University of Alberta

Design Theory and Software Design

by

Kent McPhee

Technical Report TR 96-26
October 1996
(Revised May 1977)

DEPARTMENT OF COMPUTING SCIENCE
The University of Alberta
Edmonton, Alberta, Canada

Abstract

Software design methods share many characteristics with design methods in other fields. All these methods are the progeny of philosophies of design that are in turn influenced by more general philosophic movements. This essay begins with the influence of philosophies of science on the study of design, highlighting the effects on design discourse of Cartesian rationality, the hypothetico-deductive account of scientific progress, and Kuhnian paradigms. Next, the influence of the constructivist and humanist movements on design thinking are considered, culminating in the introduction of a philosophy of design based on hermeneutics, or interpretation. The influence of design philosophy on software design methods begins a categorization of several software design methods according to the design theory framework, with some emphasis on design methods that support a hermeneutical style of design. Some justification for a pluralistic approach to software design methodology rounds out the essay.

Chapter 1

Introduction

1.1. The software "crisis"

For the first time, the "software crisis" was discussed openly at a 1968 North Atlantic Treaty Organization (NATO) science committee conference [170]. The demands of developing increasingly complex computing systems had overwhelmed the computing professionals of the day. Too often, the computing industry was not able to deliver working software on time and on budget. Conference attendees agreed on the necessity of a change in the way software was developed. The main theme of their recommendations revolves around the elimination of ad-hoc approaches to software development. They advocated the introduction of techniques based on sound theoretical foundations derived from science and mathematics, thus giving rise to the field of "software engineering." Many modern software engineering process models and formal methods build upon these foundations.

Quintas [170] groups software engineering practitioners into two camps, the "formalists" and the "pragmatists". Formalists base their work on the theoretical foundations of computing science while pragmatists take the view that software engineering is a practical discipline that can benefit from more than just formal techniques. Underlying both approaches is the desire to capture what Brooks [35] calls the "conceptual constructs" of complex systems and to manage the transformation of those constructs into physical software systems. Formal methods have been used most successfully in application domains like compiler technology, real-time computing, data transaction processing, and numerical software. Pragmatic methods and tools are used often to solve problems that seem to defy successful application of strictly theoretical approaches. For example, software developers building user interfaces with uncertain or incomplete requirements are more likely to use less formal techniques.

Today, close to thirty years after the NATO conference, the demand for timely delivery of reliable, complex, high quality software systems still strains the ability of the computing community to provide them [188] [200]. Why is this so? Has nothing happened during those thirty years? The answer, of course, is that a great deal of progress has been made in the industry's ability to deliver complicated software systems. Systems developed today are many orders of magnitude more complex than their counterparts from the 1960's. Yet, modern software engineering techniques have not been able to keep up with the demand for more and

more complex systems. The question that was asked in 1968 is just as valid now: “How can we predictably create better software systems?”

1.2. The current state of affairs

The effort of the software engineering community over the last thirty years has been massive and, for the most part, successful. A great many methods, techniques, and tools are available to today’s computing professionals. Many problems that were once difficult or intractable can now be solved routinely or even automatically. Complicated software systems are used to control airplanes, run businesses, operate stock exchanges, and monitor patients in hospitals.

Despite successes from both the formalist and pragmatist camps, difficult problems arise. Software projects and software products fail, sometimes spectacularly [211] [154] [228] [160]. Observers commonly blame these failures on the software developer’s lack of rigour, failure to use the right tools, or misunderstanding of technical issues. Often, management decision makers promulgate a solution indoctrinating developers in a new method or tool that promises to solve the problem [165]. Nearly as often, the indoctrination fails and developers end up falling back to their old ways. They build new systems, experience new problems, try another new method or tool, and the cycle continues. Fred Brooks [35] likens the search for the “one true software method” to the quest for a “silver bullet” that will vanquish all of our software development ills just as werewolves are laid to rest by projectiles made of silver.

It is no accident that the activities surrounding the development of software are often called “software *engineering*”. The software industry seems united in its desire to make software development a profession. To that end, great strides have been made in the technical aspects of software development. Computing science has produced a large body of knowledge, some of which is applicable to software development[1]. It is interesting to note that the application of the “science” in computing science is more likely to succeed the farther a target application gets from the world of humans. It seems that, the more control software developers have over how their software is used, the more likely it is they can use formal scientific development methods.

The software industry is less capable of dealing with failures due to aspects of software development that are outside the realm of the technical issues addressed by engineering-based methods and processes. The industry needs to learn more about how people design software and how that knowledge can be put to use supporting, organizing, and managing software development activities. To this end, software professionals must try to learn more about what

[1] ;-)

developing software really involves. Why do some methods help and others hinder? Why do techniques work in some situations but not in others? How have quite successful software systems been built *without* the benefit of the most up-to-date software engineering methods? Software design researchers are working hard on many fronts to answer these questions. This essay will look at the work of many of them.

Like many human endeavours, software system development involves the creation of an artifact satisfying some criteria for "goodness". The criteria for goodness of a software system is often measured by the success the software system has in meeting the needs of its users. Before, during, and after designing the software, designers must interact with users and others to help determine the suitability of their designs. As the software industry realizes that software development is just as much a "people" problem as it is a "technical" problem, interest has developed in finding methods and tools that support the more pragmatic, people-oriented, aspects of software design.

Design forms the foundation for the practical application of computers in society. Software design shares many characteristics with design as it occurs in other fields, a fact neglected by the software community in the past. Relatively little work has been done with the goal of determining how general design knowledge can, or should, be applied to better software development. By studying how design works in other fields and why it is similar across many domains, software professionals might be able to answer many questions about how to approach both the technical and "people" aspects of software design.

The philosophical basis of design is an important place to start studying design. A philosophy of design provides a foundation for work in developing design techniques, tools, and methods. Few designers have an appreciation for how the philosophical underpinnings of design have influenced the techniques that they use in daily practice. Fewer still *question* the validity of those techniques based on an understanding of alternative philosophical approaches to designing. By studying general design issues, software designers can learn when certain techniques are called for and when they should be eschewed.

The remainder of this essay will expand on the design topics mentioned above. Chapter 2, *Design Theory*, will start with a survey of general design research, concentrating on broad trends in design philosophy and design methods. It presents two major viewpoints in how humans have come to think about design. The advantages and disadvantages of both are discussed. Chapter 3, *Software Design*, discusses the similarities between general design processes and software design processes. The roots of several software design methods are illuminated and interesting new software design methods are discussed.

Chapter 2

Design Theory

2.1. Introduction

Software design shares many characteristics with design in other fields. Much can be learned about software design by examining various philosophical and methodological viewpoints underlying design in general. In many cases, these viewpoints have shaped attitudes towards software design and software design methods, techniques, and tools. Reflecting the pervasiveness of design activities in nearly all areas of human endeavour, design research continues to be carried out in an expanding range of disciplines [205] [78] [44] [177] [175] [161]. The extensive diversity and volume of design discourse makes a comprehensive review a difficult task. For brevity, Chapter 2 tries to concentrate on the main themes of design research. Chapter 2 emphasizes the historical roots of design and provides the background required for the discussion in Chapter 3, *Software Design*, of the current state of software design.

The term "design" can have a variety of meanings. Minneman recounts [150] Dilnot's three different meanings of design. In Dilnot's view, design can refer to an activity (the act of designing), to the results of that activity (the designed artifacts), or to a value (as in the aesthetic of "good design"). Typically, design research has concentrated on the first of these definitions. In the context of design activity, design researchers have tried to answer two related and fundamental questions.

The first fundamental question is:

- What are the essential characteristics of "design"?

This question relates to understanding when an activity is "designing" and when it is not. Answers to this question deal with the characteristics and classification of design situations. These answers are explored in section 2.2, *Recognizing design situations*.

The second, and perhaps more important, fundamental question asked by design researchers is:

- What processes are used by designers?

Related sub-questions are numerous. What types of processes do people follow when they design? Is one process better than another, constituting "right" and "wrong" ways to

design? Why are some processes favoured over others? Do different processes lead to different qualities of results? How is process related to and affected by the configuration of the design situation and its participants? Should designing even be viewed as a process? The section 2.3, *Approaches to designing*, gives a brief history of design processes and discusses the philosophical, methodological, and pragmatic concerns raised by these questions.

Using these questions as a guide, exploring the design landscape will help to determine the nature of appropriate software design methods, tools, and techniques. The characteristics of design situations and the processes used by designers directly affect, in a fundamental manner, the form and function of the tools most useful to designers.

While researchers generally agree that design results from the desire to satisfy a human need or want, they debate the nature of the process of designing. Does design fit within the logical and rational framework traditionally associated with the "scientific" approach to problem solving? Or are design problems best solved using more ad-hoc techniques dependent on the nature of the individuals involved? As the remainder of Chapter 2 illustrates, some design processes seem very similar to the traditional "scientific" approach to problem solving while others seem highly dependent on human behaviour that is not easily explained within the rational tradition of science. Design processes may contain elements of both "science" and "art"[2]. Acknowledging this assumption is a necessary first step in establishing the form and function of tools, methods, and techniques that must support design processes. Coyne and Snodgrass [41] go even further. They propose that the "science" versus "art" dichotomy depicts an artificial distinction that impedes effective design discourse. The debate's philosophical roots, and an evaluation of the implications of the competing philosophies appear in section 2.3, *Approaches to designing*.

2.2. Recognizing design situations

What are the fundamental characteristics of "design"? As this question implies, an important part of the study of design attempts to characterize the situations in which humans find themselves "designing."

Once thought to be exclusively within the domain of craftsmen and artists, design is now generally believed to be fundamental to human existence. This belief is exemplified by the widely varying application domains in which design research activity is conducted [54] [78] [44] [192]. Design occurs under circumstances traditionally thought of as "design" situations, such as planning the layout of buildings or specifying a software system's

[2]Here, "art" is defined as "those parts of creativity and human nature that lie outside the realm of science".

architecture. To some, it also occurs in less obvious situations, like proving a simple theorem in Euclidean geometry [203] or planning a shopping trip [94]. Wittgenstein [221] tries to show how classification of many conceptual entities transcends simple logical correspondence between the members of the class. Instead, certain "family resemblances" can be used to decide whether or not an entity is a constituent of the class. For example, the members of the family of "design situations" share an indefinite number of "features." Each feature's presence or absence either strengthens or weakens the claim that the activity is a "design situation." How an activity is classified as design depends on how the person involved interprets any "design-like" features. Classification proceeds as a metaphor-like interpretation. If an activity is enough "like design", then it is designing. In some sense, classes are determined by the attributes of their members while simultaneously influencing decisions regarding membership candidates.

In the absence of a precise definition, "design" often is described by pointing out examples and counterexamples. For instance, Archer [6] describes architecture and typographical page layout, but not sculpture and mathematical calculation, as design activities. According to Archer, architecture and page layout have "practical purpose" and are not "conducted mechanically." Often, the difficulty of classification is evident by the relative ease with which examples and counterexamples can be reversed. Archer's assertion that sculpture is without purpose seems counter-intuitive. A sculpture may be designed to evoke a particular psychological state, perhaps national pride or remembrance. Certainly some sculpture has purpose, making its creation a design activity with respect to purpose.

Perhaps because of the difficulties associated with classification of conceptual entities, design researchers debate the precise set of features that characterize design situations. The wide range of disciplines studied by design researchers leads to different opinions based on varying criteria. Very often, the aspects identified as important enough to merit a situation's admission to the field of design are influenced by the researcher's educational background and academic specialty. Typically, a researcher with a background and expertise in the fine arts appears more inclined to agree that a work of art is "designed." While another researcher, firmly grounded in the engineering sciences, is likely to disagree.

2.2.1. Characteristics of design situations

Even within the somewhat murky waters of design science, a certain degree of consensus exists. Although there will always be those who disagree with any rendering of the characteristics of "design", the following distillation of ideas appear nearly universally among design science researchers. The following discussion presents general characteristics of design situations.

- *Design situations start with a need and require intention*

The aphorism "necessity is the mother of invention" aptly and succinctly describes the opinion of many design researchers. In summarizing characteristics that are basic to design, Lawson [136] and Dasgupta [54] note that a real or perceived need, influenced by the value systems of the individuals involved, forms the basis for the definition of a design project. A need so identified acts as the initial motivational force that provides the basis for starting design work. While many researchers use words like "proactive" [192] in implicitly acknowledging that design can only be undertaken intentionally, Willem [213] expresses this viewpoint explicitly. He believes that the universal feature of design is simply the intentional devising of a plan or prototype for something new.

- *Design situations involve transformation*

A second aspect of design, almost universally accepted (either implicitly or explicitly) among design researchers, is the transformational nature of design. Dasgupta [54] maintains that need acts as a seed that design transforms into a form that is eventually used to guide the implementation of an artifact, plan, or process. Simon writes that design is the restructuring of a current situation to achieve some preferred situation [192]. Each design conceptually transforms the environment to satisfy the need posed by the design situation. Willem prefers to use the term "development" to describe the transformation that occurs during design [213]. Meanwhile, Freeman [72] describes the transformation (of needs or desires) to a realizable form as "operationalization."

- *Generation of new ideas is fundamental to design situations*

Another commonly cited characteristic of "design" is the requisite generation of new ideas during the design act. As Page [204] writes, design occurs whenever there is an "imaginative jump from present facts to future possibilities." Willem [214] describes design as an activity where designs occur by the generation of new thoughts. Kaplan [116] points out the importance of creative insight in the design of computer systems.

Creativity is often held up as an important and fundamental characteristic of all designs. Creativity is held in high regard even though it remains an elusive subject, beyond science's firm grasp. The precise manner in which new ideas are generated cannot be codified. Some researchers, such as Freeman [72], have postulated that idea generation is not entirely a haphazard activity. He believes that two styles of idea generation exist: abstraction and elaboration. Together, abstraction and elaboration form a tension. Abstraction is used to make generalizations [101] while elaboration attempts to develop into great detail the specifics of a design.

- *Constraint satisfaction*

An initial need determines the most basic constraints and requirements on a design situation. The design must satisfy the need. In general, more constraints are eventually discovered during the design work itself. Many researchers agree that a major part of designing involves discovery and satisfaction of constraints on the eventual form of the design.

Mostow [152] characterizes design in terms of the constraints that apply both to the designed artifact and to the processes and participants involved during the design activity. To him, design is an activity with the goal of creating an artifact description that satisfies constraints derived from functional and performance specifications of the artifact, limitations of the medium and process by which the artifact is rendered or produced, and aesthetic criteria on the form of the artifact. Mostow summarizes that design is "largely a process of integrating constraints imposed by the problem, the medium, and the designer."

Willem [214] also believes that design solutions are constrained by the terms and conditions provided by their media. Such constraints are part of a group of constraints that Willem labels "external constraints." External constraints are constraints that are independent of the designer, for example, function and economy. Willem believes that there are also "internal constraints" that play an important role in the designer's creative approach to a design situation. For Willem, internal constraints are shaped by cultural and experiential factors that play a large part in the development of human mental habits and attitudes. Such constraints play a role in determining the types of "new thoughts" produced when a design problem is pondered.

Writing in the context of architectural design, Lawson [136] presents design problems as the assembling of constraints along three dimensions. The three dimensions are indexed by the generator of the constraints, the domain of the constraints, and the function of the constraints. Constraints are generated from eventual users of the artifact, from designers themselves, from legislators (e.g. safety related constraints), and from design clients (i.e., the people who have

commissioned or sponsored the design and who may or may not be eventual users of the artifact). Lawson's constraints fall into one of two domains, external and internal. Example of external constraints are provided by Mostow's "implementation medium" constraints. External constraints are imposed by factors not under the designer's control, while internal constraints give the designer at least some ability to control them. Lawson's third dimension, constraint function, relates to the rationales behind the imposition of each of the constraints. Constraints can exist for reasons relating to symbolism and social norms, formal intentions of the designer, practical implications wrought by the implementation technologies, and "radical" reasons which deal with the primary purpose of the artifact. Lawson characterizes design as a process of constraint satisfaction with an exact form determined in large part by the configuration of constraints present in a particular design problem.

Constraints are essential information supplied to the design process. Freeman [72] believes that their quantity and quality are "critical parameters" for the success or failure of a particular design. Consistent with other authors [177] [152], he names design representation and the experience of the designer as two other critical parameters.

- *Problem solving or decision making*

Typically, design methodologists from the science and engineering communities characterize design as a type of problem solving or decision theory where the initial conditions, the goal, and the allowable transformation operations are all ill-defined [191]. For them, the solution space for design problems is very large and its sheer size eliminates exhaustive search as a possible problem solving technique. Design, for many scientists and engineers, invariably involves the application of some sort of logical analysis on the set of known inputs. Others, including Willem [214], believe that various design problem solutions are not necessarily connected through logic to their initial problem state. Design problems are often described as "ill-structured" problems, referring to their complexity and the difficulties in determining their associated constraints and requirements.

Rittel and Webber [174] believe that planning and environmental design, because they are intended to change the human course of affairs constitute so-called "wicked" problems. These are problems where human actions become the focus of the activity, and consequently the problems are not easily formulated or solved (i.e., they are ill-structured). By situating a design in the world of people, the design is affected by the people and the people are affected by the design. The same design issues may be viewed differently by each of the individuals involved. Each divergent perspective may influence the progress of the design in different and unpredictable ways.

Thomas and Carroll [203] investigate design from a psychological viewpoint. They describe "design" as a "way of looking at" a problem, not as a "type" of problem. For Thomas and Carroll, even problems that are ordinarily solved using rote techniques of calculation or rule application can be thought of as design problems if "the *problem-solver* views his/her problem or acts as … ill-defin[ed] in the goals, initial conditions, or allowable transformations." That which constitutes a design problem is not amenable to objective definition. The designer's (i.e., problem solver's) perceptions and experience play a much larger role in determining what is and what is not a design situation.

Freeman [72] prefers to use a decision making analogy to discuss design problem solving. For him, "design" is characterized by a series of decisions between various design alternatives, each alternative determined by the current state of abstractions, elaborations, operational statements and other known and unknown factors. Like design-as-problem-solving, the basic characteristic of design-as-decision-making is goal directed activity and navigation of a design configuration space. Design is information intensive, with more information being generated, gathered, and used as inputs to decisions assimilated into the design representation.

- *Design results in a scheme for implementing an artifact*

Like many designers, Dasgupta believes that output or product of design is a symbolic representation of the artifact for implementation. "Design" is essentially "the formulation of a prescription or model for a finished work in advance of its embodiment" [6]. Design representation serves as the basis to conceptualize and compare various design decisions.

Representation is important as a foundation on which to generate design critiques. MacLean, et al., [142] feel that the true output of design is more than a plan or symbolic representation. They maintain that, while the final output of design includes a plan, it also includes what they call the "design space." A design space is a body of knowledge about the artifact, its environment, its intended use, its actual use, and the decisions that went into creating the design. Designers must consider the representations of this kind of meta-knowledge about how they arrived at a particular design.

Sometimes, a design does not result in a distinct "plan-then-implement" situation. Often the design output occurs incrementally while the design and the artifact evolve together. This is especially true when the artifacts being designed must be deployed in partially completed forms; and can be changed relatively easily. For example, both the design of urban areas and large software systems entail an enormous amount of complexity. Their complexity often cannot be managed without allowing parts to be implemented before the "plan" is complete.

Further design is informed by the results of these early implementation episodes. Some might argue that the implemented parts are themselves part of the design, like throw-away prototypes. But often the intermediate parts are pressed into production and used as the final artifact would be used.

- *Diversity and evolution*

Commonly, design researchers acknowledge the diversity and dynamism present in almost all design situations. Any particular design situation could be drawn in many different directions. This diversity is derived from the complexity of the inputs to the design, the different experiences of the designers, the media for the solutions [214], the explicit and implicit tradeoffs between various constraints and requirements [54], and the varying amounts of information available at different times in different situations.

Diversity often leads to uncertainty, because the knowledge that there exists many other solutions to the same design problem causes designers to question the optimality of their initial solution. Thus, they test and modify their design. Designers compensate for weaknesses exposed during testing, they test modifications and redesign as necessary until they are satisfied with their design. Abstractions, elaborations, and operational statements [72] are verified against known information. Possibly, this process causes changes that must be verified themselves. The designer's act of making decisions among the various identified design alternatives ties together this evolution of the design.

The evolution of a design is often closely linked to the consolidation of the constraints and requirements applied in a particular design situation. Design requirements are often imprecise and incomplete [163]. The consequences of design decisions often cannot be forecast with complete accuracy so design solutions evolve in tandem with known problem constraints and requirements. Eventually, a successful design process includes a convergence of requirements, constraints, and knowledge about the design and its effects on the implementation environment.

Various authors [54] [78] [66] [184] have enumerated the characteristics fundamental to design situations. They argue these characteristics are observable, to varying degrees, in all design situations. However, few authors back their claims with statistical evidence. Nevertheless, from surveys of works by independent authors in a wide variety of application domains, the preceding section has presented a general picture of the fundamental attributes of design situations.

2.3. Approaches to designing

At the design situation's nucleus is a human need unsatisfied by prevailing circumstances. If the desire is strong enough, human ingenuity is brought to bear on the situation. Requirements and constraints are identified, reorganized, extended, and examined. Configurations are proposed, tested, and accepted or abandoned based on their ability to satisfy the need, requirements, and constraints. Eventually, a design is produced.

Section 2.3 examines the second question posed in this chapter's introduction. Specifically, what influences design processes and how do these processes unfold? Understanding the answer is vital to the designer of computer-based design tools, methods, and processes. These design aids must be sensitive to the processes that structure the work of the designer. Much of the remainder of this chapter deals with a central issue in design science research. The issue is the relative influences of subjective and the objective viewpoints on design processes. These two ways of looking at design influence design processes by setting the tone for the precise methods used to discover constraints on the design, manage constraints, generate and communicate ideas, test tentative design solutions, etc.. After a brief historical perspective on the design process, the next sections discuss various answers to the question "How do people design?"

2.3.1. A brief history of design processes

From prehistoric peoples' creation of the first primitive tools to the erection of the Egyptian pyramids, from the architectural marvels of Renaissance Europe to the moon landings, history has clearly demonstrated that the design of artifacts distinguishes humans from more primitive species. Design does not belong exclusively to the realm of monumental works of civilization. Everyone engages in design in seemingly ordinary activities such as arranging furniture in the living room, selecting the menu for a wedding, or planning a holiday.

A historical view of design focuses on its craft-like properties. Craft-like activity yields artifacts through the application of what Alexander [4] calls "unselfconscious" thought processes. Such vernacular design processes integrate an artifact's design and construction. Design is embodied in the artisan's knowledge of how the artifact should be built and each artifact is a unique expression of that knowledge. Lawson [136] presents the traditional Inuit igloo and the wheel of the English horse-drawn cart as examples of vernacular, "unselfconscious" design. In both cases, the designs evolved over a long period of time with features added and rejected from time to time dependent on the success or failure of the artifacts that were built with those particular features. Lawson discovered that the reasons for the specific geometric configuration of the cart wheels were unknown to the craftsmen who

made the wheels. Passed from one generation of cart builders to the next, the design was retained in an almost folkloric way. Succeeding generations learned the design by demonstration and apprenticeship.

At some point, people created artifacts too complex for a single craftsman to build alone or with apprentices. Therein the seeds were sown that led to the separation of the conceptualization and building of artifacts. In order to help coordinate the work of multiple craftsmen, design became more "selfconscious" [4] and separate from implementation. "Design-by-drawing" is the label Jones [110] [111] uses for this first step away from vernacular design. The product of design became a visual representation of an artifact drawn in advance of the artifact's embodiment. In an example of shipbuilding in the 18th century, Jones notes that design-by-drawing is effective only in situations where one person, the designer, can retain the entire design in his head. In addition to shipbuilding, architecture also used design-by-drawing to coordinate the work of many craftsmen as they constructed the various parts of a structure. In both cases, much of the design often omitted details of construction. The designer left many details to the skill and experience of the craftsmen.

Until this century, little scholarly thought has been directed towards discovering the meaning of the general processes that define "design." During the last half of this century design processes themselves became objects of intense study. Successful technological advances made during and after World War II, with an incumbent increase in the complexity of the artifacts humans build, encouraged attempts to systematize design [44]. Thus was born the field of "design science." Simon [192] defines design science as "a body of intellectually tough, analytic, partly formalizable, partly empirical, teachable doctrine about the design process." During the 1960's and 1970's design science researchers concentrated on trying to apply scientific thinking to design. More recently, some have started to take a different approach to design science. They have looked at other aspects of design that fall outside the boundaries established by the initial design science work. In addition to the technical and objective aspects of design, they have begun to reexamine the social, psychological, and subjective aspects of design.

2.3.2. Characterizing the process of designing

Theorists in the Jones' "selfconscious" school agree that design transforms human desires into a form that guides the satisfaction of human desires. Or, as Archer puts it, the design process is "the purposeful seeking of a solution" to a problem formulated from those desires [6]. Essentially, the design process involves some form of constraint satisfaction, although the constraints may neither be articulated or even known. Theorists disagree, however, on the extent of possible objectivity in the design process. Is designing a strictly personal experience

that defies generalizations? Is design based solely upon human values and experiences? Or is it an entirely rational process that eventually will be performed mechanically? Are design processes subjective or objective? Is designing an “art” or a “science”?

According to Coyne and Snodgrass [41], the major assumption behind the respective roles of science and art in design is the “dual knowledge thesis.” They discuss the argument that there are two distinct ways of thinking. On one hand, a scientific way of thinking relies on rationality, logic, and analysis; while on the other hand, an artistic way of thinking relies on intuition and exhibits irrational and idiosyncratic characteristics. The concepts of subjectivity and objectivity derive from the dual knowledge dichotomy. The dichotomy itself reflects the Cartesian separation of the rational self from the world of objects (i.e., Descartes’ subject-object opposition).

The dichotomy’s consequences are apparent in the various ways in which scholars and practitioners have tried to describe the process of “designing.” Historically, design processes have been thought of as subjective manifestations dependent on each designer. Coyne and Snodgrass [41] call this the “mystery” of designing that is thought to involve “a special kind of knowledge that is fundamentally difficult to grasp.” Some researchers accept that “mystery” is a fundamental characteristic of design. On the other hand, many researchers in the design science movement declare that subjectivity must be overcome to make design “good” [4] [5]. The appeal of the view that design should proceed objectively lies with the perceived role of objectivity in the success of fields like operations research, systems engineering, medicine, etc.. Objective models provide better equipment for the construction of explanatory models. They accommodate the mass accumulation of knowledge that can be used reflexively, making them much easier to record, discuss, and teach.

Design-by-drawing was the start of a movement towards Schön's [182] “professionalization” in the field of design. Separating design and implementation enables a closer examination of design itself. The separation suggests that perhaps better designs result from a similar separation of design knowledge and the act of designing. The subsequent development of design science produced a widening gap between the body of knowledge about design and its application. Again, Descartes’ subject-object dichotomy influence is apparent, reflected in the professional body’s desire to separate thought and practice.

In many areas of technological endeavour, the “professional” generally receives higher regard than an equally intelligent and productive “nonprofessional” counterpart. Presumably, the professional is intimately familiar with the theoretical knowledge that constitutes the foundation of the profession. This specialized knowledge elevates the professional’s status to that of an authority. This authority resonates in the generally accepted

practice of promulgating prescriptive models to guide the application of professional knowledge. The goal of prescriptive models is to reduce pragmatic aspects to merely following rules set out by the professional. This work style typifies the success of fields like engineering and medicine in modern society[3].

The development of design process prescriptive models is the principal goal of many design professionals. Often, they look to the scientific method as the standard from which to elaborate an equivalent "design method." The influence of the "design-as-science" view shows in the amount of effort expended in trying to equate "design" and "science."

As might be expected, the question of the similarity between science and design cannot be answered in absolute terms. There is a continuum of opinion regarding the verity of science-based models of design. Many design scientists relate science to design by asking the question: "How is design like science?" An equally valid question, and one that leads to answers that question the truth of the design-as-science hypothesis, is: "In what ways does design differ from science?" The answer leads to other ways of describing designing. These other ways of viewing design are not based on the tenets of science and rationalism, yet they provide insight into how design can occur in practice. Section 2.3.4, *Criticisms of design-as-natural-science*, reviews some of the differences between science and design offered by design studies researchers. These differences motivate the claim that science should be used only as a "convenient benchmark from which to view design" [213], not as a foundation for design processes. Other views of designing are discussed in Section 2.3.5, *Design and the human sciences*.

2.3.3. Design and the natural sciences

Design science, with its "intellectually tough" approach, has produced a large body of work that relates design to science. Design science researchers seek to define a philosophy to capture the meaning of design. Underlying their desire is the hope that such a universal design philosophy will establish a consistent unifying foundation for design in the same way that the logical positivist approach to science provided a foundation for its many different branches. A design philosophy is regarded as a necessary precursor to a universal language and methodology of design. Once defined, the "universal philosophy" could then be drawn upon at each unique design situation [182]. Naturally, an improved understanding of the design process would lead to a better ability for design research to improve design processes

[3]This is achieved less often than some professionals would admit. Practitioners often encounter new situations that don't fit the prescriptive models. In these cases, they must rely on professional knowledge to design a solution to the new problem. So we're back to design and the design process!

and outcomes. To this end, design processes in many fields, including software development, are heavily influenced by the desire to build a large body of general design knowledge accompanied by a set of methods and prescriptions for "correct" approaches to design. The trend towards separation of theory and practice is consistent with Schön's [182] observation that fields of endeavour with a desire to attain higher status as a "profession" tend to adopt the methods of science, engineering, and medicine.

When Winograd and Flores [215] discuss the "rationalistic tradition" and "logical empiricism" of Western science and technology, and when Schön [182] speaks of the "technical rationality" of the professions, they describe how Cartesian rationality and the logical positivism of the nineteenth and early twentieth centuries influenced the growth of modern analytic thinking. A rationalist attempts to describe the world in terms of formal systems that:

- provide a representation in terms of fundamental objects that have well-defined properties,
- provide a set of rules for manipulating the objects,
- allow people to use the rules logically to create a new system that somehow represents the world in a manner that allows them to better understand it.

The influence of the rationalistic tradition is seen in the development of the classical scientific method. Scientists first carefully observe certain parameters associated with a phenomenon, then they derive (via inductive or abductive logic) a theory that models the phenomenon. To be useful, the model must contain detail sufficient enough to motivate predictions about the phenomenon's future behaviour. If the predicted behaviour is subsequently observed, the theory is validated and becomes a natural law.

The rationalist tradition influences and predominates modern Western thought. In the words of Winograd and Flores:

> "The rationalistic orientation not only underlies both pure and applied science but is also regarded, perhaps because of the prestige and success that modern science enjoys, as the very paradigm of what it means to think and be intelligent."

Encouraged by the success of science, designers began to take a more rationalistic approach to design. Le Corbusier [40] was one of the first architects to be influenced by the new rationalist view. His approach to architectural design tried to produce "optimal" designs by strictly rational processes of data gathering, analysis, synthesis, and optimization. According

to Le Corbusier, matters of style are relegated to the periphery by proper definition[4] of design problems and thorough analysis using techniques from ergonomics, systems analysis, operations research, and computer science [19].

Alexander and Poyner [5], March [143], and Zeng and Cheng [227] propose rational approaches to design. Their approaches stem from Wittgenstein's ideas of atomism in language [143], in this case, the language of design. They attempt to reduce design problems to the specification of atomic constraints. Then, they employ methods based on logic and mathematics to find optimal design solutions.

In summary, Western society's rationalistic tradition influenced the earliest attempts to exposit a philosophical basis for design. Attempts based on the assumption that design, like science, is rooted in the rational and objective and that the details attending design processes can be modeled systematically. The higher social status of science, the particularly successful application of rational thought in science, and the tremendous progress made by science led design researchers to emulate science when proposing a foundation for design.

During the emergence of a "universal" design philosophy, the generally accepted philosophy of science was itself undergoing revisions. Design scientists noticed the shift. Popper's account of scientific progress [167] [168] refutes the idea that an absolute truth can be discovered by the application of a logical-positivist style of inquiry. Instead, Popper sees science as a continuing stream of increasingly stronger hypotheses from which logical conclusions are deduced. The strength of a hypothesis is determined by the cogency of tests constructed to *disprove* the hypothesis. If a hypothesis survives all currently devised tests, it holds as a law. This law, however, is not fixed. A falsifying test might eventually be discovered. Popper asserts that any hypothesis, even the strongest, has the potential to be disproved. In addition, Popper notes that, in trying to test a hypothesis, new hypotheses are often generated. Some of these new hypotheses must be tested before the original can be tested, others end up replacing the original hypothesis. Popper ignores how hypotheses are devised. He requires only that the hypotheses be stated so that they are falsifiable. By doing so, he allows hypotheses to be generated by systematic analysis, logic, creativity, and random chance. While admitting that certain parts of scientific progress can occur outside the bounds of rational explanation, Popper's view maintains a positivist bias for the processes that occur within its scope. Describing conjectures, devising falsifiability tests, and applying those tests remain objective, logic-based processes.

---

[4]That problems can be "properly defined" is an assumption that many rationalist arguments are built upon.....unfortunately, properly defining a problem is often a very difficult task.

Just as the philosophy of science has progressed beyond logical positivism, design science has progressed beyond the belief that atomism and synthesis form the basis for the design method. Hillier, Musgrove, and O'Sullivan [97] and Broadbent [19] provide an example of how newer philosophies of science have influenced the study of design. Hillier, et al., take up Popper's philosophy as a starting point for their views on design. Broadbent describes design in terms of Kuhn's [127] ideas of paradigmatic influences in science.

Hillier, et al., suggest that the post-WWII thinking regarding design, largely based upon a empiricist point of view, is outdated. The Hillier model of design incorporates an evolving series of conjecture/analysis cycles and appeals to supporters of Popper's philosophy of science. They claim that design is not a wholly logical activity and that a designer's preconceptions play a role. The driving force for progress in design is based on a designer's "prestructuring" of the problem at hand. The designer uses tacit knowledge and preconceptions derived from his experience with previous design situations, tools, and solution types. Design is resistant to inductive rationality because design is *essentially* a matter of this prestructuring. In contrast, logical approaches to design exclude the values and experiences of the designer from playing a role in the design. Logic-based approaches assume that facts exist independent of theory and investigator, and that designs result from logical operations on known facts. Hillier, et al., refute this position by noting that "a complete account of the designer's operations during design, would still not tell us where the solution came from."

The main characteristic of the design process, according to Hillier, et al., is that it involves "variety reduction." Variety reduction is the reduction in possible design outcomes by the application of external and internal constraints on the design. External constraints emanate from user requirements, costs, standards, appearance norms, and the availability of appropriate technology. Internal constraints result from the designer's experience with the problem area, solution types, and the tool set employed. Even before the designer begins to further specify the design problem by gathering and organizing data, variety reduction begins by the creation of a conjecture of an approximate solution to the design problem. Darke [53] believes that the origins of these conjectures arise from "primary-generators", a relatively few main design objectives implicitly selected based upon the values and preconceptions of the designer. At first, conjectures form little more than a statement of direction for the design activity. Guided by the conjectures themselves, during data collection the conjectures become detailed enough to be tested for their suitability. In a manner analogous to the falsifiability testing of hypotheses in Popper's model of science, the conjectures structure the designer's understanding of the design situation and provide a basis

for an analysis to test this understanding. The specification of the problem and the conjecture/analysis cycle proceed concurrently. At some point, the conjectural approximation of the design is substantial enough to use as the input to the final design preparation and implementation.

A more recent development in the philosophy of science was popularized by Kuhn [127]. As had happened before, the philosophy of design was soon influenced by these new thoughts about science. Supplementing Popper's description of the day to day details of scientific progress, Kuhn proposes that macroscopic scientific progress is manifested by a succession of "paradigms." Each paradigm is comprised of shared ideas and assumptions, shared commitments, and shared values that form the context within which the daily activities of science are conducted. The conduct of "normal" science accepts as its foundation the prevailing paradigm. However, eventually a problem appears that is only solvable by radical new ideas or theories. At first, these new ideas will be treated with skepticism by the majority of the scientific community. Eventually, more and more scientists corroborate the new theories and soon the prevailing paradigm is replaced with a new paradigm founded on the new theories. Thus Kuhn's paradigms portray science as periods of stable and conventional activity, separated by "non-cumulative" displacements of underlying ideas and values.

Broadbent [19] applies Kuhn's concept of paradigms to design. Broadbent notes that design has traditionally progressed through a series of stylistic themes such as Byzantine, Romanesque, Gothic, Renaissance, and so on. He further suggests that modern designers have educational backgrounds, social pressures, and shared commitments exactly like those that Kuhn describes for scientists. Design shares with science the same kind of paradigmatic progress with the four constituent parts of a design paradigm being:

- technical knowledge,
- professional skills,
- shared beliefs and convictions,
- shared examples.

Dasgupta [54] further notes that, due to the fact that design involves the creation of many possible models of the world, design paradigms aren't necessarily as entrenched as their scientific counterparts. For example, designers operate under local paradigms determined by their organization's particular standards for design methods, tools, and technologies. In addition many different paradigms may be in use simultaneously.

From the empiricist's views on reductionism and verifiability to Popper's falsifiability and Kuhn's paradigmatic constraints, many members of the design research community are influenced by the prevailing philosophical thoughts regarding science. The conjecture/analysis design cycle involves the gradual bridging of the gap between need and artifact. Conjectures are based on the designer's belief that they will contribute to the design solution. They are tested and accepted or rejected based on their suitability for advancing the progress of the design. Finally, both conjecture and analysis occurs within the context determined by Broadbent's design paradigms. The similarities between science and design, combined with the mature philosophy of science, is a powerful motivator for how science-based models are applied to design. Although similarities are useful, many authors have concentrated on the differences between science and design as a means of illuminating the full character of designing. These differences are discussed in section 2.3.4, *Criticisms of design-as-natural-science*.

2.3.4. Criticisms of design-as-natural-science

Many researchers, even those who explain design in terms of science, point out that the objectives of science differ from those of design. "Science is analytic" writes Gregory [84], while "design is constructive." Simon [192] also differentiates design from science. He notes that science is "concerned with how things are" while design is "concerned with how things ought to be." Willem [213] takes a similar position. He states that science is "knowledge of the natural world" and the "goal of design is not to produce knowledge, but rather to take action." The natural sciences are concerned with discovering and explaining existing phenomena while design primarily deals with active creation and innovation. March [143] makes a distinction between science and design when he writes "Science investigates extant forms. Design initiates novel forms." Cross [45] addresses several differences between science and design. For example, design uses the techniques of "modeling, pattern-formation, and synthesis" to study "the man-made world," meanwhile, scientific methods examine "the natural world" using methods of "controlled experiment, classification, and analysis." Cross goes on to claim that the values behind the two activities differ significantly. On one hand, science is based on the values of "objectivity, rationality, neutrality, and a concern for 'truth'." On the other hand, design is concerned with "practicality, ingenuity, empathy, and a concern for 'appropriateness'."

A strong proponent of the design-as-science viewpoint, Dasgupta, tries to sidestep these concerns. He asserts that the products and goals of design and science should not be confused with the processes of design and science [54]. One should not compare the *products* of design and science, but one can compare the *methods* and ways of thinking employed by

both. Even so, Dasgupta and other authors [66] differentiate science and design on precisely these grounds. He states that science is "theory-oriented" and "rational" while design is "result-oriented" and "creative, spontaneous, and intuitive." Dasgupta's apparent contradiction weakens his claim that design is a special case of the scientific way of looking at the world.

Glanville [77] takes the opposite view to Dasgupta. Glanville claims that science is, in fact, a special case of design. Any theory of design must include, as a subset, explanatory mechanisms for science. Broadbent [19] also explores this theme. He categorizes the activities that comprise both science and design and points out that design involves several aspects ("diagnosis", "prescription", and "advocacy") that are not normally considered part of science. Willem [213] casts science into the role of a supporting character that interacts with the creative act to produce the novel structures characteristic of design. The process of design benefits from knowledge, the product of science. Science does not guide design, science informs design such that "science knowledge is part of the fabric with which designers design." Science is made visible through products of design. Cross, Naughton, and Walker [43] propose a very similar idea. For them, design is more fundamentally a technological, rather than scientific, process. Both technology and design concern the creation of artifacts to fulfill some need for their builders and users. The output of design (i.e., artifacts) comes to exist in a manner different than the more abstract output of science (i.e., knowledge).

Some researchers attack the design-as-science viewpoint by questioning the validity of the philosophies of science. By arguing that a philosophy of science is flawed, they state that applying such a philosophy to design is flawed and therefore without validity. The logical-empiricist design theorists were the first who were criticized using this line of argument. Rzevski [181] notes that even Hume, the intellectual forefather of logical empiricism, found two major flaws in the traditional scientific method. First, the inductive step has no logical explanation that adequately can describe how a theory is derived from a set of specific observations. Second, a natural law that is discovered as a result of applying the scientific method cannot be independently verified. It wasn't until Popper's ideas of "falsification", wherein it is always possible that a new experiment might be devised to contradict the conclusions embodied in the natural law, that these ideas were fully expressed in the philosophy of science. Even Alexander [4], in attempting to apply empirical traditions to design, realizes that there are no fundamental truths in design. Instead, he adopts a principle of fallibility to test designs for success [143]. Alexander acknowledges that designs may not entirely satisfy the constraints imposed upon them. There exists always the possibility that another design can better satisfy the constraints.

In criticizing the link between science and design, Cross, Naughton, and Walker [43] claim that the antecedent observations regarding the nature of science are flawed. Citing several authors who argue against Popper's and Kuhn's models of science, they point out the "epistemological chaos" that plagues the philosophies of science. They claim that design cannot be equated to science on the grounds that the epistemology of science is unstable. They don't wish to use science as a reference for describing design, because the reference is itself a moving target and poorly described. In critiquing the notion of design as a scientific activity, Naur [153] recalls that Feyerabend [70] and Medawar [145] have both come to the conclusion that "the notion of scientific method as a set of guidelines for the practicing scientist is mistaken." Naur concludes that such a set of guidelines for design is also mistaken.

Others view the very act of comparison as being flawed. For example, in his arguments that there is no methodological difference between design and the hypothetico-deductive view of science, Dasgupta implies that:

- design is a logical process, expressible using a formal language of symbolic logic, and
- the scientific models used to describe design correspond to the act of design in a logically deducible and logically expressible manner.

Snodgrass and Coyne contend that design processes cannot, in general, be expressed using a logical formulation [193]. The expression of a precise logical correspondence between a model and its referent is not possible in the same way that a precise logical correspondence between the referents of a metaphor is not possible [194]. Thus the basis of the comparison behind the design-as-science viewpoint is weakened enough to cast doubt upon its validity.

Another problem with design science is an observed gap between theory and practice. Schön believes that the emphasis on problem solving at the expense of problem setting [182] causes this gap. For rationalist designers (e.g. Le Corbusier and the proponents of formal and computational methods for software design), problem solving approaches to design often assume that the problem being solved is well defined and fixed before design starts. In reality however, design "ends" are often undefined and fluid. Any design "means" that assume otherwise will likely fail. Thus the Cartesian tendency to separate means and ends weakens, rather than strengthens, design processes. Rittel and Webber [174] suggest that, since design problems can never be described completely, they are not amenable to the kind of complete analysis that science-based methods of designing require for success.

The separation of means and ends can lead to what Heidegger [96] calls “blindness.” Blindness describes the premature selection of a preferred course of action at the exclusion of many possible interpretations. To avoid blindness, Schön [183] advocates that designers reorient their design processes to include constant reflection about the problem they are trying to solve. The increased emphasis on problem setting closes the gap between design theory and practice by closing the gap between means and ends, subject and object, thought and praxis. Coyne [42] also notes a gap between practical matters and rationalism’s theoretical ideals. He believes that pragmatic issues force designers to deal with many variables concerning the interaction of people’s emotions, ideals, and morals. These are variables that are simply not observable in a strictly scientific sense. Most science depends on observable behaviour and well defined ends. If neither exist, then the situation is outside the scientific realm. Design models cannot ignore the impact of designs on people. Unlike the popular view of scientific theories, designs are not “truths” that somehow transcend the interpretations of the people that interact with them.

Design engages its participants in communication, social interaction, and creative thought. Consequently, researchers like Dilnot [63] and Minneman [150] stress design theory based on sociological and cultural aspects rather than design’s scientific aspects. This view’s legitimacy becomes clearer upon more closely considering the nature of paradigms. Paradigms are more than repositories for facts, methods, and examples utilized by designers when they work on a problem. More importantly, paradigms form what Kuhn calls a “shared matrix”. A shared matrix includes tacit and intangible concepts that implicitly guide designers’ day to day activities. Examples are: the ways that designers present their work to each other, their “standard” methods, their particular design agenda, their approaches to obtaining funding, etc. [19]. Day-to-day activities vary from strictly logical tasks to “gut feeling” approaches. Sociological issues of prestige, economics, and cultural conformity all play roles in affecting designers’ methods and output. By isolating the rational aspects of design from the chaotic and irrational aspects, designers and design scientists have bifurcated design theory. That separation neglects a synergism that is the essence of what it means to design an artifact. Dilnot believes that a profoundly revolutionary change in the approach to design theory is required and that a socio-cultural view of design will successfully integrate the rational and aesthetic aspects of design into a philosophically united view of design.

Many authors note that creativity is central to any design process and that the creative act must be admitted by any model that attempts to explain design. Under this requirement, science-based design models have difficulty because science knows too little about the creative act. No rational formulations exist for the spontaneous insights that occur while

designers discuss a difficult design problem. Logic cannot explain how a flash of creative brilliance occurs, nor predict the next one. However, sociological studies of design [150] [22] have tried to identify the conditions, shared understandings, and progressions of ideas where brilliant insights likely occur. Some day, there might be an entirely rational logic-based approach to the subject of creativity. Although, if such a logic exists, Archer [6] believes that the original problem ceases to be, by definition, a design problem! He goes on to say that "the creative leap from pondering the question to finding a solution" is "the real crux of the act of designing."

2.3.5. Design and the human sciences

The perceived flaws of the design-as-science viewpoint naturally lead to an exploration of alternative views. A survey of non-scientific ways of understanding creativity and design processes is necessary in any examination of design. This section describes some of the non-scientific, some would say more humanistic, ways of looking at design processes. From a philosophical tradition quite different from natural science-based philosophies, this section discusses various approaches to design based on the "human sciences" of sociology, psychology, and anthropology.

Through its genesis in human need and its realization in human activity, design intimately involves the questioning of human actions and desires. Design belongs in the domain of the human sciences. To understand how designing works requires one to understand how humans think and behave. Any theory of design based on a different assumption ignores a fundamental part of designing.

The methodology of the natural sciences was once touted as the best way to achieve the Positivist goals of objectivity and certain knowledge [63], even in the human sciences. However, a large number of human sciences researchers no longer accept this view. The natural sciences are concerned with the acquisition of knowledge and the separation of subject and object. The human sciences are concerned with peoples' behaviour and the interplay between subject and object. The natural and human sciences are as different as people are different from things [193]. The human sciences are self-reflexive, the study of human activity is itself a human activity. In sharp contrast to the Cartesian ideal, the human sciences do not allow its participants to step outside the sphere of human activity to examine it as an external object. Thus objectivity does not exist in the human sciences[5]. When studying

[5]Some might argue that it does not exist in the natural sciences either. But the natural sciences can make claims of objectivity by separating and ignoring those aspects of science that do not obey the tenants of logic (such as the genesis of hypotheses).

human behaviour, it is impossible to ignore how hypotheses are generated and how facts are derived and examined. Self-referential behaviour requires that human sciences researchers concern themselves with their own background practices. Their success often hinges on their understanding of those practices.

While unique from the natural sciences, design is not an irrational, felicitous process inexplicably tied to the individual designer's whims. Snodgrass and Coyne [193] [194] offer the view that, although not strictly guided by logic, designing can be explained by appealing to Heidegger's and Gadamer's "hermeneutics". Hermeneutics originated as the study of interpretation of texts, especially religious texts. It concerns the phenomena that texts read and reread over centuries carry with them different meanings for different people at different and the same times. Gadamer's hermeneutical view maintains that texts have no meaning independent of the act of their interpretation. Modeled as an interaction between the "horizons" (i.e., experiences and ideas) of the text and the reader, interpretation is central to the understanding of the text.

The interpretive view is important in the study of the human sciences. The human sciences are concerned with developing an understanding of how humans behave. Such understanding arises from a constantly evolving interpretive process [193].

Central to Snodgrass and Coyne's discussion of "understanding" is their concept of the "hermeneutical circle". The hermeneutical circle essentially describes how understanding arises through interpretation and interpretation's influence on the fundamental relationship between the whole and its parts. An example of this relationship and its effect on understanding is seen in the meaning of a language act, such as a sentence. Hermeneutics states that the meaning of a whole sentence cannot be understood until one understands the meanings of the words that comprise the sentence. At the same time, one cannot understand the meanings of the individual words until they are situated within the context of the sentence as a whole. This circular way of building a sentence interpretation, where the meanings of the parts and the whole each influence the other, is central. The hermeneutical circle can be extended to cover any concepts situated within some overall context, a context that gives meaning to those concepts. The apparent logical contradiction arising from this view of understanding is evidence, according to Snodgrass and Coyne, of the unsuitability of logic and method in attempting to explain how understanding arises.

The cyclical nature of the hermeneutical circle implies that understanding is not complete until a series of cyclical exchanges have been made. How then, do humans understand many complex concepts, such as speech acts, *as they occur*, without the benefit of post-mortem reflection or many iterations of the circle? The answer lies within the way that hermeneutical

interpretation begins. The hermeneutical circle is entered with the projection of an *anticipated* meaning. A meaning based on the current understanding of the situation vis à vis the preconceptions of the participant making the projection. Preconceptions are derived from the past experiences, skills, and tacit knowledge. In ordinary conversation, the anticipated meaning, which continues to build as a sentence is uttered, is usually very close to the final constructed understanding. In more uncommon situations (e.g. puns and sarcasm), iteration between the whole and its parts must be more explicit and complete before full understanding emerges.

Hermeneutical philosophy asserts the universality of interpretation and understanding to all human thought. The hermeneutical process preceded, and indeed enabled, the discovery of logic, formal languages, and the scientific method. The hermeneutical process is basic to human existence because, in the words of Snodgrass and Coyne, "understanding is not one of our activities in the world, but is basic to everything we do and are."

Design can be viewed as a process of learning [153] [152]. The design tasks involve understanding the goal, constraints, and requirements. That is, designing is the process of interpreting a design situation, not solving the problem from which the design situation arose. The hermeneutical view of designing portrays design as dialogical exchanges between the designer and the designed, between the parts of the design and the whole, between the designer and the end-users. Within this account, design becomes a cyclic interpretive process whereby the final design elements emerge from a series of interchanges. A designer's understanding of the whole and its constituent parts develops via a hermeneutical circle of interpretation. An initial understanding, called a "primary generator" by Darke [53], is obtained from the parts of the situation and the designer's "horizon" of experiences and knowledge. A dialectic begins when this initial understanding is "projected" onto the design situation. Newly acquired understanding, projected back onto the design situation as a whole, changes the way the designer perceives the design. This new perception alters his understanding of the role played by the parts of the design. The roles of designer preconceptions, users, existing design solutions, and the design environment are all affected. In this way, the design is refined by deeper and more extensive interpretations until designers gain enough confidence and understanding to consider the design "finished."

Even after implementation has begun, the dialectic structure must remain fluid and open. The efficacy of the design depends on constant refinement and evolution of the designer's situational understanding. Complex software design situations where the final goal often remains unknown until it is reached (another characteristic of the "wicked" problems mentioned earlier) show the practical implications of keeping the dialectic open. Indeed, just

as there is no end to the hermeneutical circle, there is often no real “end” to a software design project - at least not until the software is abandoned [46].

Winograd and Flores believe that much of what one does is attributable to Heidegger’s idea of “thrownness.” Thrownness means being in a situation where what one does and how one acts is controlled by almost unconscious interpretations of the situation. Schön [182] calls this “knowing-in-action.” He gives an example of a baseball pitcher who instinctively throws to each successive batter’s weakness, changes his pace, and distributes his effort during a game. Breakdowns occur when thrownness fails to produce desired results. When a breakdown occurs, things that previously were tacit come to the forefront and must be dealt with explicitly. The baseball pitcher who starts to walk batters or pitch home run balls experiences a breakdown that requires conscious corrective action. In the context of design, breakdowns occur in situations unresponsive to direct application of previously learned behaviours. The most interesting and common design situations are almost always of this type [191] [174]. Breakdowns are one way in which design situations “talk back” to the designers. Each breakdown accompanies an opportunity for designers to discover new things and to modify preconceptions.

Snodgrass and Coyne [194] stress the importance of metaphor in design. Metaphor plays a central role in how humans think [132] [158] and hermeneutics successfully describes how metaphors operate. The connection between a metaphor’s compared concepts is understood by a cyclic interpretation where characteristics of one concept are mapped to the other, perhaps revealing new characteristics that can be mapped the other way. Snodgrass and Coyne contend that a metaphor’s characteristics make it impossible to describe the connection between the compared concepts using logic based methods. Since models of design are essentially metaphors of what really happens during design and designs themselves are often based heavily on metaphor, hermeneutics offers a viable explanation of the design milieu. Thus one is led to believe that the metaphor of “design as science” is as valid an interpretation as “design as art”. The appropriateness of any metaphor depends on the circumstances under which it is interpreted.

By thinking of design as an interpretive process, one acknowledges the complex interaction of many historical, social, cultural, economic, physical, structural, and environmental factors. These factors permeate many design problems and embrace the experiential nature of human thought processes. Restrained by strictly logical paths, designers very often fail to acknowledge the many dimensions of a design problem that transcend logical formulation. Design decisions, especially early in the design, often are driven more by a designer’s intuition [53] [133] than by precise analytic or empirical investigation. Much of what goes on

in design is attributable to designers “just knowing”. They know what to do based on their own interpretation of paradigmatic norms and the configurations of particular design situations [133]. Generally, designers know when they have sufficient experience and when they need more [1]. The differences between designers’ experiences causes their approaches to similar design situations to be sufficiently varied to significantly change final design outcomes. In practice, software quality assurance methods that advocate independent duplication of design effort rely on this characteristic of design to decrease the likelihood of errors in critical software components [7].

Certainly, logical analysis and empirical inquiry have their place in design. But only by interpreting the current situation do good designers know when a more careful analysis is required. The hermeneutical approach to design frees the designer from the constraints of logic based approaches without requiring them to abandon those approaches altogether. Although relegated from its implied role as the foundation of design, logic remains a useful tool that helps designers master design situations. Traditional design techniques become one of the many parts that shape the progress of the whole. Hermeneutics acknowledges the notion that design cannot be described by a single method or theory. The complexity of real design situations requires methodological pluralism by designers. Simply put, there is no “silver bullet” [35] for design.

Schön's observations of designers and their tutors [182] [183] reveal the presence of the hermeneutical circle. In his observations, designers engage in a “reflective conversation” with the design situation (i.e., design materials, configuration, participants). Design instructors encourage their students to find a position from which to start the “conversation” and to “listen” to the design by reflecting on their “moves” from the design situation’s point of view. In situations involving more than one designer, often the design situation can “talk back” through alternative points of views generated by the participants. The dialogical nature of group interaction helps to make the hermeneutical circle of understanding more explicit. The hermeneutical circle is enlarged to encompass the group’s understanding of the design and its role. In successful design situations designers achieve a “congruence of meaning” by constructing an interpretation that merges their horizons to form a shared understanding.

For some, interpretation is the central mechanism behind human understanding and influence in the world. Researchers and philosophers, especially those working within the “human” sciences, have adopted a view that reality is “constructed” in the sense that reality depends on how people construct, or create, meaning via interpretive processes. The existence of an objective reality upon which the natural sciences are founded, can never be determined because the observers cannot escape from the hermeneutical circle that determines how they

see and react to the world. Social constructivism goes further. Social constructivists [52] claim humans build interpretations of their world coloured by social norms. The interpretations are influenced by the ways in which people interact with others in various social contexts.

Most complex design situations involve teams of designers. Thus, designing occurs within a social context (Broadbent's design paradigms) and the influence of social context plays a role in determining the outcome of the activity. Elaborating and testing design ideas in social settings is useful and often governs the acceptance of those ideas. To interact socially, people must communicate – with the people they meet, with the objects they use, with the users of the artifacts they produce, and even with themselves. Communication is the basic building block of social constructivism. Schön notes that conversation plays an important role in building social reality. Others, like Reddy [171], have pointed out that tacit contextual information is just as important as that which is explicitly stated during conversation. Such tacit understanding is built up through experience and trial and error. The success of social interactions depend on how people communicate, through what medium they communicate, and with whom (or with what) they communicate.

For the most part, the "human" sciences eschew the Cartesian notions of rationality in favour of more "relativistic" views of human existence. The human sciences tend to treat individuals as part of their world, rather than as separate from it. Designing is seen by many to be highly dependent on human values. Because of this, the ideas of the human sciences are brought to bear on designing by many researchers. Interpretation seems particularly suitable to designing. Hermeneutics, which concerns itself with interpretation, emphasizes a progression towards understanding that is often seen reflected in actual design situations. Designers are constantly interpreting the constraints placed upon them by users, other designers, technology, their own experience, and their social working context. Of course these ideas are not above criticism. Like the science-based views on designing, human-science-based views have their critics. Section 2.3.6, *Criticisms of design-as-human-science*, discusses some of their criticisms.

2.3.6. Criticisms of design-as-human-science

The human science contention that no objective reality exists (or at least it is impossible to know whether an objective reality exists) and there can be no separation of thought and praxis, shakes the foundation of the natural sciences. Natural scientists complain about the "fuzzy" or "mystical" nature of the human sciences. The imprecise descriptive nature of the writing encountered in the human sciences is often difficult to grasp. The lack of formal (i.e. rational) methods for describing the ideas of the human sciences adds to the problem.

These “fuzzy” descriptions of the nature of designing are accompanied by very little prescription for precisely what should be done in lieu of rational approaches.

Some researchers have difficulty with the implication that the interpretive processes of the human sciences will never be explained using formal symbol systems. Many human science researchers imply that the current failures of rational approaches will never be overcome. This implication is dangerous as long as progress in formal analytic approaches continues to be made. Sometimes, human science researchers point out the superiority of their methods over those from the natural sciences without fully understanding the current state of the affairs in the natural sciences. Outdated philosophies of science and superseded scientific views are used to criticize science-based views of designing. These same criticisms often ignore newer ideas from the rationalist camp. For example, Winograd and Flores use the problems associated with logical-empiricism and Searle’s views on literal meaning to support their arguments regarding the applicability of hermeneutics to thought and language. Both logical-empiricism and Searle’s views are outdated and no longer supported by most rationalists. Vellino [198] expands on this in his criticism of Winograd and Flores research, citing several researchers [138] [197] whose work weakens the claims made by Winograd and Flores.

It is difficult for natural science researches to grasp concepts that cannot be phrased within the context of the rational tradition that begat the natural sciences. By the metrics of success used in the natural sciences, theories from the human sciences appear to lack rigor, predictive power, precision, and testability. Such theories are simply not valid to scientists who strive to use reason to order their worlds. The fact that many human science researchers are admittedly not interested in these metrics makes the division even worse. Because they cannot measure and analyze, rationalists claim that ideas like hermeneutics and constructivism fair no better than rational theories of language and meaning.

There are pragmatic problems associated with the interpretive view of designing. By shifting control of design towards the designers, management of the design process is affected. It becomes more difficult to tell exactly where a design effort currently stands. Predicting where a design is going and how long it will take to get there become more difficult. This is an important issue due to the increasing pace at which design work must occur today. Unlike the craftsmen of ancient times, designers no longer have the luxury of developing shared understanding over long periods of time.

Another practical problem that is aggravated by the accelerated modern world is the difficulty in recording and passing on the knowledge that is gained through interpretive processes. This is an important problem because the human sciences approach to design depends on continuity between generations of designers. Rational approaches have historically been most

suited to abstracting generalizations and explaining how things get done. In today's fast paced world, high volumes of hard-learned design lessons must be passed from one generation of designers to the next. A lack of formal methods for doing that impedes progress over time. Inexperienced designers can learn and be more productive by following rational, prescriptive methods. In addition, rational methods can detect errors in the detail that human science methods overlook.

Natural scientists try to eliminate human bias from their work. An inflammatory criticism of the human sciences is that they encourage practitioners to embrace their biases to the point that alternative views are occluded. Practitioners become "lazy" and do not expend the intellectual effort required to fully investigate phenomena, inflating the importance of their own point of view at the expense of others.

2.4. Chapter Summary

Designing is recognized as a fundamental part of human existence. Everybody "designs" when they purposefully attempt to solve a problem posed by some need. Design researchers have noticed that design is geared to situated human needs and it is therefore action-oriented (primarily concerned with transformation) and governed by the interests of involved end-users, designers, clients, and other members of society. The close association between designing and other high-level cognitive processes has led people in many fields to believe that understanding how people design is commensurate to understanding how people think. The knowledge gained in examining philosophical and methodological underpinnings to design will help determine the kinds of techniques, tools, and methods that are needed to support software design. In addition, the value of such an exploration in educating designers, making them more aware of the ways in which their designs move forward, should not be overlooked.

Design philosophy in Western society has been evolving within the context of broader movements in philosophical thought, from the Enlightenment to Post-Modernism. Ideas from the rationalistic tradition to the constructivist movements have influenced thinking about design and design methodology.

The traditional way of thinking about science, which rests on the contrast of the subject/object dichotomy, is commonly applied to design. Even the outdated ideas of logical empiricism are still seen in design methods in use today. The subject/object dichotomy is apparent in the separation of analysis and synthesis found in many design methods. The principle assumption is that an objective reality exists, a reality that can be observed and reasoned about without the observer creating a "probe effect". The emphasis is on analytical and logical

thinking, observation, empiricism, and proofs. Such thinking serves to separate mankind from nature and assert mankind's dominance over nature. The success of rationalist approaches to science has influenced design to the point where a "science" of design was promulgated by many thinkers, culminating in Simon's *The Sciences of the Artificial* [192]. Since the early days of computers, computing "science" has adopted the traditional scientific paradigm through both its theoretical teachings and its professional practice.

Rational approaches to design engender formal methods based on control and empiricism. While such methods are appropriate in many circumstances, they do not present a complete picture of the design process. In design, especially exploratory design, the creative spark seems to be struck often in situations where control is not absolute, where errors occur, where the human mind must deal with irregularity and unpredictability.

More recently, design researchers have been applying post-modern ideas of interpretation and constructivism to design. The difficulties of rationalist approaches in accounting for the human influence has provided an opportunity for what some perceive as an "anti-science" movement in design discourse, led by proponents of constructivism. Constructivists believe that a separation of object and subject is not attainable, instead a holistic view of mankind situated within nature is emphasized. Such a view embodies an awareness of how an observer constructs reality by the act of observation and how interpretation governs perceptions of reality. Interpretation is the primary process of thought and interpretations are always influenced by experiences, social situations, and context. Constructivist approaches to design hope to deal with complexity and diversity not by conquering it, but by putting into place methods that are capable of rapid adaptation and facilitation.

While science discounts and tries to eliminate the effects of human vagary, much design work directly involves, and cannot escape from, the value systems of individuals and groups. The influence of value systems has been a major part of design discourse in fields such as architecture. But it is only relatively recently that human values have become important to computing professionals who are trying to build highly interactive systems that are situated in the homes and workplaces of ordinary people. The increasing use of computers in society has led to an increased interest in applying post-modern ideas to computing science. While the subject/object dichotomy allows science to progress without regard for the individual scientist, design outcomes often are biased by the characteristics of the involved individuals and groups.

Of course, formal design methods are not mistaken or useless. A good deal of control and measurement is required for organized progress, learning, and reflection. However, the weakness of formal methods lies in their inability to take into account their own limitations (a

product of the subject/object dichotomy) and the rationalist tendency to apply them in a rigid, top-down hierarchical manner. Interpretative approaches give designers the flexibility to apply both formal and informal methods. Sometimes formalism is needed to make progress, sometimes it is not. For example, a group defining a new programming language requires the formalisms of BNF grammars and lexical analysis at some point. But during an exploratory design session the emphasis is on idea generation and establishing a shared vision of the language.

Humans have a drive to accomplish goals that they set for themselves. The rationalist inclination is to improve the degree of certainty that those goals will be accomplished by imposing some sort of structure on the thoughts and actions of those involved. Recent work in chaos theory notwithstanding, much has been accomplished by yielding to the natural tendency to break things down into parts, conduct affairs rationally, employ hierarchical organizations, etc.. However, absolute certainty can never be attained. The evolutionary nature of interpretive models of design have a built-in ability to deal with uncertain outcomes. They promote an ongoing, flexible, continuous adaptation to the situations at hand.

The nature of design is a continuum where design activities range from less structured hermeneutical activities to highly formalized methods. Good designers interpret their design situations to determine when each type of activity is appropriate. Interpretive activity sometimes prevents rational control and measurement and sometimes requires that design constructs ultimately be submitted to a rational treatment so that the design can be implemented and explained to others in a coherent manner.

This chapter has examined a wide range of ideas from rationalist to constructivist, from scientific to romantic, from logical to non-logical. No single theory of how people design has emerged as clearly superior. The strongest conclusion that can be made is that it is probable there is no “best” way to design and that flexibility is the key. While this implies that there can be no “best” technique to support design, a pragmatic approach is to determine which attributes are common in current design practices and to create methods, techniques, and tools that support those practices. To that end Chapter 3, *Software Design*, examines software design practice.

Chapter 3

Software Design

3.1. Introduction

Over the years, computing science researchers and practitioners have developed many different software design methodologies, process models, techniques, and tools. Some successfully help designers design better software, others are less successful. Some are sweeping in their scope, others specialized to a particular problem, domain, or technology. While there are no "standard" ways to design computer software, most computing professionals are familiar with several common methods and process models. Many methods are based on the technological and engineering aspects of software design, a few are based on psychology and sociology. Taking their cue from the discourse in design philosophy, many software researchers equate computer software design with "hard" sciences like mathematics [99] [98] [85] and engineering [178] [161]. Software design methods and processes have followed suit. They are often highly structured, based on formalisms [100], or employ reductionist principles.

Kapor [119] believes that most common approaches to designing software focus too much on technology and engineering. He is typical of an emerging class of software professionals [216] [153] [34] who are looking for new ways to develop the computer systems of tomorrow. Kapor calls for a change toward "the software design viewpoint", wherein software designers concern themselves equally with science and people. Cohill [34] recommends augmenting software development teams with "information architects." Information architects concern themselves less with engineering methodology and more with human factors issues, although they do not ignore technology. They all hold the belief that successful software design must combine equally the natural-science-based world of technology with the human-science-based world of the users of technology. The ultimate benefactor is the public at large, especially as software systems become integrated into their daily lives [210] [67] [151] [82] [128]. DeGrace and Hulet Stahl [57] point out that interactive computer systems design is one of Rittel and Webber's [174] "wicked" problems. They claim "traditional" engineering-based software design methods have proved unwieldy in solving these problems.

Software design's "people issues" are not limited to the involvement of users in the design process. Issues surrounding the designers themselves have surfaced. Empirical studies [51] [159] and industry surveys [139] [125] [92] often note contradictions between "industry

approved" rationalistic software development approaches and the methods software developers actually use. Many developers customize or outright abandon traditional methods after running into difficulties applying those methods. Pressures caused by budget constraints, externally imposed schedules, demand for features, and maintenance obligations often cause software designers to take "shortcuts" that maximize short term gain [225].

The development and adoption of software design processes, methods, and techniques show the influence of general design philosophies. Understanding the relationship between software design and general design will help software designers understand why software design methods succeed or fail. This chapter begins by showing how design philosophies based on the natural sciences have influenced software design methodology. The chapter goes on to discuss newer approaches to software design that are based on the human sciences discussed in Chapter 2, *Design Theory*. Finally, software design trends and research incorporating both views are discussed. This section gives some justification for adopting a mix of "hard" and "soft" [158] science in software design. The tremendous number of published software design techniques, tools, processes, and methods precludes including all in this survey. Examples are selected based on their popularity, relevance, and illustrative nature.

3.2. Natural sciences influence on software design

From its beginnings at the 1968 NATO conference, the field of software engineering has been influenced by the philosophies of science, particularly the rationalism of Descartes. This is not surprising given the many highly technical aspects of software development. In the early days of computing, the emphasis was on hardware and mathematical algorithms, both of which are direct descendants of scientific enquiry. It made perfect sense to use design models derived from analytical thinking in designing computer systems [61] [85]. Even today there is still a strong perception that, since computers are essentially math-and-logic machines at the lowest level, the activities of design and programming should also be based strictly on math and logic. Hoare [98] is representative of this belief when he states:

- computers are mathematical machines
- programs are mathematical expressions
- a programming language is a mathematical theory
- programming is a mathematical activity

Formal methods involve the building of precisely stated abstract models of the design situation followed by a constructive phase that manipulates the formal abstractions in such a way as to satisfy the design requirements [8] [113]. This implies, of course, that the

requirements themselves must be stated in very precise formal language. Usually, symbolic languages akin to the languages of logic and mathematics are used to express requirements and provide the rules for symbol manipulation. The advantage of formal methods lies in the unambiguity of the languages that are used. They rely on the ability of designers to describe requirements as a set of atomic, objective facts. Once that is done, validation and verification of design alternatives is accomplished by formally relating the precisely expressed design decisions back to the requirements. The importance of precisely and completely specified design requirements shifts the emphasis in the design process towards the "front end" analysis phase. The transformations that occur during the ensuing synthesis phase are generated and verified using well defined rules.

The combination of formal design methods, mathematical verification, and statistical testing has been used to develop "cleanroom" software development process models [189] [149]. Such models emphasize error avoidance by starting with a well-defined specification and using verifiably correct transformations to eventually produce the working software. Cleanroom processes have been shown to be effective in producing more complete, less complex software in a timely manner [186].

Example of formal methods that are used in practice are algebraic specification, model-based specification, and mathematical program verification [146] [195] [135]. Often these methods are used for safety critical systems where risk of failure must be minimized. Algebraic specification defines software entities in terms of the operations that act on the entities and the relationships between the operations. It was first used to specify abstract data types [89] and has since found use in the specification of object-oriented software [147] and as a general-purpose approach to system specification [73] [90]. Model-based specification allows designers to specify software using well defined mathematical entities such as sets, functions, and sequences. Model-based specification languages like Z [95] and VDM [112] specify system operations by using a wide variety of mathematical operations and by defining their effects on the state of the system being modeled. Formal program verification uses the machinery of mathematical proofs to establish that a program correctly implements its specification [61]. The axiomatic approach [99] to verification proceeds by inserting assertions concerning the state of the program's execution at various points and then proving that if a preceding assertion is true and the intervening code executes, then the next assertion must be true. If the next assertion cannot be shown to be true, the intervening code is incorrect.

In addition to theoretical approaches based on mathematics and logic, Cartesian rationality has influenced more pragmatic software design methods and software development processes. Most such methods are influenced by the traditional engineering fields, the methods tend to:

- prescribe desired behaviour through normative models
- stress management issues
- emphasize reductionism and atomism

Influenced by the perception that the "scientific method" serves as the foundation of "good" science, software design researchers often have tried to prescribe a "design method" for obtaining "good" designs [84] [56] [80]. Many researchers believed that design could be codified thus, even the creative aspects of design [6]. Even though application domains, technologies, and programmers are very diverse, many early software design processes and methods are very similar in their goal of prescribing desired behaviour in the form of checklists and discrete stages. These normative models concentrate on explaining how software developers *should* proceed when engaged in the design of a software system. The models explicitly or implicitly specify a series of steps to be applied during design [161]. Many also include "rules of thumb" to guide designers through the complex design process. Normative models usually include a descriptive component, but the overall intention is to prescribe a desired behavior and to dictate the way designers work.

The classic waterfall development process [178] [12] [2], with its emphasis on separate "analyze-synthesize-evaluate" phases, is an echo of the classic scientific method and logical empiricism. The analysis phase tries to break the problem into discrete, separately solvable chunks and the synthesize phase tries to derive a solution through some sort of inductive process. The solution is implemented and tested in an evaluation phase. The classic waterfall separates the "what" from the "how" [226] with no opportunity for feedback. Results of later phases do not inform earlier phases in any way, a deficiency that is overcome by later modifications to the model. One such modification is Boehm's spiral process model [14], with its "design a bit, build a bit" approach. The spiral process model is an example of an iterative design method perhaps inspired by Hillier's conjecture/analysis and Popper's hypothetico-deductive philosophies. Many other modified versions of the waterfall method have been proposed [57]. These and other prescriptive software development methods have become more sophisticated and more detailed since the original waterfall was developed and their number has grown as quickly as their degree of sophistication [164] [222] [68].

By making clear distinctions between analysis, synthesis, and evaluation, normative models are breaking the problem of designing software systems into discrete temporal chunks. Other

kinds of decomposition are also used in software design. Due to the inherent limitations in cognitive capacity of human designers, software design problems must almost always be decomposed into smaller, more manageable pieces [62]. Some methods of decomposing software design problems have been influenced by the logical positivist approach to science. Classically, these methods are algorithmic in nature and seek to create a complete decomposition of the problem before any attempt is made to solve the problem. The individual subproblems are then solved, and those solutions are combined in a grand synthesis to create the final software design. These methods are clearly influenced by the tendency in science to isolate the object of study.

Elaborating a design using functional decomposition is familiar to most software professionals. Top-down methods like step-wise refinement [220], structured analysis and design [223] [58], architectural layering [60], data decomposition [105] [209], and a host of other methods all start with a high-level view of the design problem and proceed to progressively break it down into more detailed subproblems based on the functional structure of the problem. The result is usually a hierarchical view of the system where higher levels represent more abstract concepts. Lower levels become increasing more detailed until the subproblems they represent can be directly implemented in a programming language. In theory, once all the lowest levels have been specified, the software design is complete.

There is an engineering management benefit to the systematic, disciplined decomposition of a design problem. Assuming the decomposition is detailed and accurate enough, it is relatively easy to estimate time to completion of the individual tasks implied by the lowest levels of the decomposition. These individual estimates are used to estimate and track progress of the project as a whole. Normative methods often require that specific outputs be produced by one phase and consumed in the next. These “documentation interfaces” are used to clearly separate the phases and serve as snapshots for measuring project progress. Despite some weaknesses, the waterfall method and its ilk have excelled as management tools, as was their original purpose [178]. The application of science and engineering methods to software design is a very appealing and reasonable practice, especially to control software design situations as they are scaled up [18] [137]. Design processes become tools for coordinating and organizing groups of designers. Discipline is necessary when attacking a software design problem – particularly a large one. Processes based on science and engineering have been able to provide this discipline.

The engineering and science background of early software design methodologists was an important influence on the types of methods and processes they developed. Their work was a productive and necessary step towards modern software engineering practices. They were

perhaps a little naive in not anticipating some of the problems that the explosive growth of computers in society have caused software designers. Many of these problems are mentioned in the section 3.3, *Human sciences influence on software design*. Section 3.4, *Pragmatic software design*, highlights ways in which traditional "structured" methods have been adapted to address human factors in software design.

3.3. Human sciences influence on software design

Software engineering is no longer just the prescription of "hard and fast" rules of system design. Fairly recently, software professionals began to realize there are many important aspects to software systems. Such issues, which may be just as important as the traditional technical issues, include economic, psychological, sociological, organizational, philosophical, political, and aesthetic issues [109] [124] [83] [206]. Thus, software development should be looked at from many perspectives, including the perspective of the human sciences. Two trends influence the interest in human sciences and software development:

- the widespread use of computers in all areas of society [25] [82]
- empirical and ethnographic observations that normative models are not flexible enough to accommodate the needs of individual software designers [88] [133]

These trends intimate software design's dependence on human factors involving both users and designers of software systems. As the products of software design become more ubiquitous in society, designers face difficulties trying to abstract requirements into consistent and complete specifications that capture precisely what the software should do. Users often don't know their own needs and introducing computers often changes users' needs in unpredictable ways. An important part of software design is understanding the context in which the software will operate [20]. Questions of how the software affects users, tasks, and its environment must be addressed during the software design process.

Researchers in human-computer interaction were among the first to realize that a user-centered approach to software design is different from the "traditional" technology-centered structured approaches [157]. The constant flux caused by the instability of users, tasks, and environment require software design processes where sociology and psychology play a more significant role [24] [150] [69]. Such process often emphasize flexibility. They "grow" the software design in the environment in which the software will be used. Indeed, the software itself should remain adaptable throughout its lifetime [20] [172]. While Hoare's programming is mathematical in nature, discovering the requirements for the software involves much more than mathematics and logic. This is the motive for almost all human sciences based software design methods.

A criticism voiced by many software designers and researchers points out that normative models are inflexible in the face of changing requirements and constraints [144] [202] [76] [57] [225]. These critics note that constantly changing requirements is an attribute of almost every software design project. Normative models rarely acknowledge that software design involves highly iterative, interleaved, loosely ordered tasks [87]. Software design work is not always “balanced”, partial solutions at different levels of abstraction are common. Unexplained jumps between levels of abstraction are observed [88]. The dynamic nature of software development, the ease with which software can be changed, and the complexities of dealing with errors and uncertainty conspire to make software design a globally undeterministic activity [17] [76]. Forcing software designers to follow strict normative models, especially those that separate specification from implementation, places them in a straight-jacket that prevents them from dealing effectively with these issues.

A common theme is methodological pluralism. Software designers themselves will admit they rarely follow a single normative process in a strict manner. Instead, they adapt to the current situation based on their experience or the realization that they need to learn more about the situation [1]. When asked what they do when they design software, they will reply with comments like “you use what works” [141], “it's such a personal choice” [207], or “it depends on the situation” [37]. This real-world inclination to adapt methods to both designer and design situation is perhaps what inspired Feyerabend’s [70] attitude that the idea of a fixed method rests on too naive a view of people and their social surroundings. Only one principle can be defended under all circumstances, that is, the principle “anything goes.”

Mechanisms that support a gradual building of the software designer’s understanding are important. The success of a design depends on the designer’s ability to get feedback on ideas and to adjust the design process in response to that feedback. Communication, particularly in the form of dialog, plays a large role in software design. Designs come about as designers present their understandings of the situation to each other and to users. They bring up past experiences, clarify one another’s comments, create new abstractions and metaphors, discuss scenarios, draw pictures, etc..

Using this dialogical model of designing, the software designer and the design conduct a “conversation” during which the designer comes to “know” the design [183]. Software design practice is influenced by the work of design researchers, methodologists, and tool builders interested in communication. Many have looked more closely at how designers communicate among themselves and with users. While most of the rationalist approaches to software design try to formalize and prescribe the ways in which designers communicate [129], most human science based approaches use interpretive models as a basis for providing

support to the communication and social needs of designers [36] [117] [118] [215] [93]. They try to find ways to help software designers build a shared understanding of the design with minimal adverse interference to the communication patterns they use normally.

Many software design methods that try to overcome the weaknesses of the normative models use an "exploration" metaphor. The Oxford English Dictionary defines exploration using phrases like "connected with investigation or searching" and "to range over for the purpose of discovery." Most conventional software development approaches make a clear distinction between describing *what* should be implemented and developing a plan for *how* it should be implemented. Within these approaches, software design is concerned with determining the internal structure of a software system whose functions are fully described in some sort of requirements document. In practice, however, design activities often illuminate deficiencies in the knowledge about what should be implemented. Many software design activities thus become directed toward finding, understanding, and refining the software system's requirements. These activities mark the beginning of exploratory design [17]. Almost all purposeful design activities involve some exploration of the designed artifact's constraints, requirements, context, function, and form. Exploratory design is part of a new emphasis on human science based software design as an activity to complement and inform the engineering of software [119] [217] [34].

Exploratory software design is used whenever a complete understanding of the situation is not obvious. Whenever one or more designers don't know exactly what to do next, they will engage in some sort of exploration. For example, Wirfs-Brock, et al., use the term "exploratory design" to describe the initial discovery of objects and their responsibilities during object-oriented design [219].

Some techniques seem to be particularly suited to exploratory design. Three examples are: Class-Responsibility-Collaboration (CRC) cards, scenario (or use-case) elaboration, and prototyping. The three are used independently or together to help multiple designers form a "common horizon" [171] and act as repositories for the "theory", or rationale, of the software system [153].

The CRC card method [212] [10] uses index cards to represent design entities. CRC cards give designers a way to represent exploratory design entities with physical tokens. The cards take on anthropomorphic qualities and, in a sense, become a mechanism for the design itself to communicate with the designers. Designers often spatially distribute the cards to match the current avenue of exploration, they observe the interactions implied by the cards, and then they rearrange and edit the cards. In addition to software design, the CRC concept has been used in education [9] and software development process evaluation [38].

CRC cards are used often as a kind of “stand-in” for object instances during simulations of a proposed system. Such simulations are called “scenarios” [30] [29] [121]. Scenarios are instances of “use-cases”, which specify the proposed system’s behaviour [106]. Scenarios are simulated sequences of events that provide a mechanism for focusing on specific system behaviours. Through scenarios, designers can communicate and document the semantics of the software being designed. Scenarios drive software designs by motivating discourse and helping designers explore the implications of various decisions. In addition to helping designers, scenarios can help managers create more accurate schedules by giving a good indication of the size and complexity the software system [16].

Nielsen [155], Rettig [173], and Booch [16] all discuss the relationship between scenarios and prototyping. While scenarios are a simpler low-cost relative of prototyping, sometimes more elaborate and expensive executable prototypes are required to fully explore a design situation. Prototypes are usually executable programs that model a subset of whole system functionality. They help define system requirements by allowing designers to deploy working software in the user’s environment. The software is used in experiments that are watched by designers to find flaws in the design and to answer specific questions regarding the design. Prototyping is a venerable software design technique that has found growing support and use in the software design community [81] [3]. Some researchers have even proposed that prototyping forms the basis for an entire software design process wherein prototypes are eventually converted to final products [134] [55]. Others point out the difficulties with such an approach. Alavi [3] notes that designers who use prototyping can experience difficulties managing and controlling the design process. Davis [55] points out the quality problems that can arise when “throw away” prototypes are retrofitted in the rush to release a working system.

In addition to CRC cards, scenarios, and prototyping, the recent work in software design patterns [33] [74] [39] [169] is an important part of exploratory software design. Designers determine and interpret requirements based on their experiences with similar design problems and applications domains [1]. Patterns provide a “ready made” experience repository for designers who don’t have direct knowledge of a particular software design model. Design patterns influence the primary generators [53] designers use to drive their designs. And, in turn, the design patterns they choose influence how they go about understanding and elucidating requirements. Patterns provide broad alternative approaches and architectures to investigate with an emphasis on finding a match between the current design situation and those previously encountered.

Exploratory design involves those parts of the design process characterized by exploration of the relationship between the problem domain and the solution domain, creation of scenarios of use, discovery of implementation constraints, and rapid generation of design alternatives. Unlike rational approaches to software design, exploratory design allows the introduction of possible solutions before the problem is fully defined [57]. In this way, exploratory software design acts as a bridge between requirements analysis and wholesale detailed design. Exploratory design sets the stage for more analytical "downstream" design work by creating initial frameworks. Exploratory software design clarifies the problem, creates a foundation for further design, identifies promising approaches, and tries to find potential technical problems that will require closer scrutiny.

As long as the amount of detail is appropriate, exploratory design can be done at any level of detail from sweeping architectural alternatives to particular communications protocols. Factors limiting the efficacy of exploratory software design include the experience level and the cognitive capacity of the designers themselves. Designers who lack experience may have difficulty finding appropriate patterns or determining which details are important. Human cognitive capacity is a natural limiting factor on the amount of information that can be examined simultaneously during exploratory design.

Good tools allow more information to be brought to the table during exploratory design. They help designers to overcome knowledge and short term memory capacity limitations. Developing tools to support exploratory design constitutes an important design problem in its own right. To maintain the flexibility of exploratory design, such tools must feel as natural as a whiteboard [120] [176] [126]. While many tools suitable for use in exploratory design have been developed in different contexts, Winograd [216] points out that there is a need for integrated environments that support software design. Such environments should support software design activities much like programming environments support programming. Tools that could be integrated in such an environment include: knowledge bases that offer alternative design patterns where appropriate, responsive prototyping media, design language support tools, tools to help determine user conceptual models, and tools to facilitate communication between co-located and remote designers.

Integrated design environments do not yet exist, but many current groupware tools try to facilitate software design by mediating communication among designers and between designers and users [120] [126] [116] [199]. In addition, they often provide a foundation to support specific software design activities. Groupware tools hope to encourage frequent "high bandwidth" design discourse using various representations. High bandwidth communication between designers is important during exploratory software design. Tools

must help build a shared understanding of the design situation [201] [11]. It is important at this point for designers to be able to share their ideas and past experiences with no “probe effect” caused by tools that interfere with their communication [171].

Overcoming hurdles by discussing ideas, generating alternatives, and allowing new associations to be created are attributes of exploratory software design. Exploratory design situates new information (changes in user requirements, discovery of technical information, changes in the constraints on the system, etc.) within the context of what is already known about the design. Exploratory design is really a type of learning where the concern is “learning what must be done” [153] to solve the design problem at hand. These episodes of learning occur frequently throughout the entire software development process because of the difficulties in developing a complete and consistent prior understanding of the requirements, constraints, technical issues, and human factors issues.

Users play a central role in learning how a software system should operate. Participatory, or cooperative, design is a field that was born of the desire to involve users [32]. Cooperative design elevates users from being objects of study to a role more intimately involved in the design process. Cooperative design methods overcome the tendency for normative models to separate designers from users. They are aimed squarely at reducing the damage caused by “impedance mismatch” between software designers and users.

The focus is not only on improving the usability of a software system, it is also on the political, ethical, and sociological aspects of deploying the system in society. Influenced by the need for users and designers to acquire common shared horizons, cooperative design involves finding new ways for users to learn, participate, and cooperate with software designers. They allow software designers to gain a deeper understanding of the in situ operation of their software. They empower users to co-determine how the software will affect their jobs and lives. Thus, cooperative design is influenced by social constructivism philosophy.

Greenbaum and Kyng [83] summarize cooperative design with an analogy to traditional software design approaches. Their table comparing the two is reproduced here.

TRADITIONAL APPROACH *focus is on*	COOPERATIVE APPROACH *focus is on*
problems	situations and breakdowns
information flow	social relationships
tasks	knowledge
describable skills	tacit skills
expert rules	mutual competencies
individuals	group interaction
rule-based procedures	experience-based work

3.4. Pragmatic software design

The preceding sections have established two endpoints in a continuum. Proponents of formal techniques and processes emphasize prescription, control, discipline, correctness, logical proofs, etc.. Those who advocate less structured methods value interpretation, flexibility, creativity, social values, etc.. Neither approach is the “right” or “wrong” way to design software. Control is needed, constraints must be satisfied, local determinism is necessary, and the software must compile and run. At the same time, the creative spark must be nourished, communication must not be hindered, and the effect of software on the situation into which it is deployed must be considered.

A mature, repeatable development process is beneficial in many ways to the long term viability of any software development organization [104] [103]. Yet the process must provide sufficient latitude to encourage creativity, innovation, and user involvement. Parnas and Clements [163] claim that a totally rational design process that allows for creativity and innovation is not achievable, therefore compromises must be made. They suggest that defining and *attempting* to follow a rational design process helps to create a framework for design progress and management control but such attempts will always go astray. However, the benefits of *appearing* to have followed a rational design process are so important that it is worth creating a trail of documents just like those that would have been produced had the project been completed the ideal, rational way.

In practice, software designers mix and match rational methods with non-rational methods. Sometimes they modify a rational method to accommodate the need for flexibility or creativity Sometimes they do the opposite, adding formal design aspects to informal methods. Carroll and Kellogg [28] attribute this behaviour to the drawbacks associated with pure rational (i.e. quantitative) methods and with methods based solely on the human sciences (i.e. hermeneutical interpretation). Strictly rational methods don't work because "the limited scope of quantitative theories precludes adequate grounding for design decisions" and strictly interpretive methods are weakened because "bridges from hermeneutic interpretation into design decision-making are essentially mystical. There is no systematic methodology, no conceptual framework, no explicit way to abstract from particular experiences." Carroll and Kellogg think that using design methods based on an artifact's psychological "claims" (i.e. rich qualitative descriptions) helps to reconcile formalism and hermeneutics by "enriching the vision of the former and disciplining that of the latter." Interpretations are valid insofar as they produce results that can be tested against the psychological claims.

Combining informal unstructured techniques with formal structured techniques mirrors the continuum of the informal ideas and values that spark a software project to the necessarily formal and structured software and hardware that implements the project. In addition, there is psychological, empirical, and ethnographic evidence to suggest that software designers are likely to produce better designs if they effectively combine formal and informal methods. Section 3.4.1, *Psychology and software design*, and section 3.4.2, *Empirical and ethnographic studies of software design*, discuss psychological and empirical evidence that indicate a balance is required. Finally, section 3.4.3, *Practical software design methods*, discusses some modern software design methods that might best fulfill the need for balance.

3.4.1. Psychology and software design

From a psychological point of view, claims that software design is best served by a blend of methods is illustrated by Kay's [122] review of the work of Piaget, Papert, Bruner, and Hadamard. Using the results of a famous set of experiments [166], Piaget developed a theory that traced the intellectual development of children through three stages characterized by what Kay calls "action", "image", and "symbol". Intellectual activities of children in the first stage are manifested in their actions. For children at this stage, thinking *is* doing (touching, grabbing, banging, tasting, etc.). More abstract notions like planning, consequence, and judgment are not apparent.

In the next stage, the visual channel becomes the dominant force behind intellectual activity. At this stage of development, images play a crucial role in children's perceptions of reality. For example, in one experiment children observed the same quantity of liquid poured from

identical measuring cups into each of two glasses, a short wide glass and a tall narrow glass. When asked which glass contains more liquid, children consistently choose the tall narrow glass because the liquid level *looks* higher.

The final stage of intellectual development begins in the teenage years and continues into adulthood. The acquisition of principles of logic and symbolic representations embodied in the rationalist view of intelligence characterizes teenagers and adults. At this stage, young people create abstractions of the world around them, plan, and make predictions based on manipulations of those abstractions. The degree to which a particular person is able to attain this level of intellectual development is often used to ascertain the intelligence and even the worth of the individual.

As an example of Piaget's theories, Kay recounts Papert's experiences teaching children to design software using the Logo programming environment [162]. Papert observed a marked variation in the way children at different developmental stages approached the task of drawing a circle. For the youngest children, the best way to develop an algorithm for drawing a circle was to ask them to close their eyes and draw a circle by moving their bodies in a circle. Their Logo programs for drawing a circle thus involved repeated application of "move a bit, turn a bit" steps. At the next stage of development, older children were inclined to think about drawing circles geometrically. The algorithms they developed were based on the observation that a circle consists of a series of points equidistant from the circle's center. Their programs consisted of repeated application of "move a distance equivalent to the radius from the center, draw a point, return to the center, turn a bit" steps. Finally, the oldest children, those who had reached the symbol-oriented stage, tended to think of a higher order symbolic representation of circles. Consequently, their Logo programs involved plotting points that solved the algebraic equation for a circle.

Piaget's theories affirm the rationalist notion that by the time people become adults their intellectual activities are symbol oriented. They progress through the lower order stages of action and image and attain the highest level of intellectual development. They have left their childish (and thus inappropriate for adults) behaviour behind them. Whether Piaget's theories independently confirm the rationalist stance or are merely a product of them has not been fully addressed. For example, is the third stage a product of an intrinsic rationality or merely the result of an education based on rationalist principles?

Both the development of intellect and the development of a software design have the property that later stages require more generalized knowledge than do earlier stages. Lack of generalized knowledge in the earlier stages is overcome by doing and observing. It is this doing and observing that forms the basis for most iterative software design methods.

These experimental results can be used to create abstractions and generalization useful in creating abstract symbolic representations. However, in practice this abstraction process cannot or does not occur as readily as one would think, perhaps due to the extreme complexity of a situation, the use of an inappropriate frame of reference, or even technological and socio-cultural obstacles. When the required knowledge has not been attained, earlier stages are (or must be) used to make progress. For example, building software prototypes lets software designers experiment with different solutions (the "action" part) and observe the prototype in operation (the "image" part). The experience thus obtained can be used in future similar situations but, when the situation is too complex or each instance is too different, there is no guarantee that a strictly rational stage can ever be reached. This is sometimes seen in computer science when theoretical knowledge is gained only through a constant process of exploratory programming.

While Piaget's work implied that children move through each stage in sequential order, leaving each earlier stage completely, Bruner [23] showed that although certain characteristics may dominate at certain times, the others are still present. Individual and situational variation can cause the characteristics of all stages to show up. This view is supported by an empirical study conducted by Hadamard. He asked the top 100 scientists in the world what they do when working. Very few reported they use symbol-oriented approaches. The vast majority said that they worked with imagery and visual representation. A surprising 30 percent reported that they "felt" their work in the sense that they actually had kinesthetic experiences while working. Bruner's experiments and Hadamard's empirical work seem to indicate that, regardless of society's general perception of the importance of rational/symbolic intellectual activity, intellectual discovery really occurs at many different levels simultaneously.

Kay treats the design of computer user interfaces as the creation of objects to satisfy computer users' needs for actions, imagery, and symbolic manipulation. The success of graphical user interfaces and kinesthetic devices (like the mouse) are a testament to the importance of image and action to our interaction with computers. While Kay applies these principles to the interaction with the users of the final software artifact, they can also be applied to the processes and tools used in designing those artifacts. The activities comprising software design bear striking similarities to many of the activities of software users. In the context of Chapter 2's discussion classifying "design" activities, writing a book, creating a brochure, and making a presentation are all software supported design activities.

Labouvie-Vief [131] discusses another style of thinking that contrasts, or perhaps complements, formal symbol oriented thinking. "Postformal" thinking goes beyond

context-free rational thinking to include the context in which a problem is embedded. Emotion and social concerns are recognized as important parts of thinking and being in the world. Rybash et al. [180] note that "real-life problems, in contrast to formal problems, are 'open' to the extent that there are no clear boundaries of a problem and the context within which it occurs." Thus, postformal thinking is more concerned with problem-setting, or problem understanding, rather than problem-solving, or logical analysis. Research indicates that, as people age and become more experienced, they are more likely to engage in postformal thinking before committing themselves to a formal solution-oriented thought process.

For most people, action and images are dominant in the early parts of software design. Formal symbol systems become more important closer to the implementation of the software, which must occur in a totally rational, symbol-based system. Postformal thinking is important throughout to ensure that the "right" problem is being solved. The answer to the question of how the stages of action, imagery, and symbol manipulation are used during software design can perhaps best be understood by appealing to the theory of hermeneutical interpretation. Hermeneutics serves as the glue that binds a particular software designer's approach to a particular design problem. Because individuals interpret situations differently depending on their personal experiences, knowledge, and abilities, there is no single "correct" way to approach a complex design situation. Designers try things, observe consequences, and attempt to abstract ideas as they see fit for a particular situation. The codification and dissemination of the knowledge that they gain proceeds at a pace bounded by their abilities to absorb new ideas and, perhaps more importantly, by the opportunities given to them to try these new ideas out in practice.

The single most important conclusion that can be drawn regarding software methods is that such methods must support many different modes of intellectual activity, not merely the manipulation of formal symbolic systems. Although, formal systems must be introduced to realize the software artifact, designers are just as likely to work in the kinesthetic or visual modes. For example, one software designer might prefer to manipulate physical tokens (e.g. CRC cards) that represent design objects, perhaps moving them around while thinking about how the software will operate. Another might prefer to draw various diagrams (e.g. class diagrams and object diagrams) to develop and convey the software design. A third designer might be more comfortable using formal notations (e.g. Z or VDM). Commonly, a single designer uses all three modes at different times during a specific design process.

3.4.2. Empirical and ethnographic studies of software design

Empirical methods are not new, they have been used in the design of man-machine interfaces well before their utilization in the design of computers and software [31]. Many different kinds of empirical and ethnographic studies are used by software designers and those studying software design methods and processes. Examples of such techniques are usability testing [156], quantitative studies of problem solving behaviour [94] [115], protocol and "think aloud" analysis of designer behaviour [27] [208], interviews with participants in large-scale development efforts [125], and in situ observations of design group interaction [150]. Not only are empirical techniques increasingly being used to study software designers, they are now regularly used by software designers themselves. Software designers use empirical techniques to study the interaction of their designs with the user community [79]. In addition, they use such techniques to study and improve their design processes through such techniques as software metrics [47].

The hallmark of empirical methods is the direct observation of people as they perform well defined activities under controlled conditions. The proponents of empirical methods stress that such methods provide direction to the development of software design methods tailored to the observed needs of designers and users. Because of their behavioural orientation, these methods often employ techniques borrowed from the cognitive sciences to develop models of how people perceive the tasks under study. In contrast, ethnographic studies rely more on rich qualitative description. They serve to communicate experiences in a manner that is useful for educating others. They often provide a richness of context that is not possible to obtain through purely quantitative experiments. Ethnographic studies are sometimes the only feasible way to learn from real-life situations that happened in the past or are too expensive to study using more controlled methods.

As computers leave the data processing facility and enter the offices and homes of ordinary people, interest in the interaction between users and software systems has increased. One of the first design techniques used to address this trend was "task analysis" [148] [59]. Task analysis involves observing and classifying user actions. Task analysis is commonly used to determine functionality requirements and to generate and validate design alternatives. In traditional design processes, such analysis is conducted early with little opportunity for designers and users to later update the data given the influence of the software system on the user's environment. As design processes become more user-centered, empirical "user testing" methods have been developed to overcome this weakness. Such methods have been used extensively to test user interface designs by observing users as they attempt to accomplish tasks by working with live software [155] [79] or prototypes [173]. But even

these methods tend to separate the designers and the software design from users. Cooperative design researchers believe this separation is detrimental to the eventual acceptance and successful deployment of the software. Recall from section 3.3, *Human sciences influence on software design*, cooperative design tries to make users an integral part of the design team. In some sense, empirical studies have helped to illuminate the gap in designer/user communication created by traditional methods. They have provided impetus for the development of methods that try to bridge the gap.

In addition to empirical studies involving users, the study of the interaction between software designers has influenced software design processes. The complex social and psychological issues that surface during software design are being dealt with using methods that measure the physical, psychological, and social responses of designers. Such methods are divided between those that study the "micro", or designer-centric, aspects of design and those that study the "macro", or process-centric, aspects of software development. Generally, studies of individual designers or small groups of designers are more likely to involve strictly quantitative methods while larger organizational studies involve anecdotal and ethnographic methods.

When empirical studies concentrate on the activities of individual software designers, they often observe:

- breakdowns [91] [86] [125] [208]
- opportunistic behaviour [1] [88] [51]
- cognitive biases [1] [196]
- methodological diversity [1] [176] [208]

In their studies of software design, Guindon et al. [86] observe Gadamer-style "breakdowns", especially during the early phases of software design. Breakdowns usually involve a lack of knowledge or a miscommunication that requires the designer to stop and sort out the problem. Guindon et al. find that the design is dependent on the nature of the breakdowns that occur during its elaboration. In addition, the breakdowns that occur are partially dependent on the prior experiences of the designers. Since different designers have different backgrounds, they experience different breakdowns and thus produce different designs.

Opportunistic behaviour is observed when designers move between levels of abstraction. Since insight during design requires establishing a relationship between the problem domain and the solution structures [115], designers constantly shift their attention between a "high level"

domain oriented view and a “low level” implementation oriented view [88]. Thus, empirical studies show that designers do not use strictly rational top-down decomposition methods unless both the application domain and the implementation structures are already well known [108] [50].

Guindon and Curtis [87] give an account of opportunistic design elaboration that closely resembles Snodgrass and Coyne’s [194] discussion of the hermeneutical circle in understanding metaphors. In both cases, an iterative, back-and-forth process is undertaken that eventually arrives at an understanding (i.e., a design or a metaphor meaning). Guindon and Krasner [86] report that designers often engaged in an exploratory form of design to help them better understand requirements by reinterpreting their current understanding of the design situation.

Empirical studies reveal that irrational cognitive biases influence how designers think and act. Stacy and MacMillan [196] observe that biases can “block” designers from fully evaluating all consequences of design decisions. For example, they might pay more attention to confirming evidence and ignore disconfirming evidence when testing design decisions. Biases also exist due to different levels of knowledge and experience among designers. Adelson and Soloway [1] find that different levels of experience with the application domain can profoundly affect software design outcomes. In addition, experience with particular design patterns can cause designers to ignore others that may be more suited to the design task at hand. Empirical studies often conclude that design quality depends more on individual designer abilities than on anything else [48]. Good designers are better able to adapt their previous experiences to new design situations. Adelson and Soloway note that, even when designers try to use similar methods, they end up creating designs that differ significantly. The observed range of final designs reduces the feasibility of the positivist expectation that, by following design methods based on the rationalistic tradition, any designer will produce “the” optimal design.

Formal software design methods try to avoid biases by preventing them, while pragmatic methods try to anticipate biases and include ways of dealing with them. Stacy and MacMillian believe that biases cannot be completely eliminated and that any software design process must include methods that acknowledge designer biases.

Empirical and ethnographic studies of software designers often observe a blend of formal and less formal design methods. Interviews with software designers [133] [187] [140] often illuminate the variety of methods they use. After studying software designers in a variety of situations, Rosson et al. [176] note that designers use “an array of tools appropriate to different design contexts.” They come to this conclusion after observing designers using a

variety of methods during design, from informal strategies to formal design elaboration methods. They believe that both approaches to software design are important and relevant.

While empirical and ethnographic studies of the group design process as a whole are becoming more common, such studies are less common than studies of individual designers and small groups of designers. The effort, expense, and control difficulties involved in studying larger groups may contribute to the difference. Many researchers, believe that such studies are necessary to properly understand how software design processes can be improved and supported. Minneman [150] advocates an approach that generates a broad ethnography and includes detailed analysis of specific interactions.

Most studies of group software development concentrate on the most visible aspect of group interaction, communication. Curtis et al. [49] [125] have identified those circumstances where designers believe communication facilitates or hinders design progress. Communication problems seem to occur most when development processes and organizational modes revolve around top-down, hierarchical models. While acknowledging that such models provide some high level management control, Curtis et al. point out that lower level design processes are detrimentally affected. They conclude that traditional software project management strategies do not provide the flexibility needed for design work. They point out the right blend of unstructured, exploratory techniques and traditional, normative methods might have been achieved in Japan. The Japanese “software factories” separate software development into a research-like (i.e. divergent then convergent) design phase followed by a structured manufacturing-like phase. While traditional normative process models help managers, they seem to be less effective at guiding the designers’ activities. One study by Hale [92] found that about half of the designers’ time could not be attributed to activities prescribed by the process model they were using.

A few in situ studies have been conducted to determine the effectiveness of traditional normative process models. For example, Boehm et al. [13] examined prototyping as an alternative to traditional, top-down, design methods. They conclude that prototyping yields smaller, easier to use, programs with less effort but the discipline of traditional methods yields more robust, coherent, and maintainable software designs. They conclude that both prototyping and structured methods have their place in software design.

3.4.3. Practical software design methods

Although formal and informal design methods address the problems of software design in different ways, in practice they often have influenced each other. Practitioners recognize the strengths and weaknesses of each style of designing and create a blend of methods that best

serves their needs. For example, incremental prototyping may provide the methodological framework for a real-time software project in which formal methods are embedded to ensure correctness of critical parts of the design. Empirical methods may be used early in the design of interactive software, giving way to an artificial intelligence approach as the design progresses towards implementation of an expert system or knowledge base. Data driven design demonstrates that early empirical work, in this case the observation of data flows within a real-life business process, can serve as a precursor to a structured top-down design [195]. Highly prescriptive process models, like SSADM [80], now sanction exploratory techniques like prototyping. Structured design documentation is sometimes produced at the end of an ad-hoc design process [163].

In trying to find a balance between formal and exploratory software design methods, software design methodologists have tried to combine the two approaches. For example, various researchers and practitioners use rigorous prototyping methods in an attempt to combine the flexibility of iterative techniques with the discipline of structured techniques. Davis [55] advocates “operational prototyping” where prototypes are built to high standards using rigorous methods. Each prototype implements well understood functionality and is used to uncover requirements that may have been missed. The prototypes are “operational” in the sense that they evolve until all new requirements are met at which time the “prototype” becomes the final product. Zave [226] discusses “operational specification” wherein formal specifications modeling the problem are created using executable languages. The specifications are exercised as prototypes against the known requirements in an attempt to discover weaknesses and omissions. Finally, the specifications are transformed into a running program implemented using a “real” programming language running on a “real” system. In practice, the specification language is not executed, it is only used in scenario elucidation and use-case development. For example, the data-decomposition oriented techniques and notations of Jackson System Development have been used as operational specifications [26]. Luqi [130] also presents a more formal specification based prototyping technique whose rigor allows it to be better supported by automated tools.

Another example of combining formal and informal methods involves a formal approach to scenario analysis proposed by Hsia et al. [102]. Their approach tries to retain the user and application domain focus of scenarios while introducing a systematic way to analyze, generate, and validate the scenarios. The goal is to provide a disciplined approach to avoid incomplete or missing requirements. Their approach systematizes scenarios by creating a decision tree representation of the events that comprise a use-case. The trees are converted to a corresponding formal grammar and state machine. The grammar and state machine are

used to validate the use-case for consistency and completeness and to generate particular instances, or scenarios (i.e. particular sequences of events that conform to the various representations of the use-case). Designers can use the scenarios in normal design discourse or to automatically generate prototypes and acceptance test plans.

For Hsia et al., the generation of scenarios depends on user involvement and designer interpretation of user needs. Once generated, the scenarios can be analyzed using formal techniques. Edmonds et al. [65] provide another example of augmenting interpretive processes using more formal methods. They use pattern recognition techniques from artificial intelligence to support, but not replace, interpretive design activities. Case based reasoning [184] tries to capture design experience in a more formal way by using persistent scenarios, or "cases", that are matched against the design problem at hand. These approaches are representative of a number of AI-based tools for design support [75] [114].

In their discussion of formal specification methods, Duke and Harrison [64] note that even formal, structured, top-down specifications can be developed using exploratory methods. Formalizing the output of exploratory methods illuminates important hidden issues and highlights questions that may have not been properly considered early on. These questions may then be answered by another round of exploration. Parnas and Clements [163] also discuss the presentation of exploratory results using more traditional structured documentation methods. They believe that such an approach is a good compromise between the conflicting needs of the project managers and software designers. Booch [16] advocates a process model characterized by reconciling "macro" and "micro" processes. The macro process is very similar to the traditional waterfall and serves as a controlling framework for the micro process. The macro process is used by project managers and outsiders to gauge the progress of the project. The micro process is an ongoing, iterative, and incremental development process. It is used by software designers who utilize informal development methods on a day-to-day basis.

Object-oriented models seem to provide a blend of exploratory and formal techniques that appear to be useful in practice [15] [179] [190] [106] . Proponents of object-oriented design argue that one of its strengths is how it reflects ways of thinking that are more "natural" [15]. The methods work, it is claimed, because they complement designers' ways of thinking. Furthermore, one of the strengths of object-oriented methods is their ability to smooth the transition from initial conceptual models of the world to formalized models (i.e. executable programs) that are amenable to computation. There is a natural harmony between exploratory design, which very often takes place in the application or real-world domain, and

object-oriented design methods, which often begin by identifying relevant domain-level objects.

Use-cases provide structure to object-oriented design decomposition. Scenarios are used to "drive" the discovery of objects in both the application domain and the implementation domain. The clear relationship between domain and implementation objects facilitate Kant and Newell's [115] "insight" mechanisms. In addition, the principles behind object-orientation – modularity, abstraction, encapsulation, reuse – have firm theoretical foundations [71].

Object-oriented design provides a vocabulary for communication between designers and users in cooperative design episodes. Booch [15] notes that object diagrams, used extensively in object-oriented software design, have also been used independently in fields as diverse as astronomy and banking. Object-oriented design facilitates many interpretive methods, such as CRC cards, design patterns, and scenarios. Objects are often anthropomorphized during dialogical exchanges in design sessions. They provide a way for designers to "listen" to the design by imagining themselves to be the objects. The design-a-bit, implement-a-bit strategy that is often employed in object-oriented design echoes the iterative, opportunistic nature of software design. Object-oriented design methods are one of the few places that acknowledge the exploratory nature of software design [218].

Critics of object-oriented methods have mentioned that such methods lack a discipline necessary for large-scale software design [224]. Recent work by Jacobson [107], Booch [16], and Rumbaugh et al. [179] have tried to show how object-oriented methods can be integrated into full-featured process models. Indeed, the methods proposed by these authors seem to be converging toward a standard for object-oriented software development. They hope such a standard will retain the exploratory nature of software design while providing the discipline needed to properly manage the overall process of software development.

Only time will tell if such a standard will successfully unite formal and structured methods with informal and exploratory methods. Meanwhile, software design practice and software engineering research continue to refine established design methods and discover new ways of designing software. Since designing software is an activity ultimately limited only by human imagination, it is likely that it will never be "easy." However, software design methods and tools can enable designers to reliably develop new kinds of software in important application domains. When these methods and tools are based on a sound theoretical understanding of human nature and thought, they have the potential to vastly elevate the productivity and quality of software designers.

References

[1] Adelson, B. and Soloway, E., "The role of domain experience in software design," *IEEE Transactions on Software Engineering*, vol. 11, no. 11, 1351-1360, Nov. 1985.

[2] Agresti, W.W., "The Conventional Software Life-cycle Model: Its Evolution and Assumptions," in *New Paradigms for Software Development*, Agresti, W.W., Ed. IEEE Computer Society Press, 1986.

[3] Alavi, M., "An Assessment of the Prototyping Approach to Information Systems Development," *Communications of the ACM*, vol. 27, no. 6, 556-563, Jun. 1984.

[4] Alexander, C., *Notes on the Synthesis of Form*. Cambridge, Mass.: Harvard University Press, 1964.

[5] Alexander, C. and Poyner, B., "The atoms of environmental structure," in *Developments in Design Methodology*, Cross, N., Ed. John Wiley & Sons, 1984, chap. 2.2, pp. 123-133.

[6] Archer, L.B., "Systematic method for designers," in *Developments in Design Methodology*, Cross, N., Ed. John Wiley & Sons, 1984, chap. 1.3, pp. 57-82, Originally published by The Design Council, London (1965).

[7] Avizienis, A., "The N-version approach to fault-tolerant software," *IEEE Transactions on Software Engineering*, vol. 11, no. 12, 1491-1501, Dec. 1985.

[8] Bauer, F.L., "From Specifications to Machine Code: Program Construction through Formal Reasoning," in *Proc. 6th International Conference on Software Engineering*, IEEE, 1982, pp. 84-91.

[9] Beck, K. and Cunningham, W., "A Laboratory For Teaching Object-Oriented Thinking," in *Proceedings of Object-Oriented Programming Systems, Languages, and Applications*, 1989, pp. 1-6.

[10] Beck, K., "CRC: Finding objects the easy way," *Object Magazine*, vol. 3, no. 4, 42-44, Nov./Dec. 1993.

[11] Bly, S.A., Harrison, S.R., and Irwin, S., "Media Spaces: Video, Audio, and Computing," *Communications of the ACM*, vol. 36, no. 1, 28-47, Jan. 1993.

[12] Boehm, B.W., "Software Engineering," *IEEE Transactions on Computers*, vol. C-25, no. 12, 1226-1241, Dec. 1976.

[13] Boehm, B., Gray, T.E., and Seewaldt, T., "Prototyping vs. Specifying: A Multiproject Experiment," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 3, 290-302, 1984.

[14] Boehm, B.W., "A spiral model of software development and enhancement," *ACM SIGSoft Software Engineering Notes*, vol. 11, no. 4, 14-23, 1986.

[15] Booch, G., *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, 1994.

[16] Booch, G., *Object Solutions: Managing the Object-Oriented Project*. Menlo Park, CA: Addison-Wesley, 1996.

[17] Bradley, G., "Control vs. Creativity: Software Engineering at a Crossroads," in *Human Aspects in Computing: Design and Use of Interactive Systems and Work with Terminals*, Bullinger, H.J., 1991, pp. 561-565.

[18] Branson, M. and Herness, E., "The object-oriented development process," *Object Magazine*, vol. 3, no. 4, 66-70, Nov. 1993.

[19] Broadbent, G., "Design and theory building," in *Developments in Design Methodology*, Cross, N., Ed. John Wiley & Sons, 1984, chap. 4.3, pp. 277-290.

[20] Brooke, J., "Usability, Change, Adaptable Systems and Community Computing," in *Human Aspects in Computing: Design and Use of Interactive Systems and Information Management*, 1991, pp. 1093-1097.

[21] Brooks, F.P., *The Mythical Man-Month*. Addison-Wesley, 1975.

[22] Brown, J.S. and Newman, S.E., "Issues in Cognitive and Social Ergonomics: From Our House to Bauhaus," *Human-Computer Interaction*, vol. 1, 359-391, 1985.

[23] Bruner, J., Goodnow, J., and Austin, G., *A study of thinking*. New Brunswick, NJ: Transaction Books, 1986.

[24] Bucciarelli, L.L., "An ethnographic perspective on engineering design," *Design Studies*, vol. 9, no. 3, 159-168, 1988.

[25] Burnham, D., *The Rise of the Computer State*. Random House, 1982.

[26] Cameron, J.R., "An overview of JSD," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, 222-240, Feb. 1986.

[27] Carroll, J.M., Thomas, J.C., and Malhotra, A., "Clinical-experimental analysis of design problem solving," *Design Studies*, vol. 1, no. 2, 84-92, 1979.

[28] Carroll, J.M. and Kellogg, W.A., "Artifacts as theory-nexus: hermeneutics meets theory-based design," in *CHI Proceedings*, 1989, pp. 7-14.

[29] Carroll, J.M. and Rosson, M.B., "Human-Computer Interaction Scenarios as a Design Representation," in *Proceedings of the Hawaii International Conference on System Sciences (HICSS)*, 1990, pp. 555-561.

[30] Carroll, J.M., *Scenario-Based Design: Envisioning Work and Technology in System Development*. John Wiley & Sons, 1996.

[31] Chapanis, A., *Man-Machine Engineering*, Behavioral science in industry series. Belmont, California: Wadsworth, 1965.

[32] Clement, A. and den Besselaur, P.V., "A Retrospective Look at PD Projects," *Communications of the ACM*, vol. 36, no. 4, 29-37, Jun. 1993.

[33] Coad, P., "Object-Oriented Patterns," *Communications of the ACM*, vol. 35, no. 9, 152-159, Sep. 1992.

[34] Cohill, A.M., "Information architecture and the design process," in *Taking Software Design Seriously*, Karat, J., Ed. Academic Press, 1991, chap. 5, pp. 95-113.

[35] Brooks, F.P., "No silver bullet: essence and accidents of software engineering," *IEEE Computer*, vol. 20, no. 4, 10-19, Apr. 1987.

[36] SIGOIS Bulletin: Workshop on software architectures for cooperative systems, Benford, S., Johnson, P., Rodden, T., Dix, A., and Kaplan, S., ACM Press, ACM Special Interest Group on Office Information Systems, Apr. 1995.

[37] Buie, S., Private communication.

[38] Coplien, J.O., "Examining the Software Development Process," *Dr. Dobb's Journal*, vol. 19, no. 11, 88-97, Oct. 1994.

[39] *Pattern Languages of Program Design*, Coplien, J.O. and Schmidt, D.C. (eds), Addison-Wesley, 1995.

[40] Le Corbusier, *Towards a New Architecture*. London: Architectural Press, 1946.

[41] Coyne, R. and Snodgrass, A., "Is designing mysterious? Challenging the dual knowledge thesis," *Design Studies*, vol. 12, no. 3, 124-131, 1991.

[42] Coyne, R., "Computers and praxis: How the theoretical is giving way to the pragmatic in computer systems design," Jul. 1993, Department of Architectural and Design Science, Universit of Sydney.

[43] Cross, N., Naughton, J., and Walker, D., "Design method and scientific method," in *Design: Science: Method. Proceedings of the 1980 Design Research Society Conference*, 1980, pp. 18-29.

[44] *Developments in Design Methodology*, Cross, N. (ed). John Wiley & Sons, 1984.

[45] Cross, N., "Designerly ways of knowing," *Design Studies*, vol. 3, no. 4, 221-227, 1984.

[46] Cunningham, W., Private communication.

[47] Curtis, B., "Measurement and Experimentation in Software Engineering," *Proceedings of the IEEE*, vol. 68, no. 9, 1144-1157, 1980.

[48] Curtis, B., Krasner, H., Shen, V., and Iscoe, N., "On building software process models under the lamppost," in *IEEE International Conference on Software Engineering*, vol. 91987, pp. 96-103.

[49] Curtis, B., Krasner, H., and Iscoe, N., "A field study of the software design process for large systems," *Communications of the ACM*, vol. 31, no. 11, 1268-1287, 1988.

[50] Curtis, B., "...But You Have to Understand, This Isn't the Way We Develop Software at Our Company," Tech. Rep., MCC Technical Report, STP-203-89, 1989.

[51] Curtis, B., "Empirical Studies of the Software Design Process," in *Human-Computer Interaction - INTERACT '90*, 1990, pp. xxxv-xl.

[52] Dahlbom, B., "The Idea that Reality is Socially Constructed," in *Software Development and Reality Construction*, Floyd, C., Züllighoven, H., Budde, R., and Keil-Slawik, R., Eds. Berlin Heidelberg: Springer-Verlag, 1992, chap. 3.3, pp. 101-126.

[53] Darke, J., "The primary generator and the design process," *Design Studies*, vol. 1, no. 1, 36-44, 1979.

[54] Dasgupta, S., "The structure of design processes," in *Advances in Computers*, Yovits, M.C., Ed. Academic Press, 1989, pp. 1-67.

[55] Davis, A.M., "Operational prototyping: a new development approach," *IEEE Software*, vol. 9, no. 5, 70-78, Sep. 1992.

[56] of Defense, U.S.D., *Defense System Software Development, DOD-STD-2167*, Jun. 1985.

[57] DeGrace, P. and Hulet-Stahl, L., *Wicked Problems, Righteous Solutions : A Catalogue of Modern Software Engineering Paradigms*. Englewood Cliffs, New Jersey: Yourdon Press, 1990.

[58] DeMarco, T., *Structured Analysis and System Specification*. New York, NY: Prentice-Hall, 1979.

[59] *Task analysis for human-computer interaction*, Diaper, D. (ed). New York, NY: Halsted Press, 1989.

[60] Dijkstra, E.W., "The Structure of the T.H.E. Multiprogramming System," *Communications of the ACM*, vol. 11, no. 6.

[61] Dijkstra, E.W., *A Discipline of Programming*. Englewood Cliffs, N.J.: Prentice-Hall, 1976.

[62] Dijkstra, E.W., "Programming Considered as a Human Activity," in *Classics in Software Engineering*, Yourdon, E., Ed. New York, NY: Yourdon Press, 1979.

[63] Dilnot, C., "Transcending science and anti-science in the philosophy of design method," in *Design: Science: Method. Proceedings of the 1980 Design Research Society Conference*, 1980, pp. 112-116.

[64] Duke, D.J. and Harrison, M.D., "FSM: Overview and Worked Examples," Tech. Rep., 1995, Amodeus-2 Technical Report SM/WP44. Filename: sm_wp44.rtf.

[65] Edmonds, E.A., Candy, L., Jones, R., and Soufi, B., "Support for Collaborative Design: Agents and Emergence," *Communications of the ACM*, vol. 37, no. 7, 41-47, Jul. 1994.

[66] Eekels, J. and Roozenburg, N.F.M., "A methodological comparison of the structures of scientific research and engineering design: their similarities and differences," *Design Studies*, vol. 12, no. 4, 197-203, 1991.

[67] Elrod, S., Hall, G., Constanza, R., Dixon, M., and Rivieres, J.D., "Responsive Offic Environments," *Communications of the ACM*, vol. 36, no. 7, 84-85, Jul. 1993.

[68] Engineering, T.B., "Software Methodology Catalog," Tech. Rep., Tinton Falls, N.J., MC87-COMM/ADP-0036, Oct. 1987.

[69] Erickson, T., "Methods and Tools: Design as Storytelling," *Interactions*, vol. 3, no. 4, 30-35, Jul. 1996.

[70] Feyerabend, P., *Against Method*. London: Verso, 1975.

[71] Fichman, R.G. and Kemerer, C.F., "Object-oriented and conventional analysis and design methods - comparison and critique," *IEEE Computer*, 22–39, Oct. 1992.

[72] Freeman, P., "The nature of design," in *Tutorial on Software Design Techniques*, Freeman, P. and Wasserman, A.I., Eds. IEEE, 1980, pp. 46-53.

[73] Futatsugi, K., Goguen, J.A., Jouannaud, J.P., and Meseguer, J., "Principles of OBJ2," in *Proc. 12th ACM Symp. on Principles of Programming Languages*, New Orleans, 1985, pp. 52-66.

[74] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[75] Artificial Intelligence in Design: Workshop Preprints, Twelfth IJCAI, Gero, J.S. and Sudweeks, F., University of Sydney, 1991.

[76] Giddings, R.V., "Accomodating Uncertainty in Software Design," *Communications of the ACM*, vol. 27, no. 5, 428-434, 1984.

[77] Glanville, R., "Why design research?," in *Design: Science: Method. Proceedings of the 1980 Design Research Society Conference*, 1980, pp. 86-94.

[78] Goel, V. and Pirolli, P., "The structure of design problem spaces," *Cognitive Science*, vol. 16, 395-429, 1992.

[79] Gomoll, K., "Some techniques for observing users," in *The Art of Human-Computer Interface Design*, Laurel, B., Ed. Addison-Wesley, 1990, pp. 85-90.

[80] Goodland, M. and Slater, C., *SSADM Version 4 - A Practical Approach*. McGraw Hill, 1995.

[81] Gordon, V.S. and Bieman, J.M., "Rapid Prototyping: Lessons Learned," *IEEE Software*, vol. 12, no. 1.

[82] *Design At Work: Cooperative Design of Computer Systems*, Greenbaum, J. and Kyng, M. (eds). Lawrence Erlbaum Associates, 1991.

[83] Greenbaum, J. and Kyng, M., "Introduction: SItuated Design," in *Design at Work*, Greenbaum, J. and Kyng, M., Eds. Hillsdale, NJ: Lawrence Erlbaum Associates, 1991.

[84] *The Design Method*, Gregory, S.A. (ed), . Butterworths, 1966.

[85] Gries, D.G., *The Science of Programming*. Berlin: Springer-Verlag, 1981.

[86] Guindon, R., Krasner, H., and Curtis, B., "Breakdowns and processes during the early activities of software design by professionals," in *Empirical Studies of Programmers : Second Workshop*, Olson, G.M., Sheppard, S., and Soloway, E., Eds. Ablex, 1987, chap. 5, pp. 65-82.

[87] Guindon, R. and Curtis, B., "Control of cognitive processes during software design: What tools are needed?," in *CHI Proceedings*, 1988, pp. 263-268.

[88] Guindon, R., "Designing the design process: exploiting opportunistic thoughts," *Human-Computer Interaction*, vol. 5, 305-344, 1990.

[89] Guttag, J.V., "Abstract data types and the development of data structures," *Communications of the ACM*, vol. 20, no. 6, 396-405, 1977.

[90] Guttag, J.V., Horning, J.J., and Wing, J.M., "The Larch family of specification languages," *IEEE Software*, vol. 2, no. 5, 24-36, 1985.

[91] Hacker, W., "Designing the designer's tasks: participative analysis and evaluation of software development tasks," in *Work with Computers: Organizational, Management, Stress and Health Aspects*, 1989, pp. 163-168.

[92] Hales, C., "Analysis of the Engineering Design Process in an Industrial Context," Ph.D. thesis, University of Cambridge, 1987.

[93] Harrison, S. and Minneman, S., "The Media Space: a research project into the use of video as a design medium," Tech. Rep., Xerox-Parc, 1990.

[94] Hayes-Roth, B. and Hayes-Roth, F., "A cognitive model of planning," *Cognitive Science*, vol. 3, 275-310, 1979.

[95] *Specification Case Studies*, Hayes, I. (ed). London: Prentice-Hall, 1987.

[96] Heidegger, M., *Being and Time*. Oxford, England: Basil Blackwell, 1962.

[97] Hillier, B., Musgrove, J., and O'Sullivan, P., "Knowledge and Design," in *Environmental Psychology: People and Their Physical Settings*, Proshansky, H.M., Ittelson, W.H., and Rivlin, L.G., Eds. Holt, Rinehart and Winston, 1976, chap. 6, pp. 69-83.

[98] Hoare, C.A.R., *The Mathematics of Programming*, Inaug. Lect., University of Oxford. Clarendon Press, Oxford.

[99] Hoare, C.A.R., "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, 576-583, 1969.

[100] Hoare, C.A.R., "Programming: Sorcery or Science?," *IEEE Software*, 5-16, Apr. 1984.

[101] Hoover, S.P. and Rinderle, J.R., "Models and abstractions in design," *Design Studies*, vol. 12, no. 4, 237-245, 1991.

[102] Hsia, P., Samuel, J., Gao, J., Kung, D., Toyoshima, Y., and Chen, C., "Formal Approach to Scenario Analysis," *IEEE Software*, 33-41, Mar. 1994.

[103] Humphrey, W.S., "Characterizing the Software Process: A Maturity Framework," *IEEE Software*, vol. 5, no. 2, 73-79, Mar. 1988.

[104] Humphrey, W.S., *Managing the Software Process*. Addison-Wesley, 1989.

[105] Jackson, M., *Principles of Program Design*. London: Academic Press, 1975.

[106] Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G., *Object-Oriented Software Engineering : A Use Case Driven Approach*. Addison-Wesley, 1992.

[107] Jacobson, I., "Is object technology software's industrial platform?," *IEEE Software*, vol. 10, no. 1, 24-30, Jan. 1993.

[108] Jeffries, R., Turner, A.A., Polson, P., and Atwood, M.E., "The processes involved in designing software," in *Cognitive Skills and Their Acquisition*, Anderson, J.R., Ed. Lawrence Erlbaum Associates, 1981, pp. 255-283.

[109] Johnson, D.G. and Nissenbaum, H., *Computer Ethics and Social Values*. Prentice Hall, 1995.

[110] Jones, J.C., *Design Methods: Seeds of Human Futures*. John Wiley & Sons, Wiley-Interscience, 1980.

[111] Jones, J.C., *Essays in Design*. New York, NY: John Wiley, 1984.

[112] Jones, C.B., *Systematic Software Development using VDM*. London: Prentice-Hall, 1986.

[113] *Case Studies in Systematic Software Development*, Jones, C.B. and Shaw, R.C.F. (eds). Englewood Cliffs, N. J.: Prentice-Hall, 1990.

[114] Jones, R.M. and Edmonds, E.A., "A framework for negotiation," in *CSCW and Artificial Intelligence*, Connolly, J. and Edmonds, E.A., Eds. London: Springer-Verlag, 1994, pp. 13-22.

[115] Kant, E. and Newell, A., "Problem Solving Techniques for the Design of Algorithms," *Information Processing and Management*, vol. 28, no. 1, 97-118, 1984.

[116] Kaplan, S.M., "ConversationBuilder: An Open Architecture for Collaborative Work," in *Human-Computer Interaction - INTERACT '90*, 1990, pp. 917-922.

[117] Kaplan, S.M., "Flexible, Active Support for Collaborative Work with ConversationBuilder," in *Proceedings of the 1992 Conference on Computer-Supported Cooperative Work: Sharing Perspective*, Toronto, 1992.

[118] Kaplan, S., "Space as a Basis for Collaborative Systems," *SIGOIS Bulletin*, vol. 15, no. 3, 21-22, Apr. 1995.

[119] Kapor, M., "A Software Design Manifesto," *Dr. Dobb's Journal*, vol. 16, no. 1, 62-67, Jan. 1991.

[120] Karat, J. and Bennett, J., "Supporting effective and efficient design meetings," in *Human-Computer Interaction - INTERACT '90*, 1990, pp. 365-370.

[121] Karat, J. and Bennett, J.L., "Using scenarios in design meetings - a case study example," in *Taking Software Design Seriously*, Karat, J., Ed. Academic Press, 1991, chap. 4, pp. 63-94.

[122] Kay, A., *Doing With Images Makes Symbols: Communicating with Computers*, Video - Stanford University Video Communications, Oct. 1987.

[123] Kidder, T., *The Soul of a New Machine*. Avon Books, 1982.

[124] Kling, R., *Computerization and Controversy: Value Conflicts and Social Choices*, 2. Academic Press, 1996.

[125] Krasner, H., Curtis, B., and Iscoe, N., "Communication breakdowns and boundary spanning activities on large programming projects," in *Empirical Studies of Programmers : Second Workshop*, Olson, G.M., Sheppard, S., and Soloway, E., Eds. Ablex, 1987, chap. 4, pp. 47-64.

[126] Krueger, M.W., "Environmental Technology: Making the Real World Virtual," *Communications of the ACM*, vol. 36, no. 7, 36-37, Jul. 1993.

[127] Kuhn, T.S., *The Structure of Scientific Revolutions*, 2. University of Chicago Press, 1970.

[128] Kuhn, S. and Muller, M.J., "Participatory Design," *Communications of the ACM*, vol. 36, no. 4, 24-28, Jun. 1993.

[129] Kwiatkowska, B., "A Communication Model for the Software Systems Development Process - a Unifying Approach," Master's thesis, University of Alberta, 1991.

[130] L., L., "System Engineering and Computer-Aided Prototyping," *Journal of Systems Integration*, vol. 6, no. 1, 15-17, 1996.

[131] Labourvie-Vief, G., "Intelligence and Cognition," in *Handbook of the Psychology of Aging*, Birren, J.E. and Schaie, K.W., Eds. Van Nostrand Reinhold, 1985, pp. 500-530.

[132] Lakoff, G. and Johnson, M., *Metaphors We Live By*. Chicago: University of Chicago Press, 1980.

[133] Lammers, S., *Programmers at Work*. Microsoft Press, 1986.

[134] Lantz, K.E., *The Prototyping Methodology*. Englewood Cliffs, NJ: Prentice Hall, Inc., 1987.

[135] Larsen, P.G., Fitzgerald, J., and Brookes, T., "Applying Formal Specification in Industry," *IEEE Software*, vol. 13, no. 3, 48-56, 1996, World Wide Web page at http://www.computer.org/pubs/software/abs96.htm.

[136] Lawson, B., *How Designers Think*. The Architectural Press Ltd.: London, 1980.

[137] Lee, M.M., "Object-oriented analysis in large-scale projects," *Object Magazine*, vol. 4, no. 3, 45-49, Nov. 1993.

[138] Lewis, D., "Scorekeeping in a language game," *J. Philos. Logic*, vol. 8, 339-359, 1979.

[139] Lyytinen, K., "Different perspectives on information systems: problems and solutions," *ACM Computing Surveys*, vol. 19, no. 1, 5-46, 1987.

[140] Maccoby, M., "The Innovative Mind at Work," *IEEE Spectrum*, vol. 28, no. 12.

[141] MacDonald, S., Private communiction.

[142] MacLean, A., Bellotti, V., and Young, R., "What rationale is there in design?," in *Human-Computer Interaction - INTERACT '90*, 1990, pp. 207-212.

[143] March, L., "The logic of design," in *Developments in Design Methodology*, Cross, N., Ed. John Wiley & Sons, 1984, chap. 4.2, pp. 265-276.

[144] McCracken, D.D. and Jackson, M.A., "A Minority Dissenting Position," in *Systems Analysis and Design – A Foundation for the 1980's*, Cotterman, W.W., Ed. Elsevier Science Publishing Co., Inc., 1981, pp. 551-553.

[145] Medawar, P.B., *Plato's Republic*. Oxford, England: Oxford University Press, 1982.

[146] Meyer, B., "On Formalism in Specifications," *IEEE Software*, 6-26, Jan. 1985.

[147] Meyer, B., *Object-oriented Software Construction*. Prentice-Hall, 1988.

[148] Miller, R.B., "A method for man-machine task analysis," Tech. Rep., WADC Technical report, 53-137, 1953.

[149] Mills, H.D., Dyer, M., and Linger, R.C., "Cleanroom Software Engineering," *IEEE Software*, vol. 4, no. 5, 19-24, Sep. 1987.

[150] Minneman, S.L., "The Social Construction of a Technical Reality: Empirical Studies of Group Engineering Design Practice," Ph.D. thesis, Stanford University, 1991.

[151] Minsky, M. and Riecken, D., "A Conversation with Marvin Minsky About Agents," *Communications of the ACM*, vol. 37, no. 7, 22-29, Jul. 1994.

[152] Mostow, J., "Toward better models of the design process," *AI Magazine*, vol. 6, no. 1, 44-57, Spring 1985.

[153] Naur, P., "Programming as theory building," *Microprocessing and Microprogramming*, vol. 15, 253-261, 1985.

[154] Neumann, P.G., *Inside Risks*, Regular column in Communications of the ACM.

[155] Nielsen, J., "Usability engineering at a discount," in *Designing and Using Human-Computer Interfaces and Knowledge Based Systems*, 1989, pp. 394-401.

[156] *Usability Inspection Methods*, Nielsen, J. and Mack, R.L. (eds). New: John Wiley & Sons, 1995.

[157] *User centered system design – New perspectives on human computer interaction*, Norman, D.A. and Draper, S.W. (eds). Hillsdale, N.J.: Lawrence Erlbaum Associates, 1986.

[158] von Oech, R., *A Whack On The Side Of The Head*. Warner Books, 1983.

[159] Empirical Studies of Programmers : Second Workshop, Olson, G.M., Sheppard, S., and Soloway, E., Ablex, Norwood, NJ, 1987.

[160] Oz, E., "When Professional Standards are Lax: The CONFIRM Failure and its Lessons," *Communications of the ACM*, vol. 37, no. 10, 29-36, Oct. 1994.

[161] Pahl, G. and Beitz, W., *Engineering Design*. The Design Council, 1984.

[162] Papert, S., *Mindstorms: Children, Computers, and Powerful Ideas*. New York, NY: Basic Books, 1980.

[163] Parnas, D.L. and Clements, P.C., "A rational design process: how and why to fake it," *IEEE Transactions on Software Engineering*, vol. 12, no. 2, 251-257, Feb. 1986.

[164] Peters, L., *Software Design*. New York, NY: Yourdon Press, 1981.

[165] Peters, L., "The 'Chinese Lunch' syndrome in software engineering education: causes and remedies," in *IEEE Computer Society Workshop on Software Engineering Technology Transfer,* IEEE Computer Society, Apr. 1983.

[166] Piaget, J. and Inhelder, B., *The Psychology of the Child.* New York, NY: Basic Books, 1969.

[167] Popper, K.R., *Conjectures and Refutations: The Growth of Scientific Knowledge.* New York, NY: Harper and Row, 1965.

[168] Popper, K.R., *The Logic of Scientific Discovery.* New York, NY: Harper and Row, 1968.

[169] Pree, W., *Design Patterns for Object-Oriented Software Development.* New York, NY: Addison-Wesley, 1995.

[170] Quintas, P., "Software Engineering Policy and Practice: Lessons From the Alvey Program," *J. Systems Software,* vol. 24, 67-88, 1994.

[171] Reddy, M.J., "The Conduit Metaphor - A Case of Frame Conflict in Our Language about Language," in *Metaphor and Thought*, Ortony, A., Ed. Cambridge University Press, 1979, pp. 284-324.

[172] Rettig, M., "Cooperative Software," *Communications of the ACM*, vol. 36, no. 4, 23-28, Apr. 1993.

[173] Rettig, M., "Prototyping for Tiny Fingers," *Communications of the ACM*, vol. 37, no. 4, 21-27, Apr. 1994.

[174] Rittel, H.W.J. and Webber, M.M., "Dilemmas in a general theory of planning," *Policy Sciences*, vol. 4, 155-169, 1973.

[175] Roozenburg, N.F.M. and Cross, N.G., "Models of the design process: integrating across the disciplines," *Design Studies*, vol. 12, no. 4, 215-219, 1991.

[176] Rosson, M.B., Maass, S., and Kellogg, W.A., "The designer as user: building requirements for design tools from design practice," *Communications of the ACM*, vol. 31, no. 11, 1288-1298, Nov. 1988.

[177] Rowe, P.G., *Design Thinking.* Cambridge, MA: MIT Press, 1987.

[178] Royce, W.W., "Managing the development of large software systems," in *Proc. WESTCON*, Calif., U.S.A., 1970.

[179] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorenson, W., *Object-Oriented Modelling and Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1991.

[180] Rybash, J.M., Hoyer, W.J., and Roodin, P.A., *Adult Cognition and Aging*. Elmsford, NY: Pergammon Press, 1986.

[181] Rzevski, G., "On the design of a design methodology," in *Design: Science: Method. Proceedings of the 1980 Design Research Society Conference*, 1980, pp. 6-17.

[182] Schön, D., *The Reflective Practitioner : How Professionals Think in Action*. Basic Books, 1983.

[183] Schön, D., "Designing as a reflective conversation with the materials of a design situation," *Research in Engineering Design*, vol. 3, 131-147, 1992.

[184] Schmitt, G.N. and Chen, C.C., "Classes of design - classes of methods - classes of tools," *Design Studies*, vol. 12, no. 4, 246-251, 1991.

[185] *Participatory Design: Principles and Practices*, Schuler, D. and Namioka, A. (eds). Hillsdale, N. J.: Lawrence Erlbaum Associates, 1993.

[186] Selby, R.W., Basili, V.R., and Baker, F.T., "Cleanroom Software Development: An Empirical Evaluation," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 9.

[187] Shasha, D. and Lazere, C., *Out of Their Minds: The Lives and Discoveries of 15 Great Computer Scientists*. New York , NY: Springer-Verlag, 1995.

[188] Shaw, M., "Toward higher-level abstractions for software systems," *Data and Knowledge Engineering*, vol. 5, 119-128, 1990.

[189] Sherer, S.W., Kouchakdjian, A., and Arnold, P.G., "Experience Using Cleanroom Software Engineering," *IEEE Software*, vol. 13, no. 3, 69-76, 1996, World Wide Web page at http://www.computer.org/pubs/software/abs96.htm.

[190] Shlaer, S. and Mellor, S., *Object Lifecycles: Modeling The World In States*. Prentice Hall, 1991.

[191] Simon, H.A., "The structure of ill-structured problems," *Artificial Intelligence*, vol. 4, 181-200, 1973.

[192] Simon, H.A., *The Sciences of the Artificial*, 2 ed.. Boston, Mass.: The MIT Press, 1981.

[193] Snodgrass, A. and Coyne, R., "Is designing hermeneutical?," Tech. Rep., Sydney, Australia, 1990.

[194] Snodgrass, A. and Coyne, R., "Models, metaphors and the hermeneutics of designing," *Design Issues*, vol. 9, no. 1, 56-74, 1992.

[195] Sommerville, I., *Software Engineering*, 3. Addison Wesley, 1989.

[196] Stacey, W. and MacMillan, J., "Cognitive Bias in Software Engineering," *Communications of the ACM*, vol. 38, no. 6, 57-63, Jun. 1995.

[197] Stalnaker, R., "Presuppositions," *J. Philos. Logic*, vol. 2, 447-457, 1973.

[198] Stefik, M.J., "Understanding Computers and Cognition: A New Foundation for Design - Four Reviews and a Response," *Artificial Intelligence*, vol. 31, 213-261, 1987.

[199] Stefik, M., Foster, G., Bobrow, D.G., Kahn, K., Lanning, S., and Suchman, L., "Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings," in *Computer-Supported Cooperative Work: A Book of Readings*, Greif, I., Ed. Morgan Kaufmann Publishers, Inc., 1988, chap. 13, pp. 335-366.

[200] Stroustrup, B., *The C++ Programming Language*, 2. Reading, MA: Addison-Wesley, 1991.

[201] Stults, R., "Experimental Uses of Video to Support Design Activities," Tech. Rep., SSL-89-19 [P89-00019], Dec. 1988.

[202] Swartout, W. and Balzer, R., "On the Inevitable Intertwining of Specification and Implementation," *Communications of the ACM*, vol. 25, no. 7, 438-440, Jul. 1982.

[203] Thomas, J.C. and Carroll, J.M., "The psychological study of design," *Design Studies*, vol. 1, no. 1, 5-11, 1979.

[204] Building for People, 1965 Conference Report, UK Ministry of Public Building and Works, London, 1965.

[205] Ullman, D.G., "The status of design theory in the United States," *Design Studies*, vol. 12, no. 4, 204-207, 1991.

[206] Wagner, I., "A Web of Fuzzy Problems: Confronting the Ethical Issues," *Communications of the ACM*, vol. 36, no. 4, 94-101, Jun. 1993.

[207] Wallace, L., Private communication.

[208] Walz, D.B., Elam, J.J., Krasner, H., and Curtis, B., "A methodology for studying software design teams: an investigation of conflict behaviors in the requirements definition phase," in *Empirical Studies of Programmers : Second Workshop*, Olson, G.M., Sheppard, S., and Soloway, E., Eds. Ablex, 1987, chap. 6, pp. 83-99.

[209] Warnier, J.D., *Logical Construction of Programs*. New York, NY: Van Nostrand Reinhold, 1977.

[210] Weiser, M., "The Computer for the 21st Century," *Scientific American*, vol. 265, no. 3, 94-104, Sep. 1991.

[211] Wiener, L., *Digital Woes: Why We Should Not Depend on Software*. Addison-Wesley, 1993.

[212] Wilkinson, N.M., *Using CRC Cards: An Informal Approach to Object-Oriented Development*. New York, NY: SIGS Publication, Inc., 1995.

[213] Willem, R.A., "Design and science," *Design Studies*, vol. 11, no. 1, 43-47, 1990.

[214] Willem, R.A., "Varieties of design," *Design Studies*, vol. 12, no. 3, 132-136, 1991.

[215] Winograd, T. and Flores, F., *Understanding Computers and Cognition.* Ablex, 1986.

[216] Winograd, T., "From Programming Environments to Environments for Designing," *Communications of the ACM*, vol. 38, no. 6, 65-74, Jun. 1995.

[217] Winograd, T., *Bringing Design to Software*. Addison-Wesley, 1996.

[218] Wirfs-Brock, R. and Wilkerson, B., "Object-Oriented Design: A Responsibility-Driven Approach," in *Proceedings of Object-Oriented Programming Systems, Languages, and Applications*, 1989, pp. 71-75.

[219] Wirfs-Brock, R., Wilkerson, B., and Wiener, L., *Designing Object-Oriented Software*. Prentice Hall, 1990.

[220] Wirth, N., “Program Development by Stepwise Refinement,” *Communications of the ACM*, vol. 14, no. 4.

[221] Wittgenstein, L., *Philosophical Investigations.* New York, NY: MacMillan, 1958.

[222] Yau, S. and Tsai, J., “A Survey of Software Design Techniques,” *IEEE Transactions on Software Engineering*, vol. SE-12, no. 6.

[223] Yourdon, E. and Constantine, L.L., *Structured Design*. Prentice-Hall, 1979.

[224] Yourdon, E., “Object-Oriented Observations,” *American Programmer*, vol. 2, no. 7-8, 3-7, Summer 1989.

[225] Yourdon, E., “When Good Enough Is Best,” *Byte Magazine*, vol. 21, no. 9, 85-90, Sep. 1996.

[226] Zave, P., “The Operational versus the Conventional Approach to Software Development,” *Communications of the ACM*, vol. 27, no. 2, 104-118, Feb. 1984.

[227] Zeng, Y. and Cheng, G.D., “On the logic of design,” *Design Studies*, vol. 12, no. 3, 137-141, 1991.

[228] *comp.risks USENET Newsgroup*.