11-20-2008

A Teleological Process Theory of Software Development

Paul Ralph
University of British Columbia, paulralph@gmail.com

Yair Wand
University of British Columbia, yair.wand@ubc.ca

A Teleological Process Theory of Software Development

Paul Ralph
University of British Columbia, Canada

Yair Wand
University of British Columbia, Canada

Abstract
This paper presents a teleological process theory of software design in organizations. The proposed theory is compared to the Function-Behavior-Structure (FBS) Framework â a leading process theory of engineering design proposed by John Gero. A positivist, multiple case study methodology to empirically compare the veracity and predictive power of the two theories described. Results from a pilot case suggest that the observed behaviors of the development team are better described by the proposed theory than by the FBS Framework.

Keywords: Software Development, Process Theory, Science of Design, Case Study

Permanent URL: http://sprouts.aisnet.org/8-23

Reference: Ralph, P., Wand, Y. (2008). "A Teleological Process Theory of Software Development," Proceedings > Proceedings of JAIS Theory Development Workshop . *Sprouts: Working Papers on Information Systems*, 8(23). http://sprouts.aisnet.org/8-23

JAIS-TDW08-123

A TELEOLOGICAL PROCESS THEORY OF SOFTWARE DEVELOPMENT

Paul Ralph and Yair Wand

Sauder School of Business, University of British Columbia

paulralph@gmail.com, yair.wand@ubc.ca

Abstract

This paper presents a teleological process theory of software design in organizations. The proposed theory is compared to the Function-Behavior-Structure (FBS) Framework – a leading process theory of engineering design proposed by John Gero. A positivist, multiple case study methodology to empirically compare the veracity and predictive power of the two theories described. Results from a pilot case suggest that the observed behaviors of the development team are better described by the proposed theory than by the FBS Framework.

Keywords: Software Development, Process Theory, Science of Design, Case Study

1 Introduction

Software development and maintenance comprise a substantial economic activity: in 2006, the 500 largest software companies employed 2,914,480 and accrued revenues of $394 billion (Desmond 2007). Yet, software projects have seemingly high failure rates. It is estimated that in

2004, 18% of projects failed outright and 53% of projects were “challenged,” i.e., were delivered over budget, late, or with a reduced feature set, (Standish Group 2006). One factor that may contribute to project success is the process by which software is created (Baskerville et al. 1992, Wynekoop and Russo 1995).

Truex et al. (2000) argues that “The history of information systems development is typically interpreted as the history of *methods* for systems development,” (p. 56, emphasis added). Development has traditionally been conceived of as a series of phases, each phase dominated by a particular activity; e.g., in the waterfall method (Royce 1970), phases include “analysis,” “design,” “coding,” and “testing.”[1] Later perspectives on development included many of the same activity-centric phases organized into different sequences (e.g. the Spiral Model, discussed below), and the replacement of specific activity sequences by values and practices (i.e. Agile methods, discussed below).

In practice, however, the software developer is faced with a myriad of methodical choices: s/he can choose to follow one method completely or partially, combine aspects of two or more different methods, or follow no method at all. In this sense, the software developer enacts some, possibly unique, process. For the purposes of this paper, a software development *process* is a sequence of actions taken by an agent to create or modify a software product. This process may or may not resemble the particular set of prescriptions associated with a known software development *method*, which is a set of prescriptions regarding how to create or modify software. The Rational Unified Process (RUP) and Extreme Programming (XP) are methods. What a team of developers

1 However, design “begins when the design agent begins specifying the properties of the object, and stops when the agent stops” (Ralph and Wand 2008). Therefore, whether software design begins at problem identification or following requirements analysis, and ends before testing or continues through maintenance, is an empirical question. Hence, this paper examines research on both design and, more generally, development.

actually does is a process. This distinction is not standard; it is adopted to clarify the discussion in the next section.

From the perspective of the software developer, *neither existing* ***methods*** *nor existing* ***process models*** *explain the full spectrum of software development phenomena* (see **Section 2**, Baskerville et al. 1992, 2004, Truex et al. 2000, Wynekoop and Russo 1995). This raises our primary research question: *how do people develop software?*

One way to address this question is to develop and test a process theory of software development. A process theory is simply an explanation of how, and possibly why, something happens. A process theory's quality is determined by the accuracy of this explanation, i.e., it's empirical veracity. For example, in the case of software development, a process theory could explain the following transition. At time t_0, domain *d* contains an agent who intends to create a software product. At time t_1, *d* contains an agent and a software product. The proximate cause of this transition is the agent's intention to create the software; however, a process theory is needed to explain *how* the agents intention leads to a particular software product. *The purpose of this paper is to develop a process theory that accurately represents how the an agent develops a software product.* By "develop," we mean creating a new software from scratch or by modify existing software.

If the software development process can be described theoretically, it could be useful for developing, refining and evaluating software design methodologies, tools and practices, from the perspectives of both researchers and practitioners. Additionally, a theoretical understanding of this process could form an essential component of Simon's (1996) science of design curriculum.

To this end, Section Two reviews existing literature that attempts to prescribe or explain the process of software development. Section Three describes the genesis of a process theory of software design and situates its concepts and relationships in existing literature. Our empirical approach (a multiple-case design) for evaluating the proposed process theory is elucidated in Section Four. Section Five describes a pilot case we undertook to validate our empirical approach and summarizes its results. We conclude with a discussion of the possible contributions of the new theory (Section Six).

2 The Quest for a Theory of Software Design

We reviewed the literature on software methods and processes with two goals in mind. The primary goal was to identify an empirically tested theory of software development that could guide research. If such a theory was not available, the secondary goal was to identify theories, models or methods that could be co-opted as or adapted into a theory of software development. Since no tested theory of software development was found, we review the methods and process models we we identified and suggest why each is insufficient to describe the full spectrum of software development phenomena.

2.1 Common Software Development Process Models

Code-and-fix. The code-and-fix model (cf. Boehm 1988) is perhaps the simplest software development process model. In this model, the developer iterates between writing code and fixing code, where fixing code includes eliminating syntactic and logical errors and re-factoring. While coding and fixing code may be essential to software development, this model obviously has limited potential to explain design choices, team interactions, etc.

Waterfall. The waterfall model (Royce 1970) is a label given to several models 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 effectively explain the process engaged in by an agile development team (cf. Beck 2005), in which the requirements and the design emerge through the development process.

Spiral. The spiral model (Boehm 1988) 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. 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. Like the code and fix and waterfall models, the spiral model is comprised of a set of specific prescriptions about how developers *ought* to design software. Thus, it can be used to explain only the behavior of developers who follow these specific prescriptions.

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 can 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 in which desirable and feasible changes are identified. Again, SSM can be used to explain only the behavior of developers who adopt it.

RUP and USP. Some software development methods, such as the Rational Unified Process (RUP, Kruchten 2003) and The Unified Software Process (USP, Jacobson et al. 1999), contain specific process models. The activities of these models overlap with the steps of the waterfall method; however, the sequencing is more sophisticated with many activities occurring in parallel. Again, these models explain only the behavior of developers who adopt them.

Agile Methods. In extreme programming (Beck 2005) and other agile methods (cf. Abrahamsson et al. 2002), a precise activity sequence is abandoned in favor of 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 cannot greatly inform theory generation, because the prescriptions comprising agile methods do not include a process model.

Summary. Each of the models discussed above is composed of a set of prescriptions about how developers *ought* to create software. Although an exhaustive list of methods is not provided, the above sample of methods is intended to demonstrate the analytical generalization prescriptions for how development *ought to* occur cannot effectively represent the apparent diversity of software development behaviors. However, these models were not intended to describe all software development, we are not criticizing them by pointing this out, but merely evaluating whether they can be co-opted for a different purpose. Moreover, these models and methods can be useful as test cases for evaluating a process theory of software development. For example, since it is possible for a developer to use RUP to create software, a good process theory would be capable of describing the process implied by RUP.

2.2 The Case Against Methods

In attempting to explain software development, it is tempting to assume that software development is inherently a controlled, methodical process. This section presence evidence that *some* software development may neither use a method nor be methodical.

Many studies have suggested that software development *methods* are neither effectively nor extensively used (Avgerou and Cornford 1993, Bansler and Bødker 1993, Dobing and Parsons 2006, Whitley 1998). More specifically, 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). Turner (1987) found that similar methods applied in similar settings led to contrasting results. Bansler & Bødker (1993) found that developers may claim to follow a method while practically ignoring it. Furthermore, some evidence indicates that methods can be unsuitable for certain individuals (Naur 1993). Baskerville et al. (1992) demonstrated how organizations can change so quickly that long-term information systems development methods become ineffective. Meanwhile, Parnas and Clements (1986) argue that methodologies are "faked" and Nandhakumar and Avison (1999) argue that methodologies are used as "fiction" to make sense of actual practice. Truex et al. (2000) summarize the argument by asking "are such methods merely unattainable ideals and hypothetical "straw men" that provide normative guidance to utopian development situations?" (p. 53).

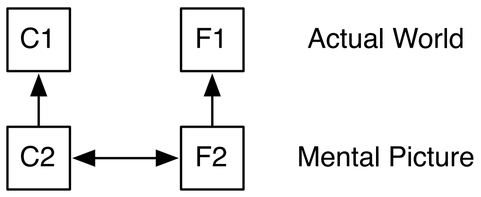
More fundamentally, Truex et al. (2000) argues that "the concept of method ... occupies an extremely privileged status in formal information systems development thought even though its origin is unstated" (p. 54), while "the possibility that *amethodical* development might be the normal way in which 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, 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, the above evidence further supports the conclusion of §2.1 that some development phenomena cannot be explained using existing software development process models. Furthermore, this evidence indicates that not all software development is as structured as it may appear from the software development literature, or even from observing practice itself.

2.3 Explanatory Models of Design

Our review of the literature also revealed general models of design (not specific to software). Unlike the models discussed above, these *are* process theories intended to describe and explain how design occurs in practice.

Alexander (1964) differentiates form (the object being designed) from context (the object's environment) and argues that a design's quality is a result of the fit between its form and context. He then suggests three classes of design process. In the "unselfconscious process," the designer directly manipulates tangible objects to eliminate misfits. (This does not apply to software design because software is intangible.) In the “selfconscious process,” (Figure 1) the designer compares his or her mental pictures of form and context to eliminate misfits mentally before or while implementing the artifact. In the unnamed third process, the designer generates formal, written pictures of his or her mental pictures to structure the design and eliminate biases.



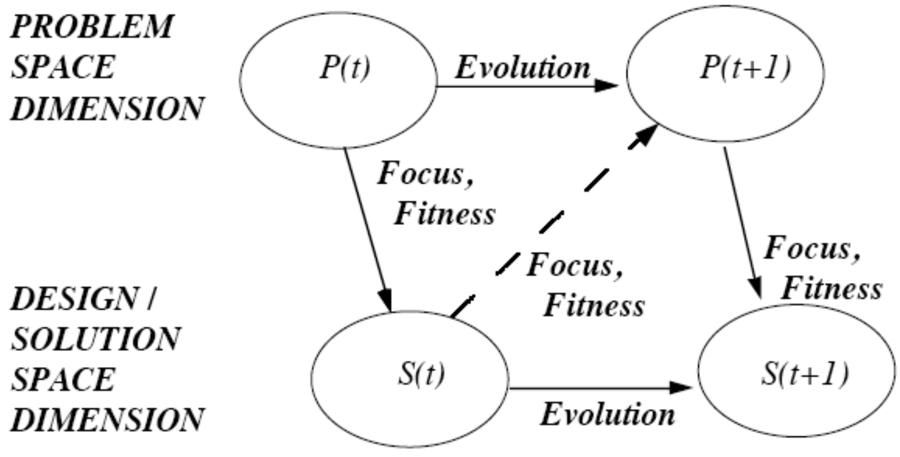
Notes
C = context
F = form
arrows indicate transitions

Figure 1: Self-Conscious Design Process (adapted from Alexander 1964)

Either of the latter two processes may describe software design. A potential criticism of Alexander’s models, however, is that they are too simple to capture the full phenomena of software design. For example, software development may include testing the software product against a set of requirements, an activity difficult to map onto Alexander’s models. Therefore, although Alexander’s models cannot serve as a process theory of design in their current form, they may inform such a theory. A second criticism is that the transitions are neither well defined nor well understood. Despite these criticisms, Alexander’s models contain important insights that may inform a process theory of software design.

Maher et al. (1995) suggest a process model of creative design (Figure 2) characterized by co-evolution of problem and solution spaces; i.e., the designer iterates between his or her ideas about the problem space (context) and solution space (form), revising both in parallel. Their 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).



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

Figure 2: Problem-Design Exploration Model (adapted from Maher et al. 1995)

Maher et al.'s model is somewhat consistent with Alexander's models in the sense that both separate the problem space or context from the solution space or form, and focus on the transitions between. Furthermore, Maher et al.'s coevolution concept is similar to the mental picture comparison in Alexander's "self-conscious process." One criticism of Maher et al.'s model is possible: while designing software may involve a search process, we can find examples in which search is not the primary activity. For instance, in designing the first version of the online social networking application Facebook, creator Mark Zuckerberg was driven by an inspiration for an online community rather than a conceptualization of the problem (Kessler 2007). Again, while Maher et al.'s model is not adequate to describe the full scope of software development, it contains important insights that may inform a process theory of software design.

Based on a survey of the design literature, Ralph and Wand (2008) define *to design* as "to create a design, in an environment (where the designer operates)," where *a design* is "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." They further present the black-box model of the design process shown in Figure 3. This seems broadly consistent with the conception of design espoused by both Alexander and Maher et al.

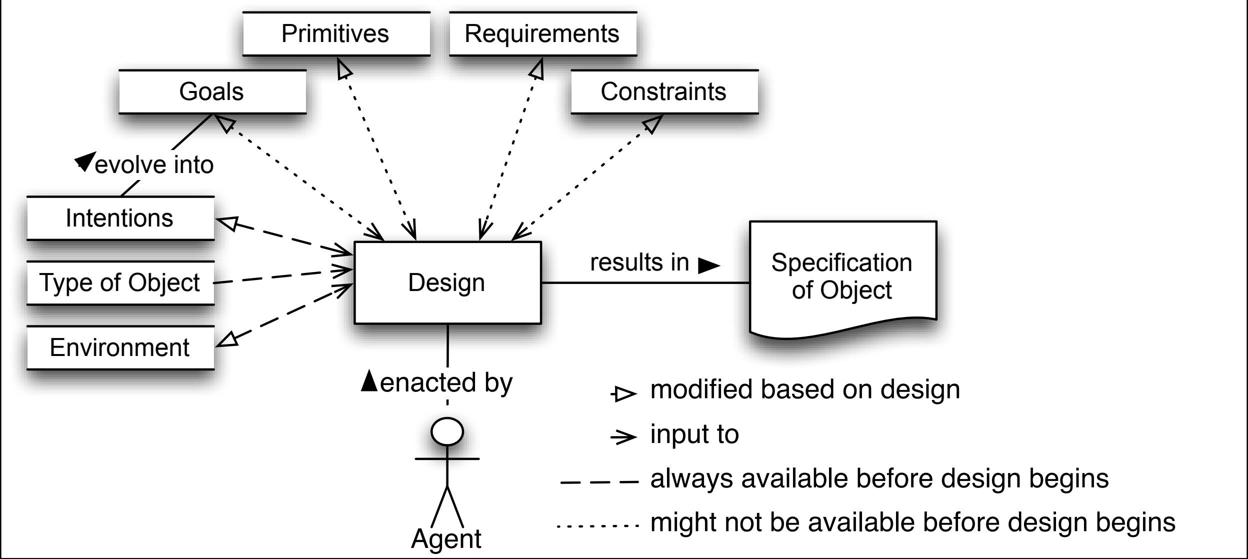


Figure 3. Conceptual Model of Design Process (adapted from Ralph and Wand 2008)

Gero (1990) suggests an engineering design meta-process, the Function-Behavior-Structure (FBS) Framework. The FBS Framework (Figure 4) describes how engineers design products using five intermediate artifacts (Table 1). It includes eight possible transformations of or operations on these artifacts (Table 2). In this model, all design proceeds from the required set of functions to a design description that is sufficiently detailed to make manufacturing possible.

The FBS Framework purports to *describe* how engineering design occurs (Vermaas and Dorst 2007). If one allows that software development is a kind of engineering design, then it seems reasonable to hypothesize that the FBS Framework can be used to describe software development. Kruchten (2005) makes a similar argument and demonstrates how can be “cast” in the FBS Framework.

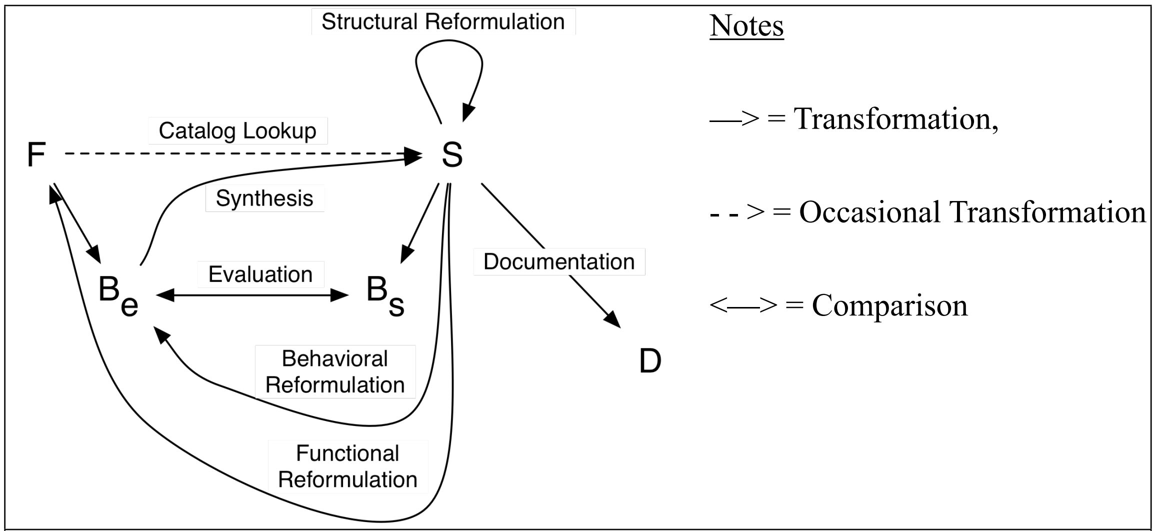


Figure 4. The Function-Behavior-Structure-Framework (adapted from Gero 1990)

Table 1. Artifacts of the FBS Framework (adapted from Gero 1990)

Symbol	**Meaning**
F	“the expectations of the purposes of the resulting artefact,” (p. 2)
S	“the artefact's elements and their relationships,” (p. 2)
B_e	the expected, or desired, behavior of the structure
B_s	“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)

However, several limitations of the FBS Framework (applied to software development) are apparent. First, it is not clear how the software structure, design description and the software product are distinguishable. Second, the FBS Framework assumes that designers are capable of predicting the behavior of an artifact from its structural description (whether developers are capable of this is an empirical question). Third, though it includes reformulation of the set of functions based on the structure, it does not describe where the functions come from. To its credit, however, the FBS Framework is significantly more specific about the artifacts and processes of software design than Alexander's or Maher et al.'s models.

Table 2. Operations/Transformations of the FBS Framework (from Gero 1990)

Activity	Definition or Description	Inputs	Outputs
Formulation	deriving expected (desired) behaviors from the set of functions	F	B_e
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)	B_e	S & B_e
Analysis	the process of deriving the behavior of a structure	S	B_s
Evaluation	comparing predicted behavior to expected behavior and determining whether the structure is capable of producing the functions	B_s & B_e	Differences Between B_s and B_e
Reformulation	changing the set of functions or expected behaviors based on the structure and its predicted behaviors	S, B_s & B_e	F, B_e
Production of Design Documentation	transforming the structure into a design description that is suitable for manufacturing	S	D
Catalog Lookup	selecting a known structure that performs the required function	F	S

Gero (2002) extends the FBS Framework, creating the (more complex) *Situated* FBS Framework. The Situated FBS Framework contains all the artifacts in the original, simpler model. Therefore, if these artifacts are not supported by an empirical test, both the original and situated FBS Framework are not supported. Given the limitations of the FBS Framework discussed above, it seems reasonable to test the original first and leave examination of the Situated FBS Framework to future work.

In summary, while each of the process models discussed in this section provides important insights for developing a (descriptive) process theory of software development, none is adequate to describe and explain the full spectrum of design phenomena in its current form. Therefore, it may be helpful to devise a new process theory. Moreover, the conceptualizations of design presented by Alexander (1964), Maher et al. (1995) and Ralph and Wand (2008) all seem compatible, while Gero's (1990) FBS Framework embodies an alternative view.

3 The Generalized Teleological Theory of Software Development (GTTSD)

To create a process theory for software design, we can draw on several resources including the models discussed in the previous section and the management literature on process theories.

3.1 A Teleological Basis

Van de Ven and Poole (1995) identify four types of process theories: life cycle, dialectic, evolutionary, and teleological, and argue that all four types can be applied to the same phenomena.

The Software Development Life Cycle model (Bourque and R. Dupuis 2004) represents an existing life cycle process theory of software development. However, this view has been considered harmful for more than a quarter of a century (McCracken and Jackson 1982). In a dialectic process model, "stability and change are explained by reference to the balance of power between opposing entities," (Van de Ven and Poole 1995, p. 517). This perspective would be problematic in describing agile software development which is based on the principle that the two main entities, *Business* and *Development*, do not oppose each other (Beck et al. 2001, Beck 2004). The evolutionary perspective might apply to either design done by evolution (e.g., using an evolutionary algorithm to design a processor chip), or to the success or adoption of a population of software products. However, explaining the development of a single product by a human design team might stretch the evolutionary perspective – it is unclear here what the population is or how individuals expire or survive and proliferate. In summary, applying three of the four types of process theories (life cycle, dialectic and evolutionary) to software development seems problematic.

Merriam-Webster defines teleological as "exhibiting or relating to design or purpose especially in nature."[2] In a teleological theory, an agent "constructs an envisioned end state, takes action to reach it and monitors the progress," (p. 516). In other words, teleological theories explain the behavior of agents taking steps to reach a purpose or goal. *Prima facie*, this is consistent with software development: the agent is the development team; the end state involves a software product; the development team takes actions such as coding and testing and the project manager monitors performance. Furthermore, agents and goals are essential aspects of design (Alexander 1964, Churchman 1971, Eekels 2000, van Lamsweerde 2004, Ralph and Wand 2008). Moreover, the above characterization of teleological theories is consistent with Alexander's self-conscious process, Maher et al.'s co-evolutionary process and Gero's FBS Framework (described in §2.3). Therefore, it seems plausible that software development may be effectively described by a teleological process theory.

3.2 Constructing a Process Theory

In this section we present the rationale behind the construction of the proposed teleological process model of design. We begin with an intuitive explanation of the theory's genesis and conclude with more precise definitions of each concept and relationship.

Alexander's (1964) self-conscious process (Figure 1, §2.3) provides a possible starting point. The first step is to give the four symbols and three relationships more descriptive names (Figure 5). Starting with the symbols, we use "Environment" for C1 and "Design Object" for F1. The labels for C2 and F2 follow Alexander's "Mental Picture of" diction. The relationship wherein the mental picture of the design object is realized in the actual design object is denoted "Imple-

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

mentation" for consistency with previous literature on software design (e.g., Beck 2005, Bourque and Dupuis 2004, Kruchten 2003). The relationship wherein the mental pictures of the environment and design object coevolve (as in Maher et al.'s theory) is labeled "Iterative Mutual Refinement." This draws attention both to iteration, which is central to software development (Berente and Lyytinen 2006) and to the idea that the mental pictures are refined *mutually*, that is, the mental picture of the design object is refined based on the mental picture of the environment, *and vice versa*. "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). The process by which the designer organizes perceptions of the environment to create a meaningful mental picture of the environment is labeled *Sensemaking*. Sensemaking refers to "The process by which individuals (or organizations) create an understanding so that they can act in a principled and informed manner," (Glossary of Sensemaking Terms, 2008).

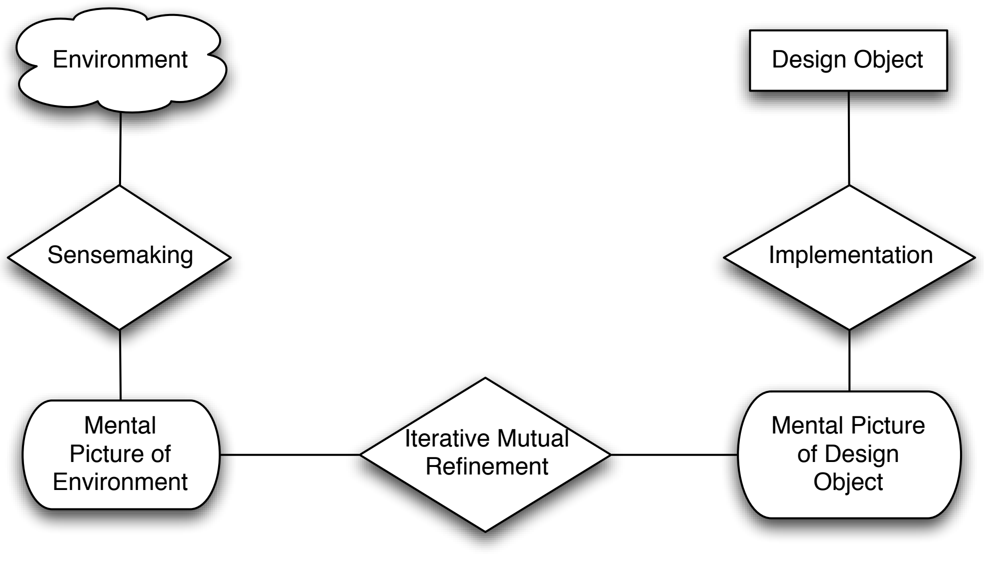

Environment
Design Object
Sensemaking
Implementation
Mental Picture of Environment
Iterative Mutual Refinement
Mental Picture of Design Object

Figure 5: Teleological Theory of Software Design - Step 1

Since the activities in the model are clearly executed my some agent, we add the design agent explicitly (Figure 6). This requires a refinement of the environment construct: the environment of the designer is not necessarily equivalent to that of the design object, especially if the designer is a person and the design object is software: one operates in a physical world, the other in a virtual world. The Design Agent is situated and operates in the Designer's environment. The Design Object exists in the Design's Environment, which (we assume) is also part of the agent's environment (otherwise the designer would not be capable of perceiving it).

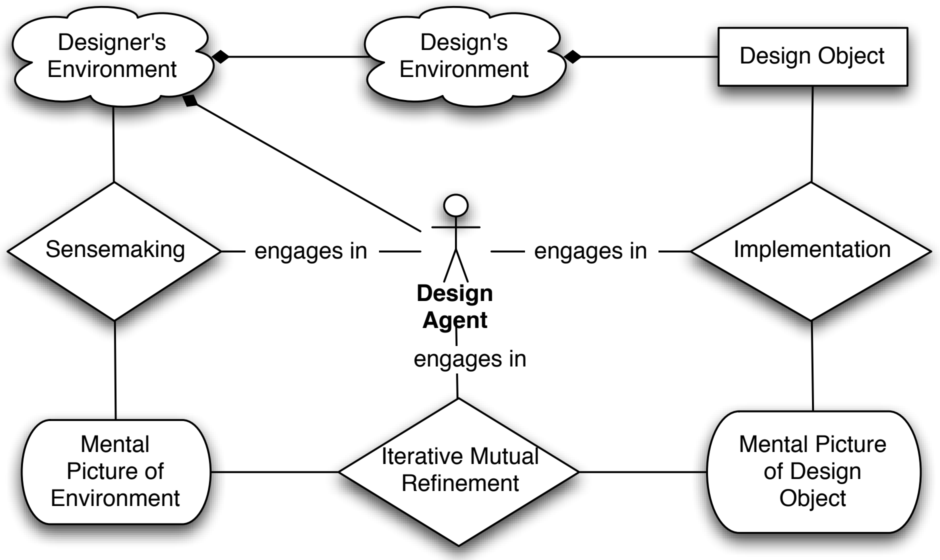


Figure 6: Teleological Theory of Software Design - Step 2

Next, we add two further concepts integral to software development: goals and primitives. Goals, of course, are also central to teleological process theories. Primitives are the objects from which the design object is constructed (Ralph and Wand 2008). Finally, since each activity involves several concepts, we give labels to the connecting lines to increase readability and provide more specific semantics. This results in *The Generalized Teleological Theory of Software Development* (GTTSD) shown in Figure 7.

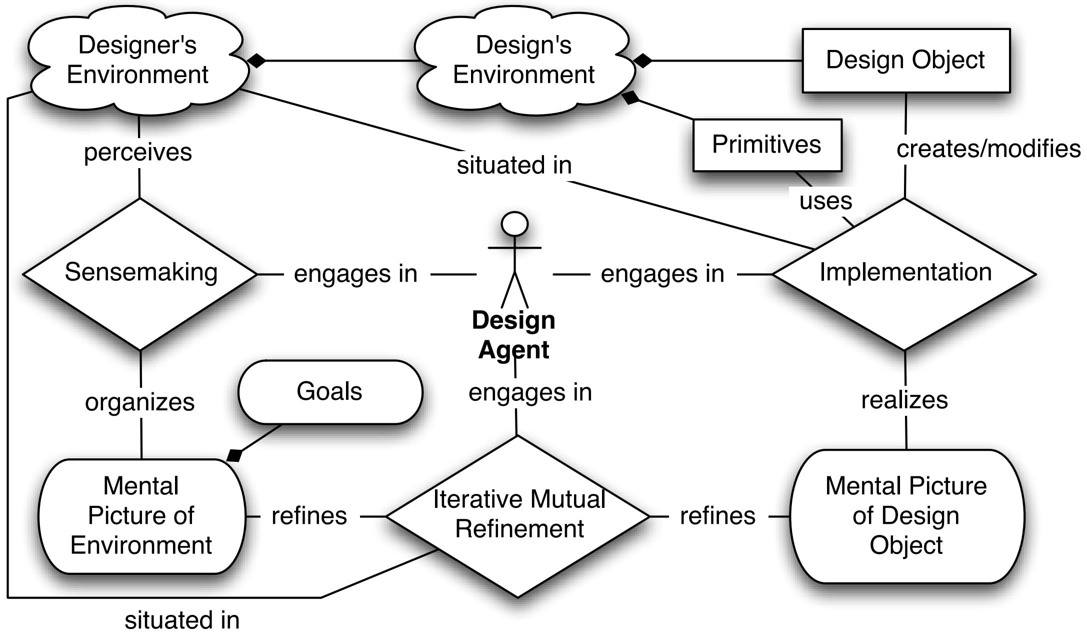


Figure 7. The Generalized Teleological Theory of Software Development

Explanation of Symbols: clouds indicate domains; diamonds indicate process; rectangles indicate objects; rounded rectangles indicate mental objects; stickfigures indicate agents.

To clarify, the GTTSD contains only three activities: *Sensemaking, Iterative Mutual Refinement* and *Implementation*. The labels on the lines connecting these activities to other concepts simply add details of how the concepts relate. For instance, the design agent engages in Sensemaking *by* organizing perceptions of the environment *into* a mental picture of the design.

Since the Mental Picture of the Design Object is an input to Implementation, the Iterative Mutual Refinement process, which initially creates the Mental Picture of the Design Object, must precede Implementation. Similarly, Sensemaking, which creates the Mental Picture of the Environment, must precede Iterative Mutual Refinement. However, once the initial mental pictures have been formed, the activities can occur in any order.

The GTTSD is not a comprehensive enumeration of all activities in which a designer might engage. For instance, designers might create formal models of their mental pictures (Alexander

1964). To create a parsimonious theory, we instead focused on the core development activities – those that we hypothesize are inherent to software development. Table 3 defines each of the concepts and relationships in the GTTSD.

Table 3: Concepts and Relationships of the GTTSD

Concept	**Meaning**	**Source**
Design Agent	the entity (e.g. group or team) that creates, or attempts to create, the design	Alexander (1964), Eekles (2000), Ralph and Wand 2008
Design's Environment	the context or scenario in which the design object is intended to be exist or operate	Alexander (1964), Ralph and Wand 2008,
Designer's Environment	the totality of the physical, organizational and conceptual surroundings of the designer	Checkland (1999), Ralph and Wand 2008
Design Object	a (possibly incomplete) manifestation of the mental picture of the design, composed of primitives, in the design's environment	Alexander (1964), Eekles (2000), Ralph and Wand 2008
Goals	optative statements (which may exist at varying levels of abstraction) about the effects the design object should have on its environment. Goals are part of the Design Agent's Mental Picture of the Environment.	Churchman (1971), van Lamsweerde (2004), Ralph and Wand 2008
Implementation	the process, situated in the designer's environment, by which the design agent realizes its mental picture of the design as a design object, composed of primitives, in the design's environment	Alexander (1964), Bourque and Dupuis (2004), Royce (1970)
Iterative mutual refinement	the process, situated in the designer's environment, by which the design agent simultaneously refines its Mental Picture of the Design Object based on its Mental Picture of the Environment, and vice versa	Alexander (1964), Dorst and Cross (2001), Maher et al. (1995), Purao et al. (2002), Schön (1983)
Mental Picture of Environment	the collection of all beliefs, held by the design agent, regarding the *designer's* environment	Alexander (1964)
Mental Picture of Design	the collection of all beliefs and decisions, held or made by the design agent, concerning the design object	Alexander (1964)
Primitives	the set of elements from which the subject *may* be composed	Ralph and Wand (2008)
Sensemaking	the process by which the agent perceives its environment and organizes these perceptions to create and refine its mental picture of that environment	Schon (1983), Weick (1995), Weick et al. (2005)

4 Case Study Design

The original research question was, *how is software designed*? This can now be operationalized as, to what extent does the Generalized Teleological Theory of Software Design explain software design practice? To determine how well the proposed process theory describes real-world development behavior, a case study approach is preferable for three reasons (Yin 2003):

1. We are interested in how things are done in practice.
2. The research focuses on contemporary events.
3. To observe real development behavior, we cannot apply behavioral manipulation.

Furthermore, the case study approach can be strengthened in at least three ways (Yin 2003):

1. testing rival theories (§4.1)
2. investigating multiple cases (§4.2)
3. running a pilot case (§5)

4.1 The FBS Framework: A Rival Theory

One hazard of a case study approach is the possibility of cherry-picking evidence to support the theory being tested. To avoid this weakness, we propose testing the GTTSD *against* a rival process theory; i.e., an alternative explanatory model of software development phenomena. For this comparison to be meaningful, the rival theory must be plausible and not simply a rhetorical strawman selected to highlight the strengths of the GTTSD.

A number of possible alternative explanations were discussed in Section 2. In selecting an appropriate rival theory, we can immediately eliminate all prescriptive process models and methods because these models obviously do not explain all software development (§2.1). Moreover, since the GTTSD extends both Alexander's and Maher et al.'s contributions (§2.3), these would make questionable rivals.

In contrast, Gero's FBS Framework remains a plausible mechanism to explain a wide variety of software development phenomena. Kruchten (2005) showed how the Rational Unified Process and the waterfall model can both be mapped into the FBS Framework. He further argues that iterative and agile development can be at least partially explained by focusing on the reformulation process in the FBS Framework. While some criticism of the FBS Framework can be made (§2.3), this criticism is not sufficient to reject it out of hand, or reduce the FBS Framework to a theoretical strawman. In summary, we conclude that, of the software development process models we have identified, the FBS Framework is the most compelling alternative to use as a rival theory.

4.2 Multiple Case Design

We propose a multiple case design to test the GTTSD against the FBS Framework.

Hypothesis. The primary hypothesis of the study is that *the GTTSD provides a more accurate explanation of how people develop software than the FBS Framework.* This hypothesis is based on the limitations of the FBS Framework (Section 2.3), which do not apply to the GTTSD. Because the two theories in question are process theories, the hypothesis is stated holistically (at the level of the theories themselves) rather than at the level of particular causal chains. Here, we seek evidence of particular concepts and activities rather than causal relationships.

Case Selection. Case studies use replication logic, not sampling logic (Yin 2003). Cases where we predict similar results are called literal replications. Since we are testing the extent to which the process theories explain all software development, any study of software development would be a literal replication. Cases where we predict “contrasting results but for predictable reasons” (p. 47) are called theoretical replications. For instance studies of a mechanical engineering design team and a software development team would be theoretical replications (since we would expect the FBS Framework to better explain the process of the former and the GTTSD to better explain the process of the latter). Ideally, a multiple-case study will involve both literal and theoretical replications.

For literal replications, the strongest test will be studying diverse development situations, for instance, a team from a small, agile-oriented project for the first case, and a team from a larger, plan- or document-driven project for the second case. A theoretical replication would also be beneficial, specifically a case on a non-software design project (e.g. architecture).

Data Collection. Data collection will consist of:

1. Interviews of participants
2. Observations of participants’ activities (recorded by the researchers in a logbook)
3. Copies of artifacts produced by participants (e.g. software diagrams, prototypes)
4. Audio or video recordings of meetings
5. Photographs or video recordings of the work environment

The initial interview will determine the project’s nature, participants’ roles and their perceptions of activities and sequences. We will then follow the project and document or record participants’ activities, meetings and workspace. Follow-up interviews will be used to clarify issues that arose in data collection and validate findings.

Coding and Analysis. The strategy here is to map observations, artifacts and statements of participants into the concepts and relationships posited by each process theory, thus creating a body of evidence for each component of each theory. Evidence may either support or discredit a particular concept. For instance, finding a requirements document and observing developers referring to that document would support the *functions* artifact in the FBS Framework. Finding that developers could not articulate beliefs about the design object's environment would discredit the *mental picture of environment* concept in the GTTSD.

Coding will proceed in parallel with the direct observations, and direct observation will end when new observations cease to provide new insights for the theories being tested. Ideally, two coders will code the data independently, allowing for measurement of inter-coder reliability. If the cost of a second coder becomes prohibitive due to the amount of data collected, inter-coder reliability can be estimated by having a second coder analyze a subset of the data collected. Coding will follow a three-step pattern matching strategy, as shown in Section 5.

Once coding is complete and conclusions have been drawn, interviews with selected participants can help to validate the findings. These follow-up interviews may be conducted a few weeks after direct observation ends. When these interviews are complete, so is the case.

5 Pilot Case and Preliminary Results

As mentioned above, running a pilot case is an important, formative step – "assisting you to develop relevant lines of questions – possibly even providing some conceptual clarification for the research design as well," (Yin 2003, p.79). Although the pilot case described below provided substantial evidence regarding the validity of both process theories, we note that the purpose of a pilot is to optimize the research protocol; supporting evidence is merely a beneficial side effect. Therefore, the data presented is intended as an illustration of the research method, **not** as comprehensive or conclusive evidence.

5.1 The Site

We conducted a pilot study with Constructive Media Inc. (CMI), a software services and development company in Vancouver, Canada. We chose CMI because the organization was familiar, accommodating and would tolerate uncertainties and experimentation. The particular team we studied had five members. A.C. and D.A. were professional web developers; J.H. was a computer science co-op student assisting with development; M.G. was the "product owner," and T.B. was a "quality assurance analyst." The team was building a web application called "Partnerpedia" (www.partnerpedia.com). More specifically, Partnerpedia is an online partner management community where businesses can find and build relationships with potential partner organizations, such as suppliers and distributors. The project was, and at the time of writing still is, on schedule and on budget, and the beta program met with significant enthusiasm from potential users.

The team takes a broadly agile approach to development and employs the SCRUM project management framework (Schwaber and Beedle 2001). In SCRUM, desired changes to the software are represented as *user stories* (Beck 2005), which are prioritized into a "product backlog." Development occurs in time-boxed iterations called "sprints" – in this case, sprints are typically two weeks. In each sprint, the team implements and tests a selection of stories chosen by the product owner.

5.2 Data Analysis and Results

Before continuing, we note that the primary purpose of this section is to illustrate the method, not to present conclusive evidence. The evidence gathered from the case supported only one of the five artifacts comprising the FBS Framework, *expected behavior* (see Table 4). The complete analysis of evidence for each artifact is provided in the Appendix. Because only one artifact was supported, analysis of the transitions between and operations on the artifacts was not meaningful. Table 5 summarizes the degree to which each concept and relationship hypothesized by the GTTSD was supported. The complete analysis of evidence for each artifact is provided in the Appendix.

Table 4. Summary of Support for Artifacts of the FBS Framework

Symbol	Meaning	Level of Support
F	The Set of Functions	Not Supported
S	The Design Object's Structure	Not Supported
B_e	Expected (desired) behavior	Medium
B_s	Predicted behavior of the Structure	Not Supported
D	Design Description	Not Supported

For the purposes of the pilot, one coder (the first author) did all of the analysis, and one of the participants thoroughly review that analysis, which was revised based on her suggestions. The coder began with the list of constructs and then identified as many relevant items of evidence as possible (shown in the Appendix). Second, the coder sorted constructs into an unlimited number of categories, depending on the level of support. Apart from constructs that were not supported, three categories emerged, which we refer to as weak, medium and strong in Tables 4 and 5. Third, the two rival theories were compared based on the level of support assigned to their respective concepts and relationships. This coding process follows a pattern matching logic (Trochim 1989), which Yin (2003) calls "one of the most desirable techniques" for case analysis (p.116).

Since some support was found for all concepts and relationships hypothesized by the GTTSD, and only one of the artifacts hypothesized by the FBS Framework, in this case *the GTTSD more accurately described the behavior of the development team than the FBS Framework*.

5.3 Lessons Learned From the Pilot Case

The primary results of a pilot case are not the empirical findings but lessons and improvements to the research design (Yin 2003) – this pilot provided five. First, we found that it was necessary to make audio recordings of meetings because important development activities occurred during meetings, sometimes too quickly to record by hand. Second, asking more direct questions about concepts and relationships from the two theories in interviews may have resulted in more direct evidence. Third, as the activities we are studying are largely cognitive, we were concerned that they would be difficult to observe; however, participants' interactions provided many clues as to their cognitive processes. To an extent, interviews can provide some confirmation of the ideas generated by these observations. Fourth, the pilot allowed us to refine our data analysis approach, resulting in the three-step pattern-matching strategy described in the previous subsection. Fifth, lack of details may make our inferences difficult for the reader to follow in some cases, e.g., during the sprint planning meeting, the team discussed how each story might be implemented. Had more details of this discussion been recorded, it might have provided more convincing evidence of Iterative Mutual Refinement. Based on these insights, in future cases we intend to record meetings, ask more concept-inspired questions and pay increased attention to interactions between participants.

6 Conclusion

This paper reviewed existing process models models that may have been used to describe software design. Finding no model entirely adequate, we synthesized the Generalized Teleological Theory of Software Design by combining and extending previous models. *The primary contribution of this paper is the GTTSD, which explains how an agent creates or modifies software.* We further proposed a multiple-case research design to test the GTTSD against its most credible rival theory, the Function-Behavior-Structure Framework of engineering design (Gero 1990). To refine the case protocol, we conducted a pilot case, which provided some evidence that the GTTSD better explains software development.

The purpose of the pilot case was to refine and test the research method. It was successful in both respects. First, it generated several insights for improving the research method (§5.3). Second, the pilot case proved by example that the proposed method is capable of discriminating among the GTTSD and the FBS framework in terms of accuracy.

The particular results of the case are promising but inconclusive – they are based on the process of a single development team, analyzed by a single coder. Furthermore, the research design was still evolving (which is partly why a pilot is useful). Therefore, the results should not be interpreted as conclusive evidence, or generalized to dissimilar contexts. However, the case does demonstrate that there exists a team of software developers whose process cannot be accurately described using the FBS Framework but can be described using the GTTSD. This motivates further study of the GTTSD and more comparison to rival process theories (§4.2).

If veracious, the GTTSD may be useful in several ways. First, it can be useful for developing and refining software design methodologies, tools and practices – the processes engaged in by all software designers are precisely what need guidance from methodologies, support from tools and addressing by development practices. For example, if a design methodology provided advice on Iterative Mutual Refinement and Implementation, but not Sensemaking, improving advice on Sensemaking may be beneficial. Second, the GTTSD can provide practitioners a lens through which to evaluate design methodologies, practices and tools. If, for example, a new tool or activity does not seem to facilitate any of the core design activities, this should raise suspicion as to the real value of the tool.

Third, Beck (2005) points out:

> *People develop software. This simple, inescapable fact invalidates most of the available methodological advice. Often, software development doesn't meet human needs, acknowledge human frailty, and leverage human strength. Acting like software isn't written by people exacts a high cost on participants, their humanity ground away by an inhumane process that doesn't acknowledge their needs. This isn't good for business either, with the cost and disruption of high turnover and missed opportunities for creative action,* (p. 24).

One area of methodological advice Beck may be referring to is the life-cycle view of software development (Bourque and Dupuis 2004), a view long considered harmful (McCracken and Jackson 1982). One interesting finding of the pilot was how the development team proceeded from an intuitive picture of a problematic environment rather than a set of functional requirements, a fact diametrically opposed to the lifecycle view. However, noting problems with the life-cycle perspective is of limited value in the absence of another perspective. The teleological perspective embodied by the GTTSD is intended to fill this gap.

From an academic perspective, the GTTSD extends or augments previous contributions by Alexander (1964), Maher et al. (2005), Ralph and Wand (2008), and Van de Ven and Poole (1995). If accurate, this model explains an important organizational activity, and provides a mechanism for unifying diverse thought on design and a foundation on which to create, evaluate and improve design methodologies, practices and tools.

Many academics and practitioners have written prescriptive accounts of how software *should be* designed (Wynecoop and Russo 1997), yet, how software *is* designed remains largely unknown (Freeman and Hart 2004, Truex et al. 2000, Wynecoop and Russo 1993, 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. As such, our next step is to examine the relative explanatory power of the FBS Framework and the Generalized Teleological Theory of Software Design in several more cases.

References

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

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

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

Bansler, J., . B. K. A reappraisal of structured analysis: design in an organizational context. *ACM Transactions on Information Systems 11*, 2 (1993), 165–193.

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

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

Beck, K. *Extreme programming eXplained : embrace change*, 2nd ed. The XP Series. 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, http://www.agilemanifesto.org/

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

Boehm, B. A Spiral model of software development and enhancement. *IEEE Computer 21*, 5 (May 1988), 61–72.

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

Checkland, P. *Systems Thinking, Systems Practice*. John Wiley & Sons, Ltd, Chichester, 1999.

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.

Desmond, J. P. The Software 500: Applications go worldwide. *Software Magazine* (Oct. 2007).

Dobing, B., and Parsons, J. How uml is used. *Communications of the ACM 49*, 5 (May 2006), 109–113.

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

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

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

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

Gero, J. S., and Kannengiesser, U. The situated function-behaviour-structure framework. In *Artificial Intelligence in Design* (Dordrecht, the Netherlands, 2002), J. S. Gero, Ed., Kluwer Academic Publishers, pp. 89–104.

Glossary of sensemaking terms, http://www2.parc.com/istl/groups/hdi/sensemaking/glossary.htm

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

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

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), 52–58.

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* (Key Centre of Design Computing, 1995), J. S. G. . F. Sudweeks, Ed., pp. 1–28.

McCracken, D. D., and Jackson, M. A. Life cycle concept considered harmful. *SIGSOFT Softw. Eng. Notes 7*, 2 (1982), 29–32.

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

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

Parnas, D. L., and Clements, P. C. A rational design process: How and why to fake it. *IEEE Trans. Softw. Eng. 12*, 2 (1986), 251–257.

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), 249–281.

Ralph, P., and Wand, Y. A proposal for a formal definition of the design concept. In *Design Requirements Engineering: A Multi-Disciplinary Perspective for the Next Decade*, K. Lyytinen, P. Loucopoulos, J. Mylopoulos, and W. Robinson, Eds., Lecture Notes on Business Information Processing. Springer, 2008 (to appear).

Royce, W. W. Managing the development of large software systems: concepts and techniques. In *Proceedings of Wescon* (1970).

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

Schwaber, K., and Beedle, M. *Agile Software Development with SCRUM*. Series in Agile Software Development. Prentice Hall, 2001.

Simon, H. A. *The Sciences of the Artificial*, 3rd ed. MIT Press, Cambridge, MA, USA, 1996.

Standish Group, The. Chaos database: Chaos surveys conducted from 1994 to fall 2004, 2006.

Trochim, W. *Outcome pattern matching and program theory*. Evaluation and Program Planning 12 (1989), 355–366.

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), 53–79.

Turner, J. Understanding the elements of system design. In *Critical issues in information systems research* (Chichester, UK, 1987), R. J. Boland and R. A. Hirschheim, Eds., Wiley, pp. 97–111.

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

van Lamsweerde, A. Goal-oriented requirements enginering: a roundtrip from research to practice. pp. 4–7.

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), 133–157.

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), 409–421.

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

Wynekoop, J., and Russo, N. System development methodologies: unanswered questions and the research–practice gap. In *the 14th International Conference on Information Systems* (Orlando, FL, 1993).

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

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

Y. Zheng, W. Venters, T. C. Agility, improvisation on enacted emergence. In *International Conference on Information Systems* (Montreal, Canada, December 2007).

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

Appendix: Data Analysis

Table 6. Summary of Support for FBS Framework Artifacts

Concept	Level of Support	Example Evidence	Interpretation
Set of Functions	Not Supported	Asked explicitly if Partnerpedia could be seen as a set of functions, M.G. replied “it’s a community... [comprised of] technology companies and individuals in them.”	This clearly indicates that M.G. (the project manager) does not see the product as a set of functions. The development team appears to view the product as a holistic entity that addresses certain goals, rather than a set of functions that addresses specific requirements.
		A.C. and D.A. consistently refer to “features.” “The content management system,” a “wiki” and “registration” were all given as example of features during interviews.	Although “feature” is the closest concept to “function” encountered, the two are significantly different. Features include whole subsystems like the content management system and non-functional characteristics like an attractive interface.
		Both M.G. and A.C. defined stories as “a promise of a future conversation,” rather than as functions.	User stories are the artifacts closest to features; however, as this quotation demonstrates, the development team does not view user stories as representing features.
		T.B.: “we don’t have requirements… I don’t have acceptance criteria, I don’t have functional specifications.”	One might expect the set of functions to exist implicitly in a requirements document or a functional specification. T.B. clearly expresses the lack of such a document.
Expected (Desired) Behavior	Medium	A.C.: “I’ll have to tell her what it’s supposed to do, the expected behavior, etc.”	Here A.C. explicitly refers to communicating the expected behavior of the product.
		The unit test suite.	A unit tests passes when the design object produces the behavior encoded in the test. Therefore, unit tests represent expected behaviors.
(Predicted) Behavior of Structure	Not Supported	Inspecting J.H.’s code, A.C. accurately predicted why it would fail and how to fix it.	This example indicates that developers form predictions of how code will behave. However, since code is not structure (see Structure row), this does not strictly support the Predicted Behavior *of Structure* artifact.
		D.A. indicated that she cannot simply imagine how repositioning a GUI element will affect the other elements; she has to change the code and view the design object to discover behavior.	This observation illustrates how developers sometimes pursue a guess-and-check strategy that does not map into the FBS Framework.

Concept	Level of Support	Example Evidence	Interpretation
Structure	Not Supported	The primary artifacts created by the team are call reports, the market requirements document, user stories, the test suite, and the source code. The first four of these include details about what the design object is supposed to accomplish, but not its “elements and their relationships.” None of these four alone or together in combination could be the design description, because they do not contain sufficient information to generate the design object. Furthermore, since the software is written in Ruby, an interpreted language, the source code *is* the software (the design object). Moreover there is no artifact that represents the structure of the design object outside of the design object itself.	
Design Descrip-tion	Not Supported		

Table 7. Summary of Support for Generalized Teleological Model of Software Development Concepts

Concept	Level of Support	Example Evidence	Interpretation
Design Agent	Weak	In the planning meeting, all team members discuss upcoming design decisions	The fact that all team members participate in the design process together indicates their shared agency.
		Retrospective meeting: D.A. says “we usually depend on [T.B.]’s feedback”	Since T.B. depends on the stories and code written by A.C. and D.A., D.A.’s statement indicates interdependence between team members.
Design’s Envi-ronment	Medium	Production server and the Internet	The Design Object exists within the production server, which is connected to the community of users through the Internet.
De-signer’s Envi-ronment	Strong	The building in which the team works.	The Design Agent (the team) is physically located in office space in a commercial building. This environment includes both physical and conceptual artifacts.
		Conceptual artifacts such as the development schedule and *mingle* project management software	
		Physical artifacts such as development workstations, desks, chairs, bulletin boards and the staging server	
Design Object	Medium	Source code and Partnerpedia website	The design object is a web application called *Partnerpedia* (www.partnerpedia.com).
Mental Picture of Envi-ronment	Medium	In the planning meeting, the team works from memory of hotmail and gmail to evaluate the reasonableness of an idea	Since the team members do not have to look at the hotmail or gmail websites to reason using these examples, this indicates that they have organized mental pictures of these aspects of their environment.
		M.G.: “it’s hard for me to separate what my ideas are from	This shows that M.G. has beliefs and ideas about “the

Concept	Level of Support	Example Evidence	Interpretation
Design Agent	Weak	In the planning meeting, all team members discuss upcoming design decisions	The fact that all team members participate in the design process together indicates their shared agency.
		Retrospective meeting: D.A. says “we usually depend on [T.B.]’s feedback”	Since T.B. depends on the stories and code written by A.C. and D.A., D.A.’s statement indicates interdependence between team members.
Design’s Environment	Medium	Production server and the Internet	The Design Object exists within the production server, which is connected to the community of users through the Internet.
Designer’s Environment	Strong	The building in which the team works.	The Design Agent (the team) is physically located in office space in a commercial building. This environment includes both physical and conceptual artifacts.
		Conceptual artifacts such as the development schedule and *mingle* project management software	
		Physical artifacts such as development workstations, desks, chairs, bulletin boards and the staging server	
Design Object	Medium	Source code and Partnerpedia website	The design object is a web application called *Partnerpedia* (www.partnerpedia.com).
Mental Picture of Environment	Medium	In the planning meeting, the team works from memory of hotmail and gmail to evaluate the reasonableness of an idea	Since the team members do not have to look at the hotmail or gmail websites to reason using these examples, this indicates that they have organized mental pictures of these aspects of their environment.
		M.G.: “it’s hard for me to separate what my ideas are from what I’ve heard from the market”	This shows that M.G. has beliefs and ideas about “the market” (part of her environment).
Mental Picture of Design	Strong	D.A. compares assumptions of stories to her memory of Partnerpedia and realizes that a story is no longer relevant.	The team members’ ability to reason from memory of the design object demonstrates that they each have a mental picture of it.
		During discussion about whether something is a bug or intentional, A.C. works from memory, without looking at the design artifact.	
		T.B. identified new features in a new build of the software by comparing it to her memory of the old build	

Table 8. Summary of Support for Generalized Teleological Model of Software Development Relationships

Concept	Level of Support	Example Evidence	Interpretation
Implementation	Medium	A.C., D.A. and J.M. were observed writing and modifying code on each day of the study. The code is "pushed live" at the end of each sprint.	Implementation includes the creation and modification of source code and the process of transferring that code to its intended environment (the production server). The developers refer to the last two steps as "pushing it live."
Sensemaking	Strong	M.G.: "I wrote *call reports*... every meeting I had I wrote a call report, just based on the information about the person, their name, where they're from, e-mail phone number facts etc.... I put the answers to specific questions, and just extra notes... and then I took an Excel spreadsheet and identified the market problems and found out how many people had this problem, how important is it... so that's how I organized that at first..."	M.G. used Call Reports to help her organize her thoughts about users and their needs. Given the complexity of the environment, the call reports act as an external cognitive aid to her sensemaking process.
		During a sprint review meeting, someone outside the team gives an example of a client's partner program organization that did not fit the developers' view of the environment. The team then discussed this example and its implications for the software.	The example that did not fit the team's mental picture of the environment triggered the sensemaking activity. During the discussion that followed, the team revised its mental picture of the environment to incorporate the new example.
		Apr 16 - J.M. begins task of fixing a file-upload bug. At first, he thinks he knows how to fix it, and tries a solution based on tag filters. When this doesn't work, he *"googled it" and found out why*: browsers don't support it. He then moves on to a second approach that he intuits.	J.M.'s mental picture of his environment includes his beliefs about how his tools andn the prototype function. When the software does not function as expected, he searches for an explanation to make sense of the situation.
		In the Planning Meeting, M.G. comments that organizing story cards into piles changes the way the participants think about them	This implies that M.G. reformulates her mental picture of her environment based on cognitive cues, and that how the story cards (cues) are laid out affects her cognition.

Concept	Level of Support	Example Evidence	Interpretation
Iterative mutual refinement	Strong	Apr 15 - D.A. and A.C. discuss the relationship between “partner applications” in the domain and in the product while cooperatively drawing a diagram to facilitate their discussion.	By the end of this discussion, D.A. and A.C. had revised their conceptions of both the environment and the design of Partnerpedia, suggesting that the two mental pictures were revised mutually. Furthermore, the refinement seem to occur in a stepwise fashion.
		Apr 15 (Sprint planning meeting): While trying to estimate the effort required by each story, the team discusses how each story might be implemented. They compare their memories of what has been built to their beliefs about desired characteristics of the site. Through their discussion, their ideas about the features become more concrete and detailed.	Through the process of agreeing on how stories should be implemented, the team makes numerous design decisions, clarifying and modifying their mental pictures of the design object. At the same time, they share and discuss ideas about potential users and their needs, clarifying and modifying their mental pictures of the environment.
		Apr 22 - T.B. asks M.G. about a story calling for the user to be able to edit an application after submitting. D.A. and M.G. discuss this and decide that this story doesn’t make sense because the application now belongs to whoever it was submitted to.	The transfer of ownership of an application is an ontological question – one dealing with the beliefs of users. Thus, in this example, a design alternative embedded in a user story (a representation of the team’s mental picture of the design object) prompted a discussion leading to a refinement of the team’s mental picture of the environment.

芽|Sprouts

Working Papers on Information Systems | ISSN 1535-6078