FUNDAMENTALS OF SOFTWARE DESIGN SCIENCE

by

David Paul Ralph

B. Comm., Memorial University of Newfoundland, 2005
B. Sc., Memorial University of Newfoundland, 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES

(Business Administration)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

October 2010

ABSTRACT

This dissertation comprises three essays on *software design science*, the philosophical, theoretical and empirical study of software creation and modification including its phenomenology, methodology and causality. The essays consider three limitations evident in the software design science literature: 1) lack of a precisely-defined, well-understood vocabulary; 2) difficulties surrounding empirical research; 3) lack of theory concerning the design process's structure and organization.

The first essay presents an extensive review of definitions of design, revealing nontrivial disagreements regarding its nature and scope. Following this, a formal definition of design and a conceptual model of design projects are constructed. The definition incorporates seven elements – agent, object, environment, goals, primitives, requirements and constraints. The conceptual model views design projects as temporal trajectories of work systems, in which human agents design systems for stakeholders, using resources and tools. This provides a detailed, defensible basis for theoretical and empirical software design science research.

The second essay addresses the difficulties of empirical software design science research by elucidating a broad, bipolar conflict in design literature between two incompatible beliefs: 1) the belief that design is an innately cognitive, approximately rational, plan-centered activity (Reason-Centric Perspective); 2) the belief that design is an emergent phenomenon, improvised through continual interaction between agents and environments (Action-Centric Perspective). Each perspective is operationalized through a software design process theory: the Function-Behavior-Structure Framework (chosen for the Reason-Centric Perspective) and the Sensemaking-Coevolution-Implementation Framework (proposed for the Action-Centric Perspective).

The third essay presents a survey study comparing these perspectives and theories. Responses from 1384 software development professionals in 65 countries indicate that the Sensemaking-Coevolution-Implementation Framework more accurately describes the structure and organization of their design processes than the Function-Behavior-Structure Framework. Gender, education, experience, nationality, occupation, team size, project duration, firm size, methodologies in use, and the nature of the software had no measurable effect on this finding. This supports a theory of the design process's structure and organization and facilitates several streams of empirical research including studies of design project success.

PREFACE

The first of the three research essays comprising this dissertation was joint work with my supervisor, Professor Yair Wand. The project described was jointly identified by the authors and the research method was designed collaboratively. I performed the data collection and analysis and wrote the text. Professor Wand was heavily involved in the paper's editing and revision. The results were presented at two conferences, once by Professor Wand and once by me. The second and third essays were single-author endeavors.

A previous version of the first essay was published in Lyytinen, K., Loucopoulos, P., Mylopoulos, J., and Robinson, W., editors, *Design Requirements Workshop* (LNBIP 14), 2009, pp. 103-136. Springer-Verlag. The second essay is under review at a top-tier MIS journal. A previous version of the third essay was published in the R. Winter, J. L. Zhao and S. Aier (Eds.): *Global Perspectives on Design Science Research* (LNCS 6105), pp. 61-76. Springer-Verlag.

This research was approved by the University of British Columbia Behavioral Research Ethics Board (certificate number H08-00416).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

This work was done with partial support from the Natural Sciences and Engineering Research Council of Canada and from the University of British Columbia Humanities and Social Science Research Fund. I am grateful to all those who provided useful feedback on my research and writing (especially J.P. and J.R.). In addition, special thanks to all those who helped keep me sane and focused throughout the process (especially D.B., B. C. and O.V.).

For the generations to follow, that they may avoid the insanity that has plagued their ancestors.

CHAPTER 1: AN INTRODUCTION TO SOFTWARE DESIGN SCIENCE

1.1 The Scope of Software Design Science

Eekels (2000, 2001) discusses the stratification of engineering design science across five different levels of abstraction, juxtaposed to an analogous software design science stratification in Figure 1-1. Level 5, the lowest level, refers to the design phenomena itself – people working on design projects. Level 4, "methodics" [1], refers to the body of design methods and systems of methods used by designers. Level 3, design science "proper" refers to the study of design practice and design methodics, including ethnographic research on designers and empirical evaluations of methods. Level 2 refers to the study of design science proper and design literature, including meta-analyses and studying the usefulness of various research methods or statistical techniques for use in design science research. Level 1 refers to the philosophical study of knowledge and science in general.

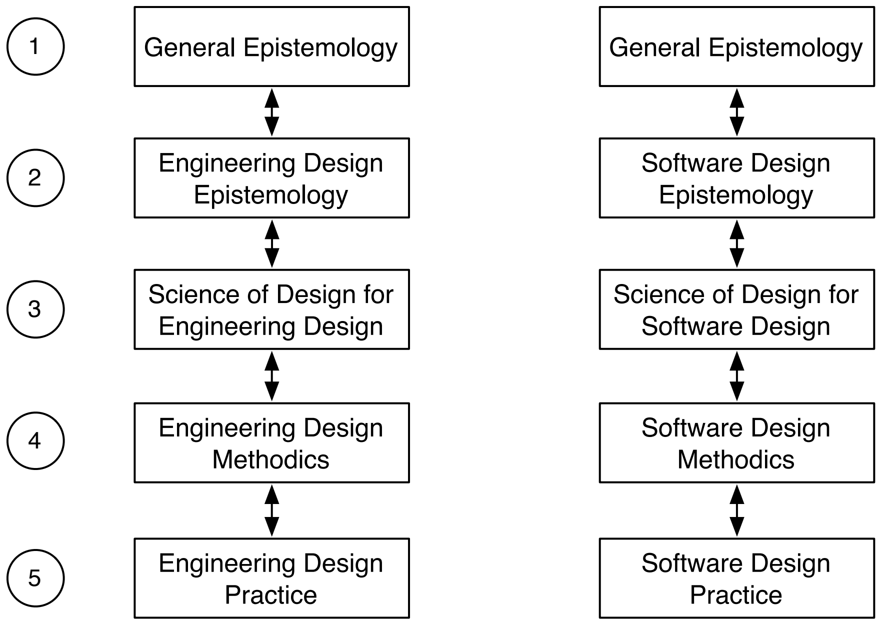

1
General Epistemology
General Epistemology
2
Engineering Design Epistemology
Software Design Epistemology
3
Science of Design for Engineering Design
Science of Design for Software Design
4
Engineering Design Methodics
Software Design Methodics
5
Engineering Design Practice
Software Design Practice

Fig. 1-1. The Stratification of Design Science (adapted from Eekels 2000)

Although software design deals with virtual objects while engineering design deals with physical objects, Eekels' stratification of engineering design science appears equally sensible for software design science. Based on this, I define software design science as follows.

[1] In this dissertation, a quotation without a page number indicates either that a term is used repeatedly by the cited source, or that source lacked page numbers, as in some electronic documents.

***Software Design Science** is the philosophical, theoretical and empirical study of the creation and modification of software artifacts, including performance (phenomenology), methods, tools and practices (methodology), and antecedents and outcomes (causality).*

When Simon (1996) calls on scientists to develop a “theory of design” and proposes a “Science of Design” curriculum, he uses “Science of Design” the same way Eekels used “Design Science” – to indicate the scientific study of the phenomenon called design. Simon further enumerates the following seven pivotal topics for design science.

1. Design alternative evaluation theories
2. Computational methods for choosing design alternatives
3. Formal imperative and declarative design logics
4. Heuristic search (for design alternatives)
5. Search resource allocation
6. Theories of design process structure and organization
7. Representation of design problems (modeling methods)

Interestingly, many of Simon’s topics embed strong assumptions regarding design process structure and organization. For example, the first and second items imply that the design process centers on discrete choice from identifiable alternatives rather than creative generation of a single solution from an infinite set of possibilities. Similarly, the need for formal design logics imply the primacy of logic over emotion in designing, and the seventh topic implies that designers create and use representations. Given these assumptions, the need for theories of the design process’ structure and organization seems paramount.

The reader may note that Simon’s design science topics are split between the third (1, 6) and fourth (2, 3, 4, 5, 7) levels of Eekels’ stratification. Meanwhile, most design literature is normative and inhabits either the second or fourth levels of Eekels’ stratification; i.e., most design literature consists either of frameworks for understanding existing literature or specific design methods, tools and practices (Wynekoop and Russo 1997). Little research has addressed the structure and organization of the design process, or how, exactly, to synthesize and evaluate alternatives (Freeman and Hart 2004; Simon 1996; Sullivan 2003; Wynekoop and Russo 1995).

1.2 Two Factions in Design Science

After conducting a literature review of topics including design, design research, design science, design science research, software design, systems design, information systems design, software-intensive systems design, engineering design, engineering design science, science of design, systems development methodologies, software development methodologies, software design methods, method engineering, systems development approaches, design paradigms, design tools, design patterns, design techniques, project management frameworks, agile development, plan-driven development and amethodical development, the first apparent adjective describing the design literature is *confused*. This confusion manifests itself in two specific, yet rarely articulated, disagreements.

The first disagreement concerns the meaning of “design science.” In the information systems discipline, “design science research” has been used to differentiate (prescriptive) research, which produces technological artifacts, from behavioral research, which produces theoretical knowledge (cf., Hevner et al. 2004; March and Smith 1995; Walls et al. 1992). Design science, then, denotes a research paradigm, like experimentalism. Contrastingly, in Eekels and Simon’s parlance, design science is a field of inquiry like psychology, wherein design is the topic of study. Recognizing this, some authors contend that design science research refers to both “design as research” and “researching design” (Hevner and Chatterjee 2010; Purao et al. 2008). Furthermore, some propose that these two research types overlap and reinforce each other (Hevner et al. 2004; Purao et al. 2008).

This situation is problematic because overloading a term with two closely related meanings may cause confusion. Anecdotally speaking, the phrase “this is not design science” on reviews of “researching design” papers has convinced me that such confusion is real. Of course, this confusion can be avoided by giving the two meanings of design science research separate names. The term “design science” should be reserved for “researching design” for three reasons. First, the use of design science to denote researching design predates its use as a research approach (cf. Cross 2001; Simon 1969). For example, Simon (1969) defines the science of design as “a body of intellectually tough, analytic, partly formalizable, partly empirical, teachable doctrine about the design process” (p. 113). Second, the English construction “X science” conventionally denotes the study of X, as in rocket science, neuroscience, political science, marketing science, environmental science, cognitive science, management science, library science, space science, earth science, life science, plant science. “Environmental science,” for example, denotes the study of the environment and environmental problems. To use the same term to indicate using the environment to produce knowledge would seem strange. By analogy, using design science to describe “design as research” may seem counterintuitive to people outside of the information systems community. Third, “design as research” already has a ubiquitous label: “engineering research.”

This distinction, however, requires two qualifications. First, it is in no way intended to denigrate “design as research.” The merits and drawbacks of this approach are unrelated to the naming issue presently addressed. Second, it does not imply a sharp distinction between the two. A “researching design” project may involve building artifacts and a “design as research” project may produce knowledge about design. Yet, this potential overlap does not justify the two concepts sharing a label any more than possible overlaps between quantum physics and mechanical engineering (e.g., building the Large Hadron Collider) justify those fields sharing a label.

The second disagreement concerns the nature of the activity commonly called “design.” On one side is the view that professionals design by optimizing or “satisficing” a design candidate vis-à-vis known constraints and objectives (Simon 1996). Design, then, is a form of plan-driven problem-solving, where an agent seeks a goal state by executing a plan in a field of constraints (Newell and Simon 1972). It is essentially *cognitive* – an exercise in logic. On the other side is the view that the designer “reflects in action … he does not keep means and ends separate … he does not separate thinking from doing” (Schön 1983, p. 69). In other words, understanding the problem and finding a solution are one and the same process. The designers act in a situated way; that is, rather than executing a plan, their actions' organization "is an emergent property of moment-by-moment interactions between actors, and between actors and the environments of their action" (Suchman 1987, p. 179). Design, then, is essentially *emergent* – comprising continuous interactions between designers and their environment (in which the details of the problem are housed).

This disagreement manifests in reactions to the Waterfall Model (Royce 1970). The waterfall model is a description of software development in terms of a series of discreet stages. While the named stages vary, "planning", "analysis", "design", and "implementation" are common. The Waterfall Model is sometimes referred to as the Systems Development Lifecycle (SDLC), although this is a contentious issue (see Appendix A). Though rarely stated, implicit claims that the waterfall model describes design practice pervade research, teaching and practice. For example, in a well-cited paper in MIS Quarterly, Fitzgerald (2006) states that "in conventional software development, the development lifecycle in its most generic form comprises four broad phases: planning, analysis, design, and implementation" (p. 3) and then describes the presence of these phases in open-source software development. In a popular introductory MIS textbook, Laudon et al. (2009) state that "systems development … consist[s] of systems analysis, systems design, programming, testing, conversion and production and maintenance … which usually take place in sequential order." Similarly, at the time of writing, the SDLC Wikipedia article states that "SDLC adheres to important phases that are essential for developers, such as planning, analysis, design, and implementation." Moreover, similar Waterfall Model phases are explicitly adopted in the official IEEE Guide to the Software Engineering Body of Knowledge (Bourque et al. 2004). These examples illustrate the how influential the Waterfall Model remains and how it is consistent with the 'design as a cognitive activity' view stated above – if "analysis" and "design" are discrete phases, then the design phase is about solving the problem elucidated in the analysis phase.

In contrast, numerous authors explicitly attack or implicitly undermine the waterfall model and its underlying separations of planning from doing and problem understanding from problem solving. For example, Schön found evidence indicating that designers do “not keep means and ends separate” or “separate thinking from doing” (1983, p. 69). Meanwhile, the author generally credited with proposing the Waterfall Model affirmed that its simplest version “has never worked on large software development efforts” (Royce 1970, p. 335). Additionally, in a study of “a large scale system development effort”, Zheng et al. (2007) found that “home-gown methods and ad hoc activities appear to dominate the day-to-day practices of systems development” (p. 1). In several case studies, Baskerville et al. (2004; 1992) found evidence of software developers acting in a fundamentally *amethodical* manner. “Amethodical systems building implies management and orchestration of systems development without a predefined sequence, control, rationality, or claims to universality. An amethodical development activity is so unique and unpredictable for each information systems requirement that even the criteria of contingent development methods are irrelevant” (Truex et al. 2000, p. 54). Design without “a predefined sequence, control, rationality” etc. is consistent with the ‘design as an emergent activity’ view.

The purpose of this dissertation is to explore the tension between these two views of design. Doing so leads to three specific problems, each of which I attempt to address in subsequent chapters.

1.3 The Three Problems in Software Design Science

In the preceding section I identified a widespread disagreement about the nature of design practice. Whether design is predominantly cognitive or emergent in practice is an empirical question – the primary question addressed by this dissertation. However, studying this raises two prerequisite questions.

First, how to operationalize this question in a testable form is not obvious. Empirical design science research, in general, is fraught with serious methodological difficulties. As it is entirely unclear how to empirically evaluate the descriptive validity or effectiveness of many design concepts, researchers avoid such studies, as evidenced by the dearth of empirical research on software design methods identified by Wynekoop et al. (1997). Consequently, the design research culture appears to have become tolerant of research that downplays empirical evaluation.

Consider, for example, how to gather evidence of the effectiveness of a software design method. The research question is, *what is the effect of method X on project success?* Three methodological options are evident – field study, survey or experiment. Let us assume that team, project and method variables all influence project success. Even ignoring generalizability issues, an ethnographic field study of a single project cannot answer this question as it cannot determine if the project would have been more or less successful without method *X*. Furthermore, a field experiment cannot work as the same team cannot complete a single project with two different methods (or once with a method, once without). Changing either team or project invalidates the conclusions as both variables may influence success as much or more than the method. Moreover, survey results would be confounded by a host of factors including the following.

1. It would be impossible to assess exactly how or to what extent real teams adapted or misused the methods in question.
2. Reasons for choosing a method may be related to the chance of project success (e.g., developers may choose more effective methods for harder project, hiding their effects).
3. The methods in question may be unpopular in industry.

Similarly, a realistic experiment would be confounded by team and context. Assuming near limitless resources, we may recruit hundreds of development professionals and randomly assign them to teams (we must have teams as most commercial development is team-based). Suppose that teams are divided into treatment group (Method A) and control group (Method B, or no method). All teams must work on the same problem. However, simply providing a problem description would invalidate the results because understanding the problem is part of the methods (see ch. 3). Therefore, teams must determine goals and requirements through stakeholder interaction. Obviously, all teams must interact with the same stakeholders, or the "different problems" confound reappears. However, stakeholders' problem conceptualizations will be polluted by each successive question-answer session, such that treatment-group teams benefit from the investigations of control-group teams and vice versa. This pollution via stakeholders confounds the difference between the methods in terms of requirements elicitation effectiveness. I cannot conceive of an experimental design that avoids these confounds. Therefore, regardless of the research methodology, scientifically testing the effectiveness of a design method is difficult, if not impossible, at the epistemological standards generally employed in behavioral research.

Returning to the specific case of whether design is cognitive or emergent, this is not a causal hypothesis that can be subjected to experimental evaluation. ‘Design is a cognitive(/emergent) activity’ is an abstract ontological statement. It is a high-level proposition about real world processes. As such, operationalizing it in an empirically testable form is a significant challenge.

Second, the underlying phenomenon of interest, design, is not consistently defined (ch. 2). Without a clear articulation of the domain of inquiry, it would be difficult to say precisely what is meant by “design is cognitive” or “design is emergent.” Lack of a clear definition of design would also hamper the operationalization of these concepts.

1.4 Summary of Contribution

Of the seven pivotal topics in design science elucidated by Simon (1996), perhaps none has received as little attention as *theories of design process structure and organization*. Hence, the purpose of this dissertation is to study empirically the process and organization of the design process. This involves three challenges:

1. Articulating exactly what is meant by “design” and specifying the domain of design phenomena
2. Fleshing out conflicting views of design practice and operationalizing them in a testable form
3. Empirically testing conflicting views of design practice and providing a theory of design process structure and organization consistent with the more accurate view.

The objective of this dissertation is to address each of these challenges. Chapter Two addresses the lack of common language describing design phenomena by formally defining design and situating it in a precise conceptual model of software design projects. This provides a working nomenclature for the subsequent studies and design science more generally. Chapter Three attenuates the difficulties of empirical work by: 1) organizing design literature through underlying conflict concerning whether design is a cognitive or emergent phenomenon; 2) operationalizing these two views using two contrasting, testable software design process theories. This facilitates not only empirically testing these particular theories but also evaluating other design related concepts with these process theories. Chapter Four reports a survey study testing the software design process theories. The results support the proposition that design is an emergent phenomenon, and the explanatory validity of a theory of the design process' structure and organization: The Sensemaking-Coevolution-Implementation Framework. Following these contributions, Chapter Five proposes a roadmap for the next decade of software design science research.

CHAPTER 2: A PROPOSAL FOR A FORMAL DEFINITION OF THE DESIGN CONCEPT[1]

[1] A previous version of this paper appeared in the book *Design Requirements Engineering: A Ten-Year Perspective*, Lecture Notes in Business Information Processing, Volume 14. Springer Berlin Heidelberg, 2009, p. 103-121

2.1 Introduction

There have been several calls for addressing design as an object of research. Freeman and Hart call for a comprehensive, systematic research effort in the science of design: “We need an intellectually rigorous, formalized, and teachable body of knowledge about the principles underlying software-intensive systems and the processes used to create them” (Freeman and Hart 2004, p. 20). Simon (1996) calls for development of a “theory of design” and gives some suggestions as to its contents. Yet, surprisingly, it seems no generally-accepted and precise definition of design as a concept is available.

A clear understanding of what design means is important from three perspectives. From an instructional perspective, it seems obvious that any designer’s education ought to include providing a clear notion of what design is. Furthermore, better understanding the meaning of design will inform what knowledge such education could include.

From a research perspective, in any theoretical or empirical work in which design is a construct, a clear definition will help ensure construct validity. Furthermore, a clear understanding of the meaning of design will facilitate developing measures of design-related constructs, such as design project success. Moreover, building a cumulative tradition of design research can benefit from a well-accepted definition of design, the alternative being different theories defining design differently, or not at all.

From a (software design) practitioner’s perspective, a clear definition of design can help organize, share and reuse design knowledge. Such sharing can enhance software project success and software development productivity. Furthermore, understanding the elements of design would be useful in determining the issues and information relevant to the process of design and in planning this process.

Given the potential value of a clear definition of design, our objective here is to suggest such a definition. We first seek to answer the question: what are the important elements of design as a phenomenon? We then seek to situate design in a network of related concepts.

The paper is organized as follows. First, we synthesize a definition of design by applying concepts and suggestions in existing literature (§2.2). We then evaluate the proposed definition in Section 2.3. Section 2.4 situates our view of design in a conceptual model of software design projects. In Section 2.5, we demonstrate how the proposed definition and conceptual model of design can be applied to indexing design knowledge for reuse and classifying design approaches, respectively. Finally, we discuss the implications of our definition of design for current themes in software design and requirements research (§2.6).

2.2 PROPOSING A FORMAL DEFINITION OF DESIGN

2.2.1 Design in the Literature

We conducted a review of existing definitions of "design" in the literature. This review combined snowball and theoretical sampling. I.e., we began by examining definitions of design in notable works on the subject drawn from our own experiences. Where authors referenced other works concerning aspects of design, we investigated these cited works and so on. To broaden the sample, we also conducted a library-shelf study, i.e., identifying physical shelves in an academic library likely to contain books from different disciplines that define design, and checking each book. Again, we investigated cited works. We concluded the review when subsequent definitions failed to uncover any new themes or concepts.

A list of definitions we examined is provided in Appendix B. We analyzed the definitions in three ways. First, we identified the concepts that were common to several definitions (Table 2-1). We then analyzed each definition and found that each had serious errors (discussed below). We found that all definitions included at least one type of error. The detailed analysis is provided also in Appendix B. Finally, we identified four main areas of disagreement among the definitions (discussed below).

Some of these areas of agreement appear problematic. First, some definitions include desirability criteria in their definitions, as evidenced by words like "optimally" (Accred. Board, 1988) and "optimizing" (van Engers et al. 2001). Just as one does not have to run in the practiced form of a professional marathoner for their action to be called "running," one does not have to design optimally for their action to be called "designing." Second, organizing does not necessarily constitute design, for example, when someone returns books to their proper shelves in a library, one is organizing the books into a pre-designed arrangement rather than actively performing a design task. Third, four definitions state or imply that design is strictly a human phenomenon. However, machines can also design objects (e.g. the design of processors using genetic algorithms (Bradel and Stewart 2004). Some research indicates that animals can also design objects (Breuer et al. 2005; Mulcahy and Call 2006). Fourth, while many designers are surely creative, not all design need involve creativity. For example, design might involve relatively minor modifications to a previously created design.

Table 2-1. Frequency of Common Concepts in Analyzed Definitions

Concept	**Frequency**
Design as a *process*	11
Design as *creation*	11
Design as *planning*	7
Design as a *physical* activity (or as including implementation)	7
System (as the object of the design)	7
Design as being *deliberate*, or having a *purpose, goal* or *objective*	7
Design as an *activity*, or a collection of activities	7
Design as occurring in an *environment* (or domain/situation/context)	7
Artifact, as the object of the design	5
Needs or *requirements*	5
Design as a *human* phenomenon	5
Design as *organizing*	4
Parts, components or elements	4
Constraints or *limitations*	3
Process (as the object of design)	2
Design as *creative*	2
Optimizing	2
Design as a *mental* activity	2
Resources	2

Finally, we identified four areas of disagreement. First, different objects of design arise: *system*, *artifact* and *process*. Second, disagreement exists concerning the scope of design: where or when a design begins and ends. Third, some definitions indicate that design is a physical activity, others a mental activity. Fourth, some disagreement concerns the outcome of design: is it a plan, an artifact, or a solution?

2.2.2 Suggesting a Definition of Design

In this section, we develop our proposed definition of design. First, Eekels (2000) differentiates between the subjects and objects of design. The subject of the design is the (often human) ***agent*** that manifests the design. The ***design object*** is the thing being designed. Design outcomes such as an artifact, a system or a process that appear in some existing definitions are encompassed here by the more general term, design object.[2]

Some definitions mention parts, components or elements of which the design object is, or is to be, composed. Obviously, all artificial physical things are made from other things. We term the lowest level of components ***primitives***. Similarly, but perhaps less obviously, if we assume that atomic conceptual things, such as single thoughts or ideas, are not designed (but are discovered or just *are available*), then all conceptual things that are designed are made from other conceptual things. Therefore, all design involves primitives, which are, or can be, assembled or transformed to create a design object[3]. March and Smith note that "[t]echnology includes...materials, and sources of power" (1995, p. 252). Materials and sources of power would be included in the set of primitives.

The outcome of a design effort is not necessarily the design object itself, but may be a plan for its construction, consistent with the definitions that characterize design as planning rather than building. The common factor here is that the agent specifies properties of the design object: sometimes as a symbolic representation, as in an architectural blueprint, sometimes as a mental representation, as in a picture in a

[2] Note: often the object is called an artifact, when designed by humans. The more general term object allows (in principle) for non-human agents such as animals and computers.

[3] What the set of available primitives is can be a relative issue. A designer might be given a set of components, or component types, where each might be in turn composed from lower level components. We consider primitives the set of component-types available to the designer, independent of whether they are natural, or the outcome of previous design. Furthermore, even if the components are not yet available, a designer might proceed assuming they will be available. The assumptions made about these components will become requirements for their design.

painter's mind, and sometimes as the artifact itself, as in a hand-carved boomerang. We call the specified properties of the design object a ***specification***. More specifically, *a specification is a detailed description of a design object's structural properties*, namely, which primitives are assembled or modified and, if more than one primitive is used, how they are linked. The specification may be purely mental, provided in a symbolic representation, presented as a physical model, or even manifested as the object itself. This notion of specification agrees with the idea that design is the activity that produces "a description of the software's internal structure" (Bourque and Dupuis 2004, p. 1-3).

Practically speaking, a specifications document might include desired behaviors as well as structural properties. From the perspective of this paper, these desired behaviors are requirements – they are not strictly part of the specifications. The object's behavior *emerges* from the behavior of the individual components and their interactions. (By behavior we mean the way the object responds to a given set of stimuli from its environment, including agents who interact with the object.)

Churchman (1971) points out that "Design belongs to the category of behavior called teleological, i.e., "goal seeking" behavior" (p. 5). Many of the definitions we surveyed also included concepts such as "goal," "purpose" or "objective." While the goal may not be explicit or well defined, design is always intentional, never accidental. For example, a social networking web application can be designed without having an explicit goal, based on the vague idea that it would be useful and fun to have an online space where people could connect. We would still say the web application was designed. On the other hand, accidental or unintentional discoveries are not designed. Thus, goals are inherent to design insofar as a designer must have intentionality. However, this should not be interpreted as a requirement that a design goal is or can be formally or explicitly articulated.

Many definitions characterize the design process as occurring within an environment, domain, situation or context. Design involves *two different environments*: the environment of the design *object*, and the environment of the design *agent*. As pointed out by Alexander, "every design problem begins with an effort to achieve fitness between two entities: the form in question and its context" (Alexander 1964, p.

15). Clearly, the design process or activity also occurs within some ***environment***, even if that environment is difficult to characterize. March and Smith mention the “organizational setting” (March and Smith 1995, p. 252) and Hevner et al. refer to “organizational context” (Hevner et al. 2004, p. 77). For instance, the software created by a developer is intended to operate in a different environment than the developer operates in. The qualifier “organizational” is not always valid for the environment of the artifact because the environment does not have to be an organization (e.g. the environment of a pacemaker is a human body).

Many definitions also mention needs or requirements and limitations or constraints. The issue of requirements needs clarification. If we interpret requirements strictly as a *formal* requirements document or as a set of mathematically expressible functions (as in Gero 1990) the system is to perform, then requirements are not absolutely necessary. The primitive hunter who fashions a spear from a branch specified the spear’s properties by creating it – without an explicit reference to formal requirements (let alone mathematically definable functions). However, in the sense that every designer expects or desires the design object to possess certain properties or exhibit certain behaviors, ***requirements*** are inherent to design. Requirements are a major construct in requirements engineering and software design (Kruchten 2003; Siddiqi and Shekaran 1996).

Similarly, all design must involve ***constraints***. Even if the design agent had infinite time and resources, physical design is still constrained by the laws of physics, virtual design by the speed and memory of the computational environment, and conceptual design by the mental faculties of the design agent. Constraints are a major construct in engineering design (Pahl and Beitz 1996; Simon 1969). However, like requirements, constraints may not be explicit.

The above analysis leads to the following suggestion for the definition of design (modeled in Figure 2-1). Table 2-2 further describes each concept in the definition.

Design

(noun) a specification of an object, manifested by an agent, intended to accomplish goals, in a particular environment, using a set of primitive components, satisfying a set of requirements, subject to constraints;

(verb, transitive) to create a design, in an environment (where the designer operates)

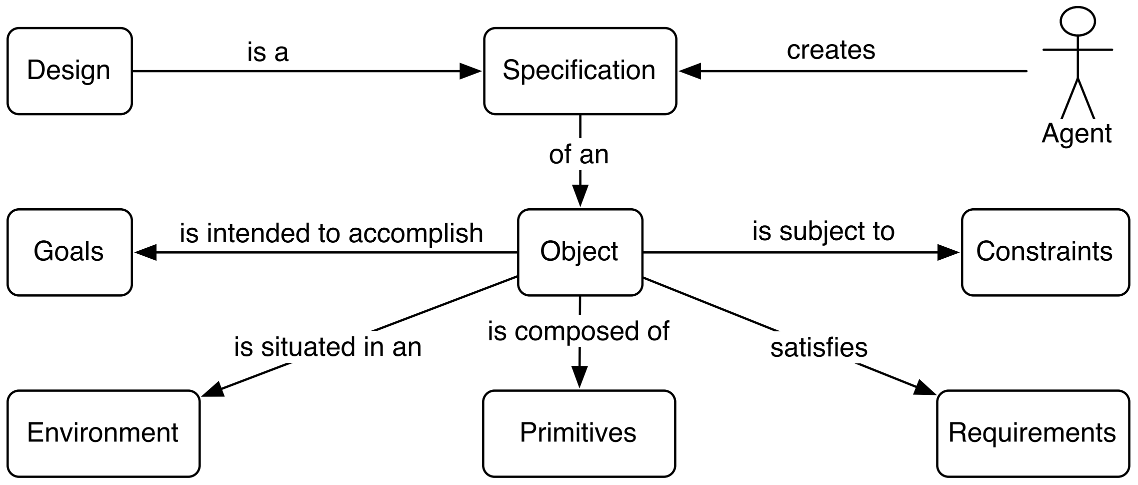


Fig. 2-1. Conceptual Model of Design (Noun)

Table 2-2. Definitions of Design Concepts

Concept	**Meaning**
Design Specification	A specification is a detailed description of an object in terms of its structure, e.g., the primitives used and their connections.
Design Object	The design object is the entity (or class of entities) being designed. Note: this entity is not necessarily a physical object.
Design Agent	The design agent is the entity or group of entities that specifies the structural properties of the design object.
Environment	The *object* environment is the context or scenario in which the object is intended to exist or operate (used for the noun form). The *agent* environment is the context or scenario in which the design agent creates the design (used for the verb form).
Goals	Goals describe the desired impacts of design object on its environment. Goals are optative (i.e. indicating a wish) statements that may exist at varying levels of abstraction (van Lamsweerde 2001).
Primitives	Primitives are the set of elements from which the design object may be composed (usually defined in terms of *types* of components assumed to be available).
Requirements	A requirement is a structural or behavioral property that a design object must possess. A structural property is a quality the object must posses regardless of environmental conditions or stimuli. A behavioral requirement is a required response to a *given* set of environmental conditions or stimuli. This response defines the changes that might happen in the object or the impact of these changes on its environment.
Constraints	A constraint is a structural or behavioral restriction on the design object, where “structural” and “behavioral” have the same meaning as for requirements.

Considering design as a process (depicted in Figure 2-2), the outcome is the specification of the design object. The goals, environment, primitives, requirements and constraints are, in principle, the inputs to the design process; however, often knowledge of these may emerge or change during the process. Nevertheless, the design process must begin with some notion of the object's intended environment, the type of object to design and intentionality – by this we simply mean that design cannot be accidental. Finally, if the type of design object changes significantly (e.g. from a software system to a policy manual), the existing design effort is no longer meaningful and a new design effort begins. The possibility of changing information is related to the possibility that the design process involves exploration. It also implies that the design may evolve as more information is acquired.

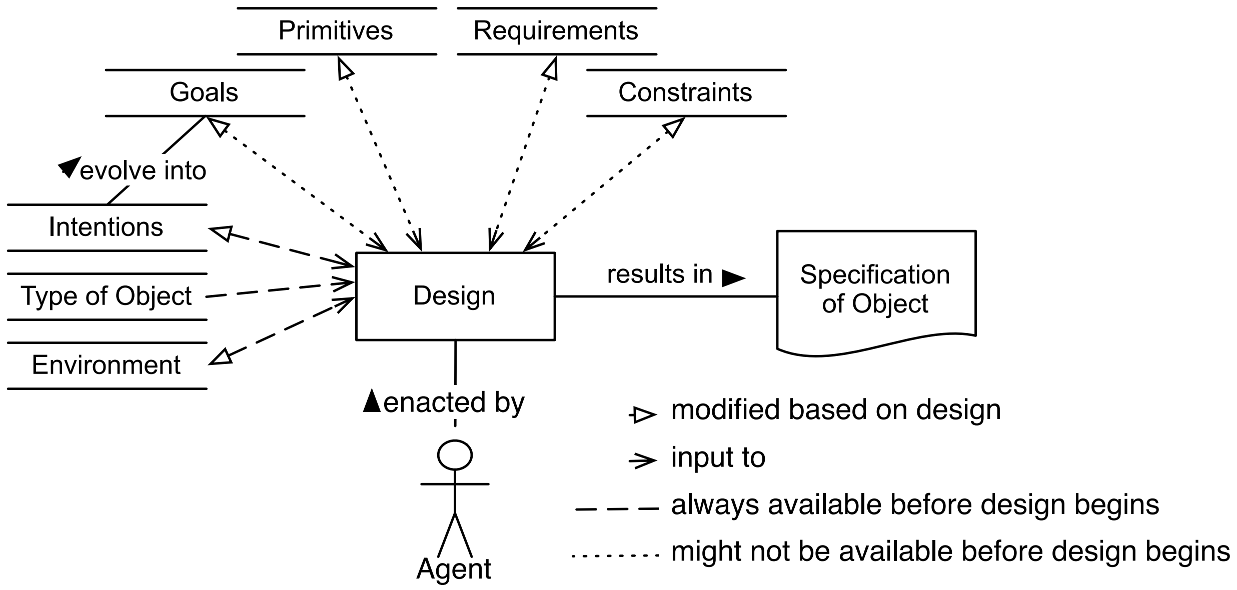


Fig. 2-2. Context-level Conceptual Model of Design (Verb)

2.2.3 What Can Be Designed and Examples of Design Elements

"What can be designed?" is a difficult ontological question, one we are not sure we can answer completely. However, we can imagine at least six, possibly overlapping, classes of design objects:

- **physical artifacts**, both simple, such as boomerangs (single-component), and composite, such as houses (made of many *types* of components)
- **processes**, such as business workflows

- **symbolic systems**, such as programming and natural languages
- **symbolic scripts**, i.e., documents written in symbolic systems, such as essays and software
- **laws, rules and policies**, such as a criminal code
- **human activity systems**, such as software design projects, committees and operas

Clearly, the nature of a specification depends on the class of design object since the structure and components of, for example, a law would be very different from those of a rocking chair. For simple artifacts, such as a one-piece racket, the specification would include structural properties such as shape, size, weight and material. For a composite physical artifact, such as a desk, the specification would include the primitive components and how they are connected. Since a process is 'a set of partially ordered activities aimed at reaching a goal' (Hammer and Champy 1993), a specification of a process may identify the activities and their order, although other approaches are possible – e.g. using Petri Nets or states and events (Alast 2000; Soffer and Wand 2004). For a symbolic system, the specification may include syntax, denotational semantics and (for a spoken language) pragmatics. A symbolic script can be specified by symbols and their arrangement. A policy or law can be specified in some (possibly formal) language. The specification of a human activity system might include agents, roles, tasks, artifacts, etc. and their relationships.

Furthermore, all elements from the definition of design may vary across object types. Table 2-3 provides examples of each design element for each class of design object.

2.2.4 Scope of Design

According to the perspective on design expressed in this paper, design (as a verb) is the act of specifying the structural properties of an object, either in a plan or in the object itself. Because design is an activity, rather than a phase of some process, it may not have a discernible endpoint. Rather, it begins when the design agent begins specifying the properties of the object, and stops when the agent stops. Design may begin again if an agent (perhaps a user) changes structural properties of the specification or design object at a later time. This defines the scope of the design activity.

Table 2-3. Examples of Design Elements

Object Type	Process	Symbolic system	Law/policy	Human activity system	Physical artifact	Symbolic Script
Object	loan approval	a special purpose programming language	criminal code	a university course	office building	a software system
Agent	loan officer	team that creates the language	legal experts and lawmakers	instructor	architect	programmer
Goals	accurately estimate risk level of loan	provide a means of expressing software instructions	provide a legal framework for dealing with crimes	facilitate learning and development of students in a given area	provide office space for a business	support management of customer information
Object Environment	bank administrative system	computing environment on which code will execute	national legal and constitutional system	university (with all resources available)	business district of a given city	personal computers and specific operating systems
Requirements	provide a decision with justification; generate audit trail for decision process	be easily readable, minimize coder effort, fit certain applications	define crimes and punishments clearly; be unambiguous	learning objectives	include open floor plan offices, be energy efficient	maintain customer information, identify customers with certain characteristics
Primitives	various actions that need to be taken, e.g., assessing the value of a collateral	the C programming language instructions	English words as used in legal documents	various common teaching actions (presentations, laboratory sessions, tests)	building materials, interior decoration materials	the instructions in the symbolic system (programming language)
Constraints	bank approval rules and risk policies (e.g. debt-service ratio allowed)	cannot violate some programming languages related standards	must not violate the country's constitution and international laws	prior knowledge students have, number of class and laboratory hours available	comply with building code, cost less than a given budget	must be able to run on a given hardware configuration with a maximum given delay

Our definition does not specify the process by which design occurs. Thus, how one interprets this scope of activities in the design process depends on the situation. If a designer encounters a problem and immediately begins forming ideas about a design object to solve the problem, design has begun with problem identification. If requirements are gathered in reaction to the design activity, design includes requirements gathering. In contrast, if a designer is given a full set of requirements upfront, or gathers requirements before conceptualizing a design object, requirements gathering is not part of design. Similarly, if the construction agent refines the specification (a possible occurrence in software development), construction is part of design, but if the designer creates a complete specification on paper that the construction agent follows deterministically, construction is not part of design. Any activity, including testing and maintenance, that involves modifying, or occurs within an effort to modify, the specification is part of design. Therefore, design practice may not map cleanly or reliably into the phases of a particular process, such as the waterfall model (Royce 1970) or Systems Development Lifecycle (Bourque and Dupuis 2004).

This distinction has particular bearing for software design, where a significant debate over the scope of design exists. On the narrow-scope side, Bourque and Dupuis (2004), for example, define design as:

> *The software engineering life cycle activity in which software requirements are analyzed in order to produce a description of the software's internal structure that will serve as the basis for its construction, (p. 1-3).*

On the broad-scope side, Freeman and Hart (2004), for example, argue that:

> *Design encompasses all the activities involved in conceptualizing, framing, implementing, commissioning, and ultimately modifying complex systems—not just the activity following requirements specification and before programming, as it might be translated from a stylized software engineering process, (p. 20)*

One way of interpreting this debate is as follows. Proponents of a narrow scope of the design process posit that all inputs to design (goals, environment, primitives, requirements and constraints) are fully defined before any property of the object has been decided. Furthermore, the design phase results in a full specification of all relevant object properties before coding begins. In contrast, proponents of a broad scope of design recognize that properties of the object are often defined during requirements elicitation, coding or even testing. Moreover, design may begin without complete knowledge of all information needed and the process may include obtaining additional information. Which side of this debate better reflects software design practice is an empirical question (see ch. 4); the proposed definition of design is compatible with either.

2.3 EVALUATING THE PROPOSED DEFINITION OF DESIGN

In this section we evaluate our definition of design, based on the degree to which it:

- Satisfies a set of four definition evaluation criteria (Appendix B)
- Incorporates areas of agreement in existing definitions (Tables 2-1 and 2-4)
- Resolves disagreements in existing definitions (§2.2.1)
- Appears usable and useful

2.3.1 Definition Evaluation Criteria

Coverage. Whether a definition covers all phenomena in the domain to which it applies, and nothing else, is an empirical question akin to a universal hypothesis. Therefore, the definition can be disproven by a counter example, but never proven. Thus, we evaluated the definition against a diverse set of examples (e.g. Table 2-3) and found that we could describe the examples using the proposed seven elements of design.

Meaningfulness. A definition is meaningful when all its terms have clear meanings. We have explicitly defined all terms having imprecise everyday meanings in Table 2-2.

Unambiguousness. A definition is unambiguous when all its terms have unique meanings. All terms not explicitly defined are intended in the everyday sense, that is, as defined in the dictionary. Where terms have multiple definitions, the intention should be clear from the context.

Ease of Use. The proposed definition is presented in natural language, and is segmented into clearly distinct elements, to ensure clarity for both practitioners and researchers. It is consistent with everyday notions of design and differentiates design from related terms such as invention, decision-making, and implementation. Table 2-3 provides examples of the elements of design to facilitate use of the definition.

2.3.2 Areas of Agreement

The relationship of each area of agreement to the proposed definition is analyzed in Table 2-4. Aspects of design mentioned in the literature that we demonstrated should not be included are marked "discounted." As can be seen in the table, all areas are explicitly or implicitly accommodated.

Table 2-4. Incorporation of Areas of Agreement

Concept	**Consistency with Proposed Definition**
Design as a process	implicit in the verb form of the proposed definition
Design as creation	explicit in the verb form of the proposed definition
Design as *planning*	encapsulated by the design 'specification' (however, planning may be lightweight)
System (as the object of the design)	included in the more abstract term, *design object*
Design as being *deliberate*, or having a *purpose, goal* or *objective*	explicitly included as *goals*
Design as an *activity*, or a collection of activities	implicit in the verb form of the proposed definition
Design as occurring in an environment (or domain/situation/context)	explicitly included as *environment*
Artifact, as the object of the design	included in the more abstract term, *design object*
Needs or *requirements*	explicitly included as *requirements*
Design as *organizing*	discounted
Parts, components or elements	explicitly included as *primitives*
Design as a *human* phenomenon	discounted
Constraints or *limitations*	explicitly included as *constraints*
Process (as the object of design)	included in the more abstract term, *design object* and listed as a class of design object
Design as *creative*	discounted
Optimizing	discounted
Resources	implicit in *primitives* and the verb form (since creating something always uses resources)

2.3.3 Areas of Disagreement

The proposed definition addresses each of the four areas of disagreement among existing definitions (see §2.2.1). First, different objects of design arise: system, artifact and process. We addressed this by using the more general term, *design object* and suggesting major categories of such objects. Second, disagreement exists concerning the scope of design: where or when a design begins and ends (resolved in

§2.2.4). Third, disagreement exists as to whether design is a physical or mental activity. Clearly, design (for humans) is a mental activity, albeit one that may be supported by physical activities (such as drawing diagrams or constructing physical models)[4]. The fourth disagreement, concerning what can be designed, was addressed in §2.2.3.

2.3.4 Usefulness and Usability

We suggest that the proposed definition of the design concept can inform practice in several ways. First, the elements of the definition (excluding agent) suggest a framework for *evaluating* designs: 1) specification – is it complete? 2) object – did we build the right thing? 3) goals – are they achieved? 4) environment – can the artifact exist and operate in the specified environment? 5) primitives – have we assumed any that are not available to the implementers? 6) requirements – are they met, i.e., does the object possess the required properties? 7) constraints – are they satisfied? Of course, answering these questions may be nontrivial or even impossible in some contexts. Second, the breakdown of design into elements can provide a checklist for practitioners. Each element should be explicitly identified for a design task to be fully explicated. For example, a project team may not be able to provide consistent and accurate estimates of design project costs if crucial elements are unknown. Third, a clear understanding of design can prevent confusion between design and implementation activities. Such confusion may lead to poor decisions and evaluation practices. For example, a manager who needs to hire team members for a project may view programmers only as implementers (not understanding the design involved in programming) and thus hire employees with the wrong sorts of skills. Fourth, the elements of design can also be used to specify and index instances of design knowledge for reuse (discussed next).

[4] This mental-physical disagreement should not be confused with the cognitive vs. emergent disagreement discussed in Chapter Three. “Emergent” does not imply “physical”, or vice versa.

2.4 A Conceptual Model for Design Projects

We now propose a conceptual model (a set of concepts and their relationships) for design-related phenomena[5]. Here, we limit our discussion to design within the information systems field. Specifically, we view design as an *activity that occurs within a complex entity, which can be thought of as a human activity system*. Alter (2006) defines a *work system* as “a system in which human participants and/or machines perform work using information, technology, and other resources to produce products and/or services for internal or external customers” (p. 11). Expanding on this concept, we suggest that a *project* is a *temporal trajectory of a work system toward one or more goals*; the project ceases to exist when the goals are met or abandoned. Following this, we define a *design project as a project having the creation of a design as one of its goals*. This relationship is shown in Figure 2-3.

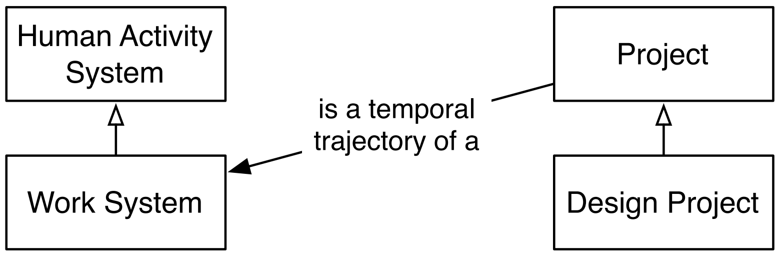


Fig. 2-3. Design Project Generalization Relationship

Note: Shaded arrow indicates relationship; unshaded arrow indicates generalization.

The design project is the central concept of our conceptual model (depicted in Figure 2-4). Each concept is defined and each relationship is discussed in the following section (except the concepts from the definition of design, defined in Table 2-2).

[5] We note that to define a conceptual model of a domain, one needs to define the concepts used to reason about the domain (and their relationships). Such a conceptual structure is an ontology. Hence, we view our proposal as a conceptual model and as an ontology of concepts.

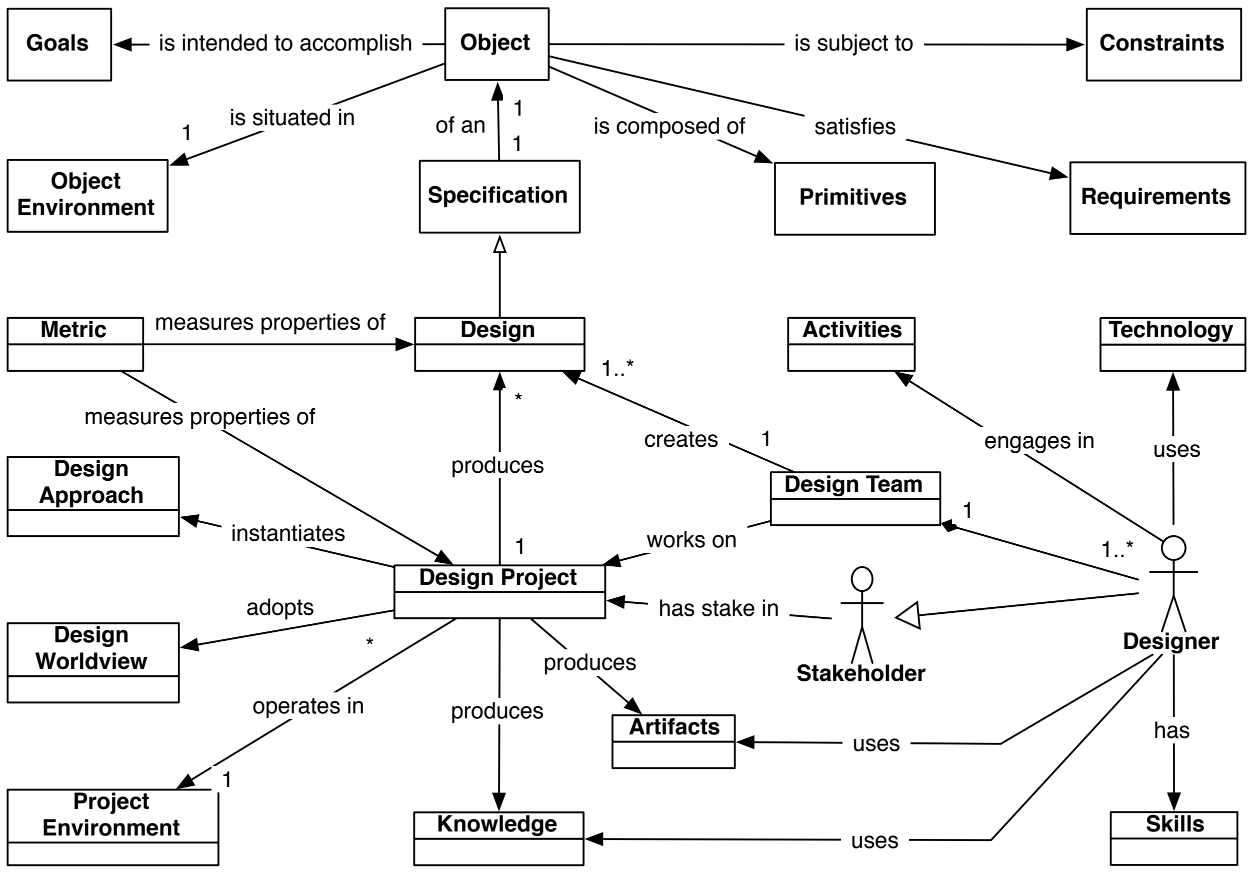


Fig. 2-4. Design Project Conceptual Model

Note: shaded arrows indicate reading direction, unshaded arrows indicate generalization, shaded diamonds indicate composition; all relationships many-to-many unless otherwise indicated.

More Notes. 1) The relationships between the definition-of-design elements (e.g. *constraints*) and the other design project conceptual model elements (e.g. *knowledge*) are omitted to maintain readability. 2) The relationships between *design approach* and elements other than design project are unclear at this time and left for future work. 3) All shown concepts are implicitly *part of* the work system within which the design project takes place. 4) *Creates* is shown in this diagram as a relationship between design team and design, whereas Figure 2-1 depicted creates as a relationship between *agent* and *specification*. In a design project, the *design team* is the *agent*. Furthermore, since the design project conceptual model includes the design concept, the model shows that the *design team creates the design*, which is a *specification*.

2.4.1 Discussion of Concepts

Alter (2006) identifies nine elements of a work system:

- Work practices
- Participants
- Information
- Technologies
- Products and services the work system produces
- Customers for those products and services
- Environment that surrounds the work system
- Infrastructure shared with other work systems
- Strategies used by the work system and the organization

Since a project is a trajectory of a work system, and a design project is particular type of project, a design project should both share all the work system elements and have properties not necessarily shared by other projects and work systems. Here, we discuss each element of the conceptual model, the relationships among elements, and the correspondence between elements of the conceptual model and elements of a work system. The conceptual model includes all the work system elements and, in addition, several elements specific to design projects, which we point out.

Activities. Activities include the specific behaviors engaged in by participants in the design project. These may include interviewing stakeholders, modeling requirements, evaluating proposed design, etc. Activities exist at differing levels of granularity; for instance, modeling can be further divided into sub-activities such as writing scenarios, drawing entity relationship diagrams and then comparing the data models with the scenarios.

Participants and Stakeholders. Participants are the “people who perform the work” (Alter 2006, p. 13). Since individual participants vary among projects, we use the generic label, *stakeholder*. A stakeholder is a person or entity with an interest in the outcome of the project (Freeman 1984). Design projects may have different types of stakeholders – we specifically include the designer type for obvious reasons.

Designer. A designer is an agent that uses his or her skills to directly contribute to the creation of a design. This concept is specific to design projects.

Knowledge. Stakeholders may have and use knowledge during their involvement with the design project. In our interpretation, *knowledge* includes the kinds of information and knowhow used by stakeholders in a design project. To define knowledge, we extend the definition suggested by (Bera and Wand 2009): given the states of the agent and the environment, knowledge is the information that enables "the agent to select actions (from those available to the agent) so as to change the current state to a goal state" (p. 7). The design project can create knowledge as it proceeds – a tenant of the design science research paradigm (Hevner et al. 2004).

Skill. A *skill* is a combination of mental and/or physical qualities that enable an agent to perform a specific action. Skills differ from knowledge as knowledge enables one to *select* actions.

Technologies. *Technologies* are artificial, possibly intangible, tools and machines. Technologies can be used by the design team to create the design.

Design. The *design*, defined above, is the product that the design project aims to produce. This concept is specific to design projects.

Environment and Infrastructure. Figure 2-4 combines Alter's *environment* and *infrastructure* constructs because both represent aspects of the project that are outside its scope (Alter 2006). Checkland (1999) argues that, to properly model a system, the modeler must first model the system it serves. This wider system served by a design project is its environment. Alter (2006) argues: "the work system should be the smallest work system that has the problems or opportunities that are being analyzed" (p. 22). Following this, then, *the environment is the smallest coherent system served by the design project*.

The environment construct is a potential source of confusion because *Design Project* and *Design* both have environments. The design project's environment is the work system in which the project occurs; the design's environment is the context in which in the object is to operate.

Design Approach and Strategy. A *design approach* is a set of beliefs about how design (and related activities) *should* be done (This should not be confused with the slightly different usage of the same term by Iivari et al. 2000). Examples include The Unified Software Development Process (Jacobson et al. 1999), and the Systems Development Lifecycle (Bourque and Dupuis 2004). "Strategies consist of the guiding rationale and high-level choices within which a work system, organization, or firm is designed and operates" (Alter 2006, p. 14). As a design approach contains rationale and is implemented as choices, it corresponds to Alter's *strategy* construct. A design project may explicitly instantiate a formal design approach by using some or all of its elements. If a broad scope of design is taken (§2.2.4), a design approach can refer to the entire development process from problem identification to implementation and maintenance.

We have adopted the more general term design "approach" instead of "process" or "methodology" because "design processes" often contain much more than sequences of activities and "methodology" is used both as a formal word for 'method' and as the systematic study of methods. This concept is specific to design projects.[6]

Design Team. All designers involved in a project comprise the design team. The design team engages in activities and uses technologies to create the design and other (intermediate) artifacts. This concept is specific to design projects.

[6] In hindsight, these differences are overwhelmed by the inconsistent, often interchangeable usage of "method," "methodology," "approach," "paradigm," etc. in the literature. At this time, no comprehensive dissection of the ontological relationships among these terms is available. For our purposes, any set of optative beliefs concerning design practice is a "design approach." Meanings of similar terms, used below, should be clear from their context.

Artifacts. In this model, *artifact* is used in the broad, anthropological sense of any object manufactured, used or modified by agents in the design project. Examples include conceptual models, software development environments, whiteboards, and e-mails. (This is not to be confused with an artifact that is the object of design.)

Metric. A metric is a way or standard of taking a measurement, where measurement refers to a process of assigning symbols (often numbers) to an attribute of an object or entity and also the symbols assigned (Fenton 1994; Finkelstein 1984; Roberts 1979). In the case of a design project, metrics are used for evaluating specifications, objects, the project, etc.

Design Worldview. A worldview or (more precisely) *Weltanschauung* is a way of looking onto the world. It is sometimes used in social sciences to indicate a set of high level beliefs through which an individual or group experiences and interprets the world. As a precise definition of this concept is elusive, we suggest some possibilities for classifying worldviews in the design context (Table 2-5). Worldviews are not mutually exclusive, i.e., some design projects may explicitly adopt one or more design Weltanschauung. However, even without such an explicit view, every project participant brings a view of design to the project, and the combination of these views comprises the project's collective Weltanschauung. This concept is not necessarily common to all work systems.

Table 2-5. Identified Design Weltanschauung

Weltanschauung	Description	Proponents / Examples
Problem Solving	Design can be seen as an attempt to solve a known problem, a view characterized by the beliefs that a problem exists and is identifiable and that the success of a design is related to how well it solves the problem.	(Hevner et al. 2004; Simon 1996), the engineering literature
Problem Finding	Design can be seen as an attempt to solve an unknown problem, implying that understanding the problem is part of the design process.	(Polya 1957; Schön 1983)
Epistemic	Design can be seen as a learning process where actions that can lead to improvements to the current situation (in the eyes of stakeholders) are discovered.	(Checkland 1999)
Inspiration	Design can be seen as a result of inspiration, i.e., instead of beginning with a problem, design begins with an inspiration of the form 'wouldn't it be great if....'	(Kessler 2007)
Growing	Design can be seen as growing an object, progressively improving its fit with its environment and purpose.	(Beck 2005; March and Smith 1995)

2.4.2 Evaluation of the Conceptual Model of Design Projects

To evaluate the set of concepts underlying the proposed conceptual model, we use evaluation techniques suggested for ontologies. Ontology evaluation can proceed in several ways. The competency questions approach involves simultaneously demonstrating usefulness and completeness by analytically proving that the ontology can answer each competency question in some question set (Grüninger and Fox 1995). The ontology is then considered complete with respect to that question set. In contrast, (Noy and Hafner 1997) suggests two dimensions of ontology quality: coverage and usefulness. Coverage can be demonstrated by comparing an ontology to a reference corpus: terms in the corpus that do not fit into the ontology indicate lack of coverage. Furthermore, “An important way of evaluating the capabilities and practical usefulness of an ontology is considering what practical problems it was applied to” (Noy and Hafner 1997, p. 72).

Since the proposed “ontology” is not intended to answer particular questions, evaluation with respect to coverage and usefulness seems preferable. Assessing the conceptual model’s coverage is beyond the scope of this paper; however, a possible approach is evident. By surveying a range of design approaches, e.g. The Rational Unified Process, Agile Methods, The Waterfall Model, The Spiral Model, etc., a list of design concepts can be generated and compared to the proposed conceptual model. Coverage can be measured by the extent to which these revealed concepts match the proposed concepts (usually as instances of the generic concepts suggested above).

We address usefulness in Section 2.5.2 by demonstrating how the conceptual model can be applied *in principle* to the practical problem of classifying and contrasting design approaches.

2.5 POTENTIAL APPLICATIONS

In this section we discuss possible applications of the proposed definition of design and of the design project conceptual model. First, we suggest the use of the elements of the definition of design to classify and index design knowledge. Second, we discuss the use of the design project conceptual model for comparing and classifying approaches to software design.

2.5.1 Application 1: Design Knowledge Management System

The importance of reuse in software development has been widely recognized. Mili et al. (1995) states that software reuse "is the (only) realistic opportunity to bring about the gains in productivity and quality that the software industry needs." Ambler (1998) suggests a number of reuse types in software engineering, divided into two broad categories: *code reuse and knowledge reuse*.

Code reuse includes different approaches to organize actual code and incorporate it into software (e.g. libraries of modules, code fragments, or classes) and the use of off-the-shelf software. Code repositories can be considered design knowledge bases. Though some authors (e.g., Beck 2005) argue that the best mechanism to communicate design is the code itself, sharing design is not the same as sharing design *knowledge*. Even well-commented code does not necessarily communicate design knowledge such as the rationale for structural decisions (e.g., why information was stored in a certain structure).

Knowledge reuse refers to approaches to organizing and applying knowledge about software solutions, not to organizing the solutions themselves. It includes algorithms, design patterns and analysis patterns[7]. Perhaps the most successful attempt to codify software design knowledge is the design patterns approach. A design pattern is an abstract solution to a commonly occurring problem. The design pattern concept was originally proposed in the field of architecture and became popular in software engineering following the work by (Alexander et al. 1977; Gamma et al. 1995)[8].

[7] Other approaches to organizing software development knowledge include architectural patterns, anti-patterns, best practices and development methods. Standards and templates (e.g. for documentation) can also be considered organized knowledge.

[8] The Portland Pattern Repository (http://c2.com/ppr/) is an example of a design pattern repository that could be called a design knowledge base.

Despite the apparent benefits of sharing design knowledge, it has been observed that it is difficult to accomplish. “Experts and veterans continue to shun reuse from public knowledge spaces” and that when the needed artifact “was not found in their private space” “it was also less costly for them to recode the desired artifact than to conduct a global search for one” (Desouza et al. 2006, p. 98). This indicates the difficulties of locating needed design knowledge (or other software artifacts). One way to facilitate searching is to classify and index design knowledge on meaningful dimensions. Next, we demonstrate by example how the proposed definition of design can provide such dimensions and thus help index instances of design knowledge.

An Example. In programming, an iterator object traverses a collection of elements, regardless of how the collection is implemented. Iterators are especially useful when the programmer wants to perform an operation on each element of a collection that has no index. The iterator design pattern is a description of how best to implement an iterator. Table 2-6 shows how the design knowledge represented by the iterator design pattern might be indexed using the elements of the proposed definition of design. Note that, in this application the goals, requirements, etc. are properties of the iterator, not of the design pattern. The goal of the design pattern, for instance, is to explain how to implement an iterator (and not how to traverse a collection).

Table 2-6. Example of Design Knowledge Indexing

Object Type	symbolic script
Object	iterator
Agent	application programmer
Goals	access the elements of a collection of objects
Environment	object-oriented programming languages
Primitives	primitives and classes available in object-oriented programming languages
Requirements	have a means of traversing a collection, be implementable with respect to a variety of collections, etc.
Constraints	must not reveal how the objects in the collection are stored, etc.

By classifying design knowledge according to these dimensions, a designer can ask questions of the form ‘are there any design patterns (*object*) for traversing a collection (*requirement*) in an object-oriented language (*environment*)?' We suggest that such classification can help organize and share design knowledge and thus help improve designers' effectiveness and efficiency in locating and applying useful design knowledge.

2.5.2 Application 2: Design Approach Classification Framework

Classifying design approaches is important for several reasons. First, practitioners need guidance in selecting appropriate design approaches for their situations. Second, such classification can facilitate comparative research on approaches. Third, it can guide the study of the methods employed by experienced developers (which, in turn, can inform research on software design and software processes).

At least two types of classifications of design approaches are possible. First, a classification can be based on the actual elements (e.g. steps, interim products) that comprise a design approach or process. This can be termed a "white-box" approach. Second, a classification can be based on the environment that surrounds a design approach. For example, specific objectives of the approach, the view of design it embeds, and the roles of stakeholders. This can be termed a "black-box" approach.

We suggest that the proposed design project conceptual model can be used to create a black-box classification scheme for design approaches. This can facilitate understanding the deep structure of design approaches and theorizing about conditions under which different approaches are appropriate. To demonstrate, using dimensions derived from the design project conceptual model, Table 2-7 classifies three design approaches: the Soft Systems Methodology (Checkland and Poulter 2006), Extreme Programming (Beck 2005) and the Rational Unified Process (Kruchten 2003). We chose these three because each is prominent in the literature and represents a significantly different perspective.

2.6 Discussion and Implications for Software Design Research

2.6.1 Completeness, Design Agency and Software Architecture

For years, researchers have argued that informal specifications may suffer from incompleteness (Reubenstein and Waters 1991). Above, we defined a specification as a detailed description of an object in terms of its structure. This allows a more precise characterization of incompleteness. We suggest that a design specification is complete when the structural information that has been specified is sufficient for generating (in principle) an artifact that meets the requirements.[9]

Table 2-7. Example Classification of Design Approaches

Approach	Soft Systems Methodology (SSM)	Extreme Programming	Rational Unified Process (RUP)
Object	human activity systems	software	software
Weltanschauung	epistemic	growing	problem solving
Metrics	situation dependent "measures of performance;" the 5 E's: efficacy, efficiency, effectiveness, ethicality, elegance	advocated, but none provided; differentiates internal and external quality	defines metrics as part of the process; fundamental quality measure: 'does the system do what it is supposed to?'
Nature of Specification	action items, i.e., some action that can be taken to improve the situation, in the eyes of the stakeholders	source code	UML models (use cases and diagrams); source code
Activities	semi-structured interviews, analysis, modeling, debate	coding, testing, listening, designing (refactoring)	broadly: requirements gathering, analysis and design, implementation, testing, deployment, configuration and change management, project management (each with sub activities)
Artifacts	interview guides and transcripts, collections of notes, rich pictures	prototypes, test suites	stakeholder requests, vision, business case, risk list, deployment plan, analysis model, etc.
Users	owner, actor, customer	programmers/developers, clients	RUP users take on one or more of six role categories: analysts, developers, managers, testers, production and support, and additional.
Stakeholders	stakeholders is an explicit concept in SSM	divided into "business" and "development"	"stakeholder" is a "generic role" that refers to "anyone affected by the outcome of the project" (p. 276)
Tools	rich pictures, interview guides, debates and group discussions	story cards, diagrams, an integration machine, several development workstations	IBM Rational Suite

[9] Since it is impossible to list every property of any object, we limit our discussion to "relevant" properties, i.e., a sufficient subset of properties to allow a "generating machine" (e.g. a human being or a manufacturing robot) to deterministically assemble the object.

Based on the notion of completeness we have defined above, we can now identify three forms of incompleteness. First, relevant components or connections may be missing. For example, the specification of a bicycle may be missing the qualification that the tires be *attached* to the rims. Second, a particular component or connection may be insufficiently described. For example, it may not be clear from the specifications *how* the tires should be attach to the rims or which tire to use. (Please note, here we are not distinguishing here between incompleteness and ambiguity.) Third, a component may not be part of the set of primitives but can be designed based on existing primitives or other components. The design will not be complete until specifications exist for all such components.

Completeness is not an end state for a design specification. Future changes in the set of primitives may render a previously complete specification incomplete. Furthermore, many researchers now agree on the importance of "the fluidity, or continued evolution, of design artifacts" (Hansen et al. 2007, p. 36). In situations where future conditions are difficult or impossible to predict, one response is to focus on the evolvability and adaptability of the design object (Gregor and Jones 2007; Simon 1996). The characterization of design advanced here provides important implications for design fluidity. First, specification completeness does not imply constancy. A design specification can be evolved to respond to changing conditions by its original creator, the design object's users, or others,. Furthermore, the elements of the proposed definition enumerate classes of possible changing conditions in response to which the design object or specification may need to evolve. For example, the specification may be modified in response to changes in the environment. Finally, the set of requirements may contain stipulations for a design object's evolvability by end-users or others.

This raises questions of who exactly, in a typical software project, is the design agent? We have defined the design agent as the entity or group of entities that specifies the structural properties of the design object. When users are involved in design, whether a user is part of the design agent depends on the nature of his or her involvement. Simply providing information, such as requirements, does not make a user part of the design agent, nor does testing and giving feedback. To share in design agency, the user must *make at least one structural decision regarding the design object*. As a complete discussion of this issue would require incorporating the vast literature on authority and organizational power (Aghion and Tirole 1997; Pfeffer 1992); here, we simply point out that official authority to make a structural decision does not necessarily coincide with the practical reality of who makes a decision. The key to identifying the design agent is in separating those individuals (or groups) who provide information about constraints, primitives and the other design elements, and those who decide on structural properties.

Another theme currently gaining significant attention is software architecture (Hansen et al. 2007). Software architecture is the level of design concerned with "specifying the overall system structure" (Garlan and Shaw 1993, p. 1). This presents a possible difficulty: if a specification is a description of the components of a design object and their relationships, which components and relationships are parts of the software architecture? How does one distinguish high-level components and relationships from low-level ones? A design specification for a complex system might exist simultaneously at many levels of abstraction. Alternatively (and perhaps more likely) high-level components are defined in terms of lower-level components and these are defined in terms of even lower-level components, etc., until everything is defined in terms of primitive components. In this multilevel view of design, the software architecture concept is a threshold above which is architecture, and below which is 'detailed design.' Is this threshold arbitrary? At this time, we can only suggest these fundamental questions about software architecture as topics for future research.

2.6.2 Implications for Research

The proposed characterization of design also gives rise to several implications for design research. To date, much design research has been prescriptive, addressing practical recommendations and guidance for software development; yet, little theoretical, and even less empirical, treatment of software design exists (Wynekoop and Russo 1997). This has led to many calls for field research in this area (e.g., Freeman and Hart 2004; Simon 1996; Sullivan 2003; Wynekoop and Russo 1997). Defining design as the process by which one specifies an object's structural properties raises several important research topics:

1. How is software designed in practice?
2. To what extent is each element of the proposed definition (requirements, primitives, etc.) known when design begins?
3. Can a single theory explain every behavior involved in software design?
4. How do designers discover each kind of information?

Put another way, academic treatment of software design may involve developing and testing interdependent *process* and *causal* theories of design. Process theories (see Appendix C) can be used to explain *how* design occurs. Causal theories deal with effects of some variables on others and can be used to suggest how to design better.

2.6.3 Goals vs. Requirements in Information Systems Development

The notion of *goal* is considered essential in requirements engineering as the concept that captures the motivation for developing a system ("why") and the way to define objectives at various levels of abstraction (van Lamsweerde 2001). Our definition of design includes both goals and requirements. We now describe briefly how these two concepts relate within this context.

We start by observing that in the information systems context, a design object is an artifact situated[10] in an environment termed the *application domain* and designed to support activities of the *application domain*. Typically, the application domain is an organizational setting such as a business or a part of a business. The application domain itself operates within an *external environment*. For example, a business is embedded within a business environment comprised of customers, suppliers, competitors, service providers, and regulatory bodies. The application domain and the external environment interact: the environment generates *stimuli* that invoke *actions* in the domain. The actions of the domain can *impact* its environment. Similarly, the artifact is situated in the domain. The domain and the artifact interact: the domain creates external stimuli which invoke actions in the artifact. The actions of the artifact can impact

[10] The word *situated* should not be taken literally in the physical sense, but in the sense that the artifact acts interacts with other components in a domain.

the domain. Once the artifact is embedded a change occurs: the domain now includes the artifact. Now the modified domain (with the included artifact) interacts with the external environment. This view is depicted in Figure 2-5.

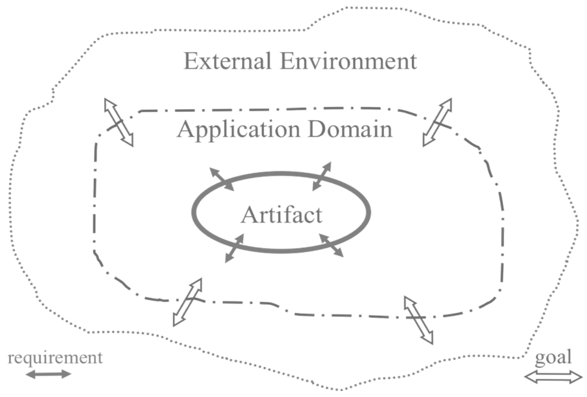


Fig. 2-5. Separate Domains of Goals and Requirements

Domain goals, or simply *goals*, are the intended impact of the actions in the domain on the external environment[11]. The purpose of the artifact is to enable the domain to accomplish these goals more effectively and efficiently. The artifact does this by responding to stimuli from the domain is ways that will support the domain in accomplishing the goals. Accordingly, requirements can be defined as the *properties that the artifact should possess in order to accomplish its purpose*. These requirements can be of two types:

1. *Structural requirements* are intended to assure that the object can match well with the other components of the domain or its external environment.
2. *Behavioral requirements* define the *desired responses* of the artifact to stimuli from the domain (or from the environment) generated when the domain is working to accomplish its goals. These responses, in turn, affect the domain (and, directly, or indirectly, the environment).

The *requirements definition* process can be viewed as identifying the properties (structural and behavioral) that the artifact should possess to support the domain in accomplishing the goals. Design can

[11] For example, while it may appear that 'profitability' is related to the business rather than to its environment, profitability is the outcome of exchanges between a business and its environment, and the business should act such that these exchanges create the desired outcome.

be viewed as the way to assemble available types of components in order to accomplish an artifact that meets the requirements.

2.7 CONCLUSION

The work we describe here is motivated by the observation that a clear, precise and generally accepted definition of the concept of design can provide benefits for research, practice and education. Our literature study indicated that such a definition was not available. Therefore, we synthesized a new definition, which views the design activity as a *process*, executed by an *agent*, for the purpose of generating a *specification* of an *object* based on: the *environment* in which the object will exist, the *goals* ascribed to the object, the desired structural and behavioral properties of the object (*requirements*), a given set of component types (*primitives*), and *constraints* that limit the acceptable solutions. As one possible application of our definition we demonstrate how it can be used to index design knowledge to support its reuse.

As a second step, we situate the design concept in a network of related concepts appropriate to the information systems and software development domain by proposing a conceptual model of design projects. The intent of this conceptual model is to facilitate study of design projects by identifying and clarifying the main relevant concepts and relationships. We demonstrate the usefulness of this conceptual model by using it to compare several approaches to system and software design.

Finally, we link our proposed definition of design to current themes in design research, in particular, the notion of requirements as used in system development.

One purpose of this work is to facilitate theoretical and empirical research on design phenomena. We hope this chapter will contribute to clarifying understanding and usage of design and related concepts and encourage scientific research on design. Another purpose is to create a set of concepts that can guide practice and education in the information systems and software design domain.

This article includes examples of design from diverse areas including prehistoric hunters, artists, and architects. The reader may question whether such a broad perspective on design is useful for studying software development. Yet, it remains unknown whether software designers are more like engineers or artists, or are not much like either. This can only be answered by *observing* the behaviors of a wide range of those who are engaged in software design: elite and amateur, engineers and hackers, formally trained and self-taught. Having a well defined set of concepts to describe and reason about phenomena related to design and design projects can provide guidance for this empirical work.

CHAPTER 3: THE SENSEMAKING-COEVOLUTION-IMPLEMENTATION FRAMEWORK OF SOFTWARE DESIGN

3.1 Introduction

Software development and maintenance constitute substantial economic activity – in 2006, the 500 largest software companies employed 2,914,480 and accrued revenues of $394 billion (Desmond 2007) and "total global spending on technology goods, services, and staff … [reached] $2.02 trillion" (Bartels et al. 2006). The magnitude of this spending makes estimates of software project failure rates far more alarming. Estimates of completely abandoned projects vary between 10% and 44% while between 16% and 52.7% experience "major truncation or simplification … prior to full implementation" (Ewusi-Mensah 2003, p. 17, 19). In comparison, if civil-engineering projects had similar success rates, abandoned construction projects would be scattered throughout our cities. The high failure rates of software projects highlight the potential value of "design methodics" (Eekels 2000) – methods, technologies, techniques and practices intended to improve project outcomes – and academic research to create or improve them.

However, developing better design methodics is hampered by the fact that the shape and organization of the software design process is not well understood (Freeman and Hart 2004; Simon 1996; Sullivan 2003; Wynekoop and Russo 1995). In this regard, software design differs from other design-related fields such as architecture and urban planning, where researchers have investigated design practices using observational methods (Schön 1983). Truex et al. (2000) argued that "the history of information systems development is typically interpreted as the history of *methods* for systems development" (p. 56, emphasis added). In a meta-analysis of research on systems development methods (SDMs), Wynekoop and Russo (1997) identified 67 studies attempting to understand or describe SDMs – of these 63 were based only on the authors' speculations or opinions with no theoretical or empirical grounding, the remaining four being surveys focusing on a small number of SDMs or on specific techniques. Empirical research to understand and to describe design phenomena and methodics is the essence of *design science* (Eekels 2000). Recognizing the relationship between design science, developing better design methodics and improving design project outcomes generated the insight motivating this study.

Motivation: *Before prescribing what software design teams should do, it is necessary to understand what they currently do.*

Curiously, our collective ignorance of design practice remains despite four decades of calls for empirical design science research (cf., Simon 1969). One explanation for why questions concerning the nature of software design practice are rarely studied empirically is that it is entirely unclear how to study them. Suppose we wish to test a theory of how software is designed. What would such a theory look like? Where could one be found? Are traditional research methods appropriate to test it? The answers to these and related questions are sufficiently complex that substantial analytical work is required to facilitate empirical study of software design practice – hence the purpose of this chapter.

Purpose: *The purpose of this chapter is to review what is believed about how professionals create software in practice and to synthesize these beliefs in an empirically testable form.*

To this end, clear definitions (from ch. 2) may help to guide analysis. A software design project is defined as a temporary trajectory of a Work System (Alter 2006) toward the goal of creating a software design. A software design is a specification of a software object, by an agent, to accomplish goals, in a particular environment, using a set of primitive components, satisfying a set of requirements, and subject to set of constraints. The specification may be a document, formal model, or software code. Moreover, software design (verb) is an activity where an agent creates or attempts to create a software product in some context. This creates the generic design situation of Figure 3-1 – at a time, an agent in an environment forms an intention to create a design. At a later time, a software artifact exists.

This raises the question how and why does the software artifact come into existence? More precisely, *what is the process whereby development teams create software, in practice?* By *the process*, I mean the activities software design agents share (not that all software designers behave identically).

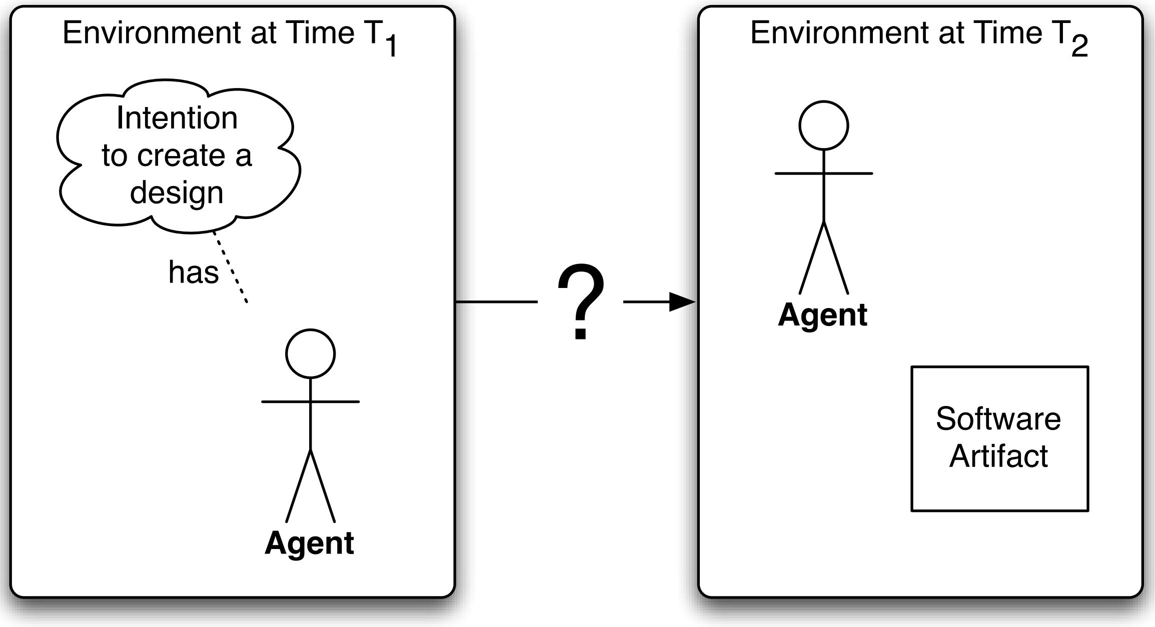

Environment at Time T_1
Intention to create a design
has
Agent
?
Environment at Time T_2
Agent
Software Artifact

Fig. 3-1. Generic Design Scenario

In summary, I posit that better design methodics may reduce software project failure rates; however, lack of theoretical knowledge regarding software design phenomenology hampers efforts to improve methodics. Moreover, how to empirically study design methodics and phenomena is unclear. Therefore, I propose to facilitate empirical design science research by organizing design literature (§3.2 and §3.3) according to a conflict between two contrasting perspectives on design (§3.4). I further propose to operationalize these perspectives into contrasting, empirically testable software design process theories, thereby representing two alternative views of the shape and the organization of the design process (§3.5). Section 3.6 concludes with a summary of contributions, implications and future work.

3.2 Literature Review – Pre-theoretical Contributions

This section reviews specific research findings in design science and related fields that are immediately useful in this chapter. Other important but not imminently applicable research is discussed in Appendix D.

3.2.1 Design Paradigms

Dorst and Dijkhuis (1995) argued that "there are many ways of describing design processes" and discussed "two basic and fundamentally different ways" (p. 261) – the Technical Problem-Solving paradigm (Simon 1996) and the Reflection-in-Action paradigm (Schön 1983). Reflection-in-Action was explicitly positioned as an alternative to Technical Problem-Solving. Similarly, Love (2000) claims that "Information processing is the most common theoretical perspective on design found in the contemporary literature of design research", while "creativity", which "forms the basis of this alternative metaphor of design research … has been unfashionable in engineering design research for some time" (p. 308). Below, I discuss the Technical Problem-Solving and Reflection-in-Action paradigms, and their relationship to information processing and creativity metaphors.

The Technical Problem-Solving Paradigm. "According to the model of Technical Rationality – the view of professional knowledge which has most powerfully shaped both our thinking about the professions and the institutional relations of research, education, and practice – professional activity consists in instrumental problem-solving made rigorous by the application of scientific theory and technique" (Schön 1983, p. 21). Technical Rationality requires that problems be given, in the sense that goals are agreed upon in advance and constraints are knowable. It is the foundation of positivist epistemology (Schön 1983).

Based on this model, Simon (1996) elucidates the Technical Problem-Solving design paradigm, positing design professionals as rational agents attempting to optimize a design candidate vis-à-vis known constraints and objectives (Newell and Simon 1972; Simon 1996). Where the problem space is so large that finding an optimal solution is beyond the designer's limited processing power, the designer will "satisfice" or "find decisions that are good 'enough'", often through heuristic search (Simon 1996, p. 27). Simon coins the term "procedural rationality" for this "finding a way of calculating, very approximately, where a good course of action lies" (p. 27), thereby differentiating a (boundedly) rational action from rational outcome.

In addition to procedural rationality, Simon's thesis rests on the assumption that "a physical symbol system … has the necessary and sufficient means for general intelligent action" (p. 23). More specifically, intelligent agents model their environments and possible actions using symbol structures; "hence the programs that govern the behavior of a symbol system can be stored, along with other symbol structures, in the system's own memory, and executed when activated" (p. 22). This is equivalent to the cognitivist view of human action. In the cognitivist view, all human action is executed and understood through a *plan*. A plan, i.e., "a sequence of actions designed to accomplish some preconceived end" (Suchman 1987), is a prerequisite to action. Unanticipated conditions trigger re-planning and evaluation is performed by comparing what was done to what was planned.

Moreover, Technical Problem-Solving is consistent with Love's Information Processing metaphor, where "design is seen as the codification, selection and management of information. The characteristic design method is the use of information selecting algorithms." Furthermore, "the designer is seen as a machine capable of rationally selecting and connecting together elemental information to satisfy a set of constraints" (Love 2000, p. 309). Meanwhile, Simon (1996) suggests that "All mathematics exhibits in its conclusions only what is already implicit in its premises … this view can be extended to all of problem solving – solving a problem simply means representing it so as to make the solution transparent" and that even "if this is too exaggerated a view – a deeper understanding of how representations are created and how they contribute to the solution of problems will become an essential component in the future theory of design" (p. 132). Therefore, information processing is central to Technical Problem-Solving. Moreover, Simon's modeling of the human mind as a physical symbol system is consistent with the information processing view of the designer as approximating a rational machine. Neither Simon nor Love formally define either Technical Problem-Solving or information processing; however, they do appear consistent.

Simon's Technical Problem-Solving paradigm has been used to code activities of industrial designers in protocol studies (Dorst and Dijkhuis 1995). However, my literature review did not uncover any empirical evidence that designers exhibit procedural rationality.

The Reflection-in-Action Paradigm. Social constructivism is an epistemological paradigm positing that knowledge is derived from social interactions (Berger and Luckmann 1966). In this sense, goals and constraints are socially constructed concepts. Building on social constructivism and empirical studies of professional practice, Schön (1983) devised the Reflection-in-Action design paradigm, where design is a reflective conversation between the designer and the situation. The designer alternates between framing (conceptualizing the problem), making moves (where a move is a real or simulated action intended to improve the situation) and evaluating those moves. Multiple agents may collectively reflect in action using boundary objects (Levina 2005). (A boundary object is an object that is simultaneously flexible enough to serve multiple parties and robust enough to maintain its identity. Examples include design drawings and physical prototypes.) Reflection-in-Action differs from Technical Problem-Solving in many ways: 1) professionals respond to a problematic situation with many possible interpretations, rather than responding to a given problem; 2) professionals form and explore hypothesis about potentially beneficial actions rather than optimizing or satisficing design candidates for known objectives and constraints.

Schön (1983) argued that "when someone reflects an action, … he does not keep means and ends separate, but defines them interactively as he frames a problematic situation. He does not separate thinking from doing" (p. 69). This rejection of planning as the foundation of action is consistent with what may be called the ethnomethodological view of human action (ethno-view). In the ethno-view, "the organization of situated action is an emergent property of moment-by-moment interactions between actors, and between actors and the environments of their action" (Suchman 1987, p. 179) while "plans are representations, or abstractions over action" (p. 186). Simply, human actions are always improvised, even when previously planned.

Both Reflection-in-Action and the ethno-view imply that the basis for innovation is the creativity and experience of the designer, Love's creativity metaphor: "where design is seen as a creative process the dominant mechanism of decision-making and evaluation is the use of 'feeling.' … All creative design methods necessarily depend on a sufficient base of experience residing within the designer(s)" (Love 2000, p. 310). Similarly, Schön (1983) explains:

> *"The practitioner has built up the repertoire of examples, images, understandings, and actions.... When a practitioner makes sense of the situation he perceives to be unique, he sees it as something already present in his repertoire…. [Furthermore,] each practitioner tries to adapt the situation to the frame … through a web of moves, discovered consequences, implications, appreciations, and further moves … The situation talks back, the practitioner listens, and as he appreciates what he hears, he reframes the situation once again..." (p. 132).*

Schön's use of "appreciation" is akin to Love's emphasis on feeling and both emphasize experience. As in the previous section, neither author formally defines his perspective; hence, I simply conclude that Reflection-in-Action is intuitively compatible with the creativity metaphor.

The Reflection-in-Action paradigm gains empirical support from Schön's (1983) own case studies. Furthermore, after a series of action research studies, Mathiassen (1998) concluded that "Reflection-in-Action provides a useful understanding of systems development practice" (p. 42).

Summary. Table 3-1 summarizes the differences between the Technical Problem-Solving and Reflection-in-Action design paradigms in terms of their underlying assumptions. Schön succinctly states his primary attack on Simon's assumptions: in situations where "Technical Problem-Solving occupies a limited place within the inquirer's reflective conversation with his situation; the model of Technical Rationality appears as radically incomplete" (Schön 1983, p. 165). Activities including goal setting and resolving disparate perspectives on design choices and appropriate methods or practices clearly lie outside Technical Problem-Solving. Therefore, the extent to which each of these paradigms applies specifically to software design depends on its scope (i.e., whether it includes problem-understanding and implementation) and the centrality of Technical Problem-Solving in the work of software developers.

Table 3-1. Summary of the Technical Problem-Solving and Reflection-in-Action Paradigms (Adapted from Dorst and Dijkhuis 1995, Figure 1)

	Technical Problem-Solving	**Reflection-in-Action**
Primary Proponent	Simon (1996)	Schön (1983)
Epistemology	Positivist	Constructivist
Theory of Action	Cognitivist	Ethnomethodological
Designer	information processor / rational agent	person constructing his or her reality
Design Problem	given evaluation criteria for alternatives, which are representable as points in a problem space	essentially unique and poorly understood at the start
Design Knowledge	knowledge of design procedures and scientific laws	artistry of design (when to apply which procedure / piece of knowledge)
Guiding Metaphor	Information Processing	Creativity
Deisgn Process	a rational search process	a reflective conversation

3.2.2 The Design Space / Problem Space Distinction

An important development in conceptual research on design involved separating the design (or solution) space from the problem space. Purao et al. (2002) described "the problem space … as the metaphoric space that contains mental representations of the developer's interpretation of the user requirements" and "the design space" as " the metaphoric space that contains mental representations of the developer's specific solutions" (p. 251-252). This distinction is evident in analytical (e.g., Dorst and Cross 2001), empirical (e.g., Gero and Mc Neill 1998) and prescriptive (e.g., Checkland 1999) design research (see Purao et al. 2002 for more examples). The basic idea of separating the designer's reasoning about what *currently* exists from that of what *may* exist in the future underlies several of the theories discussed in Section 3.3.

Furthermore, several papers discuss the relationship between the two spaces. Alexander (1964) called this "probing," while Berente and Lyytinen (2006) called it "iteration". Schön's (1983) referred to a "reflective conversation" of making and evaluating "moves". This iterative relationship is also the basis of Maher et al.'s Problem-Design Exploration Model (§3.3.2) and is related to mapping theories of design (Appendix D).

3.2.3 Design Scope and the Design Process I/O Model

In the Waterfall Model, discussed below, design is modeled as the stage between requirements analysis and implementation. This assumes that the designer is provided with a well-defined problem to solve (Royce 1970). Some have abandoned this notion, saying that “we had moved away from working with the idea of an ‘obvious’ problem which required solution, to that of working with the idea of a situation which some people, for various reasons, may regarded as problematical” (Checkland 1999, p. A8). In this view, design begins not with a well-known problem, but with an agent forming the intention to design something (ch. 2). In fact, it is possible for a problem’s structure to remain unclear even after it is solved (e.g., airplanes preceded the physics of flight).

In Chapter Two, I provided the black-box, input/output conceptual model of the design process (Figure 3-2). Each of the model’s concepts is defined in Table 3-2. In this model, design is an activity, engaged in by a design agent. The agent begins designing with intentionality, knowledge of the type of object being designed (e.g. software, a building) and knowledge of the design’s would-be environment (e.g., the Internet, a canyon). The designer may also have knowledge of specific goals, primitives (of which the design is composed), requirements and constraints. During the design process, the designer’s conceptualization of the environment, goals, primitives, requirements, constraints and its own intentions may change. The output of the process is a specification of the design object, which may be a document explaining the object’s composition or the object itself.

The Design Process Input/Output Model is compatible with the Reflection-in-Action paradigm insofar as it only posits that the designer has knowledge of the problematic situation (environment, requirements, constraints), not that its knowledge is complete or that the problem is bounded. The primary limitation of this model is that it does not attempt to explain how or why the design activity occurs – only the objects, concepts and knowledge involved. However, if we posit that these concepts apply to all forms of design, including software design, they may be useful for informing or evaluating theories of software design.

Table 3-2. Definitions of Concepts in the Design Process I/O Model (from ch. 2)

Concept	Meaning
Agent	the entity or group of entities that specifies the structural properties of the design object
Constraints	a structural or behavioral restriction on the design object*
Design	an act by which an agent intentionally creates a specification of an object, intended to accomplish goals, in a particular environment, using a set of primitive components, satisfying a set of requirements, subject to constraints; the result of this act
Environment	The object environment is the context or scenario in which the object is intended to exist or operate (used for the noun form). The agent environment is the context or scenario in which the design agent creates the design (used for the verb form).
Goals	the desired impacts of design object on its environment. Goals are optative statements (i.e. indicating a wish) that may exist at varying levels of abstraction (Lamsweerde 2001)
Intentions	the agent's "readiness" to engage in designing or achieve some purpose by way of designing (see Ajzen 2005)
Primitives	the set of elements from which the design object may be composed (usually defined in terms of types of components assumed to be available)
Requirements	a structural or behavioral property that a design object must possess.
Specification of Object	A specification is a detailed description of an object in terms of its structure, e.g., the primitives used and their connections.
Type of Object	The kind of object being specified, e.g., software, mechanical device, law.

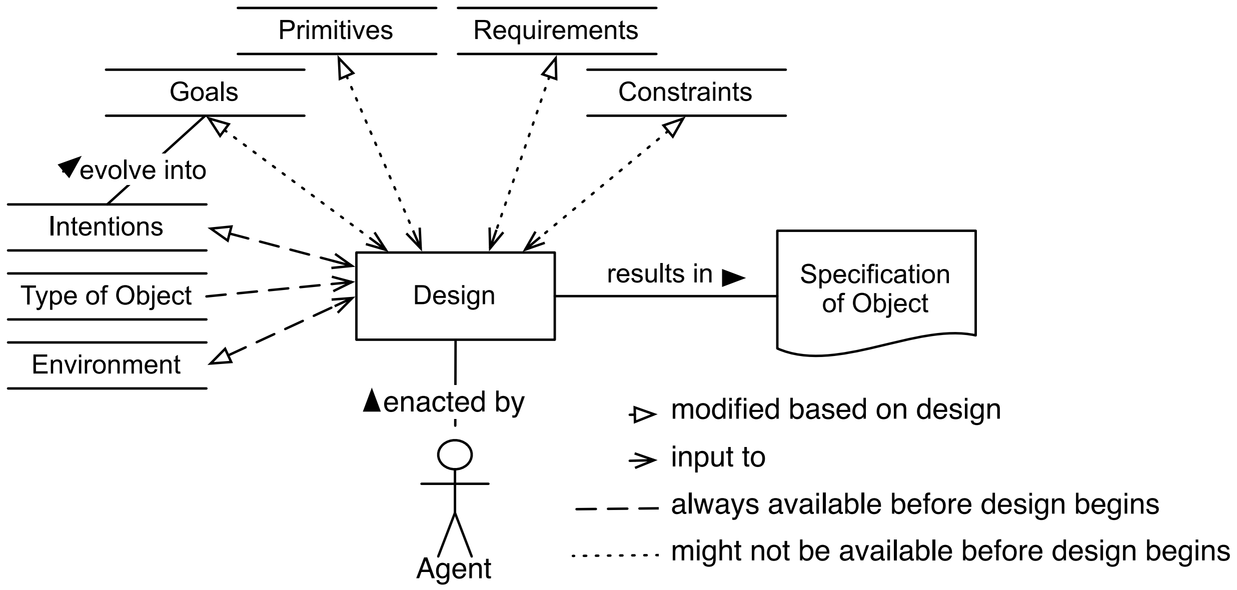


Fig. 3-2. Design Process Input/Output Model (from ch. 2)

3.2.4 Systems Development Method(ologie)s

For the purposes of this dissertation, a systems development method (SDM) is a collection of activities, practices, tools or other prescriptions, paired with an implicit or explicit claim that applying them to a design project will lead to better outcomes.

The Hardships of SDMs. The empirical study of SDMs is fraught with two primary difficulties. First, more than 1000 such methods exist (Jayaratna 1994), forming a conceptual labyrinth so bewildering that Avison and Fitzgerald called it the "methodology jungle" (1988). Iivari et al. (2000) attempted to structure the methodology jungle by providing a classification framework "for better understanding the intellectual core of methodologies and approaches and their interrelationships" (p. 180). However, the proliferation of methods still complicates efforts to understand their adoption, use and effects. Second, the effectiveness of an SDM depends on a myriad of factors including the project's context, how it is used and the degree of alignment between the SDM and the design team, making it extremely difficult to measure. Furthermore, pairwise comparisons of methods are plagued by debilitating confounds (see ch. 1). Moreover, the abundance of methods necessitates many such comparisons.

Relationships between SDMs and Design Paradigms. SDMs may be categorized according to the design paradigm that best represents their elements or underlying assumptions. Below, I illustrate this for several common SDMs (Table 3-3). I also provide an example of a possible development scenario outside the scope of each method, for reasons obvious below.

Code-and-fix. The code-and-fix model (cf., Boehm 1988) is perhaps the simplest software development method. In this method, the developer iterates between writing code and fixing code, where fixing code includes eliminating syntactic and logical errors and refactoring. This method is consistent with Reflection-in-Action as the designer iterates on the code and does not separate analysis from design. While coding and fixing code may be essential software development activities, this method does not accurately represent developers who spend much time planning.

Waterfall. The Waterfall Model (Royce 1970) is a label given to several methods that share a common core of activities including requirements elicitation, systems design, coding, implementation and maintenance. Ironically, the term "Waterfall Model" quickly came to refer to the no-backtracking version that Royce was criticizing rather than the more iterative model he was proposing. Regardless of the exact sequence prescribed, the Waterfall Model does not apply when developers engage in multiple activities (e.g. requirements analysis and implementation) simultaneously. By separating analysis from design and emphasizing linear progression, the Waterfall model is consistent with Technical Problem-Solving.

Table 3-3. Summary of Analysis of Software Development Methods

Model or Method	Key Reference	Synopsis	Example Context where method does not apply	Closest Paradigm
Code-and-fix	(Boehm 1988)	developer alternates between coding and fixing code	development driven by documents, e.g., requirements specification	Reflection-in-Action
Waterfall	(Royce 1970)	developer proceeds linearly: requirements → analysis → design → coding → testing → operations (sometimes with backtracking)	development involving simultaneous activities	Technical Problem-Solving
Spiral	(Boehm 1988)	developer iterates primarily among risk analysis, prototyping and planning	development ignoring risks	Technical Problem-Solving
SSM	(Checkland et al. 2006)	developer models the existing situation and uses models to guide a discussion about how to improve the situation	development that focuses on coding and does not involve models/diagrams	Reflection-in-Action
RUP and USP	(Jacobson et al. 1999; Kruchten 1998)	development proceeds in four phases: inception → elaboration → construction → transition. Each phase includes each of the key activities in the Waterfall Model and produces specific documents that mark progress.	development that focuses on coding and does not involve models, diagrams, use cases and other documents	Technical Problem-Solving
Extreme Programming	(Beck 2005)	a combination of particular values, principles and practices that guide development but do not prescribe a particular sequence of activities	development focusing on intermediary documents, plans and contracts	Reflection-in-Action
Scrum	(Schwaber and Beedle 2001)	a project management framework for agile development, focusing on work organization tactics	devolpment by individuals	Reflection-in-Action

Spiral. The spiral model combines many of the activities from the Waterfall Model with the iterative nature of the code-and-fix model and a predominant focus on risk mitigation (Boehm 1988). A team using the spiral model iterates between three basic activities – risk analysis, prototyping and planning, – with requirements analysis, design, testing and implementation interspersed between them, depending on the sophistication of the prototype. By separating analysis from design and emphasizing mathematical risk analysis, the Spiral model is consistent with Technical Problem-Solving. This method does not apply when developers ignore risks or cannot evaluate risks.

Soft Systems Methodology. "Soft Systems Methodology (SSM) is an organized way of tackling social situations perceived as problematical. It is action-oriented. It organizes thinking about such situations so that action to bring about improvement can be taken" (Checkland and Poulter 2006, p.xv). While SSM is not specific to software development, it may be applied in this context. The SSM practitioner makes models of purposeful activity as perceived by different people with different worldviews and uses them to structure discussion where desirable and feasible changes are identified. SSM does not apply when developers do not create models of participant perceptions of purposeful activity. SSM's iterative nature and focus on problem-understanding make it more consistent with Reflection-in-Action.

The Rational Unified Process (RUP) and Unified Software Process (USP). Some software development methods, such as RUP (Kruchten 1998) and USP (Jacobson et al. 1999), contain specific process models. The activities of these methods overlap with the steps of the Waterfall Model; however, the sequencing is more sophisticated with many activities occurring in parallel. RUP and USP are consistent with Technical Problem-Solving in their strong emphasis on planning and modeling, and their separation of design from coding. They do not apply when developers dive directly into coding without concerning themselves with planning or requirements.

Extreme Programming and Agile Methods. Extreme programming (Beck 2005) and other agile methods (Abrahamsson et al. 2002; Beck et al. 2001), promote a set of guiding values (e.g., simplicity), principles (e.g., accepting responsibility) and practices (e.g., pair programming). The software, and thereby the software design, are assumed to emerge from the actions of competent people employing these values, principles and practices. Agile methods do not prescribe a precise activity sequence (cf., Beck 2005) but they are generally consistent with Reflection-in-Action as they emphasize code over models and deemphasize planning. As agile methods reject plan-driven development, as embodied by RUP and USP, they do not apply to RUP- or USP-like development.

Scrum. Scrum (Schwaber and Beedle 2001) is a project management framework often used in conjunction with Extreme Programming and other agile methods. In Scrum, the "product owner" determines the project's direction, while the "Scrum Master" maintains the process. Small teams develop software in two to four week "sprints" assuming that the problem cannot be fully understood or defined – hence, Scrum is consistent with Reflection-in-Action. Scrum does not apply to plan-driven development.

Axiomatic Design (Suh 1990, 2001) is a collection of engineering design principles developed by Nam Suh. It assumes that "the world of design is made up of four domains: the customer domain, the functional domain, the physical domain and the process domain," and "all designs fit into these four domains" (Suh 1998, p. 204). The customer domain contains the customer's needs, which can be mapped into functional requirements in the functional domain, which in turn can be mapped into design parameters in the physical domain, and then process variables in the process domain. The residents of each domain can be organized hierarchically (Suh 1990).

Based on this worldview, Suh (1990) proposes two axioms, i.e., self-evident truths. "The independence axiom" states "Maintain the independence of functional requirements," while "the information axiom" states "Minimize the information content of the design." Numerous corollaries and theorems (in the mathematical sense of theorem) are derived from these two axioms (Strogatz 1994; Suh 1990). For example, Suh (1990) postulates that an ideal design has an equal number of design parameters and functional requirements. Suh (1995) extends these with more specific theorems for designing large flexible systems and organizations, for example, that the most efficient organizational design maximizes the organizations' ability to reconfigure its structure.

Axiomatic design is included here, as a method, because Suh's axioms are clearly prescriptive. Moreover, the theorems and corollaries are also prescriptive. In addition to the examples provided above, one corollary recommends having as few constraints and requirements as possible. While these prescriptions cannot inform a theory of software design due to the methods / theories disjunction discussed below, Suh's basic reasoning about requirements, goals and constraints may be useful. Axiomatic design is mostly consistent with technical problem solving, as evidenced by its waterfall-like process elements (cf., Suh 1998). It would not apply to most agile methods, code-and-fix or any non-technical, prototype-based approach to design that does not involve the specific formal matrix approach suggested.

Sample of Methods. Many other methods could be discussed, including Model Driven Architecture / Engineering (Schmidt 2006), the Fountain Model (Henderson-Sellers and Edwards 1993), the Hollywood Model (Gladden 1982), Feature Driven Development (Coad et al. 1999, ch. 6), Rapid Application Development (Martin 1991) and the Systems Development Life Cycle (Appendix A). However, the sample of methods discussed should sufficiently illustrate how SDMs fit with Technical Problem-Solving and Reflection-in-Action and the disjunction between methods and theories.

The SDM/Theory Disjunction. Methods of design purport to describe *good ways* of designing software; theories of design purport to describe *all ways* of designing software (cf., Gregor 2006). Consequentially, design methods *in general* are inappropriate foundations for design theories (Vermaas and Dorst 2007). Moreover, a comprehensive understanding of software development requires study of both effective and ineffective design practice so that antecedents of effectiveness may be identified. In addition, descriptive validity of methods may be challenged on the basis of identified instances of *amethodical* software development, discussed next. Therefore, SDMs have limited usefulness for understanding software design practice.

3.2.5 Amethodical Software Development

Many of the processes described in the previous section assume that developers can and do act in a methodical, if not rational, manner. Indeed, much design research is based on this assumption (Dorst and Dijkhuis 1995; Schön 1983; Truex et al. 2000). However, the many instances where designers produced software without using a method or acting methodically summarized below refute this assumption.

Many studies have suggested that SDMs are neither effectively nor extensively used (Avgerou and Cornford 1993; Bansler and Bødker 1993; Dobing and Parsons 2006; Whitley 1998). For example, in a study of “a large scale system development effort”, Zheng et al. (2007) found that “home-gown methods and ad hoc activities appear to dominate the day-to-day practices of systems development” (p. 1). Bansler & Bødker (1993) found that developers may claim to follow a method while practically ignoring it.

Similarly, Parnas and Clements (1986) argued that methodologies are “faked” and Nandhakumar and Avison (1999) argued that methodologies are used as “fiction” to make sense of actual practice. Turner (1987) found that similar methods applied in similar settings led to contrasting results. Baskerville et al. (1992) provided a possible explanation by demonstrating that organizations may change so quickly that long-term SDMs become ineffective. Furthermore, in an experimental study, Naur (1993) found that “the effectiveness of a particular technique in programming appears to be overwhelmingly dependent on the personality of the programmer using it … suggest[ing] that any methodology which imposes a particular style or manner of work on the programmers, at best may be taken to be useful to a part of the population of programmers, while any claim to general usefulness or applicability of a methodology is likely to be false” (p. 360-1). In field studies of expert designers, Schön found evidence indicating that designers do “not keep means and ends separate” or “separate thinking from doing” (1983, p. 69). Truex et al. (2000) summarized the argument by asking whether “methods [are] merely unattainable ideals and hypothetical ‘straw men’ that provide normative guidance to utopian development situations” (p. 53).

More fundamentally, Truex et al. (2000) argued that “the concept of method ... occupies an extremely privileged status in formal information systems development thought even though its origin is unstated” (p. 54) while “the possibility that *amethodical* development might be the normal way in which the building of these systems actually occurs in reality,” has “almost entirely elud[ed] the systems development literature” (p.58, emphasis added). “Amethodical systems building implies management and orchestration of systems development without a predefined sequence, control, rationality, or claims to universality. An amethodical development activity is so unique and unpredictable for each information systems requirement that even the criteria of contingent development methods are irrelevant” (Truex et al. 2000, p. 54). Baskerville et al. (1992) and (2004) found evidence of amethodical systems development in several case studies of software developers. The developers were led by practices and principles, similar to those of agile development; however, agile development “may be better described as “‘methodical-lite’ rather than amethodical” (Zheng et al. 2007, p. 2).

In summary, although the above evidence does not conclusively demonstrate (or even suggest) that *all* software design is amethodical, it certainly does call into question: 1) the model of Technical Rationality, 2) the “procedural rationality” at the heart of the Technical Problem-Solving paradigm; 3) the usefulness of methods (which represent methodical development) as foundations for understanding design. This evidence invalidates any *a priori* preference for Technical Problem-Solving (and its associated concepts) over Reflection-in-Action (and its associated concepts).

3.2.6 The Conscious Competence Learning Model

The Conscious Competence Learning Model[1] describes four stages in the development of professional competence in a skill area. In stage one, “the person is not aware of the existence or relevance of the skill area” or “they have a particular deficiency in the area concerned,” In stage two, “the person becomes aware of the existence and relevance of the skill … [and] their deficiency in this area.” In stage three, the person can perform the skill reliably and without assistance but only when concentrating. In stage four, the skill can be “performed while doing something else” and “the person might actually have difficulty in explaining exactly how they do it.” A fifth stage, “reflective competence”, may also exist, where a person can perform a skill with unconscious competence and then rationally reconstruct his actions to develop unconscious competence in others. A central claim of this model is that it “is not possible to jump stages.”

While one would not expect to find many software developers in stage 1 (since one would not become a developer while ignorant of or deficient in the profession, some developers may inhabit each of stages 2 through 5. Therefore, a thorough understanding of software development in practice requires examination not only of experts (stage 4) but also of students and amateurs (stage 2), junior professionals (stage 3) and gurus (stage 5).

[1] Chapman, Alan (2009) Conscious Competence Learning Model. Available: http://www.businessballs.com/consciouscompetencelearningmodel.htm. Retrieved 17 June 2010.

3.2.7 Summary of Pre-theoretical Contributions

This section began by elucidating two incompatible design paradigms – Technical Problem-Solving (Simon 1996) and Reflection-in-Action (Schön 1983). The former views design as rational search for a solution to a known problem; the latter views design as a reflective conversation between a designer and a situation, where problem understanding and solving are intermingled. Both paradigms distinguish between two domains – the problem space and the solution space. However, while Technical Problem-Solving is consistent with design as an activity following analysis and preceding coding, Reflection-in-Action assumes a much larger scope, consistent with the Design Process Input/Output Model, where design is the complete process from intentions to artifact.

Specific systems development methodologies can be classified according to which paradigm better represents their assumptions; however, some systems are not developed in a methodical way. This amethodical development is more consistent with Reflection-in-Action. Moreover, empirical findings on amethodical development favor Reflection-in-Action. Finally, the potentially broad range of professional competence implies that a holistic understanding of design must cover both amateurs and experts.

3.3 Literature Review - Theoretical Contributions

3.3.1 The Basic Design Cycle

Roozenburg and Eekels (1995, p. 88) argue "that design is in essence a trial-and-error process that consists of a sequence of empirical cycles, in which the knowledge of the problem as well as the solution increases spirally. We call the model of this cycle 'The Basic Design Cycle'" (see Figure 3-3). They explain that The Basic Design Cycle "is a specific implementation of the more general empirical cycle" (p. 88). The cycle begins with a known problem. The first step is to analyze this problem, determine the "technical[,] ... psychological, social, economic and cultural functions that a product should fulfill", not in detail but in "broad statements" (p. 90). This produces criteria through which design candidates may be evaluated. The second step is to synthesize a provisional design, which Roozenburg and Eekels describe as “the moment of externalization and description of an idea, in whatever form (verbally, sketch, drawing, model, etc.)” (p. 91). Next, “simulation is forming an image of the behaviour and properties of the designed product by reasoning and/or testing models, preceding the actual manufacturing and use of the product.... Simulation leads to ‘expectations’ about the actual properties of the new product, in the form of conditional predictions” (p. 91). The next step is evaluation – “establishing the ‘value’ or ‘quality’ of the provisional design” by comparing “the expected properties ... with the desired properties in the design specification” (p. 92.). Last is “the decision: continue - that is to say, elaborate the design proposal or, if it is the final design, manufacture it - or try again and generate a better design proposal” (p. 92).

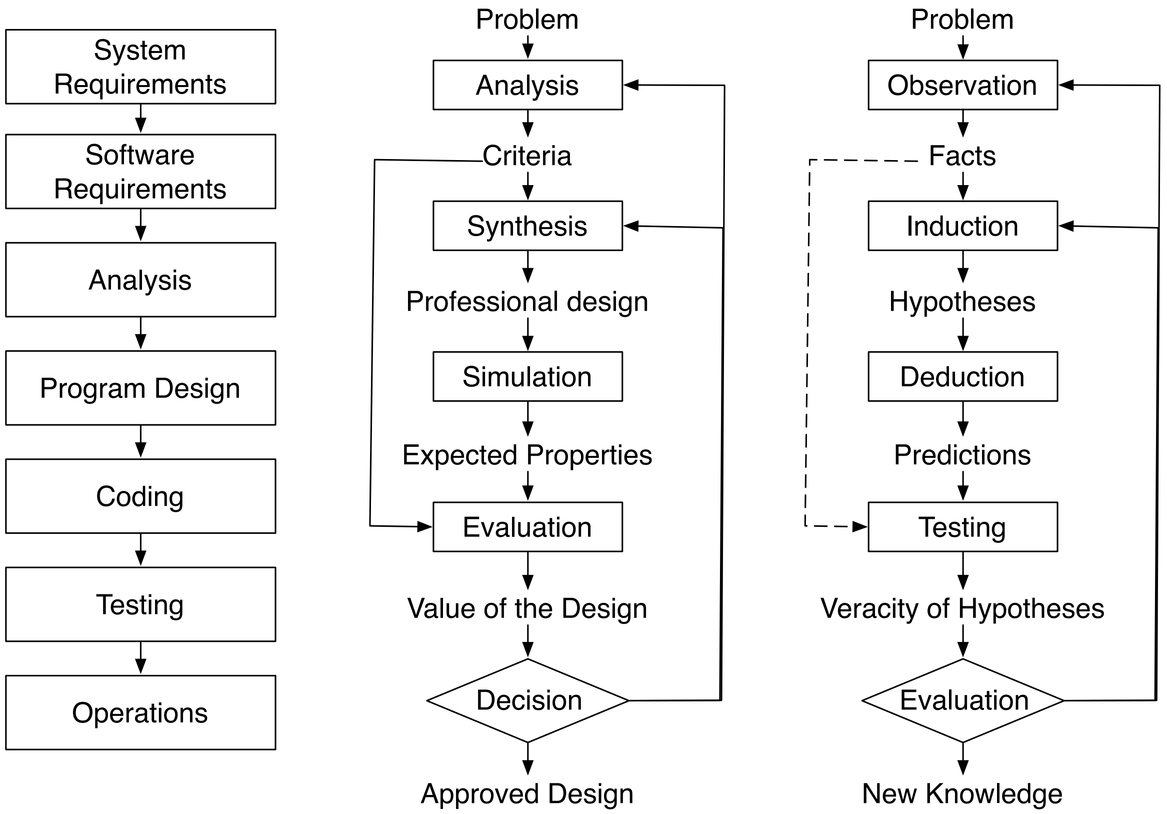


Fig. 3-3. The Basic Design Cycle (centre) with the Waterfall Model (left) and the Basic Cycle of Scientific Inquiry (right), (adapted from Roozenburg and Eekels 1995)

Limitations. The primary limitation of The Basic Design Cycle is evident in its similarity to the Waterfall Model (with backtracking). Like the Waterfall Model, it does not apply to situations where the designer begins with inspiration and moves immediately to building the design object.

3.3.2 The Problem-Design Exploration Model

Maher et al. (1995) suggest a formal model of exploration intended to describe how an artifact (not necessarily software) may be designed by genetic algorithms (Figure 3-4). Design is modeled as two interacting evolutionary systems – the problem space *P* and the solution space *S*. At time *t*, *P(t)* contains the design goal and associated requirements; while *S(t)* "defines the current search space for design solutions" (p. 5). These two systems interact over time through three different processes. First, *S(t)* is generated by searching for an artifact that satisfies the design goal represented by *P(t)* – the Focus, Fitness process. Second, *S(t)* "prompts new requirements for *P(t+1)* which were not in the original problem space, *P(t)*" (p. 5) – "refocusing" the problem space. Third, both the problem space and solution space evolve over time.

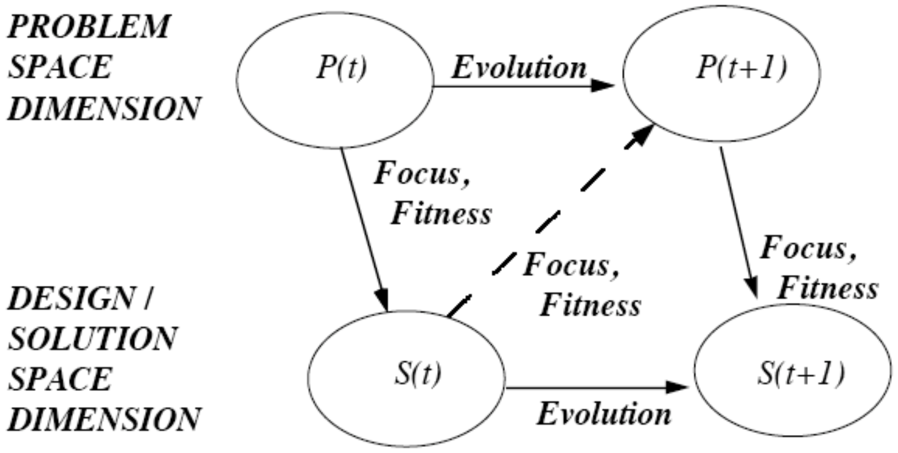


Fig. 3-4. Problem-Design Exploration Model (adapted from Maher et al. 1995)

Notes: P(t) = problem at time t; S(t) = situation at a time t; dashed line indicates situation refocusing problem; diagonal downward movement indicates a search process.

Limitations. The primary limitation of the Problem-Design Exploration Model is that it applies specifically to design using evolutionary algorithms. However, Maher et al.'s principle result (co-evolution) has been supported by a protocol study of industrial designers (Dorst and Cross 2001) and a similar study of software designers using object-oriented methods (Purao et al. 2002). Therefore, the principle of representing design as two co-evolving systems may inform general design theories.

3.3.3 Alexander's Design Processes

Alexander (1964) differentiates *form* (the object being designed) from *context* (the object's environment) and argues that a design's quality results from the fit between its form and its context. He then suggests three "possible kinds of design process" (p. 75) (Figure 3-5).

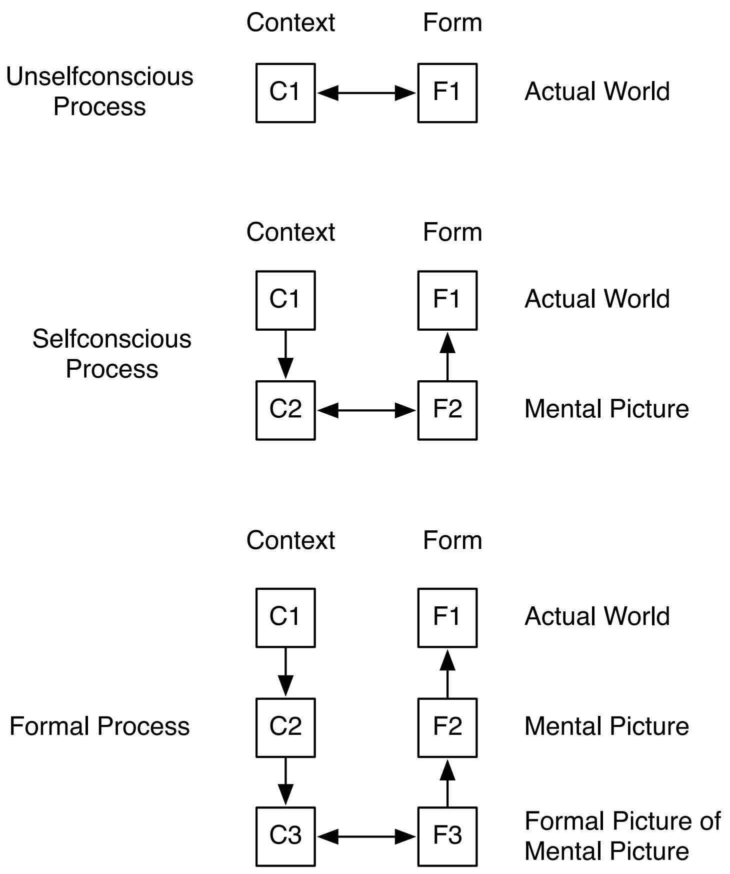


Fig. 3-5. Alexander's Design Processes (adapted from Alexander 1964)

Note: arrows indicate interactions (not sequence); Alexander does not specify the exact nature of these interactions.

In the “Unselfconscious Process”, the designer directly manipulates the design object and other items in the external world to eliminate misfits between form and context. In this case, the designer “is unlikely to impose any “designed” conception on the form” (p. 77). For example, an igloo dweller may directly manipulate the igloo’s structure to respond to temperature changes – creating vents when the temperature rises and eliminating them when the temperature falls.

In the “Selfconscious Process”, the “design process is remote from the ensemble itself; form is shaped not by interaction between the actual context's demands and the actual inadequacies of the form, but by a conceptual interaction between the conceptual picture of the context which the designer has learned and invented, on the one hand, and ideas and diagrams and drawings which stand for forms, on the other” (p. 75). For instance, a person building a new deck may observe the house and land, forming a mental conception of the situation, and then draw a rough sketch of the desired deck. He may refine this sketch several times, while comparing it to his conception of the situation, before eventually building the deck based on the sketch. Alexander suggests that the exact nature of the interaction between mental pictures of form and context is unclear.

In the third (unnamed) process, the designer creates a formal model of the mental pictures. Then, “the design F2 is preceded by an orderly complex of diagrams F3” (p.78), which constitute a formal model of the form. Alexander argues that both formal models may be represented using set theory – the problem is defined by the set of potential misfits and the solution is defined by the set of its properties. The designer may then decompose the problem into cohesive subproblems using its set-theoretical description and solve each by exploring a “hunch”, using a constructive diagram of a potential form in its context. The solutions of the subproblems may then be combined to solve the original problem.

Limitations of the Selfconscious Process. Two principal limitations are common to all of Alexander’s Processes. First, the concepts and relationships are not well-defined; indeed, Alexander explicitly expresses that the relationship between the two mental pictures is unclear. Second, concerning software design specifically, several important concepts (e.g., goals, testing) are missing.

3.3.4 The Function-Behavior-Structure Framework

Gero (1990) presented an engineering design meta-process, the Function-Behavior-Structure Framework (FBS), whose core claim is that "the purpose of designing is to transform *function*, F (where F is a set), into a *design description*, D, in such a way that the artefact being described is capable of producing those functions" (Gero 1990, p. 2, original italics). Gero further posits three intermediate artifacts – structure, predicted behavior of the structure, and expected (desired) behavior of the structure. The designer figuratively walks a path from function through behavior and structure to design description; FBS specifies the possible paths.

Vermaas and Dorst (2007) point out "that the conceptual framework underlying the FBS-model is unstable" (p. 143); more specifically, the definitions of function, behavior and structure have changed (cf. Gero 1990; Gero and Kannengiesser 2004; Gero and Kannengiesser 2007; Rosenman and Gero 1998). Vermaas and Dorst also suggest that these definitions are likely to change again as "Gero and collaborators are uncertain about making a distinction between the concepts of purpose and function" (p. 143) and as function has a pre-established meaning in other sciences that is inconsistent with Gero's definition. Moreover, FBS has been analyzed and re-interpreted by authors other than its originator (e.g., Galle 2009). Its continuing evolution impedes a definitive account; therefore, I summarize its evolution to capture its primary concepts, relationships and disagreements.

Evolution of the FBS Framework. In Gero's (1990) original account (Figure 3-6), FBS had five artifacts (Table 3-4) and six operations (Table 3-5). Although *synthesis* was not shown in the original model, it was discussed in the original paper and is shown in later versions (below).

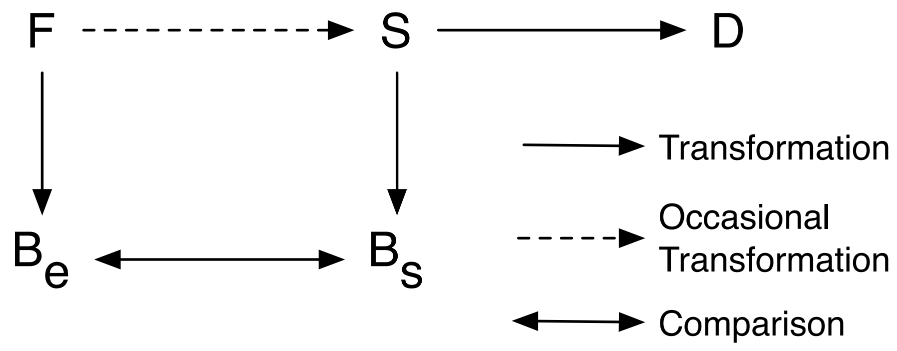


Fig. 3-6. Original FBS Framework (adapted from Gero 1990)

Table 3-4. Artifacts of the FBS Framework (adapted from Gero 1990)

Symbol	Meaning
Be	expected (desired) behavior of the structure
Bs	"the predicted behavior of the structure" (p. 3)
D	a graphically, numerically and/or textually represented model that transfers "sufficient information about the designed artefact so that it can be manufactured, fabricated or constructed" (p. 2)
F	"the expectations of the purposes of the resulting artefact" (p. 2)
S	"the artefact's elements and their relationships" (p. 2)

Table 3-5. Operations of the FBS Framework (adapted from Gero 1990)

Operation	Inputs	Outputs	Meaning
Analysis	S	Bs	the process of deriving the behavior of a structure
Catalog Lookup	F	S	selecting a known structure that performs the required function
Evaluation	Bs & Be	Differences Between Bs and Be	comparing predicted behavior to expected behavior and determining whether the structure is capable of producing the functions
Formulation	F	Be	deriving expected (desired) behaviors from the set of functions
Production of Design Documentation	S	D	transforming structure into design description suitable for manufacturing
Synthesis	Be	S & Bs	"expected behavior is used in the selection and combination of structure based on a knowledge of the behaviors produced by that structure" (p. 3)

To illustrate, the designer may begin with a desired function (carry a minimum of 2000 vehicles across the water from City X to City Y each day). First, the designer looks for a existing solution (catalog lookup). If one is not found, the designer transforms the function into a set of expected behaviors (object floats, object moves at up to 20 knots). The designer then selects and combines structures to perform the function (engines, fuel lines, navigation systems). During this synthesis step, the designer does not combine tangible objects in the physical world, rather, symbolic representations thereof. The next step is to predict how the structure will behave, perhaps by running a computer simulation. The designer than compares the predicted behaviors of the structure to its expected behaviors and, if they are sufficiently similar, generates a detailed description of the structure for manufacturing. If the predicted behaviors do not match the expected behaviors, the designer engages in a cycle of synthesis, analysis and evaluation until they do. The design description generated, the engineering process is complete and the tangible object may be fabricated.

Gero (1990) also briefly discussed *reformulation* (modifying functions and expected behavior based on changes in structure). Three types of reformulation were identified by Gero et al. (2004) – functional, behavioral and structural (Table 3-6). Figure 3-7 shows FBS updated with the *synthesis* transformation and three reformulation feedback loops.

Table 3-6. Feedback Loops of the FBS Framework (adapted from Gero and Kannengiesser 2004)

Operation	**Inputs**	**Outputs**	**Meaning**
Structural Reformulation	S, Bs	S	modifying the structure based on the structure and its predicted behaviors
Behavioral Reformulation	S, Bs & Be	Be	modifying the expected behaviors based on the structure and its predicted behaviors
Functional Reformulation	S, Bs & F	F,	modifying the set of functions based on the structure and its predicted behaviors

The Situated FBS Framework. Gero and Kannengiesser (2004) updated FBS to include the idea of *situatedness* ("the agent's view of a world changes depending on what the agent does" p. 90), leading to distinction between three *worlds*.

> *The external world is the world that is composed of representations outside the designer or design agent. The interpreted world is the world that is built up inside the designer or design agent in terms of sensory experiences, percepts and concepts. It is the internal representation*

of that part of the external world that the designer interacts with. The expected world is the world imagined actions will produce (p. 93).

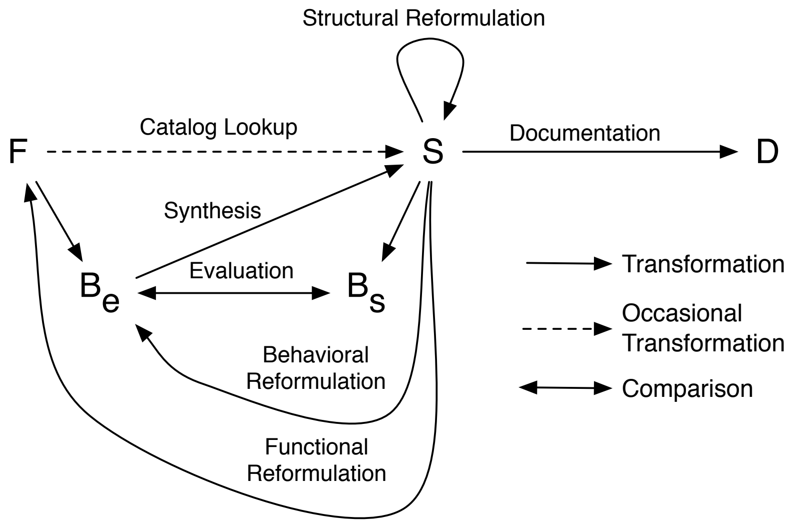


Fig. 3-7. The Function-Behavior-Structure Framework (adapted from Kruchten 2005)

Gero and Kannengiesser (2004) argued that function, behavior and structure have representations in each world and mapped the eight transformations and comparisons (now better thought of as processes) of FBS into 20 activities across the three worlds. The three-world metaphor, nine representations, eight processes and 20 activities comprise the *Situated FBS Framework* (Figure 3-8). However, since the notation and definitions assigned to the elements of the framework change in Gero and Kannengiesser (2007) and are not abundantly clear in either paper, I provide (next) an informal summary rather than formal definitions.

"The framework represents [given] external requirements related to the function (FR^e), behaviour (BR^e) and structure (SR^e)" of the design problem (Gero and Kannengiesser 2007, p. 15). S^i is the design agent's conceptualization of the "design solution in terms of a point in the structure state space" (p. 16). B^i and F^i are the design agent's conceptualizations of the how S^i will behave and the functions it will be capable of producing, were it constructed.

Se^i, Be^i and Fe^i represent the design agent's expectations about the structure, behavior and function of the design solution, respectively. B^i and F^i differ temporally from Be^i and Fe^i – the latter represent the designer's expectations before any form of testing, simulation or predictive analysis, whereas the former result from testing, simulation or predictive analysis on the design solution. Therefore, expectations may change when B^i and F^i are compared to Be^i and Fe^i.

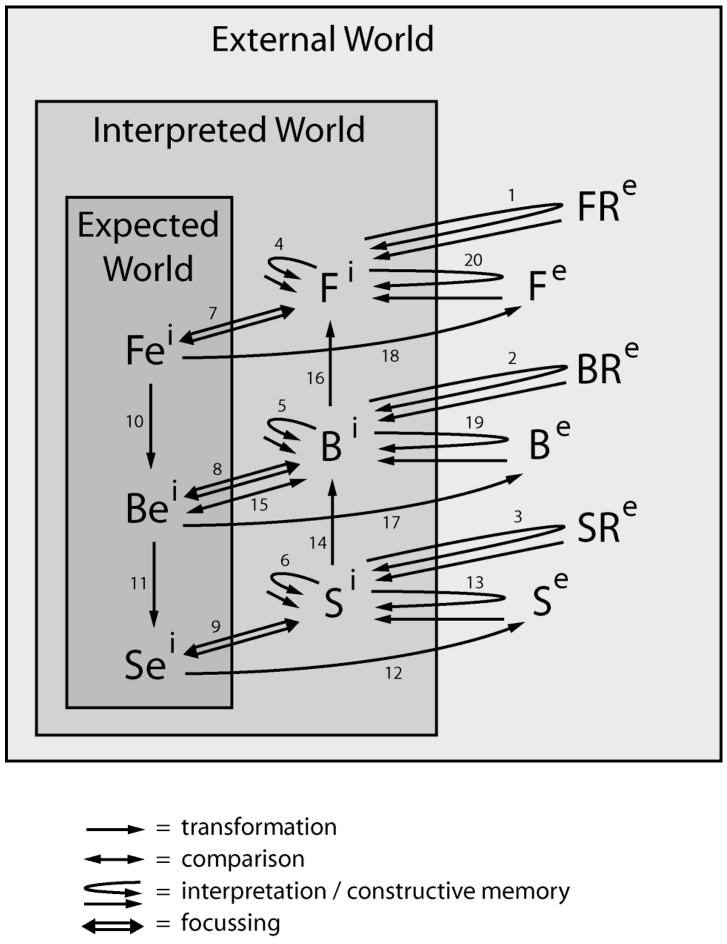


Fig. 3-8. The Situated FBS Framework (after Gero and Kannengiesser 2007)

"The structure (S) of most objects can be described in terms of geometry, topology and material" (Gero and Kannengiesser 2007, p. 2). The design agent manifests S^e in the external world 'by sketching or some similar process" (p. 383). Hence, S^e is an abstract representation of a point in the structure state space, *not* the designed object. As S^e is a representation, it is not capable of behavior or function; therefore, B^e and F^e are predictions rather than observations (as results of simulations are predictions). Gero and Kannengiesser (2004) explain that B^e and F^e comprise the design documentation (along with S^e) and may serve as inputs for behavioral and functional reformulations, respectively. B^e and F^e are produced by the documentation process (see below), not the evaluation process, further supporting the point that S^e is an abstraction. S^i and Se^i are the designer's interpretations of structure and expected structure, respectively (see below).

The activities in Figure 3-8 map into the processes in Figure 3-7 (Gero and Kannengiesser 2004) (Table 3-7). The twenty activities are neither named nor individually discussed in Gero and Kannengiesser (2004,2007). Moreover, "the numbering of the eight design steps and the 20 activities in the situated FBS Framework does not prescribe a fixed order of execution" (Gero and Kannengiesser 2007, p. 20).

Table 3-7. Meaning of Situated FBS Framework Processes (after Gero and Kannengiesser 2007)

Process	**Meaning**	**Activities**
Formulation	deriving expected (desired) behaviors from the set of functions	1 through 10
Synthesis	"expected behavior is used in the selection and combination of structure based on a knowledge of the behaviors produced by that structure" (p. 3)	11, 12
Analysis	the process of deriving the behavior of a structure	13, 14
Evaluation	comparing predicted behavior to expected behavior and determining whether the structure is capable of producing the functions	15
Documentation	transforming the structure into a design description that is suitable for manufacturing	12, 17, 18
Structural Reformulation	modifying the structure based on the structure and its predicted behaviors	9 (possibly driven by 3, 6 or 13)
Behavioral Reformulation	modifying the expected behaviors based on the structure and its predicted behaviors	8 (possibly driven by 2, 5, 14 or 19)
Functional Reformulation	modifying the set of functions based on the structure and its predicted behaviors	7 (possibly driven by 1, 4, 16 or 20)

Galle's Interpretations and Revisions. Galle (2009) pointed out that FBS implies that the design artifact exists before it has been designed (if *structure* is defined as "the artefact's elements and their relationships" (Gero 1990, p. 2), structure is a property of an object; hence, the design object must be present for structure to exist). Yet, if the design object is complete, so must be the design, leading to a contradiction. Galle then suggested two interpretations of the framework, and two corresponding modifications to resolve the difficulty.

In the *nominalist* interpretation, F, B and S “stand for function-descriptions (disposition-descriptions), behaviour-descriptions, and structure-descriptions” (Galle 2009, p. 7). Therefore, F, B and S can exist independently of any particular artifact, and FBS “becomes a model of a process of *symbol manipulation*” (Galle 2009, p. 7, original italics). Galle provides several quotations from the FBS-Framework’s originators and proponents consistent with this description view, e.g., “Gero and Kannengiesser (2004, p. 374) state that ‘the basis for Gero’s FBS framework is formed by three classes of variables describing different aspects of a design object’, namely ‘Function (F) variables’, ‘Behaviour (B) variables’, and ‘Structure (S) variables’” (p. 8). However, Galle argues, defining F, B and S as descriptions makes the “production of design description” step superfluous. Considering these arguments, Galle recommends a “nominalist modification” where F, B and S are explicitly defined as descriptions and the documentation step is eliminated. This seems consistent with the presentation of the Situated FBS Framework, summarized above, in two ways: 1) the documentation step in the Situated FBS Framework is simply the generation of F, B and S in the external world; 2) the external world is “the world that is composed of *representations* outside the designer,” (Gero and Kannengiesser 2004, p. 93, italics added), from the F, B and S in the designer’s mind; hence, F, B and S are representations.

In the realist interpretation, F, B and S are interpreted as “refer to *functions*, *behaviours* and *structures* that are regarded as *entities in their own right* even in the absence of, and prior to the making of, any artefact to ‘have’ them” (Galle 2009, p. 8). Here, F, B and S are real in the same philosophical sense as numbers and constructs. While considering functions and behaviors as abstract dispositions seems plausible, Galle concludes that “the FBS model as currently known is not amenable to the remedy of conceiving of artefact-structure as an entity in its own right” (p. 9). To overcome this limitation, Galle proposes redefining structure as a mapping of a set of materials into a 3D Euclidean space, and makes “a sharp distinction … between a structure, and material objects embodying that structure” (p. 12). Galle does not explain how these revisions translate into FBS transformations.

Summary of FBS Framework Evolution. In summary, FBS was originally introduced by Gero (1990) and later updated to include the concept of "situatedness" by Gero and Kannengiesser (2004). Vermaas and Dorst (2007) critiqued the framework and argued for a new definition of function; meanwhile Gero and Kannengiesser (2007) attempted to clarify the framework's underlying ontology. Most recently, Galle (2009) criticized the framework for referring to artifacts before they existed and proposed two modification strategies to overcome this problem.

Limitations. Overall, FBS is a well-defined, thoroughly thought-out, specific theory. Its primary limitation is its lack of empirical validation. Beyond this, several criticisms are possible.

1. It is not clearly explained how S^i differs from Se^i in the Situated FBS Framework (Gero and Kannengiesser 2004). Specifically, it is not clear how the design agent's expectations about structure differ from its interpretations of the structure. (Gero may be suggesting that a design agent may engage in a cycle of forming expectations, externalizing them as a cognitive aid and then reinterpreting the expectations based on the aid, but this is not explained.)
2. As above, definitions of core concepts continue to change.
3. FBS is both "a descriptive model aimed at describing actual designing and … a prescriptive model aimed at improving designing" (Vermaas and Dorst 2007, p. 133); i.e., some confusion exists concerning whether FBS is a theory or a method (§3.3.5).

3.3.5 Analysis of Theoretical Contributions

Theory Type. A process theory is an explanation of how and why an entity changes and develops (Appendix C). Van de Ven and Poole (1995) classified organizational process theories into four categories – lifecycle, dialectic, evolutionary and teleological (Table 3-8). None of the contributions presented in this section are *called* process theories; however, here I discuss whether each fits the definition. A description qualifies as a process theory if it meets the following three criteria (Van de Ven and Poole 1995).

1. It posits a formative relationship between a higher level phenomena (e.g., design) and several lower-level phenomena (e.g., constructing a model, identifying expected behaviors).
2. It includes a causal motor (dialectic, evolutionary, lifecycle, teleological).
3. It includes a claim to universality within a domain (e.g., the Problem Design Exploration Model is supposed to explain all design (universality) using evolutionary algorithms (domain)).

Table 3-8. Analysis of Types of Process Theories (Van de Ven and Poole 1995)

Type of Process Theory	Dialectic	Evolutionary	Lifecycle	Teleological
Proponents	(Plato; Hegel; Van de Ven et al. 1995)	(Darwin; Van de Ven et al. 1995)	(Markus and Robey 1988; Van de Ven and Poole 1995)	(Churchman 1971; Singer 1959; Van de Ven and Poole 1995)
Capsule Description	Changes result from shifts in power among conflicting entities	A population of entities changes as less fit entities expire and remaining entities change and recombine	An entity progresses through a series of stages in a predefined sequence	An agent purposefully selects and takes actions to achieve a goal
Event Progression	Recurrent, discontinuous sequence of conflict and resolution	Recurrent, cumulative and probabilistic sequence of variation, selection and retention	Linear & irreversible sequence of prescribed stages	Recurrent, agent-determined sequence of goal setting and action taking
Contemporary Example	Behavioral Negotiation Theory (Neal and Northcraft 1991)	Change in populations of organizations (Carroll and Hannan 1989)	The Organizational Lifecycle (Kimberly and Miles 1980)	Organizational decision making (March and Simon 1958)

The Basic Design Cycle clearly relates design to lower-level activities (synthesis, simulation, evaluation). Of the ideal types of process theory identified by Van de Ven and Poole (1995), this is closest to a lifecycle process theory insofar as it follows a pre-figured sequence of phases where the design characteristic accumulate across phases. Roozenburg and Eekels “consider The Basic Design Cycle the most fundamental model of designing” and conjecture that “someone who claims to have solved a design problem has gone through this cycle at least once” (p. 89). Therefore it is a process theory.

The Problem-Design Exploration Model relates design to lower-level activities (focus, fitness). Of the ideal types of process theory identified by Van de Ven and Poole (1995), this is clearly an evolutionary process theory – it explicitly explains design by evolution. Specifically, variation is represented by the Evolution process, while selection and retention are represented by the Focus/Fitness process. The

Problem-Design Exploration Model purports to explains all design by evolutionary algorithms. Therefore, it is a process theory.

Each of Alexander's three design processes relate design to one or more unnamed lower-level activities. However, Alexander explicitly *proposed* the third process as a reaction to the limitations of the Selfconscious Process – therefore, the third process is a method, not a process theory. However, both the Unselfconscious and Selfconscious Processes imply teleological motors – they describe processes engaged in by a designer (the agent) seeking to eliminate misfits (the goal) with no explicit restriction on the sequence of activities. Moreover, Alexander claimed that the Unselfconscious Process describes the design process of ancient cultures and the Selfconscious Process describes the then status quo in architecture. Therefore, Alexander's first two "design processes" are process theories.

FBS relates design to lower-level activities including synthesis, analysis and reformulation. Since it describes processes executed by an agent, which can choose the order of execution, FBS has a teleological motor. Gero and Kannengiesser (2004) claimed that "the eight processes depicted in the FBS Framework are … fundamental for all designing" (p. 90). Gero and Kannengiesser (2007) repeated the claim that the these processes are the "eight fundamental steps in designing" (p. 15). Therefore, it is a process theory.

Application to Software. Nothing in the The Basic Design Cycle appears to preclude its application to software. Indeed, if software development practice reflects the Technical Problem-Solving paradigm, The Basic Design Cycle may provide a reasonable, high-level view of the process. In contrast, the Problem-Design Exploration Model would apply only where the software were designed by genetic algorithm.

Alexander's Unselfconscious Process describes design of tangible objects in a physical context. As software is an intangible artifact and exists in a virtual context, the Unselfconscious Process does not apply to software design. In contrast, Alexander's Selfconscious Process separates conceptualizing of an artifact from its manifestation in reality. The transition from context to mental picture of context appears consistent with the ideas of systems analysis and requirements engineering. The transition from mental picture of form to form itself appears consistent with the act of coding. Furthermore interaction between mental pictures of context and form appears consistent with the act of devising the software's structure from goals. Therefore, the Selfconscious Process appears applicable to software design.

FBS is primarily intended for material objects, as revealed by the definition of structure: "*Structure* (S) of an object is defined as its components and their relationships, i.e. "what the object consists of". The structure (S) of most objects can be described in terms of geometry, topology and material" (Gero and Kannengiesser 2007, p. 2, original italics). Therefore, applying it to software requires some degree of interpretation or adaptation. At least two interpretations are possible: one provided by Kruchten (2005), and one more consistent with Galle's nominalist interpretation.

Kruchten (2005) attempted to "cast" software design in FBS by mapping the artifacts and processes of the Rational Unified Process (Kruchten 2003) and the Waterfall Model into FBS. In doing so, he mapped software code into S, and redefines documentation from the act that "produces the design description (D) for constructing or manufacturing" (p. 54) to "completing the bill of materials and preparing a master CD-ROM with the installers, binaries, data, help files, and other elements to be reproduced and delivered" (p. 55). This is a significant adaptation in the sense that, while both physical (e.g., robotic submarines) and software artifacts (e.g., web browsers) are created and deployed, the original FBS Framework includes neither creation nor deployment – "The result of the activity of designing is a design description. This design description generally is represented graphically, numerically, and/or textually. The purpose of such a description is to transfer sufficient information about the designed artefact so that it can be manufactured, fabricated or constructed" (Gero 1990, p. 2). Similarly, the situated FBS-Framework ends with artifacts in the external world, "the world that is composed of *representations* outside the designer or design agent" (Gero and Kannengiesser 2004, p. 377, italics added), not physical artifacts. In summary, to make FBS more amenable to the software domain, Kruchten (2005) moves the object being designed and the processes of creation and deployment into FBS. Henceforth I refer to this as the "consolidated interpretation" of FBS.

Alternatively, one can adopt a perspective more consistent with Galle's (2009) nominalist interpretation by assuming the F, B and S, are descriptions. These can then be mapped into common software design arftifacts: function-description include software goals; structure-description includes models of the software's architecture; behavior-description includes the software requirements (Be) and predicted behavior based on inspection or simulation of the structure-description (Bs). Hence, the FBS-Framework remains a model of a process of symbol manipulation and creation and deployment of the design object remain outside of its scope. In keeping with Galle's (2009), I refer to this as the "nominalist interpretation" of FBS.

Design Paradigm Affiliation. As The Basic Design Cycle assumes a given problem, emphasizes planning and separates analysis from synthesis, it is more compatible with Technical Problem-Solving. Roozenburg and Eekels state explicitly that "the problem-solving model of systems engineering" is "quite close to The Basic Design Cycle" (p. 87). Moreover, they refer repeatedly to "optimising" and equate The Basic Design Cycle with the basic cycle of empirical scientific inquiry (positivism), clearly aligning The Basic Design Cycle with Technical Problem-Solving.

The Problem-Design Exploration Model assumes a bounded problem space and is built on a search metaphor. This suggests compatibility with Technical Problem-Solving. However, co-evolution of problem and solution spaces superficially suggests a constructivist epistemology, consistent with Reflection-in-Action. On closer examination, however, the problem space here refers to the system requirements, rather than the physical or virtual environment the design artifact is intended to inhabit. Therefore, this model posits co-evolving sets of requirements and design features (consistent with Technical Problem-Solving), rather than the coevolution of actual context and design artifact (consistent with Reflection-in-Action). Consequently, The Problem-Design Exploration Model is more consistent with Technical Problem-Solving.

In Alexander's Unselfconscious and Selfconscious Processes, the designer considers an unbounded problem, acts on hunches, and simultaneously alters (or considers) the design object and context (or mental pictures thereof), making them more compatible with Reflection-in-Action.

FBS's design paradigm affiliation depends on the interpretation applied. Under the nominalist interpretation, FBS appears closer to Technical Problem-Solving than to Reflection-in-Action for at least two reasons. First, Gero and Kannengiesser (2004) describe the solution structure as a point in a structure state space, composed of a definitive number of relevant structure variables. This indicates that FBS applies to bounded problems. Second, the Technical Problem-Solving idea of an agent using scientific methods and techniques is exemplified by the prediction of behavior from a structure-description. The analysis process involves "deriving" behavior from structure, a term connoting a scientific approach rather than mere guessing. Third, being "a model of a process of symbol manipulation" (Galle 2009, p. 7); FBS's guiding metaphor is information processing, as in Technical Problem-Solving.

Under the consolidated interpretation, the situation is less clear. Here, FBS assumes a given problem (F), which is consistent with Technical Problem Solving. However, the designer iterates on the design object itself, rather than an abstraction, forcing the design process out of the designer's cognitive system. Meanwhile, analysis and testing of the design object, including post-deployment testing in a real-world context make the process at least partially constructivist, which is more consistent with Reflection-in-Action.

Summary of Analysis. In summary, both Technical Problem-Solving and Reflection-in-Action are associated with a teleological process theory applicable to software (Table 3-9); however, the concepts and relationships of the Selfconscious Process are not formally defined.

Table 3-9. Comparison of Design Process Theories

Theory	Proponent	Ideal Type	Paradigm	Applicable to Software?
The Basic Design Cycle	(Roozenburg et al. 1995)	Lifecycle	Technical Problem-Solving	Yes
Problem-Design Exploration Model	(Maher et al. 1995)	Evolutionary	Technical Problem-Solving	Partially*
Unselfconscious Process	(Alexander 1964)	Teleological	Reflection-in-Action	No
Selfconscious Process	(Alexander 1964)	Teleological	Reflection-in-Action	Yes
(Situated) FBS Framework	(Gero 1990; Gero et al. 2004)	Teleological	Depends on interpretation	Yes

**Note: The Problem Design Exploration Model only applies to software designed using genetic algorithms.*

3.4 Reason- and Action-Centric Perspectives

3.4.1 Defining the Two Perspectives

The preceding sections presented two incompatible design paradigms and classified a wide swath of the design concepts, methods and theories according to these paradigms. Here I identify core beliefs underlying this classification and enumerate the elements compatible with each perspective.

To review, the Technical Problem-Solving paradigm intuitively appears compatible with positivism, the cognitivist (plan-centric) theory of human action, a designer-as-information-processor metaphor, The Basic Design Cycle, the Problem-Design Exploration Model, FBS (nominalist interpretation), plan-driven SDMs including the Waterfall Model, and a narrow interpretation of the scope of design. These concepts are all “reason-centric” – they assume a logical, (boundedly-)rational designer, methodically creating or refining a design object according to a pre-programmed frame. More generally, the Reason-Centric Perspective (RCP) holds that design is a cognitive phenomenon – it occurs primarily within the designer’s cognitive system. (This does not preclude externalizing cognition using diagrams or other boundary objects.)

In contrast, Reflection-in-Action intuitively appears compatible with constructivism, the ethnomethodological (improvisation-centric) view of human action, a design-as-creativity metaphor, Alexander's process theories, agile SDMs, amethodical development and a broad interpretation of the scope of design. These concepts are all "action-centric" – they focus on the designer's actions, not thoughts. More generally, the Action-Centric Perspective (ACP) holds that design is an emergent phenomenon comprising continuous interactions between designers and their environment. "Emergence … refers to the arising of novel and coherent structures, patterns and properties during the process of self-organization in complex systems" (Goldstein 1999).

Table 3-10 summarizes RCP and ACP – they may be defined as follows.

> ***Reason-Centric Perspective (RCP):*** *1) The belief that design is a cognitive phenomenon; 2) the collection of paradigms, models, methodics and theories associated with this belief.*

> ***Action-Centric Perspective (ACP):*** *1) The belief that design is an emergent phenomenon; 2) the collection of paradigms, models, methodics and theories associated with this belief.*

I posit that the conceptual tension between RCP and ACP manifests in real-world design projects, design education and design science. For example:

- Beck (2005) discussed common tensions between managers attempting to drive projects through cost estimates and developers unable to make reliable estimates. The developers cannot provide accurate estimates because they neither conceptualize their work through detailed plans, nor have sufficient information about the problem to accurately estimate its solution's difficulty.
- Graham (2003) explained misalignment between programming education and practice – "I was taught in college that one ought to figure out a program completely on paper before even going near a computer. I found that I did not program this way.... I tended to just spew out code that was hopelessly broken, and gradually beat it into shape."
- Truex et al. (2000) identified the tension between the largely nonempirical literature on SDMs, which takes for granted that design is inherently methodical, and empirical literature on amethodical development, which consistently finds that design is not methodical.

Table 3-10. Summary of Reason- and Action-Centric Perspectives

Dimension	Reason-Centric Perspective	Action-Centric Perspective
Design Paradigm	Technical Problem-Solving	Reflection-in-Action
Primary Proponent	Simon (1996)	Schön (1983)
Epistemology	Positivist	Constructivist
Theory of Action	Cognitivist	Ethnomethodological
Designer	information processor / rational agent	person constructing his or her reality
Design Problem	given evaluation criteria for alternatives, which are representable as points in a problem space	essentially unique and poorly understood at the start
Design Knowledge	knowledge of design procedures and scientific laws	artistry of design: when to apply which procedure / piece of knowledge
Guiding Metaphor	Information Processing	Creativity
Process Theories	FBS Framework, Basic Design Cycle	Selfconscious Process
Related Methods	Waterfall Model, Rational Unified Process	Agile Methods, Amethodical Development
Related Models	Model of Technical Rationality	Design Process I/O Model
Scope of Design	phase between analysis and implementation	everything from intention to completed project

3.4.2 How to Test Perspectives

As mentioned above, "the concept of method ... occupies an extremely privileged status in formal information systems development thought" (Truex et al. 2000, p. 54). RCP has occupied an analogously privileged status in design research despite little empirical evidence concerning its assumptions and ramifications. As the Reason and Action-Centric Perspectives are very general, ontological notions, it is not clear how to test them directly vis-à-vis either their empirical accuracy or usefulness. Furthermore, testing SDMs is fraught with difficulties (ch. 1) and it is unclear how conclusions concerning specific SDMs generate insight into their associated perspectives. In contrast, testing the *usefulness* of Technical Problem-Solving and Reflection-in-Action is possible – Dorst and Dijkhuis (1995) evaluated the descriptive capabilities of both using a protocol study of individual industrial designers, finding that design activities could be coded according to either. However, how to test the *explanatory validity* of these broad models remains unclear.

Nevertheless, process theories such as FBS can be tested. The claim that FBS is an accurate description of how software is designed in practice implies predicted, observable characteristics of design projects. For example, if FBS is accurate, a set of functions (goals) of the software is provided to the developers at the beginning of the project. Moreover, although evidence concerning FBS cannot automatically generalize to RCP, such evidence on the former may generate specific insights into the latter. For instance, if a set of functions was not provided during a design project and instead the designers constructed the functions through iterative interaction with their environment and various stakeholders, it would suggest not only that FBS is flawed but also that the separation of problem-setting and -solving inherent to Technical Problem-Solving lacks descriptive validity. This, in turn, would undermine the view of design as a cognitive, rather than emergent phenomenon, since the goals of the system were emerging from the interaction between the designers and their environment, rather than being provided. This suggests the following key methodological insight.

> ***Research Methodology Insight:*** *the Reason- and Action-Centric Perspectives can be tested by operationalizing each as a specific software design process theory.*

Moreover, it may be more methodologically sound to test these process theories comparatively (Poole et al. 2000; Sober 1999; Wolfe 1994; Yin 2003), that is, to empirically evaluate the descriptive and explanatory validity of two or more process theories *relatively*.

3.4.3 Operationalizing the Perspectives

Three process theories identified above are consistent with RCP. The Problem-Design Exploration Model is inappropriate for operationalizing this perspective as its intended scope (design with genetic algorithms) is much more narrow. FBS (consolidated interpretation) is inappropriate because it is only partially consistent with RCP. The Basic Design Cycle and FBS Framework (nominalist interpretation) are both reasonable choices; however, FBS (nominalist interpretation) is more appropriate for at least three reasons.

1. It subsumes The Basic Design Cycle, i.e., the latter's artifacts and process map onto the former.
2. Gero and his collaborators have defined its concepts and relationships, minimizing the interpretation required for testing it.

3. FBS has generated an identifiable body of research and commentary, giving it more academic credibility and influence.

Therefore, RCP may be operationalized using FBS (nominalist interpretation). This is similar to claiming RCP represents FBS's assumptions.

Alexander's Selfconscious Process is the only existing process theory consistent with Reflection-in-Action and applicable to software design, providing a solid foundation for a design process theory as it is relatively simple and posits concepts that are difficult to deny. That designing leads to a conceptual or physical form (design object) and that this form exists in a context (environment) are central to design by definition (ch. 2). The conjecture that a design agent forms beliefs about both form and context is often assumed in design literature (e.g., Dorst and Cross 2001) and is supported by empirical studies (e.g., Purao et al. 2002; Schön 1983). Furthermore, although Alexander admits that the nature of the relationship between the two mental pictures is unclear, this relationship is found in other studies (e.g., Dorst and Cross 2001; Purao et al. 2002) and is the basis for mapping theories of design (Appendix D). Therefore, while the Selfconscious Process may provide a basis for a software design process theory consistent with ACP, further theory development is required. Several requirements for such a theory are discussed next.

3.4.4 New Theory Criteria

Based on the preceding discussion, I identify the following criteria for a new design theory.

- **Action-Centric.** The proposed theory must be consistent with elements of ACP, including Reflection-in-Action.
- **Teleological Process Theory.** Obviously, comparing unlike theory types (e.g., process theory to variance theory) would be problematic. Therefore, to facilitate comparison with FBS, a process theory is required. Van de Ven and Poole (1995) explain that theories with different causal motors (e.g., dialectic, evolutionary) may all describe the same phenomenon. However, as "design belongs

to the category of behavior called teleological, i.e., ‘goal seeking’ behavior” (Churchman 1971, p. 5), and FBS is teleological, it would be best if the proposed theory shared this causal motor.

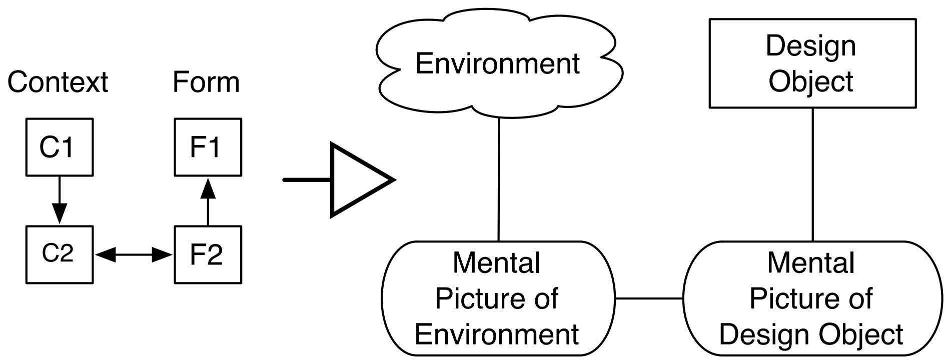
- **Overcome Limitations of Selfconscious Process.** The proposed theory must overcome the two limitations of the Selfconscious Process identified above: 1) the absence of clearly defined concepts and relationships, which obstructs application of the theory; 2) the absence of elements fundamental to software design.
- **Testable Propositions.** The proposed theory should lead to testable propositions; i.e., predictions concerning design projects, observably different from predictions generated from FBS.
- **Face Validity.** The proposed theory should be plausible on its face, given Action-Centric assumptions, how FBS is plausible given Reason-Centric assumptions.
- **Usefulness.** As stated above, the proposed theory should facilitate empirical design research by providing a testable FBS Framework alternative. Moreover, if the proposed theory is supported, it should be practically useful.

3.5 Proposing The SCI Framework

3.5.1 The Sensemaking-Coevolution-Implementation Framework (SCI)

This section describes how a software design process theory was generated from existing literature. The theory generation is shown in four steps with rationale; precise definitions are provided in Section 3.5.2.

I begin with Alexander’s (1964) Selfconscious Process (Figure 3-5), the plausibility of which was established in §3.3.3. The first step (Figure 3-9) is to give the concepts in the Selfconscious Process more convenient labels. I rename C1 (Context) to “Environment” and F1 (Form) to “Design Object” for consistency with the Design Process I/O Model. The renaming of C2 and F2 follow the same logic. The cloud symbol is used to indicate the environment’s unbounded nature. The rectangle indicates that the Design Object is a conceptual object (it exists in the world outside the designer), while rounded rectangles indicate that the mental pictures do not exist outside of the design agent.



Context
Form
C1
C2
F1
F2
Environment
Design Object
Mental Picture of Environment
Mental Picture of Design Object

Fig. 3-9. Theory Development Step 1

The second step (Figure 3-10) is to label the three relationships, each of which represents a fundamental activity of software design. Diamond icons indicate relationships (as in an Entity-Relationship Diagram). The first activity is one where the design agent organizes its perceptions to create a meaningful mental picture of the environment. "To convert a problematic situation to a problem, a practitioner must … make sense of an uncertain situation that initially makes no sense" (Schön 1983, p. 40). While Schön calls this "framing," the organizational literature terms it *sensemaking* (Weick 1995). Sensemaking refers to "the process by which individuals (or organizations) create an understanding so that they can act in a principled and informed manner".[2] Henceforth, I use the term sensemaking, since this further ties the proposed theory to existing organizational literature.

The relationship between the mental pictures of the environment and the design object is labeled *coevolution* (after Dorst and Cross 2001; Maher et al. 1995). Coevolution refers to the mutual, iterative refinement of the design agents mental pictures of the environment and design object. A good example of this is provided Dorst et. al.'s protocol study:

> *A seed of coherent information was formed in the assignment information, and helped to crystallise a core solution idea. This core solution idea changed the designer's view of the problem. We then observed designers redefining the problem, and checking whether this fits*

[2] From the Intelligent Systems Laboratory Glossary of Sensemaking Terms, available: http://www2.parc.com/istl/groups/hdi/sensemaking/glossary.htm (accessed Jan 28, 2009).

in with earlier solution-ideas. Then they modified the fledgling-solution they had (Dorst and Cross 2001, p. 434).

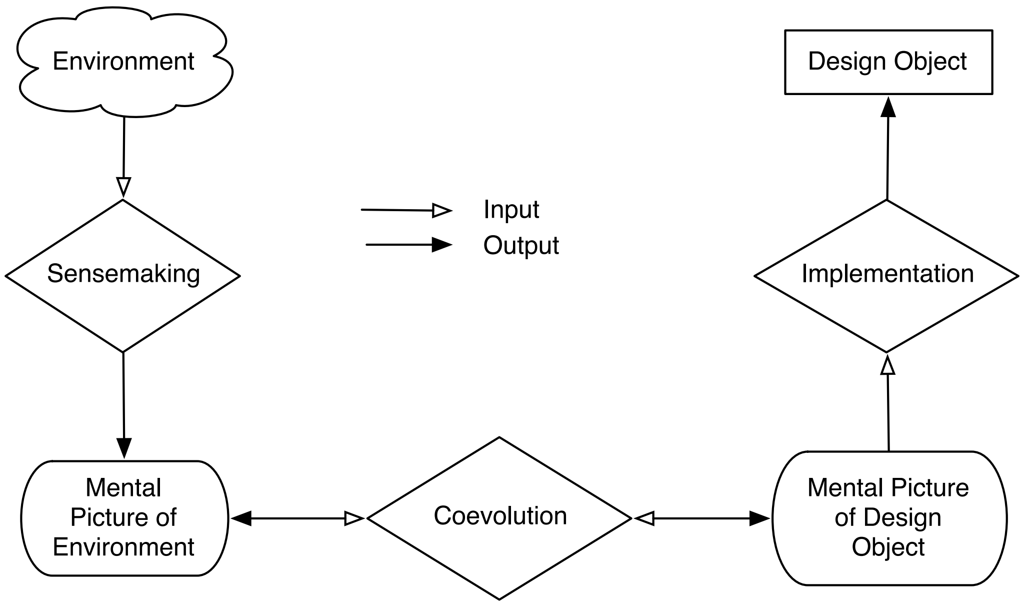


Fig. 3-10. Theory Development Step 2

Many authors consider the idea of a designer mutually and iteratively refining mental pictures of the environment (/ problem / context / problem space) and the design object (/ form / solution space) central to software development (§3.3.2).

The relationship wherein the mental picture of the design object is realized in the actual design object is denoted "Implementation" for consistency with the software industry vernacular (cf. Bourque and Dupuis 2004). As described in Chapter Two, design ends with the complete specification of the design object, either in a representation (e.g., architectural blueprints) of the design object or the object itself. Therefore, defining Implementation as a relationship between the Mental Picture of the Design Object and the Design Object itself (rather than a representation), embeds in the theory the proposition that no *complete* specification of the software is created prior to the software code (see ch. 2).

Given the centrality of these three activities, *I call the proposed theory the Sensemaking-Coevolution-Implementation Framework of Software Design (SCI Framework).*

The three core activities – sensemaking, coevolution and implementation – are each executed by an agent (Alexander 1964; Eekels 2000). Figure 3-11 shows the agent explicitly and links it to each of the three main processes. Though not a controversial point, showing the design agent explicitly (Figure 3-11) requires a distinction between the agent's environment and the object's environment (ch. 2). Obviously, the environment of designer (e.g., an office with chairs, desks, workstations) is not necessarily equivalent to the environment of the design object (e.g., the Internet). When a design agent engages in sensemaking, it considers both its environment and the environment of the existing or would-be design object. I now relabel Mental Picture of Environment to Mental Picture of Context, where 'Context' refers to both environments. Once the design object exists, it is part of the Design Object's Environment by definition.

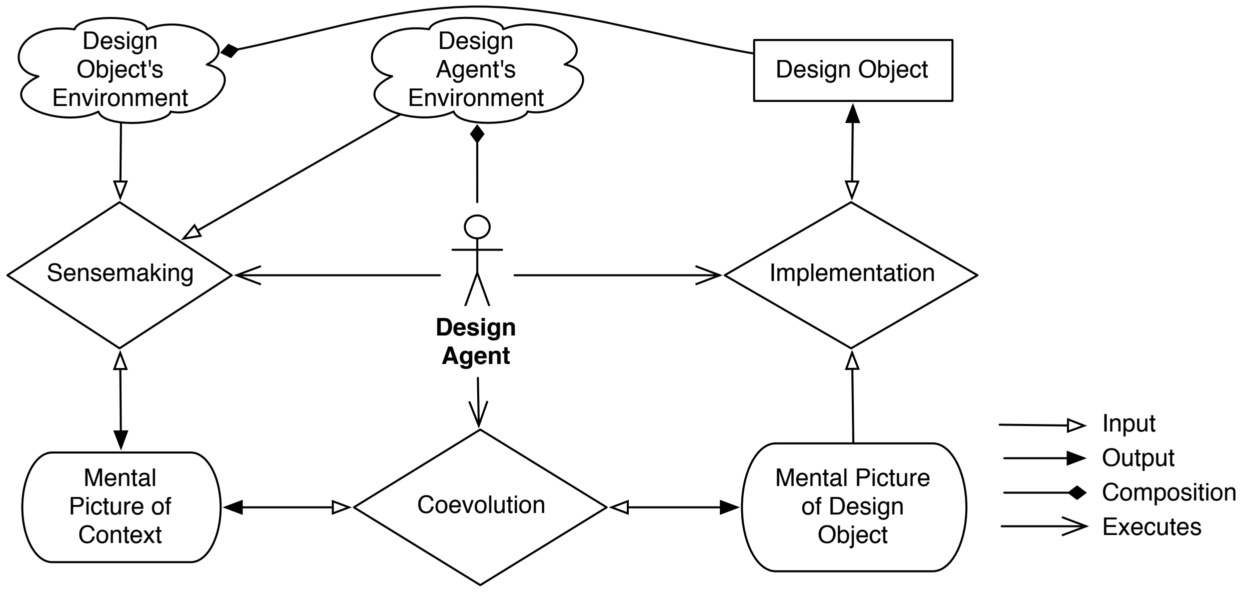

Design Object's Environment
Design Agent's Environment
Design Object
Sensemaking
Implementation
Design Agent
Mental Picture of Context
Coevolution
Mental Picture of Design Object
Input
Output
Composition
Executes

Fig. 3-11. Theory Development Step 3

Finally, I add four concepts from the Design Process Input/Output Model – primitives, goals, requirements and constraints (Figure 3-12). Primitives are the entities from which the design object is constructed (ch. 2). In software design, examples of primitives include programming languages, APIs, design patterns and data structures. Like agents, goals are fundamental to teleological process theories (Van de Ven and Poole 1995) and a central topic in the requirements engineering literature (Dardenne and Lamsweerde 1993). Requirements and constraints desired properties of and restrictions on the design object (ch. 2). (It is not entirely clear if or how requirements differ from constraints and to some extent this is an empirical question; therefore, I include both concepts and leave exploring this distinction to future work.) Requirements and constraints are central constructs in both prescriptive and theoretical literature on software and engineering design (Boehm 1988; Bourque and Dupuis 2004; Gregor and Jones 2007; Lyytinen 1987; Sommerville 1996). Goals, requirements and constraints are all part of the design agent's mental picture of the context. Although the design agent may employ external representations of them, goals, requirements and constraints are socially constructed entities, which do not exist in the physical world independent of social actors such as the design agent.

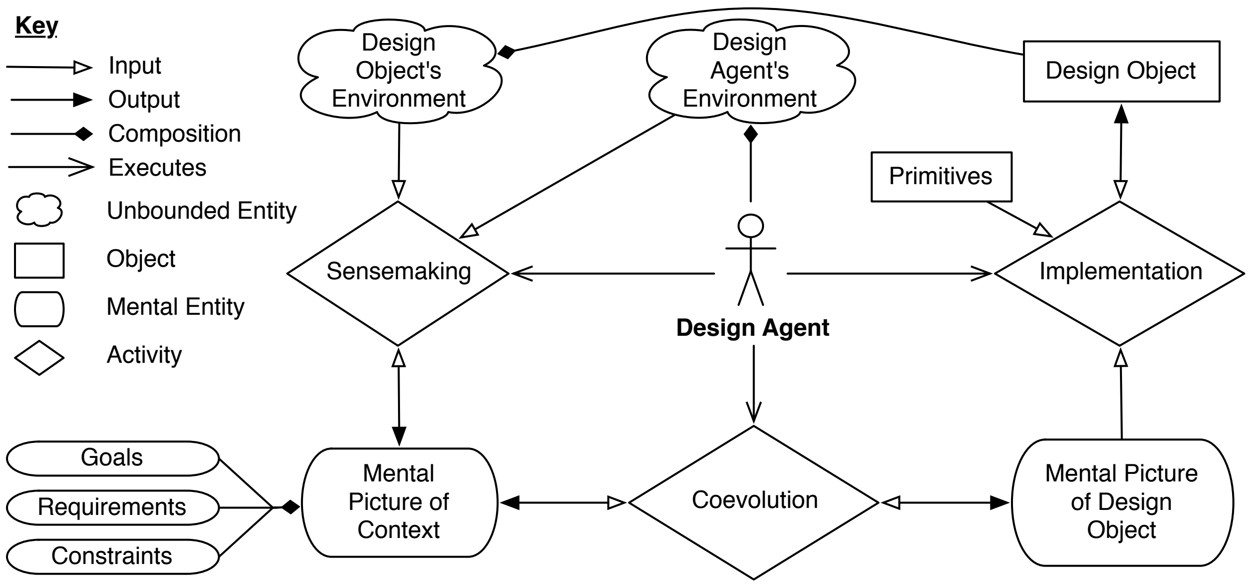


Fig. 3-12. Theory Development Step 4 – The SCI Framework

To clarify, the SCI Framework contains three activities – *Sensemaking, Coevolution* and *Implementation*. The arrows indicate *relationships* between concepts and activities, **not** the *sequence* of activities. Since implementation depends on the Mental Picture of the Design Object, which is initially generated by the coevolution process, the design agent must engage in a degree of coevolution before implementation. By equivalent logic, the design agent must do some sensemaking before beginning coevolution. However, once the two mental pictures are initially formed, the agent may transition between activities in any order. An agent-determined activity sequence is an essential characteristic of teleological process theories (Van de Ven and Poole 1995). Sensemaking, Coevolution and Implementation are activities that an agent may engage in *at will*, not phases or preconditions as in a lifecycle theory.

3.5.2 SCI Framework Definitions

Tables 3-11 and 3-12 define concepts and relationships, respectively, in the proposed theory and cite related existing work. The SCI Framework amalgamates research from engineering (including work by Eekels, Sommerville and Suh), architecture (including work by Alexander), computer science (including work by Maher and Simon), management (including work by Schön and Weick), product design (including work by Dorst and Cross) and information systems (including work by Purao and myself).

Table 3-11. Concepts of the SCI Framework

Concept	Meaning	Sources
Constraints	a restriction on a structural or behavioral property of the design object	(Simon 1996; Sommerville 1996; Suh 1990), ch. 2
Design Agent	an entity or group of entities that is capable of forming intentions and goals and taking actions to achieve those goals, and that specifies the structural properties of the design object	(Alexander 1964; Eekels 2000), ch. 2
Design Object's Environment	the totality of the surroundings where the design object exists or is intended to exist	(Alexander 1964), ch. 2
Design Agent's Environment	the totality of the surroundings of the design agent	(Checkland and Holwell 1998; Schön 1983) ch. 2
Design Object	the software artifact under construction, exclusive of non-software artifacts such as models and end-user documentation	(Alexander 1964; Eekels 2000), ch. 2
Goals	optative statements (which may exist at varying levels of abstraction) about the effects the design object should have on the design object's environment	(Churchman 1971; Dardenne and Lamsweerde 1993; Suh 1990), ch. 2
Mental Picture of Context	the collection of all beliefs, held by the design agent, regarding the design agent's environment and the design object's environments	(Alexander 1964; Maher et al. 1995; Purao et al. 2002)
Mental Picture of Design Object	the collection of all beliefs held and decisions made by the design agent concerning the design object	(Alexander 1964; Maher et al. 1995; Purao et al. 2002)
Primitives	the set of entities from which the design object may be composed	(Meyer 1988), ch. 2
Requirements	a structural or behavioral property that a design object must possess	(Bourque and Dupuis 2004; Royce 1970; Suh 1990), ch. 2

Table 3-12. Activities of the SCI Framework

Activity	Meaning	Sources
Sensemaking	the process by which the design agent perceives the design agent's environment and the design object's environment and organizes these perceptions to create or refine the mental picture of context	(Schön 1983; Weick 1995; Weick et al. 2005)
Coevolution	the process by which the design agent simultaneously refines its mental picture of design object based on its mental picture of context, and vice versa	(Alexander 1964; Dorst and Cross 2001; Maher et al. 1995; Schön 1983)
Implementation	the process by which the design agent generates or updates a design object using its mental picture of design object	(Alexander 1964; Boehm 1988; Bourque and Dupuis 2004; Royce 1970)

3.5.3 Conceptual Evaluation

Here I discuss the extent to which the SCI Framework meets the criteria described in Section 3.4.2.

- **Action-Centric.** The proposed theory combines concepts from the Selfconscious Process (Alexander 1964) and the Design Process Input/Output Model (ch. 2). It explicitly includes the four major concepts and three major relationships from the Selfconscious Process and all of the concepts in the Design Process I/O Model except *intentions* and *type of object*. Intentions are implicit included by way of the design agent, which forms intentions by definition (Table 3-2). Additionally, as the proposed theory explains software design, the type of object is always software. The proposed theory is consistent with ACP and specifically Reflection-in-Action for the same reasons as the Selfconscious Process and Design Process I/O Model from which it derives.
- **Teleological Process Theory.** The SCI Framework is an example of a teleological process theory – an explanation of how and why an entity changes wherein change is manifested by a goal-seeking agent that engages in activities in a self-determined sequence, and monitors progress (Churchman 1971; Singer 1959; Van de Ven and Poole 1995). Recalling the three criteria from §3.3.5: the SCI Framework posits a formal relationship between a high-level activity (software design) and the three lower-level activities for which it is named, it includes the teleological causal motor vis-à-vis the explicitly represented goal-seeking agent and it claims to apply to all software design. Therefore, it satisfies all three criteria to be classified as a teleological process theory.
- **Overcome Limitations of Selfconscious Process.** The primary limitations of the Selfconscious Process identified above are the absence of clearly defined concepts and relationships and lack of support for elements fundamental to software design. The proposed theory overcomes the first through the concept definition tables above. Moreover, unlike the models from which it was generated, the SCI Framework specifically explains *software* design. One software-specific aspect is the use of the design object as the primary iterative artifact rather than, for example technical drawings, as often used in product design (Roozenburg and Eekels 1995). Another software-specific

feature is the strict separation of the design agent's environment (physical) from that of the design object (virtual).

- **Testable Propositions.** The SCI Framework should lead to specific predictions, concerning design projects, observably differing from predictions generated from FBS. I submit that the two theories differ in at least three ways:
    - whether problem-setting and problem-solving are separate and sequential (FBS Framework) or cotemporal and inextricably linked (SCI Framework)
    - whether the coding process is driven by prefigured decisions (FBS Framework) or evolves iteratively with the design process (SCI Framework)
    - whether designers focus on models (FBS Framework) or code (SCI Framework)
- **Face Validity.** The plausibility of the SCI Framework may be justified in three ways. Firstly, the SCI Framework is an elaboration of an influential, pre-existing theory from another discipline, Alexander's Selfconscious Process. Therefore, Alexander's (1964) arguments and evidence for the Selfconscious Process constitute part of its theoretical basis. Second, each of the concepts and relationships in the SCI Framework is grounded in existing literature (Tables 3-11, 3-12). This further evidences the solid theoretical foundation on which the SCI Framework is built. Third, the core claim of the SCI Framework is that developers must engage in at least three activities to produce software – making sense of the project context, iterating between ideas about the context and artifact, and implementing the artifact in code. I submit that the first and third activities are uncontroversial and that the second is inherent to ACP generally and Reflection-in-Action specifically; therefore, the SCI Framework is intuitively sensible given Action-Centric assumptions.
- **Usefulness.** The proposed theory is immediately useful insofar as it facilitates empirical design research as an alternative against which to test FBS. Furthermore, if the SCI Framework is supported, it is useful for several practical purposes (§3.6.2). Moreover, the SCI Framework links three different bodies of research: 1) largely qualitative organizational literature (sensemaking); 2) general and product design literature (coevolution); 3) computer science and engineering literature

(implementation). Hence, the proposed theory may support theory integration across these three fields.

- **Further Evaluation.** The SCI Framework is a novel contribution. Although, it extends existing models by Alexander (1964) and others, the concepts from these models have been elaborated and combined in a previously unseen manner. Additionally in Section 3.2.7, it was suggested that a veracious software design process theory would describe the behavior of all designers, not just 'consciously competent' designers. I see no *a priori* reason to believe that any of the three core activities of the SCI Framework is specific to consciously competent designers. However, an unconsciously competent designer may not be aware of the engaged-in core activity, while an incompetent designer may not execute any core activity effectively.

3.5.4 Empirically Evaluating the SCI Framework

Whether the concepts and relationships of the SCI Framework accurately represent how software is designed in practice is an empirical question. However, due to the complexity of theoretically justifying the SCI Framework and positioning it as an antithesis to FBS, relative to the Action- and Reason-Centric Perspectives, empirical testing of the SCI Framework is left to future work (ch. 4). Therefore, this section demonstrates that such testing is possible, in principle.

In the past, testability was considered a property of an individual theory (Popper 1959). Generally speaking, a theory was testable if and only if one could establish observations supporting or refuting it. Following the refutation of logical positivism by Popper and the subsequent refutation of Falsificationism by philosophers including Feyerabend, Hempel, Lakatos and Quine[3], the idea of testability in philosophy of science began to be perceived as obsolete (Sober 1999). In response, Sober (1999) establishes that testability is a probabilistic relationship between two theories that make contrasting predictions. This underlies recommendations by methodologists (e.g., Yin 2003), to evaluate a theory against a plausible

[3] A full account of the philosophical arguments surrounding testability is beyond the scope of this paper; for more, see The Stanford Encyclopedia of Philosophy: http://plato.stanford.edu/entries/popper/#Crit

rival theory. Above, I have positioned the SCI Framework as a plausible rival theory to FBS. Both the FBS and SCI Frameworks are teleological process theories, making them, in a sense, compatible for comparison.

While one may argue that FBS remains a theoretical straw-man as it lacks empirical support within the software development domain, the same is true of all of the process theories uncovered by my literature review. It is appropriate to test the SCI and FBS Frameworks because these are, as far as I can determine, the best alternatives available.

Hence, taking a comparative approach to testability (Sober 1999), my research question (*what is the process whereby development teams create software, in practice?*) may now be operationalized as *of the FBS and SCI Frameworks, which better describes how development teams create software in practice?*

Two broad strategies for comparatively evaluating process theories are evident (Figure 3-13). In one strategy (left), the two frameworks are compared analytically, creating a set of contrasting predictions (§3.5.3). These predictions may then be evaluated against one or more real situations. The theory that produces more correct predictions would be supported. (Note that under comparative testability, theories are not 'proven' or 'falsified'; rather, a theory is supported if it is more valid on a balance of evidence, than the alternative theory). In the other strategy (right), both frameworks are independently compared to one or more real situations, producing a base of evidence for (and against) each construct and relationship of each theory. For example, finding a written requirements document may support the *expected behavior* artifact in FBS. These two collections of evidence may then be compared to determine which is more complete and convincing.

Much less methodological advice is available concerning evaluating process theories than variance theories; however, Wolfe (1994) identified two common approaches to studying innovation processes – cross-sectional surveys and in-depth field studies. More generally, Poole et al. (2000) considers three research designs appropriate to studying change processes – cross-sectional survey, panel (longitudinal) survey and process (field) studies. The first generic strategy described above seems amenable to a survey approach. For each contrasting prediction identified, a set of survey items may be generated, validated and given to a random sample of software developers representing diverse organizations. Similarly, the second strategy appears amenable to a field study. Collecting in-depth, longitudinal data from a small number of software development teams may provide a rich evidentiary base for each theory. This evidence may be compared, using an a priori coding scheme based on the two theories, to determine which is more veracious. Furthermore, combining the two approaches enables multi-method triangulation – the survey allows for random sampling and reliability, while the field study facilitates gathering deep insights into developer behaviors and cognitive processes.

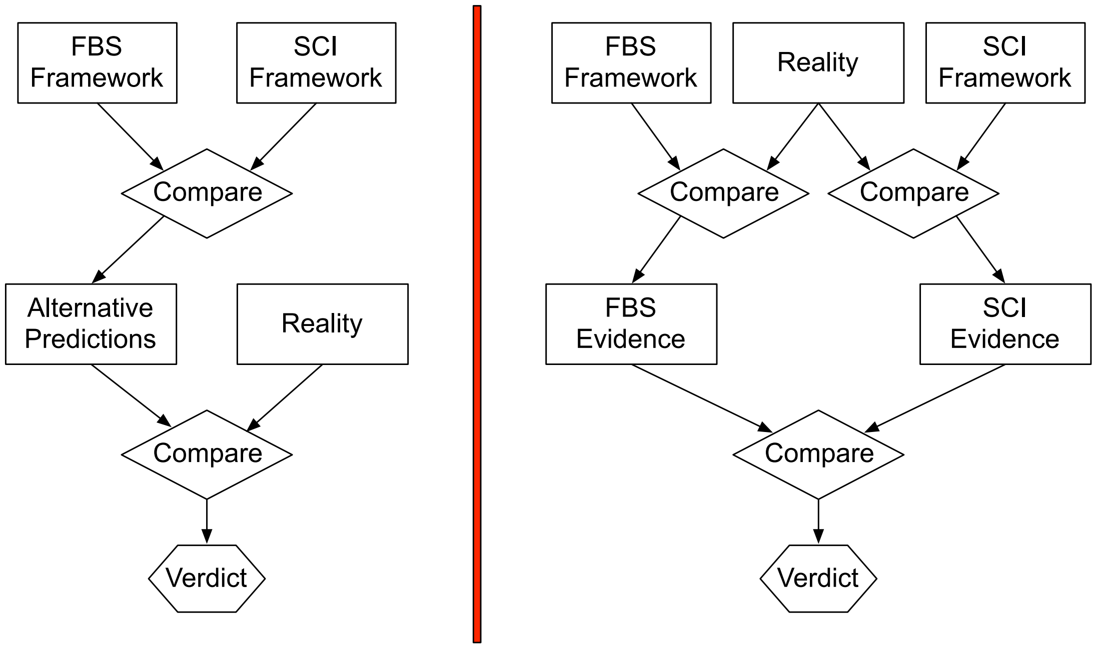


Fig. 3-13. Two Empirical Strategies for Comparing the FBS and SCI Frameworks

In summary, while empirically evaluating the SCI and FBS Frameworks is left to Chapter Four, this section demonstrates that the SCI Framework is testable, in principle in at least two different ways: 1) using a survey to evaluate contrasting predictions; 2) using a field study to gather evidence for the concepts and relationships of each theory.

3.5.5 Limitations of Proposed Theory and Future Additions

Firstly, the proposed theory is not a comprehensive enumeration of all design-related activities. I attempted to include only elements inherent to design (those without which design cannot occur). For example, a designer may externalize his or her mental picture of the design object using an UML diagram but, since such diagrams are optional, they are not included. Please note that none of the theories or models reviewed in this chapter attempt to enumerate all design-related activities; however, Sim and Duffy (2003) proposed an "ontology of generic engineering design activities" including activities such as "abstracting", "decomposing", "decision making", "simulating", and "searching." If initial empirical testing of the SCI Framework is promising, it may be useful to map Sim and Duffy's ontology onto it.

Secondly, as indicated in the introduction, the SCI Framework focuses on process. This omits characteristics of both the design agent and the design context. While both agent and context are clearly important, hypothesizing effects of their characteristics would have significantly increased the complexity of the proposed theory. Therefore, consideration of these topics is left to future research.

Third, it is easy to brainstorm important design-related concepts omitted from the proposed theory, including time, software quality, politics and power; however, no identified design process theories address these concepts either. Moreover, while these and many other concepts may play important parts in software design, at this stage I deem it more important to propose a theory that was as simple and theoretically grounded as possible. While one may argue endlessly about which concepts are the *core concepts*, practically speaking, this is somewhat arbitrary and simply not including a concept in the first rendering of a new theory does not delegitimize it. Other important and relevant concepts are simply left for future work. Including a notion of software quality, in particular, is exceedingly complex and explaining the antecedents of quality is a research program onto itself. Again, if empirical testing supports the SCI Framework, it may be reasonable to add other concepts.

Fourth, in suggesting a comparative test between the SCI and FBS Frameworks, I have omitted the possibility that a particular project, firm, or even a whole field, may gradually shift from a process consistent with the SCI Framework to one consistent with the FBS Framework. If, for example, a field's body of knowledge grew such that problems became easier to specify and articulate, and systems became more predictable and exhibited fewer emergent behaviors, its designers might shift to more FBS-like processes.

3.6 CONCLUSION

3.6.1 Contribution and Scope

As stated above, this chapter's purpose is to review what is believed about how professionals create software in practice and to synthesize these beliefs into an empirically testable form. Consequentially, this chapter contributes to research on software design science in two ways.

First, it reviews diverse literature relevant to software design and discerns a conflict between two perspectives: Reason-Centric and Action-Centric. This powerful conceptualization aides understanding of the untested assumptions underlying much design literature and the privileged position of Reason-Centric assumptions, despite lacking empirical evidence. This is not unlike the primacy of the theories of rationality and expected utility before they were debunked by psychologists including Kahneman and Tversky (1979).

Second, it operationalizes each perspective in an empirically testable process theory. RCP is operationalized by FBS (nominalist interpretation). ACP is operationalized by the SCI Framework, which was generated by elaborating the Selfconscious Process using concepts from existing literature, including the Design Process I/O Model, and conceptually evaluated using pre-figured criteria. Positioning the SCI Framework as an antithesis to FBS and giving suggestions as to their comparative empirical evaluation constitutes the "empirically testable form" I set out to produce.

Proposing the SCI Framework is a necessary preliminary step to support empirical research in software design science. I conjecture that defining the SCI Framework and convincingly justifying its concepts and relationships exhibits complexity similar to that of designing and executing a thorough empirical evaluation of either or both of the SCI or FBS Frameworks. As combining these endeavors would produce an unreasonably long chapter, this chapter presents the theory building phase and leaves empirical analysis to future work.

3.6.2 Implications for Academics and Managers

The SCI Framework is immediately useful for facilitating an empirical comparison with FBS to establish which is a better working theory of the software design process. Whichever of these theories is supported will then be useful to both academics and practitioners in several ways. As uses of FBS are discussed elsewhere (cf., Gero 1990; Gero and Kannengiesser 2004; Kruchten 2005), this section discusses only potential uses of the SCI Framework.

Academics. The SCI Framework may be useful for research in at least three ways.

1. It may facilitate evaluating and improving design methods, tools and practices. For example, in evaluating an SDM, we may ask "does this methodology provide guidance concerning all three fundamental design activities – sensemaking, coevolution and implementation?" If not, can the methodology be improved by considering those omitted?
2. The SCI Framework may be useful for evaluating and updating courses and curricula. For instance, if the ACM model curriculum for software engineering[4] lacked treatment of one of the fundamental design activities (e.g., sensemaking) this would indicate a potential avenue for improvement. Moreover, if the SCI Framework is supported over FBS (which subsumes the Systems Development Lifecycle) it may be more useful to teach supported SCI Framework concepts rather than unsupported SDLC concepts in design-oriented courses.
3. It may inform development of an antecedent theory of design project success. In a strict interpretation of causality, causal theories imply precedence relationships. The SCI-Framework dispenses with Waterfall-like, artificial activity sequences. Therefore, it may eliminate extraneous

[4] see http://www.acm.org/education/curricula-recommendations

causal relationships during theory building (e.g., the hypothesis that analysis quality causes design quality is *a priori* incorrect as analysis and design are cotemporal in practice).

Managers. The SCI Framework posits that problem-understanding and -solving are cotemporal in practice, that code is written iteratively and that the primary iterative artifact used by developers is source code (rather than models). This does not mean that attempting to separate analysis from design, write code linearly or iterate on models is less effective. However, if the SCI Framework is accurate, it implies several issues for software project managers.

1. Developers may resist attempts to pressure them into separating analysis from design, writing code linearly or iterating on models. Developers may fake adherence to design methodologies that are inconsistent with their natural way of working (Parnas and Clements 1986).
2. It is unlikely that implementing a tool, practice or method that is inconsistent with iterative coding and simultaneous analysis and design will be effective without a corresponding change in development practice.
3. If developers do not understand the problems they are solving until the solution is well into development, it means that any upfront budget and schedule estimates are made without any substantive understanding of the problem. It seems incredulous that anyone could accurately estimate the cost of solving a problem without knowing what the problem is.
4. Since many computer science, computer engineering and information systems programs teach development according to RCP, new graduates may be deeply confused about the nature of development practice.

3.6.3 The Way Forward

Many academics and practitioners have written prescriptive accounts of how software *should be* designed (Wynekoop and Russo 1997), yet how software *is* designed remains largely unknown (Freeman and Hart 2004; Simon 1996; Sullivan 2003; Wynekoop and Russo 1995). This research flows from the commonsense premise that it may be useful to describe what design teams *actually* do before trying to prescribe what they *should* do. The ubiquity of design behooves social scientists to study it empirically. Once a dominant perspective and process theory are established (one perspective and process theory is found to better represent design practice than the other), several research streams become available, including evaluating popular SDMs, evaluating software design curricula and developing a theory of design project success (see ch. 5). These are all essential steps in software design science.

CHAPTER 4: A COMPARATIVE, EMPIRICAL EVALUATION OF TWO SOFTWARE DESIGN PROCESS THEORIES[1]

[1] A previous version of this paper was accepted for the International Conference on Design Science Research in Information Systems and Technology (DESRIST 2010), St. Gallen, Switzerland, and published in R. Winter, J. L. Zhao and S. Aier (Eds.): *Global Perspectives on Design Science Research, LNCS 6105*, Springer, pp. 61-76.

4.1 INTRODUCTION

A key element of design science, in the “researching design” sense used in this thesis, involves theories of the shape and organization of the design process (Simon 1996). Yet the shape and organization of the design process of software, in particular, is not well understood (Freeman and Hart 2004; Simon 1996; Sullivan 2003; Wynekoop and Russo 1995), as most academic work on software design is prescriptive, rather than explanatory or descriptive (see Wynekoop and Russo 1997; ch. 3). The purpose of this chapter is to study empirically the shape and organization of the software design process – hence, my primary research question.

Research Question: *What is the nature of the process by which development teams create software in practice?*

Following Chapter Two, *software design* (verb) is the act of creating a specification of a software object, by an agent, intended to accomplish goals in a particular environment, using a set of primitive components, satisfying a set of requirements, subject to set of constraints. The software design literature is characterized by two contradictory perspectives on the nature of software design: one perspective holds that design is cognitive, the other holds that design is emergent (ch. 3). If design is an essentially cognitive phenomenon, then design activities occur within a designer’s mind (or cognitive system). If design is an essentially emergent phenomenon, then design activities are dominated by interactions with elements outside the designer’s cognitive system, such as stakeholders.

Each of these perspectives underlies at least one design process theory. A *process theory* is an explanation of how and why an entity changes and develops (see Appendix C), which is distinct from a process model – “an abstract description of an actual or proposed process” (Curtis et al. 1992, p. 76). A process theory seeks to explain how outcomes materialize *in general*, not simply one or several historical or possible activity sequences. As the two perspectives are sufficiently abstract as to make direct evaluation difficult, this chapter attempts to evaluate them vicariously through related process theories. The primary purpose of the chapter is to determine which of the perspectives better reflect the reality of software design - the process theories act analogously to mediating variables, enabling this purpose.

This chapter is organized as follows. Section 4.2 organizes existing research on software design into two mutually-exclusive clusters of interrelated theoretical and philosophical concepts (the Reason- and Action-Centric Perspectives) and their operationalization through two software design process theories. Section 4.3 describes the survey methodology used to comparatively test these process theories – the results are described in Section 4.4. Section 4.5 concludes the chapter with a summary of its contributions and limitations, and an outline of the next phase of the study.

4.2 Two Perspectives on Software Design

The design literature is characterized by a fundamental disagreement as to whether design is a cognitive or emergent phenomenon (ch. 3). I refer to the view that design is cognitive and the view that design is emergent as the Reason- and Action-Centric Perspectives, respectively. These are defined formally as follows. The remainder of this section summarizes and analyzes their composition and implications.

> ***Reason-Centric Perspective (RCP):*** *1) The belief that design is a cognitive phenomenon; 2) the collection of paradigms, models, methodics and theories associated with this belief.*
>
> ***Action-Centric Perspective (ACP):*** *1) The belief that design is an emergent phenomenon; 2) the collection of paradigms, models, methodics and theories associated with this belief.*

4.2.1 The Reason-Centric Perspective

RCP, which holds a privileged position in design literature, is the belief that design is a *cognitive* phenomena – it occurs primarily within the designer's cognitive system (ch. 3). A cognitive system is a set of interrelated components capable of learning and reasoning. In cognitive science, significant debate exists concerning the scope of a person's cognitive system – sometimes referred to as the "extended mind" issue (see Clark and Chalmers 1998). While this debate is beyond the scope of this chapter, here I assume that designers may integrate simple elements into their cognitive systems, such as pencil and paper, but cannot integrate complex artifacts such as a calculator, or debugger. To illustrate, doing mental division is clearly a cognitive process and doing long division using pencil and paper is still, arguably, a cognitive process, because of the high cohesion of the mind plus pencil and paper system; however, inputting a division problem into Wolfram Alpha[2] does not make it part of one's cognitive system because of the low cohesion between Wolfram Alpha and the designer's mind. While the exact boundary of the extended mind remains a gray area, what is important for this chapter is that things like models, sketches and note cards can be part of the designer's cognitive system, but tens of thousands of lines of software code, test suites and APIs cannot because they are too complex to allow high cohesion.

This belief that design is cognitive is at the heart of Technical Rationality, the view that "professional activity consists in instrumental problem-solving made rigorous by the application of scientific theory and technique" (Schön 1983, p. 21). Technical Rationality requires given problems – goals are agreed in advance and constraints are knowable. Schön argues that Technical Rationality (and therefore RCP) is foundational to both positivism (Hacking 1982) and the Technical Problem-Solving design paradigm (Simon 1996). The latter posits that professionals design by optimizing or "satisficing" a design candidate vis-à-vis known constraints and objectives. RCP is also consistent with the *cognitivist view* of human action, where actions are executed and understood through a plan and defined as "a sequence of actions designed to accomplish some preconceived end" (Suchman 1987). Plans are prerequisites to action. Unanticipated conditions trigger replanning; evaluation is performed by comparing resultant and planned actions and outcomes. In this view, design is a form of plan-driven problem-solving, where an agent seeks a goal state by executing a plan in a field of constraints (Newell and Simon 1972). Moreover, this view is guided by an Information Processing metaphor – "the designer is seen as a machine capable of rationally selecting and connecting together elemental information to satisfy a set of constraints" (2000, p. 309).

[2] http://www.wolframalpha.com/

Software development methods including the Waterfall Model (Royce 1970) and Rational Unified Process (Kruchten 2003) embody these assumptions.

RCP can be operationalized using a compatible software design process theory. One process theory that can be compatible, depending on how it is interpreted for software, is the Function-Behavior-Structure Framework (FBS) (ch. 3). FBS (Gero 1990; Gero and Kannengiesser 2004) (Figure 4-1) claims that "the purpose of designing is to transform *function*, F (where F is a set), into a *design description*, D, in such a way that the artefact being described is capable of producing those functions" (Gero 1990, p. 2, original italics). Gero posited three "intermediate artifacts" – structure (S), structure's behavior, (B_s) and expected behaviors (B_e).

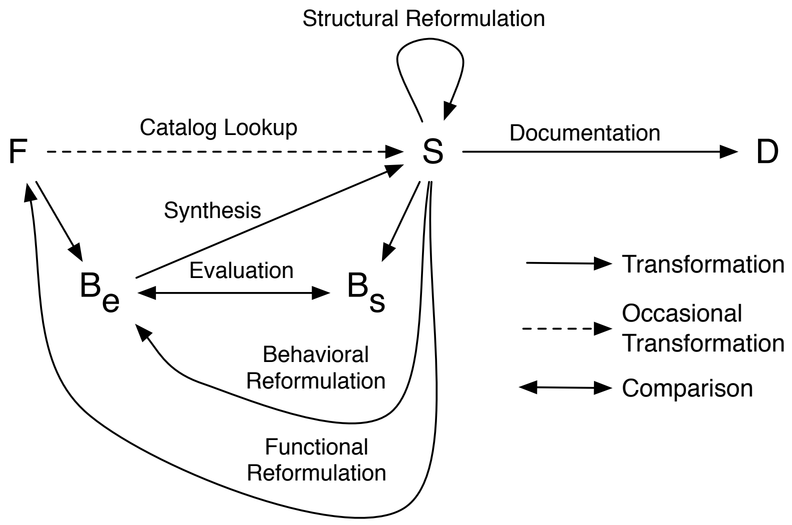


Fig. 4-1. The Function-Behavior-Structure Framework (adapted from Kruchten 2005)

Numerous papers have analyzed, applied and proposed revisions to the original FBS Framework. For instance, Gero and Kannengiesser (2004) situated function, behavior and structure in three different "worlds" – desired, internal and external – where each concept exists in each world; e.g., desired functions, the designer's interpretation of the functions of the current design candidate, and external representations of said interpretations. Vermaas and Dorst (2007) critiqued the model's "double aim of describing actual designing and prescribing improved designing" (p. 133) and argued for a new definition of function. Galle (2009) offered two re-interpretations of FBS to address ontological issues surrounding its core artifacts. In Galle's nominalist interpretation, functions, behaviors and structures are defined as descriptions; hence, FBS describes a process of symbol manipulation in which the designer generates a structure-description from function- and behavior-descriptions – creating and deploying the design object are outside of this process. For reasons previously described (see ch. 3), the Reason Centric Perspective can be operationalized using Galle's nominalist interpretation of FBS (see Tables 4-1 and 4-2 for definitions of FBS artifacts and operations). In the remainder of this chapter, "FBS Framework" refers to a nominalist interpretation of FBS unless otherwise stated.

Table 4-1. Artifacts of the FBS Framework (adapted from Galle 2009; Gero 1990)

Symbol	Meaning
Be	a desrciption of the desired behavior of the structure
Bs	a desrciption of the predicted behavior of the structure
D	a graphically, numerically and/or textually represented model that transfers information about the design object sufficient to manufacture, fabricate or construct it
F	a desrciption of the purposes of the design object
S	a desrciption of the design object's elements and their relationships

Table 4-2. Operations of the FBS Framework (adapted from Galle 2009; Gero 1990)

Operation	Inputs	Outputs	Meaning
Analysis	S	Bs	the process of predicting the behavior of a structure
Catalog Lookup	F	S	selecting a known structure that performs the required function
Evaluation	Bs & Be	Differences Between Bs and Be	comparing predicted behavior to desired behavior and determining whether the structure is capable of producing the functions
Formulation	F	Be	deriving desired behaviors from the set of functions
Production of Design Documentation	S	D	transforming the structure description into a design description suitable for manufacturing
Synthesis	Be	S & Bs	generating a structure description from the desired behaviors based on knowledge of the behaviors produced by that structure
Structural Reformulation	S, Bs	S	modifying the structure description based on itself and its predicted behaviors
Behavioral Reformulation	S, Bs & Be	Be	modifying the expected behaviors based on the structure description and its predicted behaviors
Functional Reformulation	S, Bs & F	F	modifying the set of functions based on the structure description and its predicted behaviors

4.2.2 The Action-Centric Perspective

ACP holds that design is an emergent phenomenon comprising continuous interactions between designers and their environment (see ch. 3). “Emergence … refers to the arising of novel and coherent structures, patterns and properties during the process of self-organization in complex systems” (Goldstein 1999). This is consistent with social constructivism, which posits that knowledge derives from social interactions (Berger and Luckmann 1966). ACP also underlies the Reflection-in-Action design paradigm, devised by Schön (1983), who built on social constructivism and empirical studies of professional practice. Reflection-in-Action models design as a reflective conversation between the designer and the situation. The designer alternates between framing (conceptualizing the problem), making moves (where a move is real or simulated action intended to improve the situation) and evaluating moves. Multiple agents may reflect collectively in action using boundary objects (Levina 2005). These concepts are broadly consistent with the *ethnomethodological view* of human action (ethno-view), where “the organization of situated action is an emergent property of moment-by-moment interactions between actors, and between actors and the environments of their action” (Suchman 1987, p. 179), while “plans are representations, or abstractions over action” (p. 186). Indeed, Schön (1983) argued that “when someone reflects in action … he does not keep means and ends separate … he does not separate thinking from doing” (p. 69). This further implies that innovation is based on the creativity and experience of the designer; consequently, the guiding metaphor of ACP is creativity (Love 2000). Software development methods including Scrum (Schwaber and Beedle 2001) and Extreme Programming (Beck 2005) embody these assumptions. ACP is also broadly consistent with the Agile philosophy (Beck et al. 2001) and amethodical development (Truex et al. 2000).

ACP is also compatible with a software design process theory, the Sensemaking-Coevolution-Implementation (SCI) Framework (ch. 3). The SCI Framework (Figure 4-2) claims that software design includes three primary activities (in no set order) – making sense of context; iteratively evolving mental pictures of context and software artifact; writing code based on the mental picture of the software. While

FBS was originally proposed as a model of engineering design, the SCI Framework (whose concepts and relationships are defined in Table 4-3) is specifically intended to describe software design.

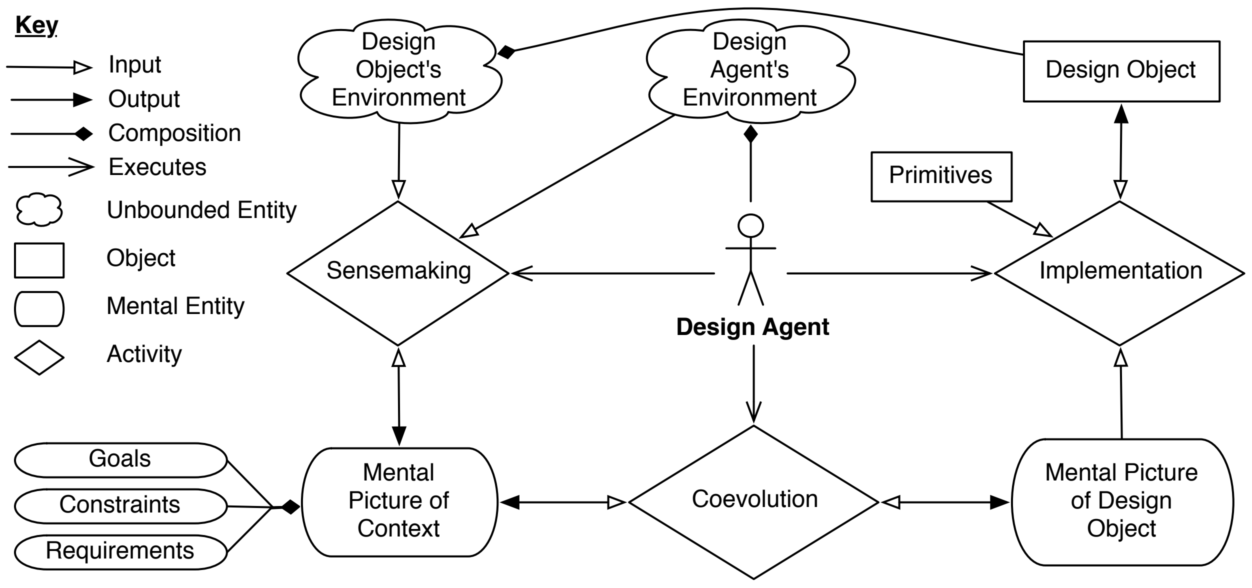


Fig. 4-2. The Sensemaking-Coevolution-Implementation Framework

Table 4-3. Concepts of the SCI Framework

Concept	**Meaning**	**Sources**
Constraints	a restriction on a structural or behavioral property of the design object	(Simon 1996; Sommerville 1996; Suh 1990), ch. 2
Design Agent	an entity or group of entities that is capable of forming intentions and goals and taking actions to achieve those goals, and that specifies the structural properties of the design object	(Alexander 1964; Eekels 2000), ch. 2
Design Object's Environment	the totality of the surroundings where the design object exists or is intended to exist	(Alexander 1964), ch. 2
Design Agent's Environment	the totality of the surroundings of the design agent	(Checkland and Holwell 1998; Schön 1983) ch. 2
Design Object	the software artifact under construction, exclusive of non-software artifacts such as models and end-user documentation	(Alexander 1964; Eekels 2000), ch. 2
Goals	optative statements (which may exist at varying levels of abstraction) about the effects the design object should have on the design object's environment	(Churchman 1971; Dardenne and Lamsweerde 1993; Suh 1990), ch. 2
Mental Picture of Context	the collection of all beliefs, held by the design agent, regarding the design agent's environment and the design object's environments	(Alexander 1964; Maher et al. 1995; Purao et al. 2002)
Mental Picture of Design Object	the collection of all beliefs held and decisions made by the design agent concerning the design object	(Alexander 1964; Maher et al. 1995; Purao et al. 2002)
Primitives	the set of entities from which the design object may be composed	(Meyer 1988), ch. 2
Requirements	a structural or behavioral property that a design object must possess	(Bourque and Dupuis 2004; Royce 1970; Suh 1990), ch. 2

4.2.3 Comparative Analysis of Perspectives and Process Theories

Perspectives in Conflict. Table 4-4 contrasts RCP and ACP. The conflict between these perspectives is evident in Beck's discussion of common tensions between managers who try to drive projects through cost estimates and developers who cannot reliably estimate complex projects (Beck 2005). Similarly, Graham explains misalignment between programming education and practice – "I was taught in college that one ought to figure out a program completely on paper before even going near a computer. I found that I did not program this way ... I tended to just spew out code that was hopelessly broken, and gradually beat it into shape" (Graham 2003). Moreover, "the concept of method ... occupies an extremely privileged status in formal information systems development thought" while "the possibility that amethodical development might be the normal way" of building systems has "almost entirely elud[ed] the systems development literature" (Truex et al. 2000, p. 54, 58). RCP has occupied an analogously privileged status in design research despite little empirical evidence concerning its assumptions and ramifications.

Table 4-4. Comparison of Reason- and Action-Centric Perspectives

Dimension	**Reason-Centric Perspective**	**Action-Centric Perspective**
Epistemology	Positivist	Constructivist
Theory of Action	Cognitivist	Ethnomethodological
Design Paradigm	Technical Problem-Solving	Reflection-in-Action
Guiding Metaphor	Information Processing	Creativity
Process Theory	FBS Framework (nominalist interpretation)	SCI Framework

Process Theory Similarities. The FBS and SCI Frameworks are similar in at least the following ways.

1. They are both teleological process theories (explanations of how and why an entity changes *wherein change is manifested by a goal-seeking agent that engages in activities in a self-determined sequence, and monitors progress* (Churchman 1971; Singer 1959; Van de Ven and Poole 1995)). Therefore, they share fundamental aspects of teleological process theories, including goals and an agent. In Gregor's (2006) typology, they are both theories for describing and explaining (ch. 3).

2. They share fundamental design concepts; e.g., FBS's *expected behavior* and *structure* concepts are similar to the SCI Framework's *requirements* and *mental picture of the design object* concepts (see Tables 4-1 and 4-3).
3. *Both frameworks are consistent with models*. In FBS, the designer necessarily creates an external representation of the design artifact's structure and may also model functions and behaviors. In the SCI Framework, the design agent may model mental pictures of the context (conceptual models) or the design object (design models) or use models to externalize or share cognition (Levina 2005).

These similarities have no apparent bearing on the Reason and Action-Centric Perspectives. First, either perspective could underly numerous process and causal theories. Second, the concepts shared by the two frameworks are specific to the domain (design), but have nothing to do with whether design is emergent or cognitive. Third, models can be used in both emergent processes (e.g., as boundary objects to facilitate information sharing) and cognitive processes (e.g., to extend one's cognitive capacity through external storage). However, Model-Driven Engineering (Schmidt 2006), which prescribes generating progressively more detailed models until the final model becomes executable code, is consistent with neither the FBS nor SCI Framework.

Process Theory Differences. Notwithstanding these similarities, the two theories differ in at least the following three ways.

1. Whether problem setting and problem solving are separate (FBS Framework) or cotemporal and inextricably linked (SCI Framework).
2. Whether the coding process is driven by prefigured decisions (FBS Framework) or evolves iteratively with the design process (SCI Framework).
3. Whether designers focus on models (FBS Framework) or code (SCI Framework).

Each of these differences is directly related to the Reason and Action-Centric Perspectives. First, since a problem is a relationship between an actor and a context, problem setting requires interaction with (or at least observation of) the actor and context. Therefore, if design involves both problem setting and solving, design must be emergent, but if problem setting and solving are separate, design may be a separate, entirely cognitive, activity. Second, if design is an entirely cognitive activity then coding is not part of design, because the software code is not part of the designer' cognitive system. Third, if design is essentially cognitive, and the complexity of realistic design problems strain the designer's limited short term memory, one might expect the designer to externalize part of his or her cognition using textual or diagrammatic descriptions of the problem or design object – models. In contrast, if design is emergent, the designer may externalize his or her ideas by building the software. Focusing on the software code, which is outside the designer's cognitive system, is consistent with designing as an emergent phenomenon, while focusing on models or similar cognitive aids is consistent with a designer working primarily within his or her cognitive system. These differences are summarized in Table 4-5.

Table 4-5. Key Differences Between SCI and FBS Frameworks

Dimension	**Reason-Centric Perspective**	**Action-Centric Perspective**	**FBS Framework**	**SCI Framework**
relationship between problem setting and problem solving	separate – since problem setting is emergent, problem solving must be separated to be cognitive	cotemporal and inextricable – both problem and solution emerge from designer's interactions with context	Separate – represented by F, which is given ("Functional reformulation," involves modifying F based ONLY on changes to structure, not based on changes to the sociopolitical context of the designer or problem)	cotemporal and inextricable – Coevolution modifies both mental pictures of context and design artifact
primary iterative artifact	models – used to enhance designer's cognitive system	code – designers focus on the real-world instantiation of the design, i.e., software code	models - referred to as S, Be and Bs (descriptions of structure and behavior). No artifact in FBS corresponds to code	code - the primary iterative cycle is between the context, mental pictures, and code, only the last of which is an artifact
coding progression	linear – since the code is not part of the designer's cognitive system, where design occurs, little iteration is necessary in coding	iterative – the design, expressed through the code, emerges from continual interaction with the context. The emerging design changes the context, necessitating code iterations	Linear - all design decisions are encompassed by the design description, making coding straightforward; hence, little iteration is needed.	Continual changes to the context (some caused by the design activity itself) necessitate iterative revisions to the Mental Pictures and code.

Testability. The Reason and Action-Centric Perspectives are very general, ontological propositions, concerning the nature of a phenomenon. They are not causal hypotheses amenable to experimental testing. They are not directly observable or measurable in a quantitative sense. Yet, they each may be more or less accurate in their description of an observable phenomenon. They are also mutually exclusive: either design occurs primarily within the designer's cognitive system or involves substantial external interactions; no third option is apparent. Furthermore, although they cannot be measured directly, indirect evaluation may be possible through their associated software design process theory.

As described above, FBS is closely related to RCP. It depicts a closed system to which the only input, the set of functions, is provided upfront. The designer than iterates between the set of functions, a structure-description, and two sets of behaviors. All of these are part of the designer's cognitive system. Similarly, ACP is closely related to the SCI Framework. It depicts an open system where the designer continually interacts with the project context and an iterative artifact outside the designer's cognitive system, namely, the design object itself. Therefore, empirically comparing the FBS and SCI Frameworks can produce insight into the corresponding perspectives.

Simply showing the FBS (or SCI) Framework is more accurate does not intrinsically support the Reason (Action)-Centric Perspective. It is also necessary to show that the *reasons* for concluding that one framework is superior are related to the underlying perspective. However, as discussed in the previous section, the three key differences between the process theories directly reflect the tension between the two perspectives. Therefore, comparatively testing the process theories may provide insight into the perspectives.

This paper began with the research question: *What is the nature of the process by which development teams create software in practice?* This question can now be reformulated as *Which of the Reason- and Action-Centric Perspectives more accurately represents how software is created in practice?* The following section describes the research method applied to this question below.

4.3 RESEARCH DESIGN

My literature review did not uncover any previous empirical evaluations of either theory in the software domain. Moreover, I uncovered much less methodological advice on evaluating process theories than is commonly available for variance theories. However, Wolfe (1994) identified two common approaches to studying innovation processes – cross-sectional surveys and in-depth field studies. More generally, Poole et al. (2000) considers three research designs appropriate to studying change processes – cross-sectional survey, panel (longitudinal) survey and process (field) studies. Combining two or more of these approaches enables multi-method triangulation. More generally, numerous authors recommend taking a comparative approach to testability to avoid cherry-picking evidence (Sober 1999; Yin 2003).

Based on this, I propose a two-phase comparative evaluation of the SCI and FBS Frameworks. By focusing on the three key differences described above, each of which maps to a difference between the Action- and Reason-Centric Perspectives, evidence supporting one of the process theories should vicariously support its associated perspective. A cross-sectional survey (phase one) allows for larger sample size and reliability while a multiple case study (phase two) facilitates gathering deep insights into developer behaviors and cognitive processes. This paper focuses on the survey, which I designed along commonly-used guidelines (DeVellis 2003; Fowler 1995; Straub 1989). Possibilities for the case study phase are described in §4.5.5.

4.3.1 Hypothesis

Despite the conventional wisdom favoring Reason-Centric assumptions, I hypothesize that the SCI Framework is more accurate, as its underlying design paradigm (Reflection-in-Action) and theory of human action (Ethno-View) are better supported by empirical studies than their Reason-Centric alternatives (Schön 1983; Suchman 1987). This leads to the following hypothesis and alternative hypothesis.

> ***Hypothesis H_1****: The SCI Framework more accurately reflects how software is created in practice than the FBS Framework.*

> ***Hypothesis H_a****: The FBS Framework more accurately reflects how software is created in practice than the SCI Framework.*

For simplicity of exposition, I state the hypothesis at the theory level (rather than the construct level, as is common for variance theories).

4.3.2 Instrument Development and Validation

The steps in the instrument development and validation were as follows.

1. The author identified differences between the two theories.
2. A colleague with expert knowledge of software design reviewed these differences (see below).
3. The author generated approximately 80 items concerning these differences.
4. Items were reviewed by two faculty members at the author's university – one with extensive experience in questionnaire-based research, the other with extensive knowledge of design.
5. Items were revised and a draft questionnaire was created.
6. A pilot was conducted with three professional developers and seven PhD students in the author's university to get research-oriented feedback.
7. Items were revised to enhance construct validity.
8. A second pilot with 12 professional developers was conducted. Results indicated that the questionnaire was too long and too difficult to understand.
9. The questionnaire was shortened and items were simplified.
10. A third pilot with 10 professional developers was conducted. Feedback indicated that the questionnaire was of acceptable length and readily understood.
11. Minor revisions resulted in the final version of the instrument.

In step two, the software design expert was asked to examine three things: 1) whether any of the proposed differences were unwarranted; 2) whether any salient differences between the two theories were missed; 3) whether any bias in the interpretation of either theory was evident. No unwarranted differences, omissions or biases were reported. Similarly, in steps four and six, reviewers were asked for feedback on the validity of the survey. In this way, the instrument development process was used to promote high content validity.

Following this process, the questionnaire comprised 13 items, each with six responses – one strongly supporting each framework, one supporting each framework, one neutral, and one “Not Applicable / Don’t know.” The question order was randomized; the answer order varied by question. Table 4-6 lists each item and describes its relationship to the process theories and perspectives.

4.3.3 Side Note on Measurement Issues

Please note, as process theory testing is less common than causal theory testing, many readers may be unfamiliar with the interpretation of items in this study. Here I address this issue with a brief exposition.

Using either process theory, one may draw a variety of predictions about design practice. In some cases, both theories lead to the same predicted observation; e.g., design is done by an agent capable of choosing actions to achieve goals. In other cases, differences between the theories lead to contrasting predicted observations; e.g., Difference 1 (whether problem setting and solving are separate), would lead one to predict that the process of designing the software would (SCI) or would not (FBS) improve the team’s understanding of the software’s intended context. Each contrasting prediction may then be encapsulated in a bipolar item where one pole represents the observation consistent with the SCI Framework and the other that of FBS. Consequently, each item concerns a behavioral dimension where the two theories predict contrasting observations.

Based on this understanding of the items, one should expect neither high inter-item agreement nor higher levels of agreement for items associated with the same difference. These items are not separate reflective indicators of the same latent construct; rather, each item concerns a behavioral dimension on which the two theories predict contrasting observations. Since construct validity (in the broadest sense of asking whether the items measure what is intended) could not be evaluated *post hoc* using statistical measures of inter-item agreement, special care was taken in generating and validating the questionnaire, as described in the previous section.

Table 4-6. Questionnaire Items

Item	Dimension	Relationship to Perspectives	Relationship to Theories
No one thing drives all design decisions – they are made based on a variety of information	relationship between problem setting and problem solving	In RCP, problems are given, design decisions are based on problems. In ACP, problems are constructed based on a multifarious context, so design decisions are based on many properties of the context.	In FBS, all design decisions are driven by functions. In SCI, design decisions are influenced by any number of aspects of the context.
Changes to my team's understanding of what the software is supposed to do were triggered by changes in our understanding of the problem/situation		In RCP, problems and goals are given and do not change. In ACP, problem understanding is part of design, so goals are constructed and revised based on a fluctuating understanding of the problem.	In FBS, understanding of the problem (F) is given and does not change over time. In SCI, understanding of the problem and solution coevolve.
My understanding of what the software is supposed to do has been influenced by several factors (e.g., management, marketing, clients, the dev team, standards, my own values, experience on previous products, etc.)		In RCP, the problem is given and requirements are generated solely from the given problem. In ACP, problems are constructed based on multifarious context, so design decisions are based on many properties of the context.	In FBS, expected behavior is determined solely by the known functions. In SCI, requirements are part of the mental picture of the context, and are affected by many contextual variables.
My understanding of the software's purpose has been influenced by several factors (e.g., management, marketing, clients, the dev team, standards, my own values, experience on previous products)		In RCP, someone or something provides the purpose, so the designer's understanding is based on this. In ACP, the purpose is constructed based on many properties of the context.	IN FBS, F is given. In SCI, goals are part of the mental picture of the context, and are affected by many contextual variables.
The process of designing the software has NOT helped my team better understand the context in which the software is intended to be used		In RCP, design is a rational search within a known solution space – it does not include any rethinking of the context of use. In ACP, understanding the problem and designing the solution occur together and contiunially interact	In FBS, changes to S can lead to changes in F, but this "functional reformulation" is unrelated to understanding of context - it reflects new things the artifact could do, or physical impossibilities. In SCI, the two mental pictures coevolve.
I do detailed design: "Exclusively with models" … "Exclusively with code" My team does detailed design: "Exclusively with models" … "Exclusively with code"	primary iterative artifact	In RCP, detailed design occurs within the designer's cognitive system, including his mind, and external representations of cognitive artifacts, i.e., design models. In ACP, detailed design emerges from the continual interaction between designers and the design object; hence, design occurs primarily in the since the context of coding. This is the essence of design as a cognitive phenomenon vs. design as an emergent phenomenon.	In FBS, detailed design is completed within the S-Bs-Be loop and during the documentation process, before coding. S and D are models. In SCI, detailed design is part of the implementation and coevolution processes, and manifests primarily in the code itself.
Which of these is more consistent with how your team does testing? "Exclusively prediction" … "Exclusively observation"		In RCP, testing is a cognitive activity; hence, it involves predicting what the design object will do once created based on the current design specification. In ACP, testing is an emergent activity in which the behavior of the design object is observed in some context.	In FBS, evaluation is the comparison of Bs (predicted behavior) to Be (desired behavior). In SCI, evaluation consists of examining the changes in context produced by the artifact (sensemaking).
A complete, correct specification of low-level design decisions was available before coding began Low-level design decisions were primarily made before the first line of code was written	coding	Without a sufficiently complete specification, linear coding is practically impossible, as working out new details may necessitate unforseeable changes. With a complete specification, iterations are unnecesary and inefficient. Hence, RCP requires an advance spec, ACP does not. Furthermore, in ACP, building the code changes the understanding of the problem, necessitating changes to the design.	In FBS, documentation produces the required specification and coding follows. The construction (coding) step is not shown, and there is no loop back to change structure. In SCI, implementation changes the context, triggering changes to the design; hence the spec cannot be complete until the design object is (nearly) complete.
The software was coded iteratively		In RCP, coding is the straightforward translation of an available, detailed specification (plan) into software code. In ACP, coding changes the understanding of the problem, necessitating code iterations. (See left.)	In FBS, the construction (coding) step is not shown, and there is no loop back to change structure. In SCI, coding coding changes the understanding of the problem, necessitating code iterations (see above).
My team has revised the software code based on new information		In RCP, the problem is given. There is no "new information." Any substantial change to the problem would require replanning and redesign prior to code changes. In SCI, continual changes to the understanding of the context drive code iteration.	FBS depicts a closed system – there is no mechanism by which new information can enter after F is given. In SCI, new information is absorbed via sensemaking and may lead to code changes.
My team now understands what the software is supposed to do better than we did when we started coding		In RCP, problem understanding precedes problem solving, which precedes coding. In ACP, problem understanding and solving are cotemporal.	In FBS, there is no reformulation of F (representing the problem) based on code. In SCI, the designer switches between implementation, sensemaking and coevolution, improving understanding with every evaluation cycle.

Furthermore, please note that although respondents were asked descriptive questions concerning their work, this is a *confirmatory* study. Using descriptive questions to gather evidence for confirmatory research is inherent to survey research methodologies. What separates descriptive or exploratory research from confirmatory research is whether the hypotheses come before or after the data collection (Jaeger and Halliday 1998). In this case, the hypotheses came first; therefore this is a confirmatory study.

4.3.4 Sampling Strategy and Administration

The population of interest includes all members of all software development teams, worldwide. However, for practical reasons, I limited the sample to English speakers. As no comprehensive population list was identified, random sampling was impractical. This left three potential recruiting strategies.

1. Randomly select software development firms from a specific geographical area where a population list is available (several companies, including Manta, maintain such lists).
2. Viral sampling through twitter, the blogosphere and online social networking applications including Facebook and LinkedIn.
3. Snowball sampling using the author's professional network.

The first strategy would be best able to support claims of representativeness but its generalizability would by limited since, for practical reasons, the sample would be limited by geography and culture. The second strategy has the potential to generate a large, diverse response, but the response rate would be incalculable as the sample size is undefined. The third strategy is dominated by the other two – it would be geographically and culturally limited **and** have an undefined response rate. Therefore, I attempted both random and viral sampling.

As the questionnaire was administered online, link tokens tagged the origin (but not the identity) of each respondent. Browser cookies attenuated duplicate responses.

4.3.5 Interpreting the Results

Before presenting the results, I enumerate the possible patterns and their interpretations, assuming responses are coded from 1 (strong support for FBS) to 5 (strong support for the SCI Framework).

1. A symmetric distribution (median 3) would indicate that neither framework is substantially more accurate than the other.
2. A positively-skewed distribution (median 1 or 2) favors FBS.
3. A negatively-skewed distribution (median 4 or 5) favors the SCI Framework.
4. A bimodal distribution (e.g., modes of 2 and 4) would indicate two groups of developers exist – one supporting each framework.
5. A variety of symmetric, positively and negatively skewed items would suggest a problem with the survey instrument.

4.4 RESULTS

4.4.1 Sample and Demographics

Between December 2, 2009 and January 11, 2010, 1384 participants responded to the survey. Random sampling of software development firms produced negligible responses as gatekeepers were unwilling to facilitate contact with developers. Hence, most respondents came from the viral sampling strategy (§4.4.3 gives further sample analysis). Since the sample size is undefined, the response rate cannot be calculated. However, of 4410 individual visitors to the survey page, 1384 completed it (31%), 1118 partially completed it and 1908 bounced (Table 4-7 and Table 4-11 give demographic analysis).

Responses were received from 65 countries across six continents, reporting a variety of roles in their projects, from managers and graphics designers to developers and testers. Respondents also reported using a wide variety of agile (e.g., Scrum) and plan-driven (e.g., Waterfall) methodologies. When asked "is your project more 'social' (like a website) or 'technical' (like a device driver)", participants answered: more social - 34%; more technical - 29%; in between - 36% (further analysis in §4.4.3).

Table 4-7. Summary of Sample Demographics

Dimension	Mode	Minimum	Maximum
Years of Experience	1 to 5 years (31.5%)	< 1 year (2.9%)	> 25 years (3.6%)
Education	Bachelor's Degree (48%)	Some School (1.7%)	PhD (4.1%)
Company Size	1 to 10 (29%)	1 to 10 (29%)	>10 000 (10.5%)
Team Size	11 members	83 members	3000 members
Project Length	1.9 years	2.4 years	20 years

4.4.2 Testing Hypothesis H_1

Many methodologists and statisticians disagree on whether Likert scales produce interval or ordinal data, and consequently on whether to apply parametric or nonparametric tests (Harwell and Gatti 2001). Here, I take the more cautious route, treating this data as ordinal.

The results are given in Table 4-8 and diagramed in Figure 4-3. Visually inspecting Figure 4-3 reveals that the overall distribution favors the SCI Framework. Table 4-8 shows the frequency of responses for each of the 13 items, e.g., for item 9, 17 respondents selected the response strongly favoring the FBS Framework while 620 selected the response strongly favoring the SCI Framework. It also shows that the median response for each item was 4 or 5, indicating support or strong support, respectively, for the SCI Framework. Similarly, 96.6% of respondents had a median response (across the 13 items) of 4 or 5. In summary, the response distribution is negatively skewed, supporting Hypothesis H_1 (SCI Framework supported).

Nonparametric tests (such as chi-square), may be used to evaluate the statistical significance of this distribution; however, these tests require an expected distribution to compare with the observed distribution. Since no *a priori,* theoretically-justified distribution is available, the "expected distribution" must be generated somehow. Three alternatives are apparent.

1. Uniform Distribution - on any given item, responses split evenly between all categories.
2. Pseudo-Normal Distribution - an approximated normal distribution on a five point scale.

3. FBS-supporting distribution - a distribution that favors FBS to the same extent that the observed distribution favors the SCI Framework.

Table 4-8. Questionnaire Results By Item

	Item													Individual	
	1	2	3	4	5	6	7	8	9	10	11	12	13	Median	Mode
Strong FBS	7	13	14	20	62	22	22	13	17	58	23	13	9	0	2
FBS Framework	38	66	42	76	161	61	97	39	63	168	173	174	67	4	21
Neutral	72	162	109	120	195	78	113	55	122	148	320	299	303	43	32
SCI Framework	597	662	576	572	572	398	539	452	539	492	671	623	562	932	717
Strong SCI	656	446	628	575	349	819	592	796	620	505	173	155	425	406	613
Item Median	4	4	4	4	4	5	4	5	4	4	4	4	4		
Item Mode	5	4	5	5	4	5	5	5	5	5	4	4	4		

Note: columns do not total 1384 as each question had a "N/A" option.

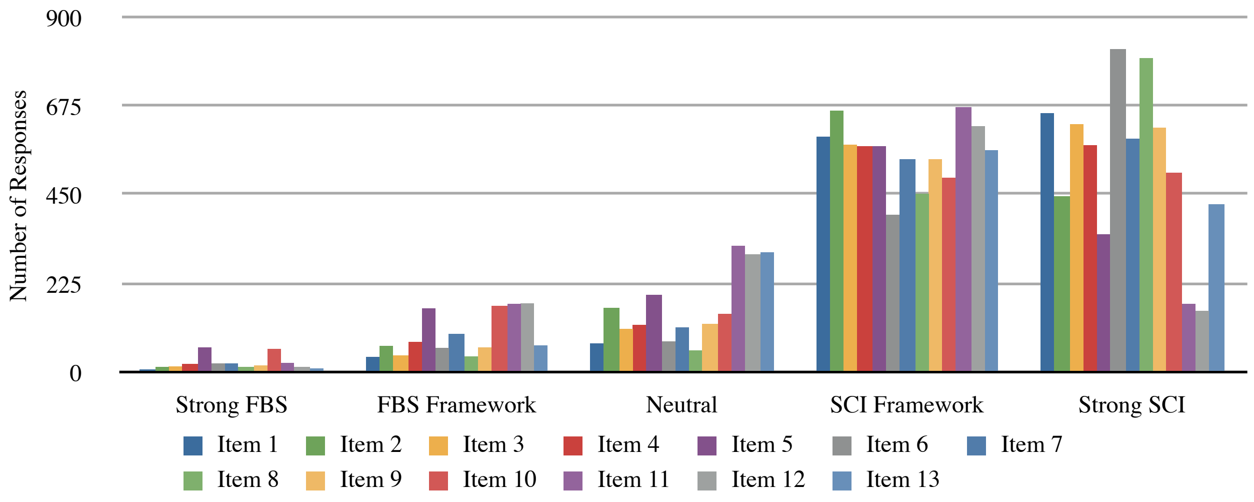


Fig. 4-3. FBS/SCI Agreement Across 13 Items

The uniform and normal distributions are automatically generated from the data by the SPSS Statistics software package when using the Kolmogorov-Smirnov (K-S) Test. I generated the FBS-supporting distribution by reflecting the observed distribution (subtracting each response from 6). Using the normal distribution addresses the question *is the extent of negative skew in the observed distribution significant?*

Using the FBS-supporting distribution addresses the question, *is the observed distribution significantly different from an equally compelling distribution supporting the alternative hypothesis?*

Table 4-9 shows the results of comparing the observed distributions with each of the three expected distribution alternatives. The results across all items and expected distributions are significant at the $p < 0.001$ level. However, generating a pseudo-normal distribution using the K-S test involves calculations (e.g., mean) inconsistent with interpreting Likert-scales as ordinal data; therefore, these statistics should be interpreted with caution. The most defensible test is the chi-square goodness of fit test using the reflected (FBS-Supporting) distributions – with the significance via the sign test – as this compares Hypothesis H1 (SCI-Framework more accurate) with the alternative hypothesis (FBS-Framework more accurate) with a minimum of assumptions. Moreover, the practical significance of the observed distribution should be evident from visual inspection of the above histogram (Figure 4-3).

Table 4-9. Chi-Square Test Results

	Uniform Distribution		**Pseudo-Normal Distribution**		**FBS-Supporting Distribution**	
Item	**K-S Test Z**	**Significance**	**K-S Test Z**	**Significance**	**Sign Test Z**	**Significance**
1	24.59	p < 0.001	10.45	p < 0.001	-33.49	p < 0.001
2	21.00	p < 0.001	10.35	p < 0.001	-29.9	p < 0.001
3	23.21	p < 0.001	9.851	p < 0.001	-32.21	p < 0.001
4	21.90	p < 0.001	9.727	p < 0.001	-29.92	p < 0.001
5	15.92	p < 0.001	10.25	p < 0.001	-20.47	p < 0.001
6	23.51	p < 0.001	12.64	p < 0.001	-31.45	p < 0.001
7	21.32	p < 0.001	9.648	p < 0.001	-28.53	p < 0.001
8	24.72	p < 0.001	12.57	p < 0.001	-33.18	p < 0.001
9	22.12	p < 0.001	9.677	p < 0.001	-30.48	p < 0.001
10	17.70	p < 0.001	9.837	p < 0.001	-22.13	p < 0.001
11	13.65	p < 0.001	10.82	p < 0.001	-20.1	p < 0.001
12	12.94	p < 0.001	10.35	p < 0.001	-18.84	p < 0.001
13	17.50	p < 0.001	8.782	p < 0.001	-27.87	p < 0.001

Given the sample size and magnitude of skewness of the observed distribution, these results are fairly robust against minor changes in the expected distribution. Again, although measures of effect size are not available for non-parametric tests, given that 96.6% of individuals had a median response agreeing or strongly agreeing with the SCI Framework, it would be fair to say that the effect size is practically significant.

In summary, Hypothesis H_1 (the SCI Framework more accurately reflects how software is created in practice than FBS) is supported and the effect is statistically significant. In the following section, I report some descriptive statistics and exploratory findings that may prove useful for theory-building or inspiring new inquiries.

4.4.3 Exploratory Analysis

In addition to the thirteen items, several demographic and project variables were included in the questionnaire, both to characterize the sample and determine whether differences in agreement with the FBS or SCI Framework were associated with any particular demographic or project dimension. Analyzing this relationship requires a measure of an individual's overall orientation toward the frameworks.

The Dependent Variable. For the following analyses, an individual's process theory agreement score (or simply "score") is defined as follows.

> ***Process Theory Agreement Score (score):*** *a bipolar measure of the extent to which the individual's beliefs about his or her work practices conform to either the FBS or SCI Frameworks, operationalized as the respondent's median response across the 13 items.*

The score indicates how well a respondent's beliefs correspond to the assumptions of the FBS or SCI Framework on a scale of 1 (FBS) to 5 (SCI).

Development Methods. Perhaps the most surprising finding to the author concerns the methods used by respondents. Respondents were asked, in an open-response format, to list any development methods they were currently using. Of the 684 respondents who answered this question, many indicated that they use or draw from a variety of methods. Some listed things that are not methods, such as "python" (a language) and "MVC" (the Model-View-Controller architectural design pattern). A few took the question as an opportunity to denounce “buzz words” or specific methods. Responses were coded by the author according to a simple scheme – a responded was considered to have been influenced by any method mentioned except methods mentioned explicitly in the negative (“We use Agile but not XP,” would be coded Agile = True; Extreme Programming = False). This coding was sufficiently simple that it did not warrant a second coder. The results are summarized in Table 4-10 – “Median Score” refers to the median of the scores of all respondents indicating the influence of the corresponding method.

Table 4-10. Median Score by Methods Used

Method	N	Median Score
Scrum	323	4
Extreme Programming (XP)	130	4
Agile	115	4
Test Driven Development (TDD)	89	4
Service Oriented Architechture (SOA)	79	4
Other	62	4
Waterfall	47	4
Kanban	17	4
Lean	16	5
Rational Unified Process (RUP)	14	4
“Cowboy Coding” or “Seat-of-the-Pants”	11	4

Notes: 1) Methods mentioned by fewer than ten respondents are grouped under “other.” 2) Some of the “methods” listed by respondents are not technically methods, e.g., TDD is a technique associated with XP; SOA is an architectural design pattern.

These results are surprising in at least three ways.

1. *Agile methods and concepts dominate the list*. Scrum, XP, Test Driven Development and Lean are all explicitly Agile philosophies. While Agile is often treated as a fringe movement by research, managers and textbooks (e.g., Baltzan and Phillips 2008), these results suggest that Agile methods are more popular than plan-driven methods such as Waterfall and RUP.
2. *Methods used had no practical effect on scores*. Since Waterfall and RUP are closely linked to FBS (Kruchten 2005), one would expect developers using these methods to have lower median scores

than developers using Agile methods. However, the median score was four for all methods expect Kanban. This clearly undermines the possibility of a plan-driven subculture that is better described by FBS.

3. The Waterfall Model explicitly mandates separating analysis from design and designing using models rather than code. These are assumptions of FBS. The SCI Framework explicitly posits that analysis and design are cotemporal and inextricably linked and that designers iterate primarily on code rather than models. The Waterfall Model and SCI Framework are manifestly incommensurate; one cannot use the Waterfall model and act in accordance with the SCI Framework at the same time. Yet, the median score for respondents using the Waterfall model was four, indicating agreement with the SCI Framework. One explanation for this is that the respondents claiming to use the Waterfall model are not actually using it – a suggestion supported by several previous studies (McCracken and Jackson 1982; Parnas and Clements 1986; Walz et al. 1993). Another possibility is that respondents understand the term "Waterfall model" differently than academics.

These method use frequencies should be interpreted with caution for two reasons. First, the purpose of this survey was to evaluate the two process theories, not to estimate method usage. The question format (open response) and sampling strategy may have introduced bias into these estimates. Second, this question does not allow inferences concerning the prevalence of homegrown methods or amethodical development (Truex et al. 2000). Although 40 respondents explicitly indicated use of no method, or a homegrown method, more than half of respondents left the question blank, which may indicate either question-skipping or use of no method. Respondents may also have difficulty distinguishing between homegrown methods and an amethodical design process. Again, the primary purpose of this analysis is to evidence that the methods respondents report using have no practical effect on their Process Theory Agreement Score.

Demographic Dimensions. As noted above, participants were asked about their gender, education, and experience. None of these factors had a significant effect on individual median responses; put more precisely, for every category of each factor the median score was four (Table 4-11). Moreover, while the sample includes a wide variety of countries, median score did not vary by country (Table 4-12).

Table 4-11. Median Score by Demographic Dimension

Dimension	Category	N	Median Score
Gender	Male	1241	4
	Female	56	4
	Other	3	4
Education	Some school	18	4
	High school Diploma	53	4
	Some college, university, trade school, etc.	198	4
	Diploma from technical college, trade school, etc.	62	4
	Bachelors degree	626	4
	Masters degree	289	4
	Doctorate degree	54	4
Experience	less than 1	31	4
(years)	1 to 5	399	4
	6 to 10	405	4
	11 to 15	274	4
	16 to 20	89	4
	21 to 25	54	4
	more than 25	48	4

Table 4-12. Median Score by Respondent Country

Country	N	Median Score	Country	N	Median Score	Country	N	Median Score
United States	549	4	Netherlands	33	4	New Zealand	16	4
Canada	176	4	India	23	4	South Africa	16	4
other	174	4	Italy	19	4	Romania	14	4
United Kingdom	118	4	Sweden	19	4	Brazil	13	4
Australia	73	4	Beligium	17	4	Spain	12	4
Germany	44	4	Poland	17	4	Israel	11	4
France	40	4						

Note: Countries with fewer than ten responses are grouped as "other."

One danger in viral sampling is the possibility that the invitation to participate will circulate within only one community, seriously curtailing the generalizability of the results. To avoid this, I attempted to inject invitations into several communities, not unlike the strategy of starting a heuristic optimization algorithm in several different parts of the solution space to avoiding sticking in the same local optimum. As shown in Table 4-13, most responses came from one of two starting points: 1) Brett Cannon, Chair of the Python Software Foundation promoted the survey via both his coding blog and Twitter account; 2) Jeff Atwood, author of the blogs Coding Horror and Stack Overflow, promoted the survey via Twitter. While Mr. Cannon primarily writes about issues specific to the language Python, Mr. Atwood covers a broad range of programming-related topics. However, starting points had no effect on median Score. As sampling origin was tracked using link tokens, the origin of respondents who manually typed the survey address or otherwise removed the tokens is unknown.

Table 4-13. Median Score by Sampling Origin

Invitation Injection Point	N	Median Score
Brett Cannon's Twitter and Python Blog	686	4
Jeff Atwood's Twitter	515	4
other/unknown	102	4
Jurgen Appelo's Twitter	69	4
Damien Guard's Twitter	12	4

Note: Injection points resulting in fewer than ten responses and responses having no URL tokens are grouped as "other/unknown."

Project-Specific Dimensions. One may expect that larger projects would be heavier in planning and modeling and therefore more consistent with FBS. However, when respondents were divided into two groups (median < 4 and median >= 4), respondents with neutral or FBS-centric dispositions had *lower* mean project and team sizes, but this difference was not significant (Table 4-14). Participants were also asked whether their projects were more social or technical in nature. Neither this nor firm size had any effect on scores (Table 4-15).

Table 4-14. Independent Samples T-test for TeamSize and Project Length

Variable	Median Score	N	Mean	Std. Deviation	Std. Error Mean	t	p
TeamSize	≥ 4.00	1290	11.10	84.85	2.36	0.382	0.703
	< 4.00	54	6.69	6.92	0.94		
ProjectLength	≥ 4.00	638	1.94	2.38	0.09	0.540	0.589
	< 4.00	18	1.63	1.71	0.40		

Table 4-15. Median Score by Project-Specific Dimensions

Dimension	Category	N	Median Score
Firm Size	1 - 10	393	4
(employees)	11 - 100	374	4
	101 - 1000	219	4
	1001 - 10 000	170	4
	> 10 000	144	4
Nature of Project	More Social	444	4
	Somewhere in between	475	4
	More Technical	372	4
	NA	9	4

Similarly, while respondents indicated a variety of roles in the project and current occupations, no effect on scores was discernible (Table 4-16).

Table 4-16. Median Score by Role in Project and Occupation

Role			Occupation		
Response	N	Median Score	Response	N	Median Score
developer	1325	4	developer	1211	4
analyst	569	4	team lead	440	4
QA	533	4	manager	269	4
manager	266	4	analyst	197	4
graphics	195	4	QA	165	4
			graphics	59	4

Note: Respondents could select multiple roles and occupations.

Summary. In conclusion, the median individual Process Theory Agreement Score was four, indicating that, on average, respondents report behaving in accordance with the SCI Framework. No demographic nor project-specific variable had a practically significant effect on this result. Despite being heterogenous on all demographic and project variables, the surveyed developers are practically homogenous concerning the process theory where views and behavior coincide.

4.5 Discussion and Conclusion

4.5.1 Contributions

Three contributions of this research are evident. Specifically, it supports the Action Centric Perspective and SCI Framework, and provides a proof-of-concept of a method for testing process theories.

The results of this study supports ACP. This implies a refutation of the privileged position of design as cognition, design as rational action and design as methodical action. The presented evidence lends further support to a growing body of evidence debunking procedural rationality and planning as the basis of design and other professional activity (e.g., Love 2000; Nandhakumar and Avison 1999; Parnas and Clements 1986; Schön 1983; Truex et al. 2000; Zheng et al. 2007). Moreover, these results undermine claims that the phases and sequence of the Systems Development Lifecycle (which are subsumed by FBS) are in any way inherent to software design – rather, the activities inherent to software design are sensemaking, coevolution and implementation.

Second, the study supports the SCI Framework. To the best of my knowledge, this study is not just the first empirical evaluation of the FBS and SCI Frameworks but of *any* software design process theory.

Third, this study provides a proof-of-concept for a new method of testing process theories. Although some authors have previously recommended survey studies to test process theories (Poole et al. 2000; Wolfe 1994), the method used here combines three more specific principles:

- **Test Comparatively.** Simultaneously testing two plausible explanations of the same phenomenon helps to mitigate the proclivity to cherry-pick evidence, focuses the study on controversial elements (differences between theories) and guards against overemphasis of statistically significant but practically insignificant findings.
- **Bipolar Items.** When testing two theories comparatively, one can create bipolar items such that each theory is represented by one of the two poles.
- **Distribution Interpretation.** The skewness of the distribution indicates which theory is favored. The extent of this skewness indicates the effect size.

One challenge of this type of research is in determining whether the skewness of the distribution is statistically or practically significant. In this study, the skewness was so pronounced and the sample size so large that significance was obvious. However, in smaller sample sizes and less skewed distributions, I suggest that the most appropriate comparison is between the observed distribution and its reflection. Comparing an observed distribution against a normal distribution is akin to evaluating the hypothesis *Variables A and B are related* against the alternative hypothesis *Variables A and B are unrelated*. When taking a comparative approach to testability, the hypothesis is *Theory X is better* and the alternative hypothesis is *Theory Y is better*. A normal distribution does not represent this alternative hypothesis; a reflected distribution does. Comparing the observed distribution to a normal or pseudo-normal distribution tests the hypothesis against the conjecture that Theory X and Theory Y are not statistically differentiable on the dimension in question. Comparing the observed distribution to a reflected distribution test H_1 against H_a. However, as I have done above, one may include a variety of statistical comparisons and leave their interpretation to the reader.

4.5.2 Implications for Academics and Managers

Academics. As the epistemological stance employed here is based on comparative testability (Sober 1999), results favoring the SCI Framework are taken as direct support for this theory. Hence, the implications described in this section are based on this philosophy and readers coming from different philosophical stances may disagree. This caveat noted, three implications for academics are evident. The finding that design is emergent rather than cognitive has one primary implication for academics: paradigms, theories, research design and design methodics that implicitly or explicitly assume that design is cognitive should be heavily scrutinized. In addition, the SCI Framework is immediately useful for research in at least three ways.

1. In a sense, the SCI Framework is a first draft of an Action-Centric design process theory. Hence, it requires further testing (including in-depth field studies), fleshing out of some under-developed concepts (including the internal dynamics of design agents), and extending to domains similar to software (e.g., as greenfield engineering projects).
2. As this study undermines the Reason-Centric Perspective, it motivates critical examination of many things based on Reason-Centric assumptions, including theories, paradigms, design methods, tools, textbooks and educational curricula. For example, textbooks presenting design as a sequence of analysis, design and implementation (e.g., Baltzan and Phillips 2008) may be misleading.
3. It may inform development of an antecedent theory of design project success. In a strict interpretation of causality, causal theories imply precedence relationships. The SCI Framework dispenses with Waterfall-like, artificial activity sequences. Therefore, it may help to eliminate extraneous causal relationships during theory building (e.g., the hypothesis that analysis quality causes design quality is incorrect *a priori* as analysis and design are cotemporal in practice).

Managers. For managers, assuming that design is emergent, rather than cognitive, implies that designers will be more effective if they have better access to the design object's intended environment, including stakeholders and information about their sociopolitical context. Furthermore, the SCI Framework posits that problem understanding and problem solving are cotemporal in practice, that code is written iteratively and that the primary iterative artifact used by developers is source code (rather than m odels). This does not mean that trying to separate analysis from design, write code linearly or iterate on models is less effective. However, it does suggest the following.

1. Developers may resist attempts to pressure them to separate analysis from design, write code linearly or iterate on models. Developers may fake adherence to design methodologies that are inconsistent with their natural way of working (Parnas and Clements 1986).
2. Managers who believe that their employees build software according to RCP (that is, rationally) or using a Reason-Centric method (e.g., Waterfall) may be mistaken or deceived.
3. Implementing a tool, practice or method that is incompatible with iterative coding and simultaneous analysis and design will likely be ineffective without corresponding changes in development practices.
4. If developers do not understand the problems that they are solving until the solution is well into development, it seems incredulous that they could produce accurate upfront budget and schedule estimates. This questions the wisdom of organizing development using fixed price and schedule contracts.
5. Since many computer science, computer engineering and information systems programs teach development according to RCP, new graduates may be deeply confused as to the nature of development practice.

4.5.3 Lessons Learned Regarding Viral Sampling

One challenge of this research was how to get a large number of busy professionals to take the time to respond to a survey without any economic incentive. To this end, I devised a strategy comprising four principles – the “four Ns” of viral sampling, listed below. Although I followed this strategy and subsequently received a substantial number of responses, this obviously does not prove the strategy’s effectiveness. I simply report these ideas hoping that they may benefit researchers with similar sampling challenges.

1. **The Name.** At the time of writing, almost half a million people have taken the “Personality Type Quiz” on Quibblo.com. I suspect that “Personality Type Quiz” sounds far more interesting to most people than, for example, “Software Project Work Practices Questionnaire”. Instead of a dry,

academic name, I marketed the survey as "Software Developer Personality Test," which sounds interesting as it addresses one's own personality and is specific to one's occupation.

2. **The Nudge.** In behavioral economics, a *nudge* "is any aspect of the choice architecture that alters people's behavior in a predictable way without forbidding any options or significantly changing their economic incentives" (Thaler and Sunstein 2009, p. 6). Asking an individual to participate in a study presents a choice architecture. I encouraged participation by promising immediate feedback – at the end of the study, participants were classified into one of three "developer personality types" based on their answers. This eliminated the need for an economic incentive.
3. **The Network.** Online social networks, including Twitter, Facebook and the blogosphere in general, allow the astute researcher (or marketer) to exploit network effects in promoting a survey. For instance, when Mr. Atwood tweeted about my questionnaire, many of his more than approximately 27 000 followers not only responded but also retweeted the link, that is, sent it to all of their followers. This positive network effect effectively crowd sourced much of the invitation work.
4. **The iNquiry**. This questionnaire concerned a conflict between two perspectives on software design. As this conflict manifests as arguments within software projects and developer education, it engages developers. When asking people to participate or to encourage others to do so, having a question relevant to their daily lives seemed to help.

In post-hoc analysis of Twitter traffic, I did not identify a single instance where an individual re-tweeted the invite from someone with fewer followers. This leads responses to dwindle over time. It also suggests that Twitter's social network topology approximates a hub-and-spokes network (an interesting topic for future research). If this is the case, overall responses and avoiding sampling bias both depend on the number of hub twitterers recruited.

4.5.4 Limitations

The results of this study should be considered in light of five limitations:

1. This study compared the SCI Framework to a particular interpretation of FBS, Galle's (2009) nominalist interpretation. Therefore, the evidence presented applies only to this interpretation – it does not generalize to other FBS interpretations such as Galle's realist interpretation, or Kruchten's (2005) software adaptation. Again, Galle's nominalist interpretation was selected because it was most compatible with RCP.
2. The sample is not random and may include some bias. For example, the popularity of agile methods displayed in the sample has at least two explanations: 1) Agile methods have largely replaced plan-driven methods in development practice; 2) Users of agile methods are over-represented in the sample. If the second explanation holds, the conclusion that the SCI Framework better represents design practice may be over-generalized. However, the finding that respondents who specifically report using plan-driven methods score no differently than those using agile methods undermines this challenge. A more thorough study of the issue could be accomplished by conducting followup surveys or field studies on a sample specifically chosen for its heavily plan driven methods, such as aerospace and defense contractors.
3. The limitations inherent to survey research, including lack of depth and responder bias, obviously apply here. Phase two of the study (described below) is designed to mitigate these shortcomings.
4. As the test was comparative, the results do not indicate that the SCI Framework is unequivocally "right" or "true"; the results simply favor the SCI Framework over its alternative.
5. Exploratory analyses should be interpreted with caution. While substantial effort was made to collect and to report the data in an unbiased fashion, these results are incidental to the study's primary purpose; consequently, they do not enjoy as high a degree of confidence as the primary conclusion.

4.5.5 Future Work (Phase 2)

As mentioned above, a multimethodological research design combining a survey with one or more in-depth fields studies would provide more convincing evidence than either approach alone. Following this, the next phase of the current study involves comparatively evaluating the FBS and SCI Frameworks using a field study to corroborate (or to contradict) and to add nuance to current evidence.

One form of field study with a rich methodological foundation in organizational research is the case study (c.f. Benbasat et al. 1987; Dube and Pare 2003; Eisenhardt 1989; Lee 1989; Yin 2003). A case study is a "comprehensive research strategy" that "investigates a contemporary phenomenon within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident … [and] relies on multiple sources of evidence, with data needing to converge in a triangulating fashion" (Yin 2003, p. 13-14). A case study approach is preferable when 1) the research focuses on how things are done in practice, 2) the research focuses on contemporary events and 3) the research does not necessitate behavioral manipulations (Yin 2003). The present situation clearly meets these criteria.

I propose a three-case design comprising two literal replications and one theoretical replication (two studies of software development teams where the same result (SCI Framework superior) is predicted and one study of an engineering design team, where a different result (FBS Framework superior) is predicted). The proposed design is informed by the incisive summary of recommendations in (Yin 2003). Data collection may include interviews, recording meetings, direct observation and copying relevant artifacts (e.g., design diagrams). The resulting collection of statements, observations and artifacts can be then coded according to a closed coding scheme based on the two theories. Specifically, for each concept and relationship of each theory, related items of evidence would be classified ether as *supporting* or as *opposing*. The extent of support for each theory would reflect the cumulative support for each concept and relationship. At least two coders will be used to facilitating measurement of reliability through intercoder agreement (Benbasat et al. 1987; Dube and Pare 2003; Lee 1989; Yin 2003).

Furthermore, the SCI Framework is in a first draft of sorts. After further testing, it may be improved and fleshed out with new concepts, or with concepts borrowed from other fields such as architecture, engineering and computer science. Furthermore, it may be used to study the relationship between software design and design in other fields.

4.5.6 Concluding Remarks

Simon (1996) argued that the shape and organization of the design process is an essential component of a theory of design. Since “the shape and organization of the design process” in the software domain is poorly understood (Freeman and Hart 2004; Simon 1996; Sullivan 2003; Wynekoop and Russo 1995), this study began with the question, *What is the nature of the process by which development teams create software in practice?*

In this paper, I addressed this question at two levels. At the more abstract level, this study investigated whether software design is essentially cognitive or emergent. At the less abstract level, this study investigated whether software design is better described by the SCI Framework or FBS. The SCI Framework was explicitly created for compatibility with the view that design is emergent (ch. 3). The degree to which FBS is compatible with the view that design is cognitive depends on how it is interpreted; therefore, I used the most compatible interpretation, the nominalist interpretation (Galle 2009). Both the FBS and SCI Frameworks are telelological (goal oriented) process theories of design.

The results of a survey of more than 1300 programmers, analysts, testers and managers indicate that their work practices are essentially emergent and better described by the SCI Framework, where design is modeled as an improvised, emergent activity wherein a self-directing agent alternates between three primary activities: 1) making sense of context; 2) iteratively evolving mental pictures of context and software artifact; 3) writing code based on the mental picture of the software. Since the view of design as a cognitive phenomenon has held a privileged position in design literature for many years, this evidence calls into question much of the field’s conceptual research, the potential usefulness of popular design methodologies and the conventional wisdom surrounding how software designers are educated and how software projects are managed.

CHAPTER 5: A SOFTWARE DESIGN SCIENCE RESEARCH PROGRAM FOR THE NEXT DECADE

This dissertation addresses three fundamental problems afflicting software design science. I addressed the lack of a consistent demarkation of design phenomena by conducting a meta-analysis of definitions of design and proposing conceptual models of design and software design projects (ch. 2). I addressed the difficulties of understanding and operationalizing different perspectives on the nature of design by synthesizing the Reason and Action-Centric Perspectives and operationalizing them using the FBS and SCI Frameworks, respectively (ch. 3). Finally, I addressed the question of whether design is predominately cognitive or emergent using a survey study, which supported the Action-Centric Perspective and SCI Framework (ch. 4).

Now that a veracious software design process theory is available, numerous areas of software design science research are evident. This chapter discusses four such areas (see Figure 5-1) and constructs a list of topics for software design science.

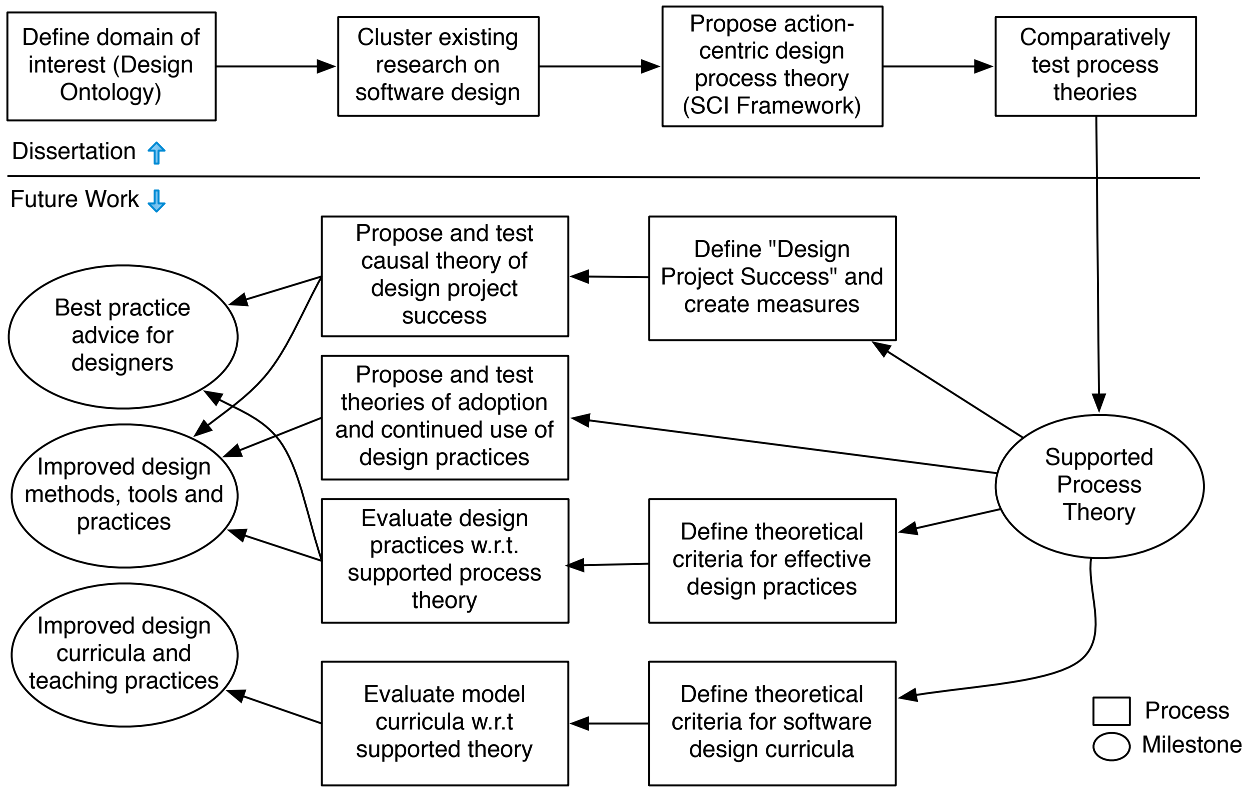

Define domain of interest (Design Ontology)
Cluster existing research on software design
Propose action-centric design process theory (SCI Framework)
Comparatively test process theories
Dissertation
Future Work
Propose and test causal theory of design project success
Define "Design Project Success" and create measures
Best practice advice for designers
Propose and test theories of adoption and continued use of design practices
Improved design methods, tools and practices
Supported Process Theory
Evaluate design practices w.r.t. supported process theory
Define theoretical criteria for effective design practices
Improved design curricula and teaching practices
Evaluate model curricula w.r.t supported theory
Define theoretical criteria for software design curricula
Process
Milestone

Fig. 5-1. A Software Design Science Research Program

5.1 Design Project Success

The first area involves developing a variance theory of design project success. Obviously, determining what factors lead to successful design outcomes would be relevant to practitioners.

I suggest that one fundamental question of software design science is, ‘how to design better?’ One way of answering this question is with a theory of the antecedents of better design. I suggest that Design Project Success (DPS) is a suitable dependent variable for such a theory. Informally, DPS is a multidimensional measure of the outcomes associated with a design project. Design Project Success is a broader, more meaningful concept than design success because the former includes both process and product considerations. Defining DPS may require combining aspects of stakeholder theory (Freeman 1984), the information systems success construct (DeLone and McLean 1992, 2003), software quality metrics (Boehm 1978) and project management success (Shenhar et al. 2001). Once DPS has been clearly defined, instrument development studies may be necessary to devise a methods of measuring it. Next, a meta analysis of the critical success factors literature (cf., Reel 1999) may facilitate theory generation. Theory testing may involve some combination of survey, field and experimental research. This gives rise to the first topic for software design science.

Topic 1: *Theories of the antecedents of design project success*

5.2 Adoption and Use of Design Methodics

The second area concerns adoption and use of design practices, tools and methods. Adoption and continued use are popular topics in information systems research. The primary theoretical finding of this literature is that adoption and use are primarily determined by effort and performance expectancy (Venkatesh et al. 2003), that is, users’ expectations about the amount of effort using the technology will require and the resulting impact on their performance. However, I suspect that design methodics are adopted and abandoned through a complex negotiation between developers and managers (cf., Beck

2005) where each group perceive different value propositions. I further conjecture that these value propositions are largely determined by cognitive biases including the bandwagon effect, confirmation bias and illusion of control, rather than the characteristics of, and evidence supporting, the design methodics themselves. In support of this conjecture I point to the strong feelings often expressed by managers and developers concerning the effectiveness of design methodics that have never been subjected to any systematic empirical validation. These strong beliefs absent of strong supporting evidence indicates low epistemic rationality (Stanovich 2009) among participants. Thoroughly investigating this process through grounded-theory research or surveys and field studies may inform generation of an alternative theory of adoption and use, which may then be tested against existing adoption and use theories in the software development domain. Moreover, understanding the relationship between features of, participants' beliefs concerning, and subsequent adoption and use of design methodics may reveal conditions beyond effort and performance expectancy required for good design methodics to proliferate. This gives rise to the second topic for software design science.

> ***Topic 2:*** *Theories of the adoption and use methods, practices and technologies supporting design*

5.3 Effectiveness of Design Methodics

The third area involves examining the effectiveness of existing design practices, tools and methods to make practical recommendations concerning their use and improvement. As described in Chapter One, empirical research on design methodics is extremely challenging. It is much easier and more efficient to evaluate design methodics vis-à-vis the design process knowledge encapsulated in the SCI Framework. In this way, obviously ineffective design methodics may be eliminated *a priori*, allowing empirical work to concentrate on legitimate contenders.

Concerning methods, suppose a method is denoted "complete" if it contains substantial prescriptions concerning all three fundamental design activities: sensemaking, coevolution and implementation.

Furthermore, Chapter Four provides some indication of the methods most commonly used in practice. By scoring the completeness of these methods we can gauge their *potential* usefulness. For example, a method providing advice on all three fundamental design activities has higher *potential* usefulness than one providing advice on only one activity; however, actual usefulness remains an empirical question. Still, showing where methods are lacking is an important step toward improving them. This suggests two more topics for software design science

Topic 3: *Theories or measures of design method completeness*

Topic 4: *Completeness evaluations of common and novel design methods*

Mathiassen observed that "current systems development methods … do not deal much with the crucial issue of invention or what we have called the invention leap, with how one goes from the analysis of the existing system to the concept of the new one" (1998, p. 102). In the SCI Framework, Mathiassen's "invention leap" is akin to coevolution. Furthermore, anyone familiar with the most commonly used methods (i.e., Scrum, Extreme Programming, Test Driven Development, Service Oriented Architecture) will recognize that they provide little to no guidance concerning coevolution. This raises the question, what sort of advice *can* be given concerning coevolution?

Simon (1996) and Hevner et al. (2004) advise structuring design as a search process within a multidimensional solution space. The conceptual problem with this is that real solution spaces are unbounded and the dimensions may be unknown. Furthermore, people seldom engage in fully-disjunctive reasoning (Stanovich 2009); i.e., considering the full range of possibilities. Indeed, prescribing a separation of analysis from design, as in the Systems Development Lifecycle, may have been motivated by the desire to promote fully disjunctive reasoning. However, since analysis and design are inextricably linked in practice, it may be more effective to devise specific design practices intended to attenuate known cognitive biases (another topic for software design science). Some such practices already exist; for example, some developers estimate task complexity by laying down estimation cards simultaneously to avoid anchoring bias (Tversky and Kahneman 1974).

Topic 5: *Specific design practices intended to attenuate known cognitive biases*

Similarly, specific technologies may assist in each of the fundamental design activities. Sensemaking may be aided through technologies such as diagrams software, qualitative coding software, whiteboards, and physical story cards. Meanwhile, implementation may be aided by design patterns, automated testing software and a host of integrated development environments. The precise effects of each of these technologies on the overall design process warrants careful examination. In contrast, I am not aware of any software tool designed to facilitate coevolution. Hence, topic six:

Topic 6: *Technologies supporting the coevolution activity*

5.4 Reforming the Software Design Curricula

The fourth area encompasses comprehensive analysis and reform of software design curricula. If sensemaking, coevolution and implementation are the primary activities engaged in by software designers, their education should include guidance on each of these activities. While a comprehensive study of various software-related degree programs in hundreds of universities and technical colleges in dozens of countries would be informative, it would be more practical to start with model curricula, such as those offered by the Association for Computing Machinery[1], and the Association for Information Systems[2], and accreditation requirements of bodies such as the Association to Advance Collegiate Schools of Business.

Reforming the software design curricula requires a process comprising at least three stages. Firstly, a meta-analysis of existing, empirically validated design guidance may be conducted, with guidance coded

[1] http://www.acm.org/education/curricula-recommendations

[2] http://blogsandwikis.bentley.edu/iscurriculum/index.php/Main_Page

according to a scheme based on the SCI Framework. Secondly, a sample of existing model curricula and requirements may be coded according to the same scheme. Thirdly, the results of these analyses may be compared, revealing in the curricula both the absence of useful guidance and the presence of unvalidated concepts. This should produce specific recommendations for improving the model curricula, another topic crucial to software design science research.

***Topic 7:** Recommendations for improving the education of software designers*

Integrating better design guidance into model curricula and especially accreditation requirements will generate intense pressure on both accredited and accreditation-seeking schools to incorporate better design education into their programs. However, software developers are not like engineers – they come from a variety of educational backgrounds and need not seek professional certification. Developing professional degree programs and certifications for software developers may be pragmatically necessary to substantially improve design education outcomes.

5.5 A Research Agenda for Software Design Science

In this chapter, I enumerate a seven-part research program for software design science in the next decade. With a concerted effort, I believe most or all of these topics can be addressed within the next ten years. However, this requires numerous researchers to concentrate on these and other pragmatically critical issues, resisting the temptation to migrate toward easier but less important studies. Although not comprehensive, the following list touches on many key areas for software design science.

The Software Design Science Research Agenda

1. Theories of the antecedents of design project success
2. Theories of the adoption and use methods, practices and technologies supporting design
3. Theories or measures of design method completeness
4. Completeness evaluations of common and novel design methods

5. Specific design practices intended to attenuate known cognitive biases
6. Technologies supporting the coevolution activity
7. Recommendations for improving the education of software designers

In addition to these topics, the SCI Framework may be useful outside of software. Applying or adapting the SCI Framework to other design disciplines, such as architecture, product design and engineering, would require coordinated studies to understand how these disciplines differ from software. As discussed in Chapter Three, disciplines may undergo predictable shifts from Action-Centric to Reason-Centric ideals as more Reason-Centric assumptions are met by expanding scientific knowledge. Furthermore, a discipline's adherence to Reason- or Action-Centric ideals may be related to identifiable factors, such as the discipline's maturity, the complexity of the design objects and the experience of designers.

Many authors have called for empirical work on software design (cf. Freeman and Hart 2004; Simon 1996; Sullivan 2003; Wynekoop and Russo 1995, 1997), yet, as I have shown in Chapter Four, the literature is still largely non-empirical. The idea that design is an inherently cognitive activity continues to occupy a privileged status, underlying many prominent concepts in design literature. This situation is analogous to the primacy of the expected value hypothesis and theory of rationality in the economics literature before Kahneman and his colleagues debunked these taken-for-granted concepts (Kahneman and Tversky 1979; Tversky and Kahneman 1974). In this dissertation I have attempted to provide a similar empirical refutation of the Reason-Centric Perspective by operationalizing it using a version of the FBS Framework and providing evidence that core assumptions within this framework describe real development behaviors less accurately than a process theory embedding contradictory assumptions. In doing so, I created and gathered evidence supporting a new software design process theory, the SCI Framework. This framework represents a holistic understanding of how software is created in practice. It can be used to initiate or advance a variety of software design science research initiatives, including generating theories of the causes of design project success and better understanding the use and effects of all manner of design methodics. Such endeavors require of researchers the tenacity to choose difficult but practically important research over straightforward but inconsequential studies.

BIBLIOGRAPHY

"Annual report for the year ending september 30, 1988," Accreditation board for engineering and technology, Inc., New York, USA.

Abrahamsson, P., Salo, O., Ronkainen, J., and Warsta, J. *Agile software development methods: Review and analysis*, VTT Publications, Espoo, 2002.

Aghion, P., and Tirole, J. "Formal and real authority in organizations," *Journal of Political Economy* (105:1) 1997, p 1.

Alast, W.M.P.v.d. "Workflow verification: Finding control-flow errors using petri-net-based techniques," Business Process Management: Models, Techniques, and Empirical Studies,(LNCS 1806), Springer-Verlag, Berlin, 2000, pp. 161-183.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., and S., A. *A pattern language: Towns, buildings, construction.*, Oxford University Press, 1977.

Alexander, C.W. *Notes on the synthesis of form*, Harvard University Press, 1964.

Alexandrou, M. "Systems development life cycle (SDLC)," 2010. Available: http://www.mariosalexandrou.com/methodologies/systems-development-life-cycle.asp. Accessed: Mar 18, 2010

Alter, S. *The work system method: Connecting people, processes, and it for business results*, Work System Press, 2006.

Ambler, S. "A realistic look at object-oriented reuse," *Software Development* (6:1) 1998, pp 30-38.

Ambler, S. *Agile modeling: Effective practices for extreme programming and the unified process*, Wiley, 2002.

Archer, B. "Design as a discipline," *Design Studies* (1:1) 1979, pp 17-20.

Avgerou, C., and Cornford, T. "A review of the methodologies movement," *Journal of Information Technology* (8:4) 1993, pp 277-286.

Avison, D.E., and Fitzgerald, G. *Information systems development: Methodologies, techniques and tools*, Blackwell, Oxford, 1988.

Baltzan, P., and Phillips, A. *Business driven information systems*, McGraw-Hill Irwin, 2008.

Bansler, J., and Bødker, K. "A reappraisal of structured analysis: Design in an organizational context," *ACM Transactions on Information Systems* (11:2) 1993, pp 165-193.

Barnett, J. *An introduction to urban design*, Harper and Row Publishers Inc., New York, USA, 1982.

Bartels, A., Holmes, B.J., and Lo, H. "US slowdown in 2007 will dampen the $1.6 trillion global it market," Forrester Research, 2006.

Baskerville, R., and Pries-Heje, J. "Short cycle time systems development," *Information Systems Journal* (14:3) 2004, pp 237-264.

Baskerville, R., Travis, J., and Truex, D.P. "Systems without method: The impact of new technologies on information systems development projects," Proceedings of the IFIP WG8.2 Working Conference on The Impact of Computer Supported Technologies in Information Systems Development, North-Holland Publishing Co., Amsterdam, The Netherlands, 1992, pp. 241-269.

Beck, K. *Test driven development: By example*, Addison-Wesley Professional, 2002.

Beck, K. *Extreme programming explained: Embrace change*, (2nd ed.), Addison Wesley, Boston, MA, USA, 2005.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. "Manifesto for agile software development," 2001. Available: http://www.agilemanifesto.org/. Accessed: June 22, 2010

Benbasat, I., Goldstein, D., and Mead, M. "The case research strategy in studies of information systems," *MIS Quarterly* (11:3) 1987, pp 369-386.

Bera, P., and Wand, Y. "A framework to clarify the role of knowledge management systems," in: *Pacific Asia Conference on Information Systems (PACIS '09)*, Hyderabad, India, 2009.

Berente, N., and Lyytinen, K. "The iterating artifact as a fundamental construct for information system design," in: *1st International Conference on Design Science in Information Systems and Technology*, Claremont, CA, USA, 2006.

Berger, P., and Luckmann, T. *The social construction of reality : A treatise in the sociology of knowledge*, Penguin, London, 1966.

Blumrich, J.F. "Design," *Science* (168:3939) 1970, p 1151.

Boehm, B. "A spiral model of software development and enhancement," *IEEE Computer* (21:5), May 1988, pp 61-72.

Boehm, B.W. *Characteristics of software quality*, North-Holland, Amsterdam, 1978.

Bourque, P., and Dupuis, R. (eds.) *Guide to the software engineering body of knowledge (SWEBOK)*. IEEE Computer Society Press, 2004.

Bradel, B., and Stewart, K. "Exploring processor design using genetic programming," in: *ECE1718 Special Topics in Computer Hardware Design: Modern and Emerging Architectures*, University of Toronto, Toronto, Ontario, 2004.

Breuer, T., Ndoundou-Hockemba, M., and Fishlock, V. "First observation of tool use in wild gorillas," *PLoS Biol* (3:11), October 2005.

Buchanan, R. "Definition of design," Personal Communication. Received: Jan. 30, 2006.

Carroll, G., and Hannan, M.T. "Density delay in the evolution of organizational populations: A model and five empirical tests," *Administrative Science Quarterly* (34) 1989, pp 411-430.

Casti, J. *Paradigms lost.*, Avion Books, New York, USA, 1989.

Checkland, P. *Systems thinking, systems practice*, Wiley, Chichester, 1999.

Checkland, P., and Holwell, S. "Action research: Its nature and validity," *Systemic Practice and Action Research* (11:1) 1998, pp 9-21.

Checkland, P., and Poulter, J. *Learning for action*, Wiley, 2006.

Churchman, C.W. *The design of inquiring systems: Basic concepts of systems and organization*, Basic Books, New York, 1971.

Clark, A., and Chalmers, D. "The extended mind," *Analysis* (58:1) 1998, pp 7-19.

Coad, P., LeFebvre, E., and Luca, J.D. *Java modeling in color with UML: Enterprise components and process*, Prentice Hall, 1999.

Complin, C. "The evolutionary engine and the mind machine: A design-based study of adaptive change," Doctoral Dissertation, University of Birmingham, UK, 1997.

Curtis, B., Kellner, M.I., and Over, J. "Process modeling," *Communications of the ACM* (35:9) 1992, pp 75-90.

Dardenne, A., and Lamsweerde, A. "Goal-directed requirements acquisition," *Science of Computer Programming* (20) 1993, pp 3-50.

DeLone, W.H., and McLean, E.R. "Information systems success: The quest for the dependent variable," *Information management* (3) 1992, p 60.

DeLone, W.H., and McLean, E.R. "The DeLone and McLean model of information systems success: A ten-year update," *Journal of Management Information Systems* (19:4), March 2003, pp 9-30.

Desmond, J.P. "The software 500: Applications go worldwide," in: *Software Magazine*, 2007.

Desouza, K.C., Awazu, Y., and Tiwana, A. "Four dynamics for bringing use back into software reuse," *Commun. ACM* (49:1) 2006, pp 96-100.

DeVellis, R. *Scale development: Theory and applications*, (2nd ed.), Sage, Thousand Oaks, CA, USA, 2003.

Dobing, B., and Parsons, J. "How UML is used," *Communications of the ACM* (49:5), May 2006, pp 109-113.

Dorst, K., and Cross, N. "Creativity in the design process: Co-evolution of problem-solution," *Design Studies* (22), September 2001, pp 425-437.

Dorst, K., and Dijkhuis, J. "Comparing paradigms for describing design activity," *Design Studies* (16:2) 1995, pp 261-274.

Dube, L., and Pare, G. "Rigor in information systems positivist case research: Currect practices, trends and recommendations," *MIS Quarterly* (27:4), December 2003, pp 597-635.

Eckroth, J., Aytche, R., and Amoussou, G.-A. "Toward a science of design for software-intensive systems," Proceedings of the Second International Conference on Design Science Research in Information Systems and Technology, Pasadena, CA, USA, 2007.

Eekels, J. "On the fundamentals of engineering design science: The geography of engineering design science. Part 1," *Journal of Engineering Design* (11), December 2000, pp 377-397.

Eekels, J. "On the fundamentals of engineering design science: The geography of engineering design science. Part 2," *Journal of Engineering Design* (12), September 2001, pp 255-281.

Eisenhardt, K.M. "Building theories from case study research," *The Academy of Management Review* (14:4) 1989, pp 532-550.

Ewusi-Mensah, K. *Software development failures*, MIT Press, 2003.

Fenton, N. "Software measurement: A necessary scientific basis," *IEEE Transaction on Software Engineering* (20:3) 1994, pp 199-206.

Finkelstein, L. "A review of the fundamental concepts of measurement," *Measurement* (2:I) 1984, pp 25-34.

Fitzgerald, B. "The transformation of open source software," *MIS Quarterly* (30:3) 2006.

FitzGerald, J., and FitzGerald, A. "Fundamentals of systems analysis," Wiley, USA, 1987.

Fowler, F.J. *Improving survey questions: Design and evaluation*, Sage, Thousand Oaks, CA, USA, 1995.

Freeman, P., and Hart, D. "A science of design for software-intensive systems," *Communications of the ACM* (47:8) 2004, pp 19-21.

Freeman, R.E. *Strategic management: A stakeholder approach*, Pitman, Boston, 1984.

Galle, P. "The ontology of Gero's FBS model of designing," *Design Studies* (30:4) 2009, pp 321-339.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design patterns: Elements of reusable object-oriented software*, Addison-Wesley, Boston, MA, USA, 1995.

Garlan, D., and Shaw, M. "An introduction to software architecture," Advances in software engineering and knowledge engineering, World Scientific, 1993, pp. 1-39.

Gero, J.S. "Design prototypes: A knowledge representation schema for design," *AI Magazine* (11:4) 1990, pp 26-36.

Gero, J.S., and Kannengiesser, U. "The situated function-behaviour-structure framework," *Design Studies* (25:4) 2004, pp 373-391.

Gero, J.S., and Kannengiesser, U. "A function-behavior-structure ontology of processes," *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* (21:4) 2007, pp 379-391.

Gero, J.S., and Mc Neill, T. "An approach to the analysis of design protocols," *Design Studies* (19:1) 1998, pp 21-61.

Gladden, G.R. "Stop the life-cycle, I want to get off," *SIGSOFT Software Engineering Notes* (7:2) 1982, pp 35-39.

Goldstein, J. "Emergence as a construct: History and issues," *Emergence* (1:1) 1999, pp 49-72.

Grabowski, H., Lossack, R.-S., and El-Mejbri, E.-F. "Towards a universal design theory," in: *Integration of process knowledge into design support systems*, University of Twente, Enschede, The Netherlands, 1999, pp. 47-56.

Graham, P. "Hackers and painters," 2003. Available: http://www.paulgraham.com/hp.html. Accessed: 21 June 2010

Gregor, S. "The nature of theory in information systems," *MIS Quarterly* (30:3) 2006, pp 611-642.

Gregor, S., and Jones, D. "The anatomy of a design theory," *Journal of the Association for Information Systems* (8:5), May 2007, p 312.

Grüninger, M., and Fox, M.S. "Methodolgy for the design and evaluation of ontologies," Proceedings of the IJCAI Workshop on Basic Ontological Issues in Knoweldge Sharing, AAAI Press, Menlo Park CA, USA, 1995.

Hacking, I. *Scientific revolutions*, Oxford University Press, New York, USA, 1982.

Hammer, M., and Champy, J. "Reengineering the corporation: A manifesto for business revolution," *Business Horizons* (36:5) 1993, pp 90-91.

Hansen, S., Berente, N., and Lyytinen, K. "Requirements in the 21st century: Current practice and emerging trends," in: *The Design Requirements Workshop*, Cleveland, Ohio, USA, 2007.

Harris, D. *Systems analysis and design: A project approach*, Dryden Press, Texas, USA, 1995.

Harwell, M.R., and Gatti, G.G. "Rescaling ordinal data to interval data in educational research," *Review of Educational Research* (71:1), January 1 2001, pp 105-131.

Hatchuel, A., and Weil, B. "C-K theory: Notions and applications of a unified design theory," in: *The Herbert Simon International Conference on "Design Sciences"*, Lyon, 2002, p. 22.

Hatchuel, A., and Weil, B. "A new approach of innovative design: An introduction to C-K theory," in: *International Conference on Engineering Design*, Stockholm, Sweden, 2003.

Henderson-Sellers, B., and Edwards, J.M. "The fountain model for object-oriented system development," in: *Object Magazine*, 1993, pp. 71-79.

Hevner, A.R., March, S.T., Park, J., and Ram, S. "Design science in information systems research," *MIS Quarterly* (28:1), March 2004, pp 75-105.

Hinrichs, T.R. "Problem-solving in open worlds: A case study in design," Doctoral Dissertation, Georgia Institute of Technology, Atlanta, GA, USA, 1992.

Hirschheim, R., Klein, H.K., and Lyytinen, K. *Information systems development and data modeling: Conceptual and philosophical foundations*, Cambridge University Press, New York, NY, USA, 1995.

Iivari, J., Hirschheim, R., and Klein, H.K. "A dynamic framework for classifying information systems development methodologies and approaches," *Journal of Management Information Systems* (17:3) 2000, pp 179-218.

Jacobson, I., Booch, G., and Rumbaugh, J. *The unified software development process*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

Jaeger, R.G., and Halliday, T.R. "On confirmatory versus exploratory research," *Herpetologica* (54), June 1998, pp S64-S66.

Jayaratna, N. *Understanding and evaluating methodologies: NIMSAD, a systematic framework*, McGraw-Hill, Maidenhead, UK, 1994.

Jobs, S. "Apple's one-dollar-a-year man," Fortune, 2000. Available: http://money.cnn.com/magazines/fortune/fortune_archive/2000/01/24/272277/index.htm. Accessed: June 22, 2010

Kahneman, D., and Tversky, A. "Prospect theory: An analysis of decision under risk," *Econometrica* (47:2) 1979, pp 263-291.

Kessler, A. "Wsj: Weekend interview with facebook's mark zuckerberg," 2007.

Kimberly, J., and Miles, R. *The organizational life cycle*, Jossey-Bass, San Francisco, 1980.

Kroenke, D., Gemino, A., and Tingling, P. *Experiencing MIS*, (Second Canadian ed.), Pearson Prentice Hall, Toronto, 2010.

Kruchten, P. *The rational unified process: An introduction*, (1st ed.), Addison-Wesley Professional, 1998.

Kruchten, P. *The rational unified process: An introduction*, (3rd ed.), Addison-Wesley Professional, 2003.

Kruchten, P. "Casting software design in the function-behavior-structure framework," *IEEE Software* (22:2) 2005, pp 52-58.

Laudon, K., Laudon, J., and Brabston, M. *Management information systems: Managing the digital firm*, (Fourth Canadian ed.), Pearson, Prentice Hall, Toronto, 2009.

Lee, A.S. "A scientific methodology for MIS case studies," *MIS Quarterly* (13:1) 1989, pp 33-50.

Lee, G., and Xia, W. "Toward agile: An integrated analysis of quantitative and qualitative field data," *MIS Quarterly* (34:1) 2010, pp 87-114.

Levina, N. "Collaborating on multiparty information systems development projects: A collective reflection-in-action view," *Information Systems Research* (16:2) 2005, pp 109-130.

Love, T. "Philosophy of design: A meta-theoretical structure for design theory," *Design Studies* (21) 2000, pp 293-313.

Love, T. "Constructing a coherent cross-disciplinary body of theory about designing and designs: Some philosophical issues," *Design Studies* (23:3), May 2002, pp 345-361.

Lyytinen, K. "A taxonomic perspective of information systems development: Theoretical constructs and recommendations," Critical Issues in Information Systems Research, John Wiley & Sons, Inc., New York, USA, 1987, pp. 3-41.

Maher, M., Poon, J., and Boulanger, S. "Formalising design exploration as co-evolution: A combined gene approach," in: *Preprints of the Second IFIP WG5.2 Workshop on Advances in Formal Design Methods for CAD,* J.S.G.F. Sudweeks (ed.), Key Centre of Design Computing, 1995, pp. 1-28.

March, J.G., and Simon, H.A. *Organizations*, Wiley, New York, 1958.

March, L. "The logic of design," in: *Developments in design methodology,* N. Cross (ed.), John Wiley & Sons, Chichester, 1984, pp. 265-276.

March, S.T., and Smith, G.F. "Design and natural science research on information technology," *Decision Support Systems* (15:4) 1995, pp 251--266.

Markus, M.L., and Robey, D. "Information technology and organizational change: Causal structure in theory and research," *Management Science* (34:5) 1988, pp 583-599.

Martin, J. *Rapid application development*, Macmillan Publishing, Indianapolis, IN, USA, 1991.

Mathiassen, L. "Reflective systems development," Doctoral Dissertation, Aalborg University, Aalborg, Denmark, 1998, 537 Pages.

McCracken, D.D., and Jackson, M.A. "Life cycle concept considered harmful," *SIGSOFT Software Engineering Notes* (7:2) 1982, pp 29-32.

Meyer, B. "Reusability: The case for object-oriented design," in: *Software reuse: Emerging technology*, IEEE Computer Society Press, 1988, pp. 201-215.

Mili, H., F., M., and A., M. "Reusing software: Issues and research directions," *IEEE Transactions on Software Engineering* (21:6) 1995, pp 528-562.

Miller, W. "The definition of design," 2005. Available: http://www.methodjournal.com/artman/publish/Digital_Design/article_13.shtml. Accessed: 1 Dec., 2009.

Mohr, L.B. *Explaining organizational behavior*, Jossey-Bass, San Francisco, 1982.

Mulcahy, N., and Call, J. "Apes save tools for future use.," *Science* (312:5776) 2006, pp 1038-1040.

Nandhakumar, J., and Avison, D. "The fiction of methodological development: A field study of information systems development," *Information Technology & People* (12:2), February 1999, pp 176-191.

Naur, P. "Understanding turing's universal machine: Personal style in program description," *The Computer Journal* (36:4) 1993, pp 351-372.

Neal, M.A., and Northcraft, G.B. "Behavioral negotiation theory: A framework for conceptualizing dyadic bargaining," in: *Research in organizational behavior,* L.L. Cummings and B.M. Staw (eds.), JAI Press, Greenwich, CT, 1991, pp. 147-190.

Newell, A., and Simon, H. *Human problem solving*, Prentice-Hall, Inc., 1972, 920 pages.

Noy, N.F., and Hafner, C.D. "The state of the art in ontology design," in: *AI Magazine*, 1997, pp. 53-74.

Nunamaker, J.F., Chen, M., and Purdin, T.D.M. "Systems development in information systems research," *Journal of Management Information Systems* (7:3), Winter 1991, pp 89-106.

Pahl, G., and Beitz, W. *Engineering design: A systematic approach*, Springer-Verlag, London, 1996.

Palvia, P., and Nosek, J. "An empirical evaluation of system development methodologies," *Information Resource Management Journal* (3:3) 1990, pp 23-32.

Papanek, V. *Design for human scale*, Van Nostrand Reinhold Company, New York, USA, 1983.

Parnas, D.L., and Clements, P.C. "A rational design process: How and why to fake it," *IEEE Transactions on Software Engineering* (12:2) 1986, pp 251-257.

Pfeffer, J. *Managing with power. Politics and influence in organizations.*, Harvard Business School Press, Boston, MA, USA, 1992.

Pinker, S. *The blank slate: The modern denial of human nature*, Penguin, 2002.

Polya, G. *How to solve it: A new aspect of mathematical method*, (2nd ed.), Princeton University Press, Princeton, New Jersey, USA, 1957.

Poole, M., Van de Ven, A.H., Dooley, K., and Holmes, M.E. *Organizational change and innovation processes theory and methods for research*, Oxford University Press, New York, NY, USA, 2000, 406 pages.

Popper, K. *The logic of scientific discovery*, Basic Books, New York, NY, USA, 1959.

Purao, S., Rossi, M., and Bush, A. "Towards an understanding of problem and design spaces during object-oriented systems development," *Information and Organizations* (12:4) 2002, pp 249-281.

Pye, D. *The nature of design*, Studio Vista, London, 1964.

Reel, J.S. "Critical success factors in software projects," *IEEE Software* (16:3) 1999, pp 18-23.

Reubenstein, H.B., and Waters, R.C. "The requirements apprentice: Automated assistance for requirements acquisition," *IEEE Transactions on Software Engineering* (17:3) 1991, pp 226-240.

Richardson, J. *Basic design*, Prentice-Hall, New Jersey, 1984.

Roberts, F. *Measurement theory with applications to decision making, utility. And the social sciences*, Addison Wesley, Reading, MA, 1979.

Roozenburg, N., and Eekels, J. *Product design: Fundamentals and methods*, Wiley, Chichester, UK, 1995.

Rosenman, M.A., and Gero, J.S. "Purpose and function in design: From the socio-cultural to the techno-physical," *Design Studies* (19) 1998, pp 161-186.

Royce, W.W. "Managing the development of large software systems: Concepts and techniques," Proceedings of Wescon, 1970.

Schmidt, D.C. "Guest editor's introduction: Model-driven engineering," *IEEE Computer* (39:2) 2006, pp 25-31.

Schön, D.A. *The reflective practitioner: How professionals think in action*, Basic Books, USA, 1983.

Schurch, T.W. "Reconsidering urban design: Thoughts about its definition and status as a field or profession," *Journal of Urban Design* (4:1) 1999, pp 5-28.

Schwaber, K., and Beedle, M. *Agile software development with scrum*, Prentice Hall, 2001.

Shenhar, A.J., Dvir, D., Levy, O., and Maltz, A.C. "Project success: A multidimensional strategic concept," *Long Range Planning* (34:6) 2001, pp 699-725.

Siddiqi, J., and Shekaran, M. "Requirements engineering: The emerging wisdom," *IEEE Software*, March 1996, pp 15-19.

Sim, S.K., and Duffy, A.H.B. "Towards an ontology of generic engineering design activities," *Research in Engineering Design* (14:4), November 2003, pp 200-223.

Simon, H.A. *The sciences of the artificial*, (1st ed.), MIT Press, Cambridge, MA, USA, 1969.

Simon, H.A. *The sciences of the artificial*, (3rd ed.), MIT Press, Cambridge, MA, USA, 1996.

Singer, E.A. *Experience and reflection*, University of Pennsylvania Press, 1959.

Sircar, S., Nerur, S.P., and Mahapatra, R. "Revolution or evolution? A comparison of object-oriented and structured systems development methods.," *MIS Quarterly* (25:4), December 2001, pp 457-471.

Sober, E. "Testability," *Proceedings and Addresses of the American Philosophical Association* (73:2) 1999, pp 47-76.

Soffer, P., and Wand, Y. "Goal-driven analysis of process model validity," in: *Advanced Information Systems Engineering*, 2004, pp. 521-535.

Sommerville, I. *Software engineering*, (5th ed.), Addison Wesley, Redwood City, CA, USA, 1996, 742 pages.

Stanovich, K. *What intelligence tests miss: The psychology of rational thought*, Yale University Press, New Haven, CT, USA, 2009, 308 pages.

Straub, D.W. "Validating instruments in MIS research," *MIS Quarterly* (13:2) 1989, pp 147-169.

Strogatz, S.H. *Nonlinear dynamics and chaos*, Addison-Wesley, Reading, MA, USA, 1994.

Stumpf, R., and Teague, L. *Object-oriented systems analysis and design with UML*, Pearson Prentice Hall, New Jersey, USA, 2005.

Suchman, L. *Plans and situated actions: The problem of human-machine communication*, Cambridge University Press, 1987.

Suh, N. *The principles of design*, Oxford University Press, New York, NY, USA, 1990.

Suh, N. "*Design and operation of large systems*," Journal of Manufacturing Systems (14:3) 1995, pp 203-213.

Suh, N. "*Axiomatic design theory for systems*," Research in Engineering Design (10) 1998, pp 189-209.

Suh, N. *Axiomatic design: Advances and applications*, Oxford University Press, New York, NY, USA, 2001.

Sullivan, K. "Preliminary report: NSF workshop on the science of design: Software and software-intensive systems," University of Virginia Department of Computer Science, Airlie Center.

Thaler, R.H., and Sunstein, C.R. *Nudge: Improving decisions about health, wealth, and happiness*, Yale University Press, New Haven, CT, USA, 2009.

The Partners of Pentagram *Living by design*, Lund Humphries, London, 1978.

Tomiyama, T., and Yoshikawa, H. "Extended general design theory," in: *IFIP WG 5.2 Working Conference on Design Theory for CAD,* H. Yoshikawa and E. Warman (eds.), North-Holland, Tokyo, Japan, 1985.

Truex, D., Baskerville, R., and Travis, J. "Amethodical systems development: The deferred meaning of systems development methods," *Accounting, Management and Information Technologies* (10:1) 2000, pp 53-79.

Turner, J. "Understanding the elements of system design," Critical Issues in Information Systems Research, Wiley, Chichester, UK, 1987, pp. 97-111.

Tversky, A., and Kahneman, D. "Judgment under uncertainty: Heuristics and biases," *Science* (185:4157) 1974, pp 1124-1131.

Urban Design Group "Urban design as a career," 2003. Available: http://www.udg.org.uk/?document_id=468. Accessed: June 22, 2010.

Van de Ven, A.H., and Poole, M.S. "Explaining development and change in organizations," *The Academy of Management Review* (20:3), July 1995, pp 510-540.

van Engers, T.M., Gerrits, R., Boekenoogen, M., Glass, E., and Kordelaar, P. "Power: Using UML/OCL for modeling legislation - an application report," ICAIL '01: Proceedings of the 8th international conference on Artificial intelligence and law, ACM Press, New York, NY, USA, 2001, pp. 157-167.

van Lamsweerde, A. "Goal-oriented requirements engineering: A guided tour," Proceedings of the Fifth IEEE International Symposium on Requirements Engineering, 2001, pp. 249-262.

Venkatesh, V., Morris, M.G., Davis, G.B., and Davis, F.D. "User acceptance of information technology: Toward a unified view.," *MIS Quarterly* (27:3), September 2003, pp 425-478.

Vermaas, P.E., and Dorst, K. "On the conceptual framework of John Gero's FBS-model and the prescriptive aims of design methodology," *Design Studies* (28:2) 2007, pp 133-157.

Walls, J.G., Widmeyer, G.R., and El Sawy, O.A. "Building an information system design theory for vigilant eis," *Information Systems Research* (3:1), March 1992, pp 36-59.

Walz, D.B., Elam, J.J., and Curtis, B. "Inside a software design team: Knowledge acquisition, sharing, and integration," *Communications of the ACM* (36:10) 1993, pp 63-77.

Weick, K. *Sensemaking in organizations*, Sage, Thousand Oaks, CA, USA, 1995.

Weick, K.E., Sutcliffe, K.M., and Obstfeld, D. "Organizing and the process of sensemaking," *Organization Science* (16:4) 2005, pp 409-421.

Whitley, E.A. "Method-ism in practice: Investigating the relationship between method and understanding in web page design," Proceedings of the 19th International Conference on Information Systems (ICIS), Helsinki, Finland, 1998, pp. 68-75.

Wolfe, R.A. "Organizational innovation: Review, critique and suggested research directions," *Journal of Management Studies* (31:3) 1994, pp 405-431.

Wynekoop, J., and Russo, N. "Systems development methodologies: Unanswered questions," *Journal of Information Technology* (10:2), June 1995.

Wynekoop, J., and Russo, N. "Studying system development methodologies: An examination of research methods," *Information Systems Journal* (7), January 1997, pp 47-65.

Yin, R. *Case study research: Design and methods*, (3rd ed.), Sage Publications, California, USA, 2003.

Yoshikawa, H. "General design theory," in: *The IFIP WG 5.2-5.3 Working Conference,* T. Sata and E. Warman (eds.), North Holland, Amsterdam, 1980, pp. 35-57.

Zheng, Y., Venters, W., and Cornford, T. "Agility, improvisation and enacted emergence," International Conference on Information Systems, Montreal, Canada, 2007.

APPENDICES

APPENDIX A: DISMISSING THE SDLC

This section is motivated by the many dichotomous discussions I have had with academics and practitioners concerning The Systems Development Lifecycle (SDLC). Anecdotally speaking, for every person I have met who believes that SDLC is an absurdity no one takes seriously, I have met another who believes that SDLC is the fundamental basis of all systems development. Many people in each group appear unaware that the other group even exists. This necessitates an open discussion of the role of SDLC in design research, practice and education. I open this discussion by asserting the following.

Position: *The Systems Development Lifecycle is a toxic concept.*

SDLC (Figure 1) is a somewhat nebulous concept that may refer to:

1. A process theory (Van de Ven and Poole 1995) that describes systems development in terms of a discrete number of sequential phases, including planning, analysis, design and coding, or variants thereof.
2. A systems development method (SDM) (Wynekoop and Russo 1997) resembling one or more variants of the Waterfall Model (Royce 1970)
3. Any set of steps for creating a technological artifact

For the purposes of this paper, toxic concepts are ideas that are both false and harmful (defined formally below). For example, in educational psychology, the blank slate hypothesis (the view that the mind lacks innate traits) is a toxic concept as it has been refuted by neurobiological studies and deleteriously affects educational methods (Pinker 2002).

Toxic Concept: a theory, construct, argument, technology, or other idea that 1) is untrue, inaccurate or refuted and 2) causes confusion, practical hardship or deleterious action.

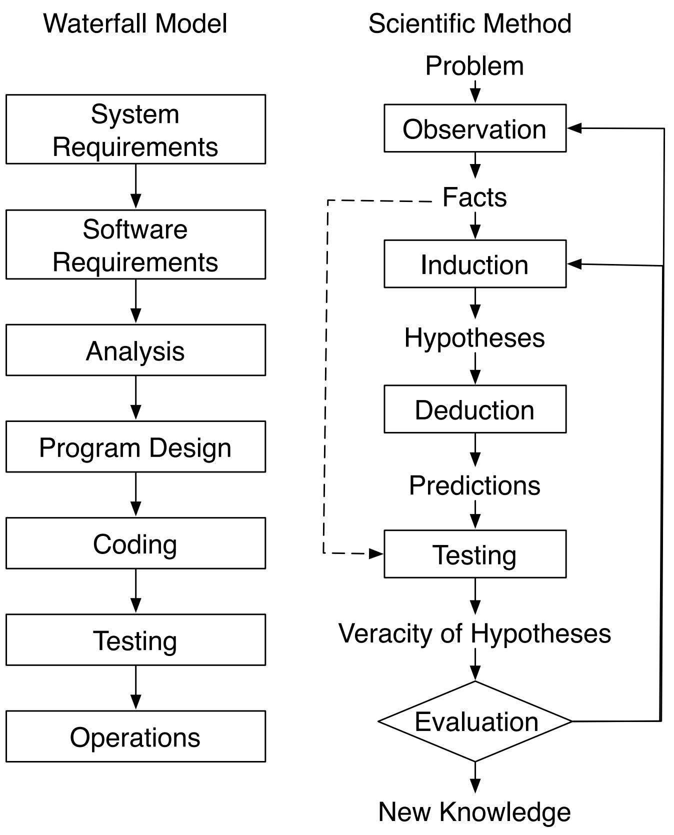


Fig. A-1. SDLC (left) and Cycle of Scientific Inquiry (Roozenburg and Eekels 1995) (right)

A.1 SDLC as a Theory

A process theory is an explanation of how and why an entity changes and develops (Van de Ven and Poole 1995). Coupling a model of SDLC (as in Figure 1) with a claim that it either describes all systems development or (equivalently) that its elements or structure are inherent to development is commensurate with claiming that SDLC is a process theory.

Though rarely stated, implicit claims that SDLC is a veracious process theory pervade research, teaching and practice. For example, in a well-cited paper in MIS Quarterly, Fitzgerald (2006) states that "in conventional software development, the development lifecycle in its most generic form comprises four broad phases: planning, analysis, design, and implementation" (p. 3) and then describes the presence of these phases in open-source software development. In a popular introductory MIS textbook, Laudon et al. (2009) state that "systems development … consist[s] of systems analysis, systems design, programming, testing, conversion and production and maintenance … which usually take place in sequential order." Similarly, at the time of writing, the SDLC Wikipedia article states that "SDLC adheres to important phases that are essential for developers, such as planning, analysis, design, and implementation." Moreover, the traditional SDLC phases are explicitly adopted in the official IEEE Guide to the Software Engineering Body of Knowledge (Bourque and Dupuis 2004), although it does not imply linear sequencing. In summary, implicit and explicit claims of SDLC universality remain prevalent in research, teaching and practice.

The claim that SDLC describes all systems development has been unequivocally refuted by empirical research. This finding is independent of the precise phases employed or their sequence. For example, in field studies of expert designers, Schön found evidence indicating that designers do "not keep means and ends separate" or "separate thinking from doing" (1983, p. 69). Meanwhile, Bansler & Bødker (1993) found that developers may claim to follow a method while practically ignoring it. Additionally, in a study of "a large scale system development effort", Zheng et al. (2007) found that "home-gown methods and ad hoc activities appear to dominate the day-to-day practices of systems development" (p. 1). Furthermore, in Chapter 4, I found that the a generalized model of SDLC does not accurately represent software design practice. Moreover, the XP and Agile Development Conferences feature multitudinous experience reports irreconcilable with SDLC-thinking. More generally, "any form of life cycle is a project management structure imposed on system development. To contend that a life cycle scheme, even with variations, can be applied to all system development is either to fly in the face of reality or to assume a life cycle so rudimentary as to be vacuous" (McCracken and Jackson 1982, p. 30).

A.2 SDLC as a Method

Some argue that a waterfall-like SDLC is a SDM, i.e., it is one way to build software. This view is common in MIS research (e.g., Lee and Xia 2010; Sircar et al. 2001). Furthermore, Royce originally proposed SDLC as "a more grandiose approach to software development" than a method comprised only of analysis and coding (1970, p. 328). This view was elaborated by Boehm (1988). SDLC is often contrasted with various Agile methods (Abrahamsson et al. 2002), and a case is made that each is effective in different circumstances. This is the approach taken by several introductory MIS textbooks (e.g., Baltzan and Phillips 2008; Kroenke et al. 2010). Moreover, proponents of agile methods often position them as more effective alternatives to SDLC (e.g., Beck 2005; Schwaber and Beedle 2001).

Positioning SDLC as a method involves two claims: 1) that SDLC is in some way effective; 2) that it is possible, in principle, to develop systems using it.

The claim that SDLC is an effective method lacks empirical support. I have never encountered an experimental study comparing SDLC to alternative methods. I have found no multiple-case studies contrasting teams using SDLC with teams employing other methods. I have identified no analyses of secondary data evaluating the effect of SDLC on outcome variables such as project success or software success. I did find one survey evaluating SDLC against a prototyping methodology (Palvia and Nosek 1990); however, it explicitly assumed that SDLC describes all software development, thus its support for SDLC as a method is circular. In summary, I found no credible evidence that SDLC is effective in any sense. While this does not refute the claim, we have several reasons to believe SDLC is ineffective. The author generally credited with proposing SDLC affirmed that its simplest version "has never worked on large software development efforts" (Royce 1970, p. 335). Furthermore, SDLC ignores end-user development and end-user involvement outside of requirements specification and "rigidifies thinking", increasing developers' resistance to change (McCracken and Jackson 1982, p. 31). Moreover, the tightly-coupled nature of the life cycle stages exacerbates problems by making it difficult to modify either requirements or the software without setting off complex downstream or upstream revisions (Gladden 1982, p. 36). Additionally, SDLC is "risky and invites failure" because testing occurs at the end and many of the phenomena of interest are "not precisely analyzable" (Royce 1970, p. 329). Also, SDLC justifies intensive upfront analysis by citing a steep cost-of-change curve, but the steepness of the curve is not a feature of software projects but a feature of waterfall-like processes (Ambler 2002; Beck 2002). Finally, SDLC assumes that human developers are capable of correctly getting the requirements, design and tests correct on the first try. As the burden of proof (of effectiveness) for any method lies with its proponents, no proof has been provided, and we have many strong reasons to question SDLC's potential effectiveness, on the balance of evidence, this claim is unsupported.

The claim that SDLC describes any systems development practice can be challenged on several grounds. First, "the development process itself changes the user's perceptions of what is possible, increases his or her insights into the applications environment, and indeed often changes that environment itself;" therefore, "systems requirements cannot ever be stated fully in advance, not even in principle" (McCracken and Jackson 1982, p. 31). Second, the descriptions of the stages of SDLC are "imprecise, ambiguous, incomprehensible" (Curtis et al. 1992, p. 75). Third, SDLC separates analysis from design, where the former generates an understanding of the problem and the latter generates a solution, without providing any guidance as to how the solution is generated. Since software problems are unbounded (unlike arithmetic problems), even a deep understanding of the problem does not necessarily make the solution evident. Fourth, a waterfall-like SDLC confuses "phases" with activities; for example, analysis is not a phase, it is an activity that is necessary not only for requirements modeling but also for coding and testing. Therefore, on a balance of evidence, this claim is also unsupported.

A.3 SDLC as a Class of Phenomena

Some suggest that a SDLC simply describes a development project's stages (Alexandrou 2010); hence, different projects have different SDLCs. In this interpretation, SDLC obviously cannot be false (and therefore cannot be toxic) as it is not coupled with any empirical claim. I return to this issue below.

A.4 How SDLC Causes Harm

SDLC causes identifiable harm in many ways. First, as SDLC is presented as either a valid description or an effective method of software design in many SA&D courses and texts, it confuses students regarding the true nature of software development and encourages unjustified faith in a deeply flawed approach. Second, it creates conflicts between managers (who try to drive projects through phases, schedules and costs) and developers (who do not adopt these phases and cannot accurately estimate costs) (Beck 2005). Third, insofar as SDLC-thinking underlies design methods, tools and practices, their practical usefulness is hampered. Fourth, the prevalence of SDLC-thinking impedes publishing engineering and behavioral research on design aides rooted in more realistic design theories. Finally, I suggest that as "SDLC" has become inextricably confounded with the stages of the waterfall model, using the same term to denote any sequence of stages resulting in a technological artifact only exacerbates the confusion and conflict described above.

A.5 Conclusion

In conclusion, if SDLC is considered a theory, substantial empirical findings refute its veracity. If SDLC is considered a method, no scientific evidence supports its effectiveness, and many sound arguments that it is impossible in principle exist. These arguments hold regardless of precisely how the stages are divided (e.g., five-stage model, seven-stage model) and whether backtracking or loops are included. Moreover, although using SDLC to denote any development process is not wrong, when combined with its historical usage, this too exacerbates confusion. Furthermore, as software development literature is replete with unreferenced, unsupported empirical claims regarding the centrality of SDLC concepts, SDLC causes significant harm and confusion among practitioners, managers and students alike. Therefore, the Software Development Lifecycle is a toxic concept.

A.6 An Alternative to SDLC

Identifying problems with SDLC is of limited usefulness without suggesting alternatives. Fortunately, better alternatives are available. SDLC may be replaced by an alternative software design process theory, specifically the Sensemaking-Coevolution-Implementation (SCI) Framework (ch. 3). Whereas SDLC is a lifecycle process theory (Van de Ven and Poole 1995), the SCI Framework is a teleological process theory, i.e., an explanation of how and why an entity changes wherein change is manifested by a goal-seeking agent that engages in activities in a self-determined sequence (Churchman 1971; Singer 1959;

Van de Ven and Poole 1995). The core claim of the SCI Framework is that developers engage in three activities to produce software – making sense of the project context, iterating between ideas about the context and artifact, and implementing the artifact in code. The results of the questionnaire study reported in Chapter 4, indicate that the SCI Framework describes software development practice more accurately than SDLC (as the SDLC is a special case of the FBS Framework, against which the SCI Framework was tested).

APPENDIX B: ANALYSIS OF EXISTING DEFINITIONS OF DESIGN

We have identified at least 33 definitions of design and sub-types of design (such as "software design" and "urban design") in the literature. Though design has several meanings, we have focused on the meaning involving plans for an object and planning or devising as a process.We employed judgment sampling and snowball sampling, i.e., we made educated guesses as to where to look, and then investigated promising references. This strategy was consistent with our goal of identifying as many relevant definitions as possible.

To evaluate the definitions we applied a set of four main criteria: coverage, meaningfulness, unambiguousness and ease of use (see Table A-1). The first three are derived from the evaluation criteria for good theories mentioned, for example, by Casti (1989, p. 44-45). The fourth is a pragmatic criterion. We do not claim that these are the *best* criteria, but, in the absence of a guiding theory for evaluating definitions, that they are reasonable and have face validity.

Table A-1. General Definition Evaluation Criteria

	Criterion	Definition	Example of Error
Necessary	Coverage	Proper coverage means including all appropriate phenomena (completeness), and only appropriate phenomena. If a definition has improper coverage, it excludes at least one phenomenon that it should include or includes at least one phenomenon it should not.	Defining "human communication" to include only speech, will not address non-verbal communication (e.g. body language).
	Meaningfulness	Each term comprising a definition must have a commonly accepted meaning in the given context or must have been pre-defined. Each combination of terms must be directly understandable from the meaning of terms, or have been predefined.	Defining a zombie as 'the living dead' is inappropriate because, even though 'living' and 'dead' have commonly accepted meanings, their juxtaposition forms an oxymoron.
	Unambiguousness	Each term comprising a definition must have exactly one meaning in the given context; furthermore, the definition as a whole must have only one valid interpretation.	Defining political oratory as 'oral rhetoric related to politics' is inappropriate because 'rhetoric' is a contronym, i.e., has two contradictory meanings.
Optional	Ease of Use	Ideally, a definition should be easy to understand and remember, applicable in disparate situations, and readily differentiate between included and excluded phenomena. Simplicity, parsimony and concreteness are all aspects of Ease of Use. These aspects are at least in part subjective and depend on who uses the definition.	Defining the Natural Numbers as 'the smallest set satisfying the two properties: A) 1 is in N; and B) if n is in N, then n + 1 is in N" while clearly correct, would score poorly on Ease of Use in a low-level mathematics class.

To give the reader a sense of the thought process behind the analysis, we discuss two representative examples of the definitions encountered. The first example is by van Engers et al. (2001), who define design as "the creative process of coming up with a well-structured model that optimizes technological constraints, given a specification." This definition has both meaningfulness and coverage problems. First, the meaning of 'optimizes technological constraints' is unclear. In optimization techniques, one optimizes the characteristics of an object subject to constraints, not the constraints themselves. Second, the use of "well-structured" paints an idealistic portrait of design. This confounds the notion of design with measures for design quality. For example, an inexperienced computer science student can design a personal organizer application. The application might not be "well-structured", but is nonetheless *designed*. Thus, this definition omits activities that are clearly design. The second example is that of Hinrichs (1992) who defines design as "the task of generating descriptions of artifacts or processes in some domain" (p. 3). This also has coverage problems. "My chair is grey" is a description of an artifact in a domain, but is clearly not a design. The problem here is that the definition relates to previously designed artifacts. Thus, this definition includes phenomena that are not design.

The complete analysis of existing definitions is presented in Table A-2. Of the 33 definitions identified, we have found that all seem to have coverage problems, at least 12 have meaningfulness problems and at least three have some form of ambiguity.

Table A-2. Analysis of Existing Definitions

Source	Definition	Criticism
(Accred. Board, 1988)	"Engineering design is the process of devising a system, component, or process to meet desired needs. It is a decision making process (often iterative), in which the basic sciences, mathematics, and engineering sciences are applied to convert resources optimally to meet a stated objective."	*Coverage* – the definition is idealistic and unnecessarily limiting in its use of "optimally." E.g., the building in which I work is far from optimal, but it was still designed. *Meaningfulness* – it is not clear what "desired needs" are.
(Alexander 1964)	"The process of inventing physical things which display new physical order, organization, form, in response to function."	*Coverage* – this definition excludes the design of intangible things, such as processes. *Unambiguousness* – it is not clear whether thing must display new physical order, organization AND form, or new physical order, organization OR form.

Source	Definition	Criticism
(Archer 1979)	"Design is, in its most general educational sense, defined as the area of human experience, skill and understanding that reflects man's concern with the appreciation and adaptation in his surroundings in the light of his material and spiritual needs."	*Coverage* – design is an activity, not an "area of human experience…" One can design with little or no experience, skill and understanding. E.g., the application programmer who designs a graphical user interface without experience in, skill in or understanding of the principles of interface design.
(Beck 2005)	"Designing is creating a structure that organizes the logic in the system"	*Coverage* – excludes forms of design that organize things other than logic, e.g., urban planning organizes space.
(Blumrich 1970)	"Design establishes and defines solutions to and pertinent structures for problems not solved before, or new solutions to problems which have previously been solved in a different way."	*Coverage* – Unnecessarily limits design to solutions not previously solved. Excludes independent invention and finding new ways to solve old problems. E.g., by this definition, new cars are not designed because we already have cars.
(Bourque and Dupuis 2004)	"Design is defined in [IEEE610.12-90] as both "the process of defining the architecture, components, interfaces, and other characteristics of a system or component" and "the result of [that] process." Viewed as a process, software design is the software engineering life cycle activity in which software requirements are analyzed in order to produce a description of the software's internal structure that will serve as the basis for its construction."	*Coverage* – even within the software domain, this definition is far too restrictive. If someone simply writes software without creating an intermediate description of its structure, this is still design. Design is, furthermore, not limited to the phase of the software engineering life cycle between requirements analysis and construction; it is in no way clear that these phases can be practically distinguished in all situations.
(Buchanan 2006)	"Design is the human power to conceive, plan and realize all of the products that serve human beings in the accomplishment of their individual or collective purposes."	*Coverage* – Design is not an ability ("power") but an activity. E.g., drawing blueprints for a house, by this definition, is not design. *Unambiguousness* – it is not clear what "products" are – does this include processes and strategies as well as consumer goods?
(Complin 1997)	"'design' is used to refer to the abstract description of the functional architecture of both real or possible systems."	*Coverage* – Excludes design of simple things, such as boomerangs. *Meaningfulness* – it is not clear what "functional architecture" entails
(van Engers et al. 2001)	"the creative process of coming up with a well–structured model that optimizes technological constraints, given a specification."	*Coverage* – excludes all suboptimal artifacts. *Meaningfulness* – the meanings of "specification" and model are unclear.
(Eckroth et al. 2007)	"Design (as a verb) is a human activity resulting in a unique design (specification, description) of artifacts. Therefore, what can be designed varies greatly. However, common to all design is intention: all designs have a goal, and the goal is typically meeting needs, improving situations, or creating something new. Thus, design is the process of changing an existing environment into a desired environment by way of specifying the properties of artifacts that will constitute the desired environment; in other words, creating, modifying, or specifying how to create or alter artifacts to meet needs. In addition, it is best communicated in terms of a particular context, as previous knowledge, experience, and expectations play a strong role in designing and understanding designs."	*Coverage* – excludes independently inventing previously created artifacts and design starting from a hypothetical situations
(FitzGerald and FitzGerald 1987)	"design means to map out, to plan, or to arrange the parts into a whole which satisfies the objectives involved."	*Coverage* – this excludes artifacts that satisfy only some of their objectives. E.g., Enterprise-Resource Planning software does not always satisfy its stated objectives [60], but surely it as still designed.

Source	Definition	Criticism
(Freeman and Hart 2004)	"design encompasses all the activities involved in conceptualizing, framing, implementing, commissioning, and ultimately modifying complex systems—not just the activity following requirements specification and before programming, as it might be translated from a stylized software engineering process."	*Coverage* – simple systems and non-systems can also be designed, e.g. an oar, a boomerang. *Meaningfulness* – the activities are not defined or clearly explained; furthermore, enumerating the tasks *encompassed by* design does not necessarily capture the *meaning of* design.
(Gero 1990)	"a goal-oriented, constrained, decision-making, exploration and learning activity which operates within a context which depends on the designer's perception of the context."	*Coverage* – The problem here is subtle. Not all design is a decision making activity; some designers, such as sculptors, may proceed fluidly without discrete decisions. It could be argued that their decisions are implicit, but then this definition would include activities such as public speaking. Decision-making is a perspective on design, not inherent to it. Furthermore, the idea of designing as leading to a new or changed artifact is missing.
(Harris 1995)	"A collection of activities designed to help the analyst prepare alternative solutions to information systems problems."	*Coverage* – excludes design for non problems outside information system. *Meaningfulness* – use of "designed" is circular
(Hevner et al. 2004)	"design is the purposeful organization of resources to accomplish a goal."	*Coverage* – includes organization tasks that do not constitute design, e.g., alphabetizing books. *Meaningfulness* – resources is undefined; e.g., what are the resources organized to create a military strategy? What are the resources that are being organized in graphics design? *Unambiguousness* – usage of "organization;" is it physical organization of resources, or mental?
(Hinrichs 1992)	"the task of generating descriptions of artifacts or processes in some domain"	*Coverage* – includes descriptions that are not, e.g., "the chair is brown."
(Hirschheim et al. 1995)	"*systems analysis* is the process of collecting, organizing, and analyzing facts about a particular [information system] and the environment in which it operates. *Systems design* then is the conception, generation and formation of a new system, using the analysis results."	*Coverage* – excludes design of non-systems and designs that end in a complete specification (e.g., of a bridge) rather than a system. *Meaningfulness* – this definition hinges on undefined terms "conception, generation and formation"
(Jobs 2000)	"Design is the fundamental soul of a man-made creation that ends up expressing itself in successive outer layers of the product or service."	*Coverage* – excludes designs not involving a product or service and designs that are not "man-made" Meaningfulness – the meaning of "fundamental soul" is unclear
(Love 2002)	"'Design'— a noun referring to a specification or plan for making a particular artefact or for undertaking a particular activity. A distinction is drawn here between a design and an artifact — a design is the basis for, and precursor to, the making of an artefact." "'Designing'—human activity leading to the production of a design."	*Coverage* – 1) the strict time sequencing implied by this definition is unnecessarily limiting; e.g., in software engineering simultaneous design and creation is arguably the preferred approach [65] [35], 2) Design is not strictly a human activity *Meaningfulness* - "Artefact" is undefined, so the scope is unknown.
Merriam-Webster Online (verb)	"*transitive senses* 1 : to create, fashion, execute, or construct according to plan : DEVISE, CONTRIVE 2 a : to conceive and plan out in the mind <he *designed* the perfect crime> 4 a : to make a drawing, pattern, or sketch of b : to draw the plans for"	*Coverage* – t would include drawing a diagram of a tree (not design), but not collaboratively writing a new search algorithm (design). *Meaningfulness* – circular reference to 'design'

Source	Definition	Criticism
Merriam-Webster Online [noun]	"1 a : a particular purpose held in view by an individual or group <he has ambitious *designs* for his son> b : deliberate purposive planning <more by accident than *design*> 2 : a mental project or scheme in which means to an end are laid down 4 : a preliminary sketch or outline showing the main features of something to be executed : DELINEATION 5 a : an underlying scheme that governs functioning, developing, or unfolding : PATTERN, MOTIF <the general *design* of the epic> b : a plan or protocol for carrying out or accomplishing something (as a scientific experiment); *also* : the process of preparing this 6 : the arrangement of elements or details in a product or work of art 7 : a decorative pattern 8 : the creative art of executing aesthetic or functional designs"	*Coverage* - Overall, this definition does not provide a unifying notion of the minimum requirements to call something a design, and does not separate designing from planning. *Meaningfulness* – circular reference to 'designs'
(Miller 2005)	"Design is the thought process comprising the creation of an entity."	*Coverage* – design can encompass more than just a thought process; e.g., drawing diagrams. Thought processes cannot create physical things.
(Nunamaker et al. 1991)	"Design ... involves the understanding of the studied domain, the application of relevant scientific and technical knowledge, the creation of various alternatives, and the synthesis and evaluation of proposed alternative solutions."	*Coverage* – if a person has a breakthrough idea and implements a single, innovative artifact, without considering any alternatives, this would still be design. Depending on how one defines "scientific knowledge," many designers throughout history would be excluded by this definition.
(Papanek 1983)	"Design is a conscious and intuitive effort to impose meaningful order.... Design is both the underlying matrix of order and the tool that creates it."	*Coverage* – Would include all ordering activities, such as alphabetizing books *Meaningfulness* – 'underlying matrix of order' is undefined. *Ease of use* – unclear how to operationalize "matrix of order"
(The Partners of Pentagram 1978)	"A design is a plan to make something: something we can see or hold or walk into; something that is two-dimensional or three-dimensional, and sometimes in the time dimension. It is always something seen and sometimes something touched, and now and then by association, something heard."	*Coverage* – This definition excludes design of an incorporeal thing, e.g., a philosophy, society or strategy.
(Pye 1964)	"Invention is the process of discovering a principle. Design is the process of applying that principle. The inventor discovers a class of system – a generalization – and the designer prescribes a particular embodiment of it to suit the particular result, objects and source of energy he is concerned with."	*Coverage* – Designing need not comply with principles; e.g., one might design a software interface with absolutely no knowledge of any principles regarding interface design. The interface is no less designed by someone.
(Richardson 1984)	"Design is a general term, comprising all aspects of organization in the visual arts."	*Coverage* – excludes design in architecture, engineering, etc.
(Schurch 1999)	"urban design might be more clearly defined as "giving physical design direction to urban growth, conservation, and change..." as practised by the allied environmental design professions of architecture (Barnett 1982, p. 12), landscape architecture and urban planning and others, for that matter, such as engineers, developers, artists, grass roots groups, etc."	*Coverage* – Though design intuitively may give direction, not all instances of giving direction are design; e.g., the mere command "give the castle a moat" gives direction, but is clearly not design *Meaningfulness* – 'physical design direction' undefined
(Simon 1996)	"Design is devising courses of action aimed at changing existing situations into preferred ones."	*Coverage* – excludes designs beginning from hypothetical situations, e.g., when a national defense agency designs a contingency plan for a nuclear attack, and designing imagined system.

Source	Definition	Criticism
(Stumpf and Teague 2005)	"Design is a process which creates descriptions of a newly devised artifact. The product of the design process is a description which is sufficiently complete and detailed to assure that the artifact can be built."	*Coverage* – includes describing an artifact that already exists, e.g. 'the cruise ship is big;' excludes partially designed objects and design of imaginary objects.
(Urban Design Group 2003)	"Urban design is the process of shaping the physical setting for life in cities, towns and villages. It is the art of making places."	*Coverage* – This definition confuses design as planning a setting with the physical process of implementing that plan; e.g., by this definition, planning the park is not designing, but laying the sods is.
(Walls et al. 1992)	"The design process is analogous to the scientific method in that a design, like a theory, is a set of hypotheses and ultimately can be proven only by construction of the artifact it describes."	*Coverage* – While a design may imply a set of hypotheses, saying the design *is* the like saying being hungry *is* making a sandwich. *Ease of Use* – representing a design as a set of hypotheses may be difficult.

APPENDIX C: BACKGROUND ON PROCESS THEORIES

Significant disagreement exists regarding the nature of process theories. For instance, following Mohr (1982), Markus and Robey (1988) described a process theory as "a recipe of sufficient conditions occurring over time" (p. 584) and "concerned with explaining how outcomes develop over time" (p. 589). In contrast, Van de Ven and Poole (1995) defined a process theory more broadly, as "an explanation of how and why an organizational entity changes and develops" (p. 512). More generally, Gregor (2006) identified five types of theories: "(i) theory for analysing; (ii) theory for explaining, (iii) theory for predicting; (iv) theory for explaining and predicting; and (v) theory for design and action" (p. 611). Clearly, Van de Ven and Poole characterize process theories as a theories for explaining. In contrast, with their focus on necessary and sufficient conditions, Markus and Robey have characterized a process theories as a theories for predicting, i.e., if the "recipe of sufficient conditions" holds, the predicted outcomes will follow. Van de Ven and Poole also classify organizational process theories into four categories: lifecycle, dialectic, evolutionary and teleological.

A lifecycle theory "is a unitary sequence (it follows a single sequence of stages or phases), which is cumulative (characteristics acquired in earlier stages are retained in later stages) and conjunctive (the stages are related such that they derive from a common underlying process)" (Van de Ven and Poole 1995, p. 515). This progression occurs because "the trajectory to the final end state is prefigured and requires a particular historical sequence of events" (p. 515). Lifecycle theories have their roots in biological life cycles. The Basic Design Cycle (discussed in ch. 3) exhibits many characteristics of lifecycle theories.

In dialectic process theories, "stability and change are explained by reference to the balance of power between opposing entities" (Van de Ven and Poole 1995, p. 517). This is rooted in the argumentative methods of classical philosophy. Dialectic process theories, therefore, posit two or more entities with manifest conflicts and model change with respect to inter-entity power. Both Soft Systems Methodology and Extreme Programming (discussed in ch. 3) have elements of dialectic process – SSM in its discussion with stakeholders and XP in its negotiation between "business" and "development."

In an evolutionary process theory, a population of "organizational entities" undergoes structural changes through "variation, selection and retention" (Van de Ven and Poole 1995, p. 518). Variation involves producing new entities through chance occurrences. Selection is the preservation of organizations with higher fitness and elimination of those with lower fitness. "Retention involves forces (including inertia and persistence) that perpetuate and maintain certain organizational forms" (Van de Ven and Poole 1995, p. 518). Maher's Problem-Design Exploration Model (discussed in ch. 3) has many characteristics of an evolutionary process theory.

Merriam-Webster defines teleological as "exhibiting or relating to design or purpose especially in nature." [1] In a teleological process theory, an agent "constructs an envisioned end state, takes action to reach it and monitors the progress" (Van de Ven and Poole 1995, p. 518). In other words, teleological theories explain the behavior of agents taking steps to reach a goal, but the agent chooses its own sequence of steps. Teleological theories were originally proposed by Singer (1959), in the domain of biology, and further elucidated by Churchman (1971). Van de Ven and Poole found that teleological process theories are the most common type in the organizational literature.

Of the four types of process theories identified by Van de Ven and Poole, only lifecycle theories satisfy Markus and Robey's criteria for being process theories. Specifically, the "historical sequence of events" of lifecycle theories is analogous to the kind of "recipe of sufficient conditions" Markus and Robey require. This is problematic both pragmatically and theoretically. Pragmatically speaking, adopting Markus and Robey's conception of process theories would exclude three of the four types identified by Van de Ven and Poole. Specifically, this would exclude evolution, with its prominence in the history of

[1] http://www.merriam-webster.com/dictionary/teleological

science; teleology, the most common type of process theory in organizational literature; and dialectic, which has occupied a central place in philosophy for more than 2000 years. I contend that any working definition of process theory that excludes Singer, Darwin and Plato is, at minimum, pragmatically awkward.

Theoretically speaking, in asking for sufficient (and necessary) conditions, Markus and Robey seem to conflate their conception of process theory with a counterfactual analysis approach to causality. That is, causality is defined by the events without which the outcome could not have occurred. Gregor (2006) identifies three other forms of causal analysis: “regularity (or nomological)”, “probabilistic” and “teleological.” Briefly, under regularity causation, X causes Y if Y always follows X; under probabilistic causation, X causes Y if the probability of Y given X is higher than the probability of Y given not-X; under teleological causation, the cause of an event is the agent that brings it about. Lifecycle theories involve counterfactual causation through its “unitary sequence” - the latter activities (and outcome) cannot occur without the earlier activities. Evolutionary theories embed notions of probabilistic causation within the selection concept – the structural properties relate to the probability of survival through the fitness function. Dialectic theories involve the regularity approach to causation insofar as change always follows sufficient shifts in power[2]. Finally, teleological theories, by definition, involve a teleological approach to causation. This analysis is summarized in Table A-3.

Markus and Robey’s conception of process theory is inappropriate for software design science for three interrelated reasons. First, this conception is inextricably tied to a counterfactual approach to causality, as described above. Second, design is a teleological activity (Churchman 1971), that is, an activity driven by an agent in response to a goal. Focusing on antecedents (necessary and sufficient conditions) marginalizes the influence of the goal and agent’s freedom to determine its course of action. Third, the focus on

[2] It could also be argued that dialectic process theories embed a teleological approach to causality in the sense that changes are enacted by agents. Similarly, whether dialectic theories embed a probabilistic or regularity approach to causation depends on whether changes occur if power shifts beyond a set tipping point or the probability of a change varies with the ratio of inter-entity power. As dialectic theories are not central to this paper, further analysis and elucidation of the causal approach of dialectic theories is left to future work.

antecedents also overshadows the potentially rich interactions between the agent, goal and activities; conceptualizing a process theory as a series of necessary and sufficient conditions marginalizes the study of situated action. Heavy criticism of precisely this kind of abstraction away from situated human action motivated the ethno-view of human action discussed above (Suchman 1987). In summary, Markus and Robey's conception of process theory is inappropriate in the science of design because it is in incompatible with teleological causality, shifts the theoretical focus away from goals and abstracts away the details of situated action.

Table A-3. Analysis of Types of Process Theories (partially adapted from Van de Ven and Poole 1995)

Type of Process Theory	Dialectic	Evolutionary	Lifecycle	Teleological
Proponents	(Plato; Hegel; Van de Ven et al. 1995)	(Darwin; Van de Ven et al. 1995)	(Markus and Robey 1988; Van de Ven and Poole 1995)	(Churchman 1971; Singer 1959; Van de Ven and Poole 1995)
Capsule Description	Changes result from shifts in power among conflicting entities	A population of entities changes as less fit entities expire and remaining entities change and recombine	An entity progresses through a series of stages in a predefined sequence	An agent purposefully selects and takes actions to achieve a goal
Event Progression	Recurrent, discontinuous sequence of conflict and resolution	Recurrent, cumulative and probabilistic sequence of variation, selection and retention	Linear & irreversible sequence of prescribed stages	Recurrent, agent-determined sequence of goal setting and action taking
Contemporary Example	Behavioral Negotiation Theory (Neal and Northcraft 1991)	Change in populations of organizations (Carroll and Hannan 1989)	The Organizational Lifecycle (Kimberly and Miles 1980)	Organizational decision making (March and Simon 1958)

Considering this analysis, for the purposes of this paper, I adapt Van de Ven and Poole's definition of a process theory as "an explanation of how and why an [entity] changes and develops" (p. 512), with two caveats: 1) I omit "organizational" from "organizational entity" because process theories could also describe actions of an individual or inanimate system; 2) I interpret "why" to encompass any combination of the four approaches to causality identified by Gregor (2006).

It is important to distinguish process theories from process models. "A process model is an abstract description of an actual or proposed process that represents selected process elements that are considered important to the purpose of the model and can be enacted by a human or machine" (Curtis et al. 1992, p.

76). Two salient differences between process theories and process models are evident – intent and scope. The intent of a process model is to *describe* a sequence of activities, whereas the intent of a process theory is to explain how and why a change occurs, where ‘how’ does not necessarily refer to an activity sequence. The scope of a process model is confined to the particular sequence of events in question, whereas a process theory seeks to explain *all of the ways* an outcome may (or may have) come about. It should, therefore, come as no surprise if process theories have little in common with process models.

Appendix D: Extended Design Literature Review

This appendix describes selected topics that are not directly relevant to the dissertation research but may seem so on first glance.

Mapping Theories. Several works attempt to provide mathematical descriptions of design in terms of mappings between two or more sets. These include General Design Theory (Tomiyama and Yoshikawa 1985; Yoshikawa 1980), Universal Design Theory (Grabowski et al. 1999), and C-K Theory (Hatchuel and Weil 2002; Hatchuel and Weil 2003). General Design Theory describes design as a mapping between an "attribute space" and a "function space." Universal Design Theory describes design as a mapping between "the set of requirements" and "the set of design parameters." C-K Theory describes design as a mapping between the Concepts (C) and Knowledge (K) used by a designer. In each mapping theory, one set contains information about the problem or situation (function space, set of requirements, knowledge) and the other set contains information about design candidates (attribute space, set of design parameters, concepts). These mapping theories are generally compatible with the Technical Problem-Solving paradigm, discussed above; however, C-K theory seeks to explicitly allow for creativity (Hatchuel and Weil 2003).

While these theories lend some conceptual support to the design space / problem space distinction discussed above, they do not provide insight into how or why designers operate beyond dividing the contents of the designer's mind into two sets and describing design as creating a mapping between them. Moreover, my literature review revealed no empirical tests of their veracity.

The Production/Deduction/Induction Model. March (1984) proposed a "model of the rational design process" (p. 270) comprised of three different forms of reasoning (see Figure A-1). In this model, the designer produces a "design proposal" from the system requirements and some presuppositions, using productive reasoning. Productive (or abductive) reasoning is the method of reasoning from facts to an explanation. The designer then predicts the characteristics of the design proposal using deductive reasoning. Next, the designer uses inductive reasoning to generate new suppositions from the design proposal and its predicted characteristics. The cycle may then repeat, using these new suppositions to generate a new design proposal and so on, until a satisfactory design proposal is found.

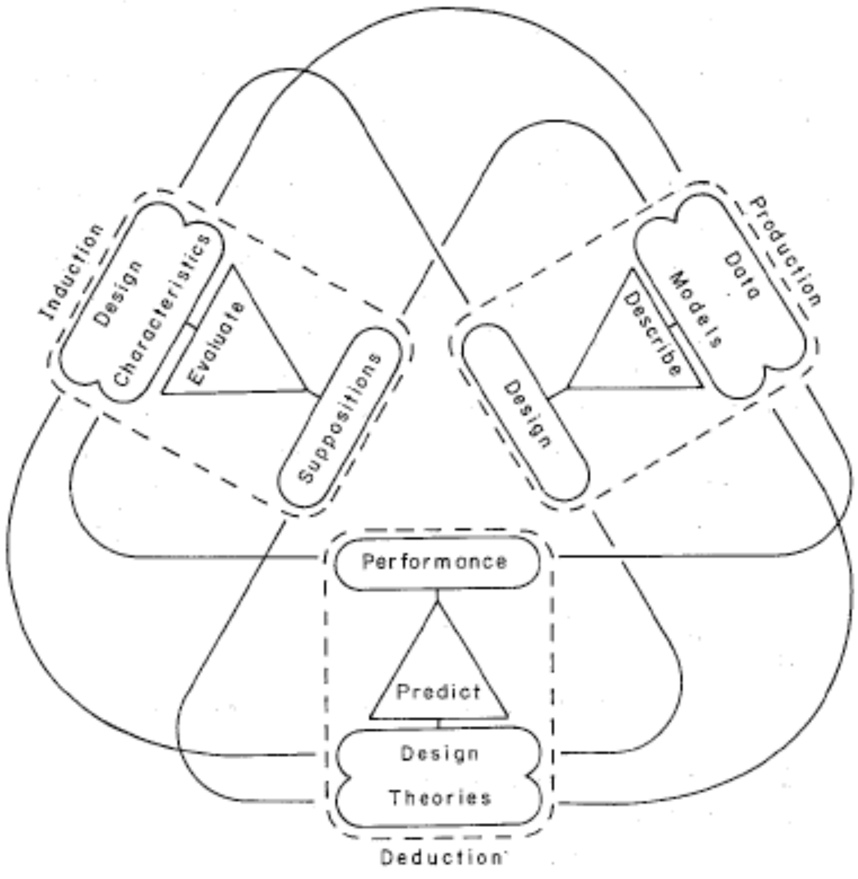


Fig. A-2. The Production/Deduction/Induction Model (from March 1984)

March's (1984) PDI Model is interesting in that it identifies three types of reasoning that may be used by designers. Furthermore, it characterizes design as "a critical, learning process in that statements inferred at later stages may be used to modify those used in earlier stages and thus to stimulate other paths of exploration" (p. 270). However, March explicitly states that the PDI model is not intended to, or capable of, describing *all* design: "If internalized personal judgment, experience, and intuition alone are relied upon, the three modes of the PDI-model become inextricably entangled" (p. 272). March further argues that such entangling is generally detrimental to the accumulation of scientific knowledge.

APPENDIX E: QUESTIONNAIRE ITEMS

A summary of the questionnaire instrument described in Chapter Four follows.

Project Summary

The purpose of this research project is to study how software is created "in the wild." At the end of the survey you will find out your "developer personality" (which is similar to a Myers-Briggs personality but specific to software development). The purpose of this study is to learn from you, not to evaluate you. This project is NOT associated with any known risks. We will not sell or share your personal information under any circumstances.

Propositions 1 and 3

Important Directions: Please answer all questions based on "your project."

Definition 1: your project is the software development project on which you are currently working. If you are not currently working on a project, your project is the last one you worked on. If you are working on multiple projects, your project is the one on which you spend the most time.

Definition 2: your team consists of everyone with whom you collaborate closely on your project, regardless of their geographic location or official title. Your team is all the people you feel like you work with.

1. Please indicate the extent to which you agree with the following statements. (Required) (Five point scale from “Strongly Disagree” to “Strongly Agree” with “N/A” option)

- No one thing drives all design decisions – they are made based on a variety of information
- Changes to my team's understanding of what the software is supposed to do were triggered by changes in our understanding of the problem/situation

- My understanding of what the software is supposed to do has been influenced by several factors (e.g., management, marketing, clients, the dev team, standards, my own values, experience on previous products, etc.)
- My understanding of the software's purpose has been influenced by several factors (e.g., management, marketing, clients, the dev team, standards, my own values, experience on previous products, etc.)
- The process of designing the software has NOT helped my team better understand the context in which the software is intended to be used
- A complete, correct specification of low-level design decisions was available before coding began (*e.g., whether to use a hashtable or array to store usernames)
- The software was coded iteratively
- My team has revised the software code based on new information (e.g., bug reports, failed unit tests, feedback from Quality Assurance, etc.)
- My team now understands what the software is supposed to do better than we did when we started coding
- Low-level design decisions* were primarily made before the first line of code was written (*e.g., whether to use a hashtable or array to store usernames)

Proposition 2

2. Important Definition: For the purposes of this survey, a model is an abstraction of the software.Models can be diagrammatic (e.g., UML class diagrams, data flow diagrams) or text (e.g., a list of requirements, a set of user stories). Just to be clear, code refers to the source code of the software.

We have found that some developers figure out the detailed design of a product through making and revising models, whereas others do their detailed design by writing and revising code. (Reminder: detailed design refers to things like deciding whether to use a hash table or an array to store passwords).

Please indicate where you and your team do your detailed design work. (Required) (five-point scale ranging from “Exclusively with models,” to “Exclusively with code.”)

- I do detailed design...
- My team does detailed design...

3. It is possible to test software in two ways: 1) Prediction: testers inspect models of the software and predict how code based on those models will behave (e.g., predict from a UML class diagram how the code will handle an error); 2) Observation: testers run the code and see what it does (e.g., unit testing, manually test the interface).

Which of these is more consistent with how your team does testing? (Required) (five-point scale ranging from “Exclusively prediction,” to “Exclusively observation.”)

About You and Your Project

Please answer all questions to the best of your knowledge. If you don't know the answer to a question, skip it, but please do not skip questions unnecessarily.

4. What is your gender?

5. What is the highest level of education you have attained?

6. How long have you been involved in the software development industry? (In-house, off-the-shelf, for-client, etc.)

7. What is your current occupation (check all that apply)

8. How many employees does your company have?

9. What roles have you held in your project? (check all that apply)

10. Approximately how many people are currently on your development team? (If the project is complete, what was the largest number of people who were on the team at one time?)

11. Please list any development methods your team is explicitly using (e.g., RUP, Extreme Programming, SCRUM, Service Oriented Architecture).

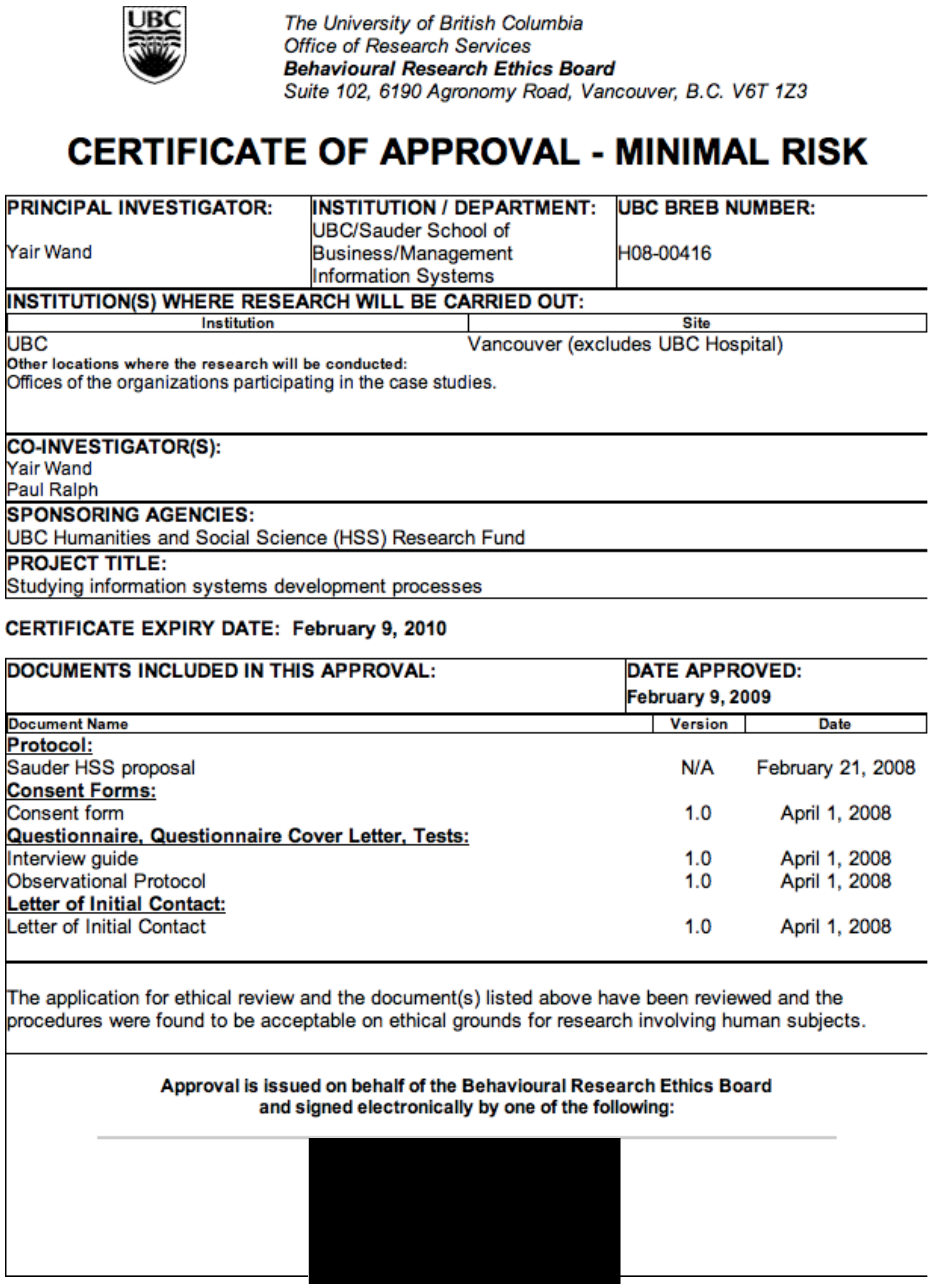
12. Approximately how long has your project been in progress?

13. Is your project more "social" (like a website) or "technical" (like a device driver)?

14. Is there anything else you would like to add to your response? Anything you feel we should know?

15. If you would be willing to discuss your results further, please enter your contact info below. (Don't worry, we hate spam as much as you do.)

APPENDIX F: BEHAVIORAL RESEARCH ETHICS BOARD CERTIFICATE OF APPROVAL



The University of British Columbia
Office of Research Services
Behavioural Research Ethics Board
Suite 102, 6190 Agronomy Road, Vancouver, B.C. V6T 1Z3

CERTIFICATE OF APPROVAL - MINIMAL RISK

PRINCIPAL INVESTIGATOR: Yair Wand	**INSTITUTION / DEPARTMENT:** UBC/Sauder School of Business/Management Information Systems	**UBC BREB NUMBER:** H08-00416

INSTITUTION(S) WHERE RESEARCH WILL BE CARRIED OUT:

Institution	Site
UBC	Vancouver (excludes UBC Hospital)

Other locations where the research will be conducted:
Offices of the organizations participating in the case studies.

CO-INVESTIGATOR(S):
Yair Wand
Paul Ralph

SPONSORING AGENCIES:
UBC Humanities and Social Science (HSS) Research Fund

PROJECT TITLE:
Studying information systems development processes

CERTIFICATE EXPIRY DATE: February 9, 2010

DOCUMENTS INCLUDED IN THIS APPROVAL:

DATE APPROVED:
February 9, 2009

Document Name	Version	Date
Protocol:		
Sauder HSS proposal	N/A	February 21, 2008
Consent Forms:		
Consent form	1.0	April 1, 2008
Questionnaire, Questionnaire Cover Letter, Tests:		
Interview guide	1.0	April 1, 2008
Observational Protocol	1.0	April 1, 2008
Letter of Initial Contact:		
Letter of Initial Contact	1.0	April 1, 2008

The application for ethical review and the document(s) listed above have been reviewed and the procedures were found to be acceptable on ethical grounds for research involving human subjects.

Approval is issued on behalf of the Behavioural Research Ethics Board and signed electronically by one of the following:

[redacted]