex systems, and the underlying principles of those systems are based on the notions of concepts, theories, and abstractions, then those other disciplines have more to learn from us because they are still arguing about those principles while we have been successfully using them.

> "Software builders have tackled – with various degrees of success – some of the most challenging intellectual endeavors ever undertaken. Few engineering projects, for example, match in complexity the multi-million line software projects commonly being launched nowadays. Through its more ambitious efforts the software community has gained precious insights on such issues and concepts as size, complexity, structure, abstraction, taxonomy, concurrency, recursive reasoning, the difference between description and prescription, language, change and invariants. All of this is so recent and so tentative that the profession itself has not fully realized the epistemological implications of its own work.
>
> Eventually someone will come and explain what lessons the experience of software construction holds for the intellectual world at large." (Meyer 1997) (p. 148)

Meyer's comments show a complete lack of understanding of both the relevant philosophical issues and the complexity of traditionally engineered systems. Based on both the ignorance and arrogance expressed in his comments, perhaps those lessons he welcomes will include theories explaining how a discipline of software engineering can progress in spite of rather than because of the way it understands those issues.

---

[37] This was argued in the Chapter 4.

7. Conclusion

This thesis has made explicit and then explored issues that influence the understanding of software engineering. The aim has been to show that understanding in software engineering research has been dominated by analogies with traditional engineering disciplines. However, an alternative approach, based on philosophical foundations, offers the potential for an improved way of thinking about software systems and how they are developed. The previous chapter specified how guiding assumptions govern the way researchers in a discipline understand the phenomena they investigate. Those guiding assumptions are not always explicitly stated and practitioners are not always aware of them. Indeed, it is not necessary for practitioners to be aware of them to operate as researchers within a discipline. However, those guiding assumptions set research agendas, direct investigations, bias observations, and justify conclusions. Moreover, those sets of guiding assumptions change as a discipline evolves and research based on different sets of guiding assumptions are not always commensurable with each other. In software engineering research, the most prevalent view has been that software development can be understood as artefact engineering. The research presented in this thesis has developed an alternative view that software development can also be understood as model or theory building. The conclusion is that this view has the potential to improve software engineering research.

Because it is not necessary to be aware of the underlying philosophical issues when performing research, it is often difficult to make others aware of their significance – let alone get them to evaluate different theories about them. Before we can question our guiding assumptions, we first need to be made aware of them. And to be made aware of them we need to encounter situations that cannot be explained without resorting to questioning, not just the situation, but how we think about that situation. Towards the end of my research, and as I was beginning to write up my thoughts, I had the chance to explain my theories to Keith Gallagher who was visiting my University at the time. He mentioned I should read Blum's *Beyond Programming* book. The immediate similarities between our work were obvious. However, what I could not understand was why I had never heard of Blum's work, or similar efforts. It may simply have been shoddy research on my part. However, I had never seen his work cited, or that of Naur, in the context I was working on – the philosophical understanding of software engineering. In

comparison with those two researchers, the advantage I have is that I am writing this at the beginning (hopefully) of my research career, rather than at its culmination. To remember how I thought about software engineering before developing a different way of understanding it I only have to reach back a few years rather than a few decades. Had I encountered Blum's book before coming to similar conclusions I probably wouldn't have fully realised its implications for software engineering. Without understanding the influence of guiding assumptions on research and without comprehending the prevailing influence of the classical but now outdated theories of how we have knowledge of reality, his conclusions would have made little sense. From those realisations, and the realisations achieved from trying to explain these theories to other software engineering researchers, the difficulty in explaining these issues to others has become apparent. Consequently, I decided to write the thesis in an order that reflects the evolution of my own understanding of software engineering rather than simply presenting a new approach. The turning point in my understanding came with the cruise control comparison so that was the logical place for the substance of this thesis to begin. That study highlighted issues that cannot be adequately explained without resorting to an analysis of the fundamental nature of software engineering and, therefore, prepares the reader for the subsequent philosophical treatment. This conclusion can now summarise the issues in a different light.

The NATO conferences established the artefact engineering view of software development. At that time, the software development community had produced many large-scale systems and was looking for a way of directing research efforts that would result in improved practices. Engineering was a natural choice. The disciplines appear similar with many terms beings used between the two. The term 'software engineering' was suggested as a means of provoking discussion, a starting point for developing a view of software development that would provide significant leverage for researchers and developers. The analogies with traditional engineering disciplines used by the conference participants were able to highlight important issues and thereby served to reinforce the artefact engineering view. However, other insightful comments were made that could not easily be explained using that same view. The conference failed to achieve consensus among the participants concerning the applicability of the artefact engineering view. Nevertheless, the term stuck and became the dominant guiding principle for the next thirty years (at least) of software engineering research.

The research from the history and philosophy of science shows that guiding assumptions are not always chosen because they solve all of the problems of a particular research discipline. They simply have to show potential for solving problems. During the 1968 NATO conference, the idea that software systems could be engineered showed enormous potential for improved development practices. For instance, McIlroy's *Mass Produced Software Components* paper provided a vision of software engineering that matched the participants understanding of traditional engineering, and which, if realised, could solve many of the concerns that led to the conference. The debate during the 1968 meeting concerned whether or not that vision was applicable. However, by 1969, without resolving the identified problems, the debate turned to how to achieve that vision. The artefact engineering view of software development provided the necessary direction for research in a field that had previously lacked a common view. Its emergence as the dominant set of guiding assumptions is consistent with the philosophical and psychological theories presented. This is not meant as a criticism of those conference participants, it is merely provided as an explanation for their actions.

Despite the fact that the analogies used between software development and traditional engineering disciplines have illuminated many issues for researchers, the analyses presented in this thesis show that many of the similarities and differences were misunderstood. On the evidence presented, the conclusion is that researchers have not had a thorough understanding of software development, nor have they had a thorough understanding of those engineering disciplines used in the comparisons. Nevertheless, the result of using those analogies has facilitated significant progress in software engineering techniques. One of the anomalies that did arise, and which has remained the source of debate, has been the question of the underlying principles of software systems and software development. Researchers realise that the methods of engineering are based on the ability to model and predict the properties of system and component designs based on the underlying principles of the discipline. The underlying principles of software systems were obviously different, however it is not immediately apparent what they are. However, those differences have not been used to question the validity of the underlying assumptions of software engineering. Rather, the research concerned with the progress of the discipline has sought to identify the relevant 'science' of software engineering. That 'science', it is supposed, will explain the underlying principles and provide the rigorous mathematical techniques necessary to make software development an 'engineering'

discipline. Unfortunately, those principles and the associated 'science' have never been discovered. Formal methods have been suggested as the science for software engineering, however a comparison of both what is represented by the mathematics, and how it is used in the development processes of both software development and traditional engineering shows the suggestion is incorrect.

By using traditional engineering development as a contrasting position from which to observe the software engineering approach, the differences become apparent and the nature of the underlying principles of the two disciplines become clear. That was evident in the cruise control case study. The engineering designs all followed a similar process. The designers began with a common design strategy, feedback control, which the history of control systems has shown to be the most useful way of approaching the problem. An initial system architecture was developed that consists of well-known generic components in a well-known structural arrangement. Those components represent standard functionality that can be applied to the underlying properties of the discipline – electrical signals. The functionality of those components and the environmental influences on the system were then modelled mathematically and a combination of experimental testing and additional mathematical and graphical analysis techniques were then used to solve the unknown parameters of those mathematical models. Because the system components are specified in terms of idealised functions that can be applied to the underlying properties of the discipline, the functionality of the components are independent of the context in which they are used.

The design approach of the software developers was completely different. They began with a design formalism that allowed them to depict the concepts and relationships required to solve the problem. The successive stages of design serve to refine and implement the functionality of the initial model. The identified concepts and relationships represent the functionality to be performed. However, because the system needs to be executed by a computer, the model also needs to include, or be supplemented by, a depiction of how the model should be executed to solve the problem. Finally, because the model needs to be realised in the implementation medium of the discipline, the model needs to be specified in a structure that maps easily onto the constructs provided by that implementation medium. The different design methods provided different ways of capturing these three aspects of the software model, however they do all capture it.

The contrast between approaches provides a glimpse of the fundamental nature of software systems and shows how their development differs from traditional engineering development. Traditional engineers build artefacts to solve real world problems. Software developers build models of reality to automate real world processes. Traditional engineers design their systems to meet the functionality that is achievable using the pre-existing components of their implementation medium. Software developers use a tremendously malleable implementation medium to realise the component functionality of their designs. Therefore, making the artefact engineering view of software development explicit and then examining it with respect to what artefact engineers actually do, reveals a different understanding of software development – the model building view.

The model building view provides a way of identifying the underlying principles of software systems (abstractions, concepts, and theories), which can be examined through disciplines that have a long history of researching those principles – the disciplines of philosophy and psychology. Both of these disciplines show there is a classical way of understanding concepts and theories. In that classical view, the observer is separate from reality and that reality is objective in that everybody experiences the same phenomena and can classify it in the same way. Our explanations of the world are inferred from what is really there and the concepts and relationships that comprise those explanations are defined in terms of essential attributes or features. This classical way of understanding matches the way traditional engineers understand their components and systems and, therefore, it appears to provide a philosophical foundation for the artefact engineering view of software development. That is, if the classical view is a valid means of understanding, then our models of reality can be treated in a similar manner to traditionally engineered artefacts. Indeed, some software engineering researchers have used aspects of the classical way of understanding, especially the notions of concept definition by intension and extension, as justification for their views.

Unfortunately, it is not the case. Philosophical analysis and experimental psychology have developed more sophisticated theories of understanding that repudiate that classical view. As a consequence, the underlying principles of software systems cannot be understood in the same way as traditionally engineered systems. Reality cannot be considered as separate from people's understanding of it. There is a myriad of detail to observe and people would not be able to function by constantly inferring concepts and relationships from the infinite amount of detail presented to them. Instead, people

automatically and subconsciously apply concepts and theories to that detail in order to understand the world and function in it. As people interact with phenomena, they apply different concepts and theories to it in order to provide a useful explanation for it. If one theory does not provide a suitable explanation, a different or more complex theory is generated and applied. The consequence is that there is no objective reality that all people share. Many different concepts and theories can explain the same reality. The reason the world is forever inaccessible is because all observation is tainted by the theories that we automatically and subconsciously use to understand that world. Therefore, it is impossible to detect which concepts and theories are closest to the truth. They can only be measured by their usefulness, not their veridicality. Furthermore, those concepts and theories become part of the culture of different groups of people and pass between them as they communicate.

The other significant contradiction to the classical way of understanding concerns the definition of concepts and relationships. The precise meaning of concepts cannot be specified independent of the context in which they are used. That context is the theory being used to understand the experienced phenomena. Therefore, while it may be possible to define a concept by essential attributes, that does not mean the same concept can be represented by the same definition in another context.

Commentaries in philosophy and psychology recognise that the classical way of understanding still dominates people's guiding assumptions in the general community and this is evident in the justifications used for software engineering theories. However, detailed analysis has provided more sophisticated explanations of human understanding that explain how people develop and use models of reality. Those theories provide the foundation for a new understanding of software development. In addition, that foundation provides a way of explaining existing research in software development, a way of explaining anecdotes provided by experienced practitioners, and highlights issues that can lead to new avenues of inquiry.

One criticism encountered when explaining these theories to other researchers is that it does not explain how software development should be performed. That was not the purpose of the thesis. A few years ago, I came across a cartoon in a book on mathematics, which unfortunately I cannot find now. It captures the conclusion this thesis has attempted to present. Two children are playing in a sandpit The first asks the other, "Have you found the answer yet?" and is met by the reply, "No, but now I know how to ask the

right questions". This treatise has examined the fundamental nature of what software is so that the right questions can then be asked about how it could be engineered. Nevertheless, a number of conjectures have become apparent that could be used as the basis for subsequent research in the areas of reuse, patterns, and architecture.

The first concerns software reuse. Substantial gains have been made as a result of our efforts to reap the benefits of widespread software reuse. However, we have yet to achieve the same scales of reuse that has been achieved by traditional engineering disciplines. The philosophical foundations of the model building view may provide some insights to explain this. The first insight concerns the difference between requirements/analysis concepts and design/implementation concepts. Concepts are identified during the requirements/analysis stage of the development process and have to be precisely defined and implemented as software constructs during the design/implementation stage. However, the concepts we entertain in our explanations of the world do not identify objective real-world parts and they cannot be universally defined by essential attributes. They are theory dependent and are subjective to the person using that theory to understand the phenomena under investigation. This results in two different types of concepts. The first (referred to here as $concepts_1$) are the fuzzy, theory-dependent concepts applied to sensory experience to assist human understanding. The second (referred to here as $concepts_2$) are the independent, rigorously defined structures of software design and implementation. The identification of a $concepts_1$ concept can result in an infinite variety of $concepts_2$ definitions. If a concept is identified during the development process of a system, then its definition, the resulting software construct, is only a realisation of that concept within the theory used to understand the problem at hand. For example, if the object-oriented analysis of a problem identifies a class, 'Customer' ($concepts_1$), then its definition ($concepts_2$) only provides the required features of a 'Customer' within the confines of the problem that the system solves. The philosophical foundations of software engineering suggest that if the analysis of a different problem also identifies a 'Customer' ($concepts_1$) during its analysis stage, then the original 'Customer' definition ($concepts_2$) may not be applicable in the new context. It may be possible to reuse the 'Customer' definition in the new situation, but it equally well may not be. This contradicts the idea of software reuse based on the classical theory of understanding and the artefact engineering view of software development.

Nevertheless, some successful reuse efforts have been achieved and they can be explained with the foundations provided by a model building view. The first concerns the observation that reuse is more successful when the designer browses an asset library before beginning design rather than searching for and retrieving assets to match the concepts of a proposed design (Mili, Addy et al. 1999). The human mind applies known concepts and theories to a situation in order to explain it. That is, humans understand a situation in terms of how they understand previously encountered situations. Having knowledge of what is already in a reuse repository before design commences exploits that innate conceptual ability by allowing the mind to devise a solution to a problem in terms of that knowledge. As the designer interacts with the problem, knowledge of those artefacts will be automatically and subconsciously applied to the situation to determine if they provide a useful explanation. Therefore, the human conceptual apparatus makes it a lot easier to design a system to reuse known artefacts than it is to find artefacts to meet a designed system.

Mili's analysis of software reuse also notes that software product lines provide the most dominant form of systematic software reuse today (Mili, Addy et al. 1999). This fact can also be explained as a consequence of the model building view. The precise meaning of concepts identified during the requirements/analysis phase of development ($concepts_1$) are dependent on the roles they play within the encompassing theory being used to explain the phenomena under investigation. That encompassing theory is the designer's conceptual model of the problem. If the same problem/phenomena was represented using a different conceptual model, a similar collection of concepts may still be identified. However, the precise meanings of the concepts that constitute that second model will be different because they play different roles and, therefore, require different definitions. Product line architectures appear to improve the potential for software reuse because they constrain the developer to utilising similar conceptual models to explain similar problems. Therefore, the concepts ($concepts_1$) that remain invariant across the different conceptual models in the product line can be realised by the same implementations ($concepts_2$), while the concepts that change can be modified as necessary.

Another successful reuse effort concerns artefacts that do not represent implementations of the fuzzy, theory dependent concepts used in human understanding ($concepts_1$). For example, user interface components are often used as examples in explanations of successful reuse theories. However, they are not the same as the concepts used to explain

real world phenomena. Rather, they have been defined independent of human experience for use in software development. That is, when they were created, they were specified precisely and those precise definitions are passed on to system developers when they learn about and use those user interface components. Therefore, it is possible to successfully reuse those software components because they have universally applicable definitions. Moreover, as those components are utilised during the design process, they can be understood the same way that engineers understand their components. This is also the case for other successfully reused software components such as math libraries, compiler designs, etc.

It may also be the case that when a group of people have been interacting with the same problem for long enough that they develop common explanatory theories for it. Those theories, and the concepts that comprise them, may become codified to the point where their meanings are similar to all members of the community. This detailed, communal understanding of the phenomena may also lead to the reuse of similar concepts. However, to extrapolate from the successful reuse of these components to the claim that all software components can be reused the same way traditionally engineered components can be reused is to misunderstand the nature of software components.

While the philosophical foundations suggest conjectures for understanding the potential of software reuse, considerably more research is required if a systematic theory of software reuse is to be developed.

These foundations can also be applied to develop an understanding of software design patterns. Software patterns have become extremely useful in software development. Their historical link with the patterns of Christopher Alexander are well documented, however the model building view may provide a better explanation as to why they are so useful and provide insights into how they can be better utilised. Alexander himself questions the validity of the analogy between software patterns and his building patterns.

> "Now, of my evaluation of what you are doing with patterns in computer science… When I look at the object-oriented work on patterns that I've seen, I see the format of the pattern (context, problem, solution, and so forth). It is a nice and useful format. It allows you to write down good ideas about software design in a way that can be discussed, shared, modified, and so forth. So, it is a really useful vehicle for communication. And, I think that insofar as patterns

> have become useful tools in the design of software, it helps the task of programming in that way. It is a nice, neat format and that is fine.
>
> However, that is not all that pattern languages are supposed to do. That pattern language that we began creating in the 1970s had other essential features. First, it had a moral component. Second, it has the aim of creating coherence, morphological coherence in the things which are made with it. And third, it is generative: it allows people to create coherence, morally sound objects, and encourages and enables this process because of its emphasis on the coherence of the created whole.
>
> I don't know whether these features of pattern language have yet been translated into your discipline." (Alexander 1999)

Despite the successful application of design pattern theories to software development, research in the area fails to resolve anomalies that exist between software systems and traditionally built artefacts. For example, recall the analysis of the Alexander's work in chapter three. "The ultimate object of design is form" (Alexander 1964) (p. 15). Software systems do not have a notion of form that is analogous to that found in traditionally built artefacts. Hence, it is not clear what Alexander's term, "the coherence of the created whole", means in the context of software systems when using the artefact engineering view of software development.

However, if software development is understood as model building rather than artefact engineering, some explanations of how patterns are utilised in the model building process become evident. People automatically and subconsciously apply their accumulated concepts and theories to the world in order to understand their experience. In software development, the subconsciously applied concepts and theories are made explicit and captured during the requirements/analysis stage of the process. They are then converted into a collection of constructs and connections that can be precisely specified and implemented during the design/implementation stage. However, the process of creating a useful analysis model and the transformation of that analysis model into a design model is quite complex. The solution embodied in the analysis model is the developer's theory for explaining the problem and that theory is not completely specified until it is implemented in code. However, as chapter five noted, it may take a long period of interacting with the problem before a satisfactory explanatory theory can be generated that cannot be falsified.

Indeed, it may not be until the design is in the implementation stage that anomalies between the requirements and the explanatory theory become apparent. However, when successfully utilised collections of concepts and theories have been used to capture the understanding of a problem, and those concepts and theories are known to implementable in the constructs of software and hardware, they can be made explicit for use by other developers. Moreover, those concepts and relationships can be represented at a higher level of abstraction – as superordinate level concepts (see Rosch in chapter 5) – to make them applicable to analogous problem situations. Software patterns appear to provide a format for capturing those superordinate level concepts and relationships. They do not capture naturally occurring aspects of an objective reality. They capture successfully used ways of understanding a subjective reality that are known to be implementable in software and hardware constructs. To reiterate, people naturally explain the situations they encounter in terms of concepts and theories they have used before and modify those concepts and theories according to the new context. Software patterns make explicit and capture aspects of the natural thought processes of human understanding. In that sense, the design patterns used in software development are similar to the structures that the philosopher Hanson termed our 'patterns of discovery' (Hanson 1965).

However, additional research is necessary to explore this conjecture. As Alexander notes, "It is a nice and useful format. It allows you to write down good ideas about software design in a way that can be discussed, shared, modified, and so forth. So, it is a really useful vehicle for communication." The formalisms used to pass on patterns between developers may be similar to the formalisms used to communicate patterns in Alexander's approach to architecture design. However, the understanding of why patterns work is similar to the way patterns are discussed in philosophy. Therefore, philosophy may also reveal further ideas for how they can be improved.

The last conjecture to be made concerns software architecture, specifically software architecture views. Chapter four of this thesis examined software architecture in detail and identified many anomalies between the practice of software architecture and the theories provided by software architecture research. Case studies in software architecture have shown that many high-level design representations are created during the development process. Existing software architecture research suggests those different representations are analogous to the many different architecture representations used during the development of traditionally engineered artefacts. Those views provide

different abstractions that capture useful subsets of the underlying implementation detail. However, chapter 4 also identified many anomalies between software architecture, traditional architecture, and their different representations. First, software systems have no form that is analogous to the form of traditionally built artefacts. Second, a computer must execute a software system in order to realise its intended purpose. Third, different high-level software representations do not capture subsets of the underlying implementation detail the way traditional architecture views do.

Those anomalies exist because of the prevailing influence of the artefact engineering view of software development. However, they can be explained by changing to a model building view. During the software development process, the designer must create an initial conceptual model that makes explicit the concepts and relationships, the explanatory theory, which the designer believes explains the problem. That conceptual model can consist of many different types of concepts and relationships, at many different levels of generality, and are limited only by the designer's experience and imagination. However, to implement that conceptual model, the collection of concepts and relationships must be transformed into a collection of constructs and connections provided by the implementation medium. Those constructs and connections may have the same labels as the concepts and relationships in the conceptual model, however the philosophical foundations of the model building view show that one is not simply a refinement of the other. The types of concepts and relationships are fundamentally different. As stated previously, one set of concepts is the fuzzy, theory-dependent concepts used in human understanding ($concepts_1$). The other set of concepts is the formally specified, context-independent concepts of software implementation ($concepts_2$). Finally, to realise the required system, a computer must execute the implemented constructs. That implementation can exist across many different machines, many different processes, and may include many different instantiations of the one software system.

Therefore, the model building view of software development suggests three different types of high-level system representation are required during the development process. Those different types of representation are not different abstractions, or subsets, of the complex implementation detail. They are fundamentally different and are required because of the unique nature of software systems. First, representations of the conceptual model are required. These are produced as the initial step in the design process and represent the model that is to be implemented as a solution to the problem. They consist

of the concepts and relationships that constitute the designer's explanatory theory for the problem. Second, representations of the static implementation are required. These depict the implementation of the system in terms of software and hardware constructs and their dependencies. They represent the structural form of the implemented system but do not contain enough explicit information to depict the control flow through the executing system. While the constructs in the static implementation may appear similar to the concepts in the conceptual model representations, they are fundamentally different and one is not merely a refinement of the other. Third, representations are required to represent the dynamic operation of the system. These depict the behaviour of the system and may consist of concepts from the conceptual model, concepts from the static implementation model, concepts used by the computer in the execution of the system, and concepts depicted to the user such as user interface constructs.

Each of these types of high-level system representation are fundamentally different and those differences can only be satisfactorily explained by rejecting the prevailing artefact engineering view of software development and accepting a model building view. This conjecture was first presented in (Baragry and Reed 1998).

Other areas of future research may include analysing the historical arguments that were used to advocate particular development methodologies in terms of the philosophical foundations presented. Furthermore, future research could determine how the philosophical foundations fit in the slowly evolving 'Software Engineering Body of Knowledge' (Bourque, Dupuis et al. 1999). And finally, the experimental procedures in cognitive psychology that were used to determine the theory-dependent nature of human observation could be applied, with suitable modification, to elucidate how software engineers perform system analysis.

This thesis has examined the philosophical and psychological foundations of software development and has used the subsequent analyses to explain difficulties with what may be considered the conventional wisdom of software engineering research. Dealing with these issues the thesis will be contentious. The criticisms faced by new ways of understanding phenomena in the research of a discipline have been detailed in the history and philosophy of science and were documented in the previous chapter. Therefore, despite the detailed analysis and critique of the conventional, artefact engineering view of

software development and the explication of an alternative model building view, which is based on philosophical and psychological foundations, it is reasonable to expect that this thesis will face criticisms. Some of them may be well founded due to errors on my part. Others will simply be because the ideas are different. Those criticisms have been exemplified in comments I have received from software engineering researchers during conversations and in reviews of articles submitted for publication. Again, some of them were well founded and assisted my own understanding of the issues while others were simply based on an innate refusal to identify and accept the influence of guiding assumptions on software engineering research. The fact these issues can inspire criticisms based on irrational justification may lead to despair for established researchers who would like to see the discipline move forward. That was exemplified in the comments of Peter Naur and Bruce Blum in the previous chapter. Therefore, a few comments are warranted concerning the implications of this thesis.

A review by Suchman of Terry Winograd's proposal for new way of understanding cognition and software systems captures some of the issues.

> "There are two kinds of books in the world. One the one hand, there are those books that fall neatly into a particular intellectual tradition, to which they contribute some development, clarification, or revision or received ideas. For such books, the critical question is what is their thesis, and how well do they succeed in its exposition. On the other hand, there are those books, which tend to come along less often, that aim to challenge the basic soundness of received ideas, and to propose radical alternatives. *Understanding Computers and Cognition* aims to be this second kind; namely, a radical book that should be read as such.
>
> Taken as a radical book, the question to ask about *Understanding Computers and Cognition*, beyond how well it succeeds in its arguments, is whether those arguments are about something important." (Stefik 1987)

This thesis also falls into that second category. It is not a threat to established research agendas, though some may view it that way. It has examined and made explicit the fundamental nature of what software systems are. It has not, apart from a few conjectures, proposed how software development or software engineering should be performed. The goal has been to provide a foundation for explaining existing research anomalies, which

can then be used as justification for or against proposed theories in future work. Moreover, the thesis does not claim that software cannot be engineered, though some may also view it that way. The goal must now be to determine how to engineer software as explanatory theories of the world rather than as another built artefact. That goal, however, may in fact require fundamental changes in research agendas. Furthermore, it does not advocate that all software engineers must have academic training in philosophy or need to understand Kant's *Critique of Pure Reason* (Kant 1933) in addition to software development texts. There exist many mainstream books that popularise the philosophical theories of the foundational issues. For example, Pirsig's *Zen and the Art of Motorcycle Maintenance* (Pirsig 1974) and Gaarder's *Sophie's World* (Gaarder 1996). Software engineering practitioners should have some idea of these issues, however the specifics of this thesis are aimed at software engineering researchers. Indeed, practitioners may never be consciously aware of these issues even though researchers must become far more informed in these areas.

> "Plato, who formulated this analysis of understanding in Euthyphro, goes on to ask in Meno whether the rules required to make behavior intelligible to the philosopher are necessarily followed by the person who exhibits the behavior. That is, are rules only necessary if the philosopher is to *understand* what is going on, or are those rules necessarily followed by the person insofar as he is able to behave intelligently? ... In the case of theorem proving ... Plato thought that although people acted without necessarily being aware of any rules, their action did have a rational structure which could be explicated by the philosopher." (Dreyfus 1992) (p.176)

As a final comment, I return to the Einstein's quote that opened this thesis.

> "If we knew what it was we were doing, it wouldn't be called research, would it?"

That understanding of research as proposing explanatory conjectures that may or may not be correct, applies equally well to the explicitly stated, specific theories of software engineering as it does to our guiding assumptions that may not be explicitly understood but which also, may or may not be correct. As to whether or not this work constitutes software engineering research, this thesis finishes with a comment by John of Salisbury, who wrote in 1159 in his book, *Policratus*,

"Who is more contemptible than he who scorns knowledge of himself?" (in (Saul 1997)).

8. Bibliography

Abel, C. (1981). Vico and Herder: The Origins of Methodological Pluralism. Design: Science: Method. Proceedings of the 1980 Design Research Society Conference. R. Jacques and J. A. Powell (eds), Westbury House.

Adelson, B. and E. Soloway (1985). "The Role of Domain Experience in Software Design." IEEE Transactions on Software Engineering **SE-11**(11): pp. 1351-1360.

Alberts, L. K. (1993). YMIR: An Ontology for Engineering Design. Doctoral Thesis. University of Twente.

Alberts, L. K., P. M. Wognum, et al. (1991). Design as Science instead of Art. Technical Report February 1991 Memoranda Informatica 91-10, University of Twente, The Netherlands.

Alexander, C. (1964). Notes on the Synthesis of Form, Harvard University Press.

Alexander, C. (1999). "The Origins of Pattern Theory: the future of the theory and the generation of a living world." IEEE Software(September/October): pp. 71-82.

Ambler, A. L., M. M. Burnett, et al. (1992). "Operational Versus Definitional: A Perspective on Programming Paradigms." Computer(September): pp. 28-43.

Amdahl (1998). "ObjectStar Software Solutions.", http://www.amdahl.com/objectstar/ September 1998.

Appelbe, B. and G. Abowd (1995). "Beyond Objects: A response." Software Engineering Notes **20**(3): pp. 45-48.

Atlee, J. M. and J. Gannon (1993). "State-Based Model Checking of Event-Driven System Requirements." IEEE Trans on Software Engineering **19**(1).

Awad, M., J. Kuusela, et al. (1996). Object Oriented Technology for Real-Time Systems: a practical approach using OMT and Fusion, Prentice Hall.

Baber, R. L. (1997). "Comparison of Electrical "Engineering" of Heaviside's Times and Software "Engineering" of Our Times." IEEE Annals of the History of Computing **19**(4): pp. 5-17.

Baragry, J. (1996). An Initial Comparison of Software and Engineering Designs of Automotive Cruise Control Systems. Proceedings Australian Software Engineering Conference, Melbourne, Australia, IEEE Computer Society Press.

Baragry, J., D. Cleary, et al. (1994). Unix Inter-component Communication within a Hypertext Based CASE Tool. Proceedings Australian Unix User Group Conference, Melbourne.

Baragry, J. and K. Reed (1998). Why Is It So Hard To Define Software Architecture? Proceedings Asia Pacific Software Engineering Conference, Tapei, Taiwan.

Barrett, W. F., W. Brown, et al. (1902). "Researches on the Electrical Conductivity and Magnetic Principles of Upwards of One Hundred Different Alloys of Iron." Institution of Electrical Engineers **XXXI**(Feb 13th).

Bass, L., P. Clements, et al. (1998). Software Architecture in Practice, Addison-Wesley.

Bass, L. and R. Kazman (1999). Architecture-Based Development. Technical Report April 1999 CMU/SEI-99-TR-007, Software Engineering Institute.

Bechtel, W. (1988a). Philosophy of Mind: an overview for cognitive science, Erlbaum Associates.

Bechtel, W. (1988b). Philosophy of Science: an overview for cognitive science, Erlbaum Associates.

Belady, L. (1989). Forward. Software Reusability: Concepts and Models. T. J. Biggerstaff. and A. J. Perlis (eds), ACM Press.

Belevitch, V. (1962). "Summary of the History of Circuit Theory." Proceedings of the IRE **50**(5): pp. 848-855.

Beltrami, S. (1998). Automobile Design. Personal Communication with Jason Baragry.

Bennett, D. (1997). Desiging Hard Software: the essential tasks, Manning Publications.

Biggerstaff, T. J. and A. J. Perlis (1989). Preface. Software Reusability: Concepts and Models. T. J. Biggerstaff and A. J. Perlis (eds), ACM Press.

Birchenough, A. and J. R. Cameron (1989). JSD and Object Orientated Design. JSP and JSD: The Jackson Approach to Software Development. J. R. Cameron (ed), IEEE Computer Society Press.

Blum, B. I. (1985). "Understanding the Software Paradox." ACM SigSoft Software Engineering Notes **10**(1): pp. 43-47.

Blum, B. I. (1993). "The Economics of adaptive Design." Journal of Systems and Software **21**(2): pp. 117-128.

Blum, B. I. (1994). "A Taxonomy of Software Development Methods." Communications of the ACM **37**(11): pp. 82-94.

Blum, B. I. (1996). Beyond Programming: To A New Era Of Design, Oxford University Press.

Boeing (1998). 777 Computing Design Facts. http://www.boeing.com/commercial/777family/cdfacts.html. March 1998.

Booch, G. (1986). "Object-Orientated Development." IEEE Trans on Software Engineering **SE-12**(2).

Booch, G. (1991). Object Orientated Design with Applications, Benjamin / Cumming Pub Co.

Bourque, P., R. Dupuis, et al. (1999). "The Guide to the Software Engineering Body of Knowledge." IEEE Software(November/December): pp. 35-44.

Bowman, I. T., R. C. Holt, et al. (1999). Linux as a Case Study: Its Extracted Software Architecture. Proceedings ICSE, Los Angeles, California, IEEE Computer Society Press.

Brackett, J. (1987). Automobile Cruise Control and Monitoring System Example. Technical Report TR 87-07, Wang Inst of Graduate Studies, Boston Univ.

Braha, D. and O. Maimon (1997). "The Design Process: Properties, Paradigms, and Structure." IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans **27**(2): pp. 146-166.

Briggs, M. S. (1959). Everyman's Concise Encyclopedia of Architecture.

Broadbent, G. (1981). Design Methods - 13 Years After - A Review. Design: Science: Method. Proceedings of the 1980 Design Research Society Conference. R. Jacques and J. A. Powell (eds), Westbury House.

Brooks, F. P. (1962). Architectural Philosophy. Planning a Computer System - Project Stretch. W. Buchholz (eds), McGraw-Hill**:** pp. 5-16.

Brooks, F. P. (1975). The Mythical Man-Month: Essays in Software Engineering, Addison-Wesley Publishing.

Brooks, F. P. (1987). "No Silver Bullet." IEEE Computer **20**(4): pp. 10-19.

Brothers, J. T. (1962). "Historical Development of Component Parts Field." Proceedings of the IRE. **50**(5): pp. 912-920.

Brown, A. W. and K. C. Wallnau (1998). "The Current State of CBSE." IEEE Software(Sept/Oct).

Bruegge, B. and A. H. Dutoit (1999). Object-Oriented Software Engineering: Conquering Complex and Changing Systems, Prentice Hall.

Bruner, J. S. (1958). On Perceptual Readiness. Readings In Perception. D. S. Beardslee and M. Wertheimer (eds)**:** pp. 686-729.

Bunge, M. (1973). Method, Model, and Matter.

Capra, F. (1983). The Tao of Physics. Second Ed.

Carmichael, L., H. P. Hogan, et al. (1932). "An Expiremental Study Of The Effect Of Language On The Reproduction Of Visually Perceived Form." Journal of Experimental Psychology **44**: pp. 163-174.

Caromel, D. (1993). "Towards a Method of Object-Oriented Concurrent Programming." Communications of the ACM **36**(9): pp. 90-102.

Castano, S. and V. DeAntonellis (1993). A Constructive Approach to Reuse of Conceptual Components. Proceedings 2nd Intl Workshop on Software Reusability, Lucca, Italy.

Cleary, D. (1997). Design Issues of the HyperCase ProTract Tool. Personal Communication with Jason Baragry.

Cleary, D. and K. Reed (1993). PROTRACT: A Computerised Tool for Tracking Software Development using Event Based Activity Specifications. Proceedings Australian Software Engineering Conference.

Clements, P. C. (1996). Software Architecture: An Executive Overview. Technical Report February CMU/SEI-96-TR-003, Software Engineering Institute.

Cockburn, A. (1996). "The Interaction of Social Issues and Software Architecture." Communications of the ACM **39**(10): pp. 40-46.

Cooper, M. (1996). A Navigation Support System for the Amdahl Australian Intelligent Tools Program Integrated CASE System. Masters Thesis. La Trobe University.

Coplien, J. (1999a). Architecture as Metaphor. http://www.bell-labs.com/~cope/ArchitectureAsMetaphor.html. March 2000.

Coplien, J. O. (1999b). "Reevaluating the Architectural Metaphor: Toward Piecemeal Growth." IEEE Software(Sept/Oct).

Corsini, R. e. (1984). Encyclopedia of Psychology. New York, NY Wiley.

Cox, B. (1991). Planning the Software Industrial Revolution. Supply-side Economic of Software Reuse. Proceedings International Workshop on Software Reuse. http://rbse.jse.gov/eicherman/wisr/wisr.html.

Cox, B. (1992). "Superdistribution and Electronic Objects." Dr. Dobb's Journal(October).

Cox, B. J. (1990). "Planning the Software Industrial Revolution." IEEE Software(November): pp. 25-33.

Cross, N., J. Naughton, et al. (1981). Design Method and Scientific Method. Design: Science: Method. Proceedings of the 1980 Design Research Society Conference. R. Jacques and J. A. Powell (eds), Westbury House.

Cumming, G. (2000). Conversations on the application of theories in psychology to software engineering research. Personal Communication with Jason Baragry.

Curl, J. S. (1993). Encyclopedia of Architectural Terms.

Currie, B. and R. A. Sharpe (1990). Structural Design, Stanley Thornes.

Curtis, B. (1989). Cognitive Issues in Reusing Software Artifacts. Software Reusability: Applications and Experience. T. J. Biggerstaff. and. A. J. Perlis (eds), ACM Press.

Curtis, B., H. Krasner, et al. (1991). "A Field Study of the Software Design Process for Large Systems." Communications of the ACM **31**(11): pp. 1268-1288.

Cybernetica, P. ASC Glossary: Model. Online: http://pespmc1.vub.ac.be/ASC/MODEL/html, Principia Cybernetica Web.

Cybulski, J. and A. Proestakis (1991). HyperEDIT: An Object-Oriented Diagram Meta Editor. Technical Report June. 1991 TR009, AAITP, La Trobe University.

Cybulski, J. L., R. D. Neal, et al. (1998). "Reuse of Early Life-Cycle Artifacts: workproducts, methods, and tools." Annals of Software Engineering **5**: pp. 227-251.

Cybulski, J. L. and K. Reed (1992). "A Hypertext Based Software Engineering Environment." IEEE Software(March): pp. 62-68.

Dahlbom, B. and L. Mathiassen (1993). Computers in Context: the philosophy and practice of systems design, Blackwell.

D'Alessandro, M., P. L. Iachini, et al. (1993). The Generic Reusable Component: an Approach to Reuse Hierarchical OO Designs. Proceedings 2nd Intl Workshop on Software Reusability, Lucca, Italy.

Darnell, P. S. (1958). "History, Present Status, and Future Development of Electronic Components." IRE Transactions on Component Parts **CP-5**(3 (Sept)).

Darnell, P. S. (1962). "Future of the Component Parts Field." Proceedings of the IRE **50**(5).

D'Azzo, J. J. and C. H. Houpis (1988). Linear Control System Analysis and Design, McGraw-Hill. Third Ed.

Denning, P. J., D. E. Comer, et al. (1989). "Computing as a Discipline. (Final report of the Task Force on the Core of Computer Science)." Communications of the ACM **32**(1): pp. 9-23.

DeRemer, F. and H. H. Kron (1976). "Programming-in-the-Large Versus Programming-in-the-small." IEEE Transactions on Software Engineering(June).

Dijkstra, E. W. (1968). "The Structure of the "THE" - Multiprogramming System." Communications of the ACM **11**(5): pp. 341-346.

Dillon, T. and P. L. Tan (1993). Object-oriented conceptual modeling, Prentice-Hall.

D'Ippolito, R. S. and K. Lee (1992a). Modelling Software Systems by Domain. Proceedings AAAI-92 Workshop on Automating Software Design.

D'Ippolito, R. S. and K. Lee (1992b). Putting the Engineering into Software Engineering. Tutorial presented at the SEI Conference on Software Engineering Education..

D'Ippolito, R. S. and C. P. Plinta (1989). Software Development using Models. Proceedings Fifth International Workshop on Software Specification and Design.

Doble, J. (1997). Change Resilience Patterns in Software Architecture & Design. Proceedings OOPSLA '97 Workshop: Exploring Large System Issues, http://www.ccsi/~brr/doble.html.

Dreyfus, H. L. (1992). What Computers Still Can't Do: A Critique of Artificial Reason, MIT Press.

Dumas, J. and P. Parsons (1995). "Discovering the Way Programmers Think About New Programming Environments." Communications of the ACM **38**(6).

Dummer, G. W. A. (1956). Fixed Resistors, Sir Isaac Putman & Sons.

Ellinger, H. E. (1985). Automotive Electrical Systems, Prentice Hall.

Feyerabend, P. K. (1979). Against Method.

Fieser, E. J. (1999). The Internet Encyclopedia of Philosophy. http://www.utm.edu/research/iep/. December 1999.

Fivush, R. (1987). Scripts and categories: interrelationships in development. Concepts and Conceptual Development. U. Neisser (ed), Cambridge University Press**:** pp. 234-254.

Flohr, U. (1995). "Hyper-G Organizes the Web." Byte magazine(November).

Fox, J. (2000). Conversations on the application of theories in philosophy to software engineering research. Personal Communication with the Jason Baragry.

Fugini, M. G. and S. Faustle (1993). Retrieval of Reusable Components in a Development Information System. Proceedings 2nd Intl Workshop on Software Reusability, Lucca, Italy.

Fugini, M. G., O. Nierstrasz, et al. (1992). "Application Development through Reuse: the Ithaca Tools Environment." ACM SIGOIS Bulletin **13**(2): pp. 38-47.

Gaarder, J. (1996). Sophie's World: a novel about the history of philosophy. London, Pheonix., P. Moller (trans).

Gallagher, K. (1997). The Nature of Software Engineering. Personal Communication with Jason Baragry.

Gamma, E., R. Helm, et al. (1994). Design Patterns: elements of reusable object-orientated software, Addison-Wesley.

Garlan, D., R. Allen, et al. (1995). Architecural Mismatch or Why it's hard to build systems out of existing parts. Proceedings ICSE'17, Seattle, Washington.

Garlan, D. and D. E. Perry (1995). "Introduction to the Special Issue on Software Architecture." IEEE Transactions on Software Engineering **21**(4).

Garlan, D. and M. Shaw (1993). An Introduction to Software Architecture. Advances in Software Engineering and Knowledge Engineering. V. Ambriola (ed), World Scientific.

Gelernter, M. (1995). Sources of Architectural Form: a critical history of Western design theory, Manchester University Press.

Gelman, S. A. (1996). Concepts and Theories. Perceptual and Cognitive Development. R. Gelman and T. K.-F. Au (eds). London, Academic Press.

Germann, S. and R. Isermann (1994). Modelling and control of longitudinal vehicle motion. Proceedings American Control Conference, Baltimore, Maryland.

Giancoli, D. C. (1988). Physics for Scientists and Engineers, Prentice-Hall International. 2nd Ed.

Giesecke, F. E., A. Mitchell, et al. (1974). Technical Drawing.

Gilb, T. (1996). "Level 6: Why We Can't Get There From Here." IEEE Software(January).

Glass, R. L. (1994). "The Software Research Crisis." IEEE Software(November): pp. 42-47.

Glass, R. L. (1998a). "Reuse: What's Wrong with This Picture." IEEE Software **15**(2): pp. 57-59.

Glass, R. L. (1998b). "Success? Failure? or Both?" IEEE Computer(May): pp. 103.

Gomaa, H. (1989). Structuring Criteria for Real Time System Design. Proceedings International Conference on Software Engineering, Pittsburgh, USA, IEEE Computer Society Press.

Gomaa, H. (1993). Software Design Methods for Concurrent and Real-Time Systems, Addison-Wesley.

Gray, H., et al (1995). Gray's Anatomy. Churchill Livingstone. 38th Ed.

Gray, P. E. (1969). Electronic Principles: Physics, Models, and Circuits, John Wiley & Sons.

Gray, P. R. and R. G. Meyer (1984). Analysis and Design of Analog Integrated Circuits, John Wiley & Sons. Second Ed.

Hanson, N. R. (1965). Patterns Of Discovery: an inquiry into the conceptual foundations of science. Cambridge, University Press.

Harel, D. (1992). "Biting the Silver Bullet: Toward a brighter future for System Development." IEEE Computer. (January).

Harel, D. and e. al (1990). "STATEMATE: A Working Environment for the Development of Complex Reactive Systems." IEEE Transactions on Software Engineering **16**(4): pp. 403-413.

Hatton, L. (1998). "Does OO Sync with How We Think?" IEEE Software(May/June): pp. 46-54.

Higgins, D. A. (1987). Specifying Real-Time/Embedded Systems using Feedback Control Models. Proceedings SMC XII: Twelfth Structured Methods Conference.

Hill, D. W. (1975). "From Torpedo to Telemetry." Electronics and Power(November): pp. 1110.

Hirschheim, R. and H. K. Klein (1989). "Four paradigms of information systems development." Communications of the ACM **32**(10): pp. 1199(18).

Hoare, C. A. R. (1969). "An Axiomatic Basis for Computer Programming." Communications of the ACM **12**(10): pp. 576-580,583.

Horowitz, P. and W. Hill (1989). The Art of Electronics, Cambridge University Press. 2nd Edition.

IEEE, A. W. G. and S. E. S. Committee (1998). Draft Recommended Practice for Architectural Description. Draft Standard December IEEE P1471/D4.1, IEEE.

Inverardi, P. and A. L. Wolf (1995). "Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model." IEEE Transactions of Software Engineering **21**(4).

Ioannou, P., Z. Xu, et al. (1993). Intelligent Cruise Control: Theory and Experiment. Proceedings 32nd Conference on Decision and Control, San Antonio, Texas, IEEE Press.

Ioannou, P. A. and C. C. Chen (1993). "Autonomous Intelligent Cruise Control." IEEE Transactions on Vehicular Technology **42**(4): pp. 657-672.

Jackson, M. (1995). The World and the Machine. Proceedings ICSE 17, Seattle, Washington.

Jackson, M. (1998a). "Defining a Discipline of Description." IEEE Software(Sept/Oct 98).

Jackson, M. (1998b). "Will There Ever Be Software Engineering?" IEEE Software(Jan-Feb): pp. 36-39.

Jacobson, I., G. Booch, et al. (1998). The Unified Software Developent Process, Addison Wesley Longman.

Jacques, R. (1981). Introduction. Design: Science: Method. Proceedings of the 1980 Design Research Society Conference. R. Jacques and J. A. Powell (eds), Westbury House.

Jones, D.-W. (1994). "Cruising With Ada." Embedded Systems Programming(November): pp. 18-44.

Jungnickel, C. and R. McCormmach (1986). Intellectual Mastery of Nature: theoretical physics from Ohm to Einstein.

Kaindl, H. (1999). "Difficulties in the Transition from OO Analysis to Design." IEEE Software(September/October).

Kant, I. (1933). Critique of Pure Reason. London, Macmillan., N. K. Smith (trans).

Kazman, R., L. Bass, et al. (1994). SAAM: A Method for Analyzing the Properties of Software Architectures. Proceedings ICSE, Sorrento, Italy, IEEE Computer Society Press.

Keil, F. C. (1987). Conceptual development and category structure. Concepts and Conceptual Development. U. Neisser (ed), Cambidge University Press**:** pp. 175-200.

Kellman, P. J. (1996). The Origins of Object Perception. Perceptual and Cognitive Development. R. Gelman and T. K.-F. Au (eds). London, Academic Pres.

Kemerling, G. (1997a). History of Western Philosophy. http://people.delphi.com/gkemerling/hy. March 1999.

Kemerling, G. (1997b). Definition and Meaning. http://people.dephi.com/gkemerling/e05.htm. October 1999.

Knudtson, P. and D. Suzuki (1992). Wisdom of the Elders, Allen & Unwin.

Kogut, P. (1994). Design Reuse: Chemical Engineering vs. Software Engineering. SARG Presentation 21 March 1994, Software Engineering Institute. Centre for Reusable Defense Software (CARDS).

Kogut, P. (1995). "Design Reuse: Chemical Engineering vs. Software Engineering." ACM SigSoft Software Engineering Notes **20**(5).

Koning, J. (1984). "Cruise Control For Cars." Electronics Australia(June): pp. 44-54.

Kostof, S. (1986). The Architect: chapters in the history of the profession, Oxford University Press.

Kozaczynski, W. and G. Booch (1998). "Component Based Software Engineering: Guest Editors' Introduction." IEEE Software(Sept/Oct).

Kruchten, P. (1995). "Architectural Blueprints - The "4+1" View Model of Software Architecture." IEEE Software(November).

Krueger, C. W. (1992). "Software Reuse." ACM Computing Surveys **24**(2): pp. 131-183.

Kruft, H.-W. (1994). A History of Archiectural Theory: from Vitruvius to the present, Zwemmer., E. C. Ronald Taylor, and Antony Wood (trans).

Kuhn, T. (1962). The Structure of Scientific Revolutions, University of Chicago Press.

Kuhn, T. S. (1977a). A Function for Thought Experiments. The Essential Tension, University of Chicago Press.

Kuhn, T. S. (1977b). The Function of Measurement in Modern Physical Science. The Essential Tension, University of Chicago Press.

Kuhn, T. S. (1977c). The History of Science. The Essential Tension, University of Chicago Press.

Kuhn, T. S. (1977d). Second Thoughts on Paradigms. The Essential Tension, University of Chicago Press.

Kumagai, A. (1998). Oriental Philosophy and Software Engineering. Position Paper for Asia Pacific Forum on Software Engineering. Proceedings ICSE '98, Kyoto Japan.

Lakoff, G. (1987). Cognitive models and prototype theory. Concepts and Conceptual Development. U. Neisser (ed), Cambridge University Press**:** pp. 63-100.

Larman, C. (1997). Applying UML and Patterns: an introduction to object-oriented analysis and design.

Last, R. J. (1978). Anatomy, regional and applied, Churchill Livingstone. 6th Ed.

Laudan, L., A. Donovan, et al. (1986). "Scientific Change: philosophical models and historical research." Synthese(69): pp. 141-223.

Laudon, K. C. and J. P. Laudon (1996). Information, Management, and Decision Making. Management Information Systems: New Approaches to Organization & Technology, Prentice Hall International**:** pp. 116-147.

Lawson, B. (1980). How Designers Think. London, The Architectural Press Ltd.

Lawson, H. W. (1990). "Philosophies for Engineering Computer-Based Systems." IEEE Computer **23**(12): pp. 52-63.

Lee, B. J., Y. W. Kim, et al. (1993). Engine Throttle Control Using Anticipatory Band in the Sliding Phase Plane. Proceedings American Control Conference, San Francisco, California.

Lee, H. N. (1973). Percepts, Concepts and Theoretical Knowledge, Memphis State University Press.

Lee, K. and A. Karmiloff-Smith (1996). The Development of External Symbol Systems: The Child as a Notator. Perceptual and Cognitive Development. R. Gelman and T. K.-F. Au (eds). London, Academic Press.

Lehman, M. M. (1980). "Programs, Life Cycles, and Laws of Software Evolution." Proceedings of the IEEE **68**(9): pp. 1060 - 1076.

Leveson, N. G. (1992). High-Pressure Steam Engines and Computer Software. Proceedings International Conference on Software Engineering, World Congress Centre, Melbourne Australia.

Levi-Strauss, C. (1962). The Savage Mind. London, Weidenfeld and Nicholson.

Levi-Strauss, C. (1986). Structural Analysis in Linguistics and in Anthropology. Semiotics: an introductory anthology. R. E. Innis (ed).

Lewis, T. G. and P. W. Omen (1990). "The Challenge of Software Development." IEEE Software(November): pp. 9-12.

Linden, F. J. v. d. and J. K. Muller (1995). "Creating Architectures with Building Blocks." IEEE Software(Novermber): pp. 51-60.

Liubakka, M. K., D. S. Rhode, et al. (1994). Adaptive Automotive Speed Control. Proceedings Proceedings of Workshop on Advances in Control and its Applications, Springer-Verlag.

Lowry, M. R. (1992). "Software Engineering in the Twenty-First Century." AI Magazine (Fall**)**: pp. 71-87.

Maibaum, T. (1997). "What We Teach Software Engineers in the University: Do We Take *Engineering* Seriously?" Software Engineering Notes **22**(6): pp. 40-50.

Marciniak, J. J., Ed. (eds). (1994). Encyclopedia of Software Engineering. New York, John Wiley.

Marco, A. (1990). Software Engineering: Concepts and Management, Prentice-Hall.

Marsten, J. (1962). "Resistors - A Survey of the Evolution of the Field." Proceedings of the IRE(May): pp. 920.

Martin, J. and J. Odell (1995). Object-oriented Methods: A Foundation, PTR Prentice Hall.

McCauley, R. N. (1987). The role of theories in a theory of concepts. Concepts and Conceptual Development. U. Neisser (ed), Cambridge University Press**:** pp. 288-309.

McConnell, S. (2000). "The Best Influences on Software Engineering." IEEE Software(January/February).

McIlroy, M. D. (1968). Mass Produced Software Components. Software Engineering Concepts and Techniques: Proceedings of the NATO Conferences.

Meddin, D. L. and W. D. Wattenmaker (1987). Category cohesiveness, theories, and cognitive archeology. Concepts and Conceptual Development. U. Neisser (ed), Cambridge Unversity Press**:** pp. 25-62.

Mehra, R. K. and K. Baheti (1995). Introduction and Motivation for the Real-Life Control Design Challenge Problem Session. Proceedings 34th Conference on Decision & Control, New Orlean, LA.

Mellor, S. J. and R. Johnson (1997). "Why Explore Object Methods, Patterns, and Architectures?" IEEE Software(January): pp. 27-30.

Meyer, B. (1997). Object-Orientated Software Construction, Prentice Hall. 2nd Ed.

Mili, A., E. Addy, et al. (1999). "Toward an Engineering Discipline of Software Reuse." IEEE Software(September/October).

Miriam-Webster Dictionary. (1997). http://www.m-w.com/netdict.htm.

Mitchell, P. (1996). Beyond Programming: To a New Era of Design. 16th August 1996. http://www.ercb.com/ddj/1996/ddj.9608.html. May 1998.

Mobray, T. J. (1998). "Will the Real Architecture Please Sit Down?" Component Strategies(December).

Muller, R. and G. Nocker (1994). Intelligent Cruise Control with Fuzzy Logic. Fuzzy Logic Control and Applications. M. J. Roberts (ed), IEEE Press**:** pp. 74-80.

Nakamura, K., T. Ochiai, et al. (1983). "Application of Microprocessor to Cruise-Control System." IEEE Transactions on Industrial Electronics **IE-30**(2): pp. 108-113.

NATO (1976a). Report on a Conference Sponsored by the NATO Science Committee. Garmisch Germany, Oct 7-11 1968. Software Engineering Concepts and Techniques: proceedings of the NATO conferences. P. Naur and B. Randell (eds), Mason/Charter.

NATO (1976b). Report on a Conference Sponsored by the NATO Science Committee, Rome, Italy Oct 27-31, 1969. Software Engineering Concepts and Techniques: Proceedings of the NATO confereces. J. N. Bruxton and B. Randall (eds), Petrochelli/Charter.

Naur, P. (1985). "Programming as Theory Building." Microprocessing and Microprogramming **15**(5 (May)): pp. 253-261.

Naur, P. (1992a). Computing: a human activity, ACM Press, Addison-Wesley.

Naur, P. (1992b). Computing and the So-Called Foundations of the So-Called Sciences (1990). Computing: a human activity, ACM Press, Addison-Wesley**:** pp. 49-63.

Naur, P. (1992c). Data and their Applications (1974). Computing: a human activity, ACM Press, Addison-Wesley**:** pp. 9-22.

Naur, P. (1992d). Datalogy and Datamatics and Their Place in Education (1966). Computing: a human activity, ACM Press, Addison-Wesley**:** pp. 175-180.

Naur, P. (1992e). Human Knowing, Language, and Discrete Structures (1989). Computing: a human activity, ACM Press, Addison-Wesley**:** pp. 518-535.

Naur, P. (1992f). Intuition in Software Development (1985). Computing: a human activity, ACM Press, Addison-Wesley**:** pp. 449-466.

Naur, P. (1992g). The Place of Strictly Defined Notation in Human Insight (1989). Computing: a human activity, ACM Press, Addison-Wesley**:** pp. 468-478.

Naur, P. (1992h). Programming Languages are not Languages - Why 'Programming Language' is a Misleading Designation (1988). Computing: a human activity, ACM Press, Addison-Wesley**:** pp. 503-510.

Neisser, U. (1987a). The ecological and itellectual bases of categorization. Concepts and Conceptual Development. U. Neisser (eds), Cambridge University Press**:** pp. 1-11.

Neisser, U. (1987b). From Direct Perception to Conceptual Structure. Concepts and Conceptual Development. U. Neisser (ed), Cambridge University Press**:** pp. 11-23.

Nierstrasz, O., S. Gibbs, et al. (1992). "Component-Oriented Software Development." Comunications of the ACM (September**)**: pp. 160-165.

Nierstrasz, O. and T. D. Meijler (1995). "Research Directions in Software Composition." ACM Computing Surveys **27**(2).

Nierstrasz, O., D. Tsichritzis, et al. (1991). Objects + Scripts = Applications. Proceedings Esprit, Kluwer Academic Publishers.

Northrop, L. M. and W. E. Richardson (1991). Design Evolution: Implications for Academia and Industry. Proceedings SEI Conference on Software Engineering Education, Pittsburgh, Pennsylvania, USA, Springer-Verlag.

Oda, K., H. Takeuchi, et al. (1991). Practical Estimator for Self-Tuning Automotive Cruise Control. Proceedings American Control Conference.

Oestereich, B. (1999). Developing Software with UML: Object-oriented analysis and design in practice, Addison Wesley Longman.

Parnas, D. L. (1972). "On the Criteria to be Used in Decomposing Systems into Modules." Communications of the ACM(December).

Parnas, D. L. (1997). Softare Engineering: An Unconsummated Marriage. Proceedings Foundations of Software Engineering, Zurich Switzerland.

Parnas, D. L. (1999). "Software Engineering Programs are not Computer Science Programs." IEEE Software(November/December): pp. 19-30.

Patel, D., Y. Wang, et al. (2000). "Comparative Studies of Engineering Approaches for Software Engineering." Annals of Software Engineering **10**.

Perrochon, L. and W. Mann (1999). "Inferred Designs." IEEE Software(Sept/Oct).

Perry, D. E. (1998). Working IFIP Conference on Software Architecture. February 1998. http://www.bell-labs.com/user/dep/prof/wicsa1/. September 1998.

Perry, D. E. and A. L. Wolfe (1992). "Foundations for the Study of Software Architecture." ACM SigSoft **17**(4).

Petroski, H. (1994). Design Paradigms: Case Histories of Error and Judgement in Engineering, Cambridge University Press.

Pevsner, N., J. Fleming, et al. (1975). A Dictionary of Architecture.

Pinker, S. (1997). How the Mind Works. New York, Norton.

Pirsig, R. M. (1974). Zen and the Art of Motorcycle Maintenance, Random House.

Podolsky, L. (1962). "Capacitors." Proceedings of the IRE(May): pp. 924.

Pooley, R. and P. Stevens (1999). Using UML: software engineering with objects and components, Addison Wesley Longman.

Popper, K. R. (1979a). The Aim Of Science. Objective Knowledge: an evolutionary approach, Oxford University Press.

Popper, K. R. (1979b). Conjectural Knowledge: My solution of the problem of induction. Objective Knowledge: an evolutionary approach, Oxford Univeristy Press.

Popper, K. R. (1979c). Epistemology Without A Knowing Subject. Objective Knowledge: an evolutionary approach, Oxford Univeristy Press.

Popper, K. R. (1979d). Evolution And The Tree Of Knowledge. Objective Knowledge: an evolutionary approach, Oxford University Press.

Popper, K. R. (1979e). Of Clouds and Clocks. Objective Knowledge: an evolutionary approach, Oxford University Press.

Popper, K. R. (1979f). On The Theory Of The Objective Mind. Objective Knowledge: an evolutionary approach, Oxford University Press.

Popper, K. R. (1979g). Two Faces Of Common Sense: An arguement for commonsense realism and against the commonsense theory of knowledge. Objective Knowledge: an evolutionary approach, Oxford University Press.

Popper, K. R. (1983). Two Kinds Of Definitions (1945). A Pocket Popper. D. Miller (ed), Fontana.

Powers, B. P. (1976). "Wheatstone's Contribution to Electrical Engineering." Electronics and Power(May): pp. 295-298.

Proestakis, A. (1993). HyperEDIT: Architectural Overview and Distributed Building Process Description. Technical Report June, 1993. TR017, AAITP, La Trobe University.

Reck, E. (1997). Frege's influence on Wittgenstein: Reversing Metaphysics via the Context Principle. Early Analytic Philosophy. W. W. Tait (ed), Open Court**:** pp. 123-185.

Reed, K. (1987). Commercial Software Engineering, The Way Forward. (keynote address). Proceedings Australian Software Engineering Conference, Canberra. ACT. Australia.

Reed, K. (1991). The Impact of Government Policies on Software Engineering: a fact of life and a necessary evil. (keynote address). Tri-Ada 91, San Jose CA. USA.

Reed, K. (2000). Conversations 1992-2000. Personal Communication with Jason Baragry.

Rosch, E. (1978). Principles of Categorization. Cognition and Categorization. E. Rosch and B. B. Lloyd (eds), Lawrence Erlbaum Associates: pp. 27-48.

Rutland, N. K. (1991). A crash course on the application of a new principle of design to vehicle speed control. Proceedings Conference on Decision and Control, Brighton, England.

Rutland, N. K. (1992). "Illustration of a new principle of design: vehicle speed control." Intl. Journal of Control **55**(6): pp. 1319-1334.

Saksena, M., P. Freedman, et al. (1997). Guidelines for Automated Implementation of Executable Object Oriented Models for Real-Time Embedded Control Systems. Proceedings 18th Real-Time Systems Symposium.

Saul, J. R. (1997). The Unconscious Civilisation, The Free Press.

Schokralla, S. H. (1998). Beoing 777 Case Study. March 1998. http://www1.needs.org/develop/b777/main.html. CAD 3D Modelling.

Sedra, A. S. and K. C. Smith (1991). Microelectronic Circuits, Saunders College Publishing. Third International Ed.

SEI (1997). Software Architecture Definitions. http://www.sei.cmu.edu/architecture/definitions.html. September 1998.

Shaout, A. and M. A. Jarrah (1997). "Cruise Control Technology Review." Computers and Electrical Engineering **23**(4): pp. 259-271.

Shaw, M. (1984). "Abstraction Techniques in Modern Programming Languages." IEEE Software(Oct): pp. 10-26.

Shaw, M. (1989). "Large Scale Systems Require Higher-Level Abstraction." Proceedings of Fifth International Workshop on Software Specification and Design, IEEE Computer Society.: pp. 143-146.

Shaw, M. (1990). Prospects for an Engineering Discipline of Software. Technical Report September CMU/SEI-90-TR-20, Carnegie Mellon University, Software Engineering Institute.

Shaw, M. (1994). Making Choices: A Comparison of Styles for Software Architecture. Unpublished Report May, SEI, Carnegie Mellon University.

Shaw, M. (1995a). Architectural Issues in Software Reuse: It's not just the functionality, it's the packaging. Proceedings IEEE Symposium on Software Reusability.

Shaw, M. (1995b). "Beyond Objects: A Software Design Paradigm Based on Process Control." ACM Software Engineering Notes **20**(1).

Shaw, M. (1995c). "Comparing Architectural Design Styles." IEEE Software **12**(6 (Nov)).

Silverman, B. G. (1983). "Analogy in Systems Management: A Theoretical Inquiry." IEEE Transactions on Systems, Man, and Cybernetics **SMC-13** (6): pp. 1049-1075.

Silverman, B. G. (1985). "Software Cost and Productivity Improvements: An Analogical View." IEEE Computer (May): pp. 86-96.

Smith, B. (1995). "Formal Ontology, Common Sense and Cognitive Science." International Journal of Human-Computer Studies(43): pp. 641-647.

Smith, C. U. and J. A. Dallen (1984). Future Directions for VLSI and Software Engineering. VLSI Engineering: Beyond Software Engineering. T. L. Kunii (ed). Berlin; New York, Springer-Verlag.

Smith, R. J. (1984). Circuits, Devices and Systems, Wiley. Fourth Ed.

Smith, S. L. and S. L. Gerhart (1988). Statemate and Cruise Control: a Case^2 Study. Proceedings COMPSAC 88: Twelth Annual International Computer Software and Applications Conference.

Soni, D., R. L. Nord, et al. (1995). Software Architecture in Industrial Applications. Proceedings ICSE '95, Seattle, Washington.

Sowa, J. F. (1983). Conceptual Structures: information processing in mind and machine, Addison-Wesley.

Spector, A. and D. Gifford (1986). "A Computer Science Perspective of Bridge Design." Communications of the ACM **29**(4): pp. 267-283.

Spelke, E. A. and L. Hermer (1996). Early Cognitive Development: Objects and Space. Perceptual and Cognitive Development. R. Gelman and T. K.-F. Au (eds). London, Academic Press.

Spooner, C. R. (1971). "A Software Architecture for the 70's: Part I - The General Approach." Software - Practice and Experience **1**(Jan-March): pp. 5-37.

Stacy, W. and J. Macmillian (1995). "Cognitive Bias in Software Engineering." Communication of the ACM **38**(6).

Standen, D. (1981). Terms in Practice: a dictionary for Australian arthitects, Royal Austrlian Institute of Architects.

Stefani, R. T., J. Clement J. Savant, et al. (1994). Design of Feedback Control Systems, Saunders College Publishing. Third Ed.

Stefik, M. J. (1987). "Book Reviews - Understanding Computers and Cognition: A New Foundation for Design." Artificial Intelligence(31): pp. 213-261.

Susskind, C. (1968a). "The Early History of Electronics: I. Electromagnetics before Hertz." IEEE Spectrum(August): pp. 90-98.

Susskind, C. (1968b). "The Early History of Electronics: II. The experiments of Hertz." IEEE Spectrum(December): pp. 57-60.

Susskind, C. (1969a). "The Early History of Electronics: III. Prehistory of radiotelgraphy." IEEE Spectrum(April): pp. 69-74.

Susskind, C. (1969b). "The Early History of Electronics: IV: First radiotelgraphy experiments." IEEE Spectrum(August): pp. 66-70.

Susskind, C. (1970a). "The Early History of Electronics: V. Commercial beginnings." IEEE Spectrum(April): pp. 78-83.

Susskind, C. (1970b). "The Early History of Electronics: VI. Discovery of the Electron." IEEE Spectrum(September): pp. 76-79.

Swartz, N. (1997). Definitions, Dictionaries, and Meanings. September 1997. http://www.sfu.ca/philosophy/swartz/definitions.htm. October 1999.

Sydney Opera House(1999). http://www.soh.nsw.gov.au. 20 April 1999.

Takasaki, G. M. and R. E. Fenton (1977). "On the Identification of Vehicle Longitudinal Dynamics." IEEE Trans on Automatic Control **AC**-**22**(4): pp. 610-615.

Teplitzky, P. (1994). A Taxonomy of Software Development Methods. From Computing Reviews. http://www.acm.org/pubs/toc/Abstracts/0001-0782/188377.html. May 1998.

The Institution of Engineers, A. (1973). Australian Standard: Engineering Drawing Practice.

Tjalve, E., M. M. Andreasen, et al. (1979). Engineering Graphic Modelling, Newnes - Butterworths.

Tsichritzis, D., O. Nierstrasz, et al. (1992). "Beyond Objects: Objects." IJICIS **1**(1): pp. 43-60.

Tsujii, M., H. Takeuchi, et al. (1991). Application of Self-Tuning to Automotive Cruise Control. Advances in Adaptive Control. K. S. Narendra, R. Ortega and P. Dorato (eds), IEEE Press**:** pp. 282-287.

Urmson, J. O. and J. Ree, Eds (eds). (1989). The Concise Encyclopedia of Western Philosophy and Philosophers. London; Boston, Unwin Hyman.

Vitruvius, P. (1931). Vitruvius, On Architecture., F. Granger (trans).

Ward, P. T. and D. A. Keskar (1987). A Comparison of Ward/Mellor and Boeing/Hatley Real Time Methods. Proceedings SMC XII: Twelfth Structured Methods Conference.

Wasserman, A. I. (1996). "Towards a Discipline of Software Engineering." IEEE Software(November).

Wasserman, A. I., P. A. Pircher, et al. (1989). "An Object-Oriented Structured Design Method for Code Generation." ACM SIGSoft Software Engineering Notes **14**(1): pp. 32-55.

Watson, D. (1990). Rule-Generated Architecture. Geelong, Australia, Deakin University Press.

Wegner, P. (1984). "Capital-Intensive Software Technology." IEEE Software(July): pp. 7-45.

Weinberg, J. (2000). Architecture as Metaphor. Personal Communication with Jason Baragry.

Wertheimer, M. (1958). Principles of Perceptual Organisation. Readings In Perception. D. C. Beardslee and M. Wertheimer (eds)**:** pp. 115-135.

Whitehead, E. J., J. E. Robbins, et al. (1995). Software Architecture: Foundation of a Software Component Marketplace. Proceedings 1st Intl Workshop on Architectures for Software Systems, Seattle, Washington.

Winograd, T. (1995). "From Programming Environments to Environments for Designing." Communications of the ACM **38**(6).

Winograd, T. and F. Flores (1985). Understanding Computers and Cognition: a new foundation for design, Ablex Pub. Corp.

Wittgenstein, L. (1968). Philosphical Investigations. Oxford, Blackwell. 3rd Ed.

Wolf, A. L. (1997). "Succeedings of the Second International Software Architecture Workshop (ISAW-2)." Software Engineering Notes **22**(1): pp. 42-56.

The Wright Brothers(1995). http://www.hfmgv.org/histories/wright/wrights.html. June 1999.

WWISA (1999). "Philosophy.", Worldwide Institute of Software Architects http://www.wwisa.org/ October 1999.

Xia, F. (1997). Software Engineering Research: A Methodological Analaysis. Proceedings Asia-Pacific Software Engineering Conference.

Xia, F. (1998). (Panel Session) How Can We Conduct Research In Software Engineering. Proceedings Asia Pacific Software Engineering Conference, Taipei, Taiwan.

Yin, W. P. and M. M. Tanik (1991). "Reusability in the real-time use of Ada." International Journal of Computer Applications in Technology **4**(2).

Zakian, V. (1991). "Well Matched Systems." I.M.A. Journal of Mathematical Control and Information(8): pp. 29-39.

Zalta, E. E. N. (1999). The Stanford Encyclopedia of Philosophy. December 1999 Archive. http://plato.stanford.edu/.