

The Iterating Artifact as a Fundamental Construct in Information System Design

Nicholas Berente *

Kalle Lyytinen

Information Systems Department
Case Western Reserve University
Weatherhead School of Management
Case Western Reserve University
Cleveland, Ohio - 44106

Abstract

Iterative development is fundamental principle of information system design methodologies. Yet, the idea of iteration remains poorly defined across literature streams. In this essay we identify the types of iterations that can occur in software development processes, and organize them into typologies by the iterating artifacts they generate. We then see how a sample of development approaches discuss these artifacts and how empirical research treats identified iterations. We conclude by observing that although prescriptive literature addresses a large number of iterating artifacts, empirical research focuses almost entirely on one form of iterating artifact: the evolutionary prototype. Findings associated with evolutionary prototypes are generally consistent with the expected outcomes.

Keywords: *Iterative Development, Iterating Artifact, Design Iteration, Evolutionary Prototyping, Iterative Enhancement, Software Prototyping, Agile Methodologies, Rapid Application Development, Iteration, methodologies, method engineering.*

Introduction

Although modern methods of software development stress iteration as a fundamental principle of effective software development practices (Cockburn 2002), the concept of iteration is not new. Information system design has always been an iterative process: from the formulations of earliest software development methodologies, concepts relating to iteration have been inherent in the discussions among researchers. Yet, surprisingly the concept of iteration has remained poorly articulated, and as a result we do not have strong evidence as to what type of iterations are common under what circumstances, and what their impacts are on software development outcomes such as speed, quality and cost.

“Design science ... creates and evaluates IT artifacts intended to solve identified organizational problems” (Hevner et al 2004, p.77). Although often these artifacts can take the form of software system instantiations, they can also be constructs, models, and methods (March & Smith 1995) geared to aid in the design of software systems, as well as organizations, policies, and work practices (Simon 1996). Therefore, the development of system development methodologies falls solidly within the realm of design science. The concept of iteration is fundamental to the design process and inherent in every design methodology, yet it is not a carefully articulated construct upon which methodologies can be based.

This paper explores the concept of iteration as it pertains to information systems development processes, and seeks to understand the types of iterations, their attributes, and their impacts as they emerge in system development. In the spirit of design science (March & Smith 1995; Hevner et al 2004) we seek to reveal and refine the concept of iteration and formulate it as a well-defined construct upon which future research - both normative and empirical - can be based. In order to do this, we will identify types of iterating artifacts that occur in a development process, then address both prescriptive (design science) and empirical (“natural” science) literature to understand how they treat these artifacts.

Concept of Iteration

The term concept is not always used to denote the same aspect of the design process. In fact, a small glimpse over the literature shows that it has been used to refer to a wide range of different types of repetitive, incrementally progressive activities. For example, iteration is often used to describe the cycle of incremental generation of functional code and its associated testing (Beck 2002), successive sub-phases within a main phase (Iivari and Koskela 1987), or cyclical comparisons of conceptual models to the real world (Checkland and Scholes 1999). Moreover, iteration often goes by different names, such as “prototyping” to iteratively elicit user input (Alavi 1984), “rounds” of iterative design activities to reduce risk (Boehm 1988), or even a “dance” of human interactions in efforts toward increased mutual understanding (Boland 1978).

The term “iteration” is used in a variety of disciplines and in different contexts of common speech. It is defined as “the repetition of a process” in computer science, “a specific form of repetition with a mutable state” in mathematics, and in common parlance it is considered synonymous for repetition in general (Wikipedia, 2005). Likewise the “iterative method” describes a problem-solving methodology in many fields, including computer science (Wirth 1971) and mathematics. Although all these uses bear a Wittgensteinian family resemblance (Blair 2005), simply equating iteration with repetition does not capture what is often the most salient aspect in its usage. For example, Cazanescu & Stefanescu (1994) while formally comparing “looping operations,” distinguish between the two. Iteration necessarily involves the idea of “deterministic computation” of input, which reflects a model of expected behavior. Simple repetition is in itself “nondeterministic.” This implies that iterative activity is associated with an objective and the progression towards that objective, whereas repetition has no such implication. Accordingly, all iterative methods share the description of techniques “that use successive approximations to obtain more accurate solutions ... at each step.” (Barrett, et al. 1994) The problem-solving system is said to *converge* when a solution that satisfies the problem criteria is reached through iteration.

No formal, single definition of the term “iteration” will be presented here. Rather, following the spirit of its many uses, we assume that key ideas associated with iteration are: 1) looping operations, and 2) a progression toward convergence. An important limitation we will impose on our usage is to apply it only to processes that result in explicit manifestations, or instantiations (March and Smith 1995). Many informal or non-codified activities in the development process repeat, evolve, or change, often incrementally, and even progress, aiding the design process, but we will not attempt to address them all. Rather, we will consider iterative aspects of system

development that are labeled as iterative in some sense by researchers, and which involve intentional, repetitive activities that result in a discrete artifact or instantiation through which the design is expected to progress incrementally toward a solution. Next we will identify key types of iterating artifacts found in system design and address ways in which these artifacts are treated across a sample of methodologies.

Iterating Artifacts

Simon (1996) in his *Sciences of the Artificial* indicates that the “generate-test cycle” forms the fundamental unit of design. He characterizes design as problem solving that involves interplay between cognitive and representational spaces. In his words: “solving a problem simply means representing it so as to make the solution transparent” (Simon 1996 p.132). Simon emphasizes the role of the design process i.e. how the cognitive and representational spaces are traversed during design and how these paths affect design outcomes. March and Smith (1995) build upon Simon’s ideas and describe the “artifacts” of design science: constructs (concepts), models (representations), methods (processes), and instantiations (software code) which all can be mapped either to cognitive or representational spaces.

In order to makes sense of alternative forms of iteration inherent in systems development we can examine whether we are iterating through conceptual, representational, or methodological artifacts, or through instantiations of the system itself (March & Smith 1995). For each type of iteration, we can uncover Simon’s generate-test cycles as a form of iteration for each artifact, and then examine the expected impacts of prescriptive iterative practices for each type of iteration. Based on this framework, we present in Table 1 typical forms of iteration based on the type of artifact, the form of generate-test cycle, and its expected impact based on typical examples from the literature.

Fundamentally, all artifacts associated with a design process are representations. Representations that model a portion of the system under development will be referred to as representational artifacts. Concepts associated with the design, but not directly representative of the design itself, will be captured through conceptual artifacts. Representations of the design process will be called methodological artifacts. Instantiations of the system itself – typically in the form of software code - will be referred to as instantiation artifacts. We will next examine each class of artifact in more detail.

	<u>Artifact</u>	<u>Generate- Test Cycle</u>	<u>Expected Impact</u>	<u>Example</u>
Concepts <i>Stages - formal</i> <i>Stages - cycles</i> <i>Agreements</i> <i>Problem & context</i>	Steps, stages, phases Iterations, rounds Verbal, written agreement Rich pictures, holons	document/code - approve code - test conflict - cooperation representation - reality	inevitable problem fixing incremental enhancement improve system use / user outcomes improve developer understanding	Davis 1975 Boehm 1988 Mumford 2003 Checkland 1981
Representations <i>Requirements</i> <i>Specifications</i> <i>Throw away</i> <i>prototypes</i>	Requirements docs, use cases Specifications, data models Interface, limited system	document - approve document - approve code - test	accommodate changing requirements accommodate learning - guide & track enhance requirements determination	Davis 1982 Royce 1970 Alavi 1984

<i>Documentation</i>	Instruction manual	track - document	record "as built" information	Royce 1970
Instantiations				
<i>Evolutionary</i>	Subset of total system	code - test	communication, learning, productivity	Cockburn 2002
<i>Refinement</i>	Blunt prototype of total system	code - test	performance	Brooks 1975
Methodologies				
<i>Between project major</i>	Radical method engineering	develop - measure	overall process improvement	Humphrey 1989
<i>Between project minor</i>	Incremental method engineering	experiment - review	incremental process refinement	Cockburn 2002
<i>Within projects</i>	Project plan	action - reflection	adjustment to changes	Checkland 1981

Table 1. Examples of iterating artifacts

Conceptual Iterating Artifacts

The reasoning process of a system designer is often described as abductive (retroductive) in contrast to it being inductive or deductive (Peirce 1992). Abduction involves generating a (design) hypothesis- guessing- in the face of an uncertain situation, and then working with this hypothesis until it is no longer practical – at which time another hypothesis will be generated. This view is consistent with Simon’s (1996) idea of heuristic search and selective trial and error learning. A (design) hypothesis is a form of construct, or conceptual understanding of the system, its context, and the interactions between the two.

In the spirit of our focus on design iterations with concrete, intentional output, conceptual iteration can in most cases be traced by looking at sets of design representations over time. These representations aid in the thinking processes of the designers (Simon 1996, Hutchins 1995, Boland et al 1994). Representations also support distributed cognition among designers (Boland et al 1994) and between the designers and users. In both instances these representations act as boundary objects share knowledge and align interests across multiple social worlds (Boland & Tenkasi 1995, Carlile 2000, Bergman et al 2005).

The most common conceptual artifact present in the design is the step, stage, or phase of the design. Stages are iterated if they are repeated during the design. Such iterations have traditionally been considered an inevitable, necessary evil in system development (Royce 1970, Davis 1975), but are now more commonly thought to enhance the system quality across multiple dimensions (Brooks 1995, Basili & Turner 1975, Boehm 1981, Floyd 1984, McCracken & Jackson 1982, Keen & Scott Morton 1978, Cockburn 2002, Beck 2002, Larman & Basili 2003). Such stages can be formal, such as the requirements determination phase which results in “frozen” requirements (Davis 1982), or they can be fairly indeterminate, such as “time-boxed” steps (Auer, Meade, & Reeves, 2003; Benyon-Davies et al 1999). Stages and phases of the process are constructs that are prescribed by a methodology, but are not directly related to the designs or code instantiations. Other terms for such repeating steps in the methodology are “rounds” (Boehm 1988) and “iterations” (Kruchten 2000; Larman 2004; Beck 2002).

Beyond stages or phases some researchers describe other forms of conceptual iterations. For example, systems design has been likened to a hermeneutic circle (Boland & Day 1989), where a designer iteratively compares an artifact to its context in order to understand its meaning. Checkland (1981) recommends specific representations, such as rich pictures and holons, to

guide a system developer in iterative cognitive cycles between the representations, personal mental models, and perceptions of reality that progressively refine his underlying concepts. Researchers have also likened forms of system development to dialectic cycles (Churchman 1971). Such cycles are evident in participatory approaches that encourage dialogs between system developers and the user community (Floyd 1998, Mumford 2003). These dialogs result in a series of explicit agreements concerning system functionality, the anticipated environment, or appropriate methodologies (Mumford 2003). They typically involve cycles of cooperation and conflict that are intended to improve user-related outcomes such as user satisfaction and system use.

Representational Iterating Artifacts

Our use of representational artifacts is limited to those representations that are intended to depict the information system. These representations abound in the design process, and can be intended to describe the entire system, or a portion of it. For example, requirements definitions, specifications, and data models depict slices of the entire system from a certain perspective and using a particular abstraction. Other representations are intended to illustrate only subsets of the system, for example, use cases, user-interface mock-ups, or user views. Throw-away prototypes (Baskerville & Stage 1996), if actually thrown away, would be considered representational artifacts even though they are representations made up of software code.

Early representations of the system in the design process typically take the form of requirements definitions and system specifications. Over the course of design these representations change, and often evolve into other representations, such as “as built” software documentation. Because of this need for downstream documentation, no software development methodology can overlook iteration across documents entirely, although some, such as XP (Beck 2002), do want to remove documentation from the critical path of development.

The extant literature addresses various forms of iteration related to representations – some to a great degree, such as requirements determinations (Davis 1982), data models (Hirschheim et al 1995), and the wide array of documentation within formal methodologies (Humphrey 1989; Kruchten 2000; Boehm 1981, 1988; Davis 1975; Mumford 2003; etc.). Although most of the literature addresses changing documentation throughout the design, the value of these changes is not elaborated beyond guiding and tracking. Even more nuanced views of documentation that treat its creation as problematic and argue its content to be flawed (i.e. Parnas & Clements 1986) have made no distinction between the value and cost of iterations across different forms of representation. There are some exceptions to this, however. For example, the “Inquiry Cycle Model” (Potts, Takahashi, & Anton 1994) describes iterative requirements refinement where stakeholders define, challenge and change requirements. Using requirement goals to drive such practice can be efficient, since many goals can typically be eliminated, refined, or consolidated (Anton 1996).

Instantiation Iterating Artifacts

The software code evolves through multiple instantiations in some development approaches including evolutionary prototypes or versions of a completed system. The common usage of

“iterative development” refers to software development that proceeds through “self-contained mini-projects” where each produces partially completed software (Larman 2004). This has traditionally been referred to as evolutionary prototyping (Floyd 1984, Benyon-Davies et al 1999, Alavi 1984). Although iterative development has been associated with stepwise refinement of a blunt system (Wirth 1971), it is typically equated with evolutionary enhancement, where a subset of the final code is developed to evolve into the final system (Basili & Turner 1975).

The justification for evolutionary prototyping centers on trial and error learning about both the problem and solution. Users and developers do not know what they need until they see something, and generation of prototypes assists communication better than traditional documentation, thus supporting mutual learning (Alavi 1984, Brooks 1995, Basili & Turner 1975, Boehm 1981, Floyd 1984, McCracken & Jackson 1982, Keen & Scott Morton 1978, Cockburn 2002, Beck 2002, Larman & Basili 2003, etc.). Although evolutionary prototyping is the concept most often associated with iterative development, all application software iterates over the course of its life-cycle even if the methodology is not referred to as iterative. In this sense, each version of a software system can be considered an iteration. As bugs are fixed or enhancements added to code in a linear, even in a supposedly “non-iterative” development process, any new instantiation of code can be considered an iteration. An iteration of the software can also be tested. When all or some portion of the code is compiled, the result is an iteration of compiled code. Anytime a designer replaces or adds to any part of working code resulting in some form of instantiation, he has developed an iteration of that code.

Methodological Iterating Artifacts

System designers work within implicit or explicit normative frameworks that guide their work. The explicit representations of such frameworks are known as methodologies. Such frameworks have stability and resist change: organizations are known to not change their methodologies even after numerous failures (Lyytinen & Robey 1999). Since all methodologies have limitations (Cockburn 2002), a conclusion that many have reached is that a contingency approach to selection of development methodologies is a necessity (Davis 1982, Cockburn 2002, Hardgrave & Wilson 1999, Matthiassen et al 1995, Coopriider & Henderson 1991). In this approach, selection of the appropriate methodology depends upon critical project variables such as the project’s level of uncertainty (Davis 1982), innovativeness, criticality, number of users (Hardgrave & Wilson 1999), and risk (Matthiassen et al 1995), as well as the size of the development team (Cockburn 2002), and organizational fit (Coopriider & Henderson 1991). In an effort to find an appropriate methodology, if a team were to change its methodology between projects, it might implement entirely different methodologies, or perhaps merely implement a few practices from current vogue methodologies. As methodologies change between projects, one could say the methodologies iterate toward the goal of an ideal fit. An articulation of each variant of a methodology would be the iterating artifact.

A better and more fine grained example of methodological iteration is the idea presented in method engineering literature (Brinkkemper 1996; Rossi et al 2005; Tolvanen & Lyytinen 1993). Because methodologies cannot specify all of the tasks to be done in a development project, and problems change during development, designers must reflect on their actions in order to be successful (Checkland 1981). Through this reflection, designers will learn and continuously

evolve their practices (Rossi et al 2005). As practices evolve and designers learn, capturing that process knowledge and rationale for method-related decisions can reduce errors and facilitate proper evolution of development methods (Rossi et al 2005). When users articulate changes in their development processes, they are iterating their methodology.

Any change to the development methodology may or may not improve specific outcomes. In this sense, changes are essentially experiments associated with trial and error learning around a methodology. A series of changes that take advantage of learning from iterations would conceivably lead to improvements. Incremental refinement of methodologies should lead to better outcomes than radical changes to methodology, because incremental refinements take process experience and continuous evolution into consideration.

Prescriptive System Design Approaches

There are a number of dualisms that are used in the academic treatment of system design methodologies. Discussions of iterative development contrast modern “agile” or iterative / incremental approaches with traditional linear, phased approaches – typically represented by the waterfall or life cycle method. A distinction is made between “lightweight” development practices that minimize formality with “heavyweight” approaches that require a great deal of documentation and “high ceremony.” A division also exists between “software engineering” (SE) and “information system development” (ISD) streams of literature. SE is generally more current, pragmatic, and focuses on the generation of code, whereas ISD tends to be more theoretical and concerns itself with the relationship of information systems to their organizational contexts.

To represent each of these dualisms, five approaches were identified as key representatives of software development methodologies: traditional, evolutionary, hybrid, socio-technical, and sensemaking. The process for selecting representative literature involved a systematic search for literature with “iterative development” and “agile development” related terms in three search engines: ISI Web (formerly the Social Sciences Citation Index), Google Scholar, and the ACM Portal. Also Agile and XP editions of *Lecture Notes on Computer Science* were reviewed. Based on initial results, the most heavily cited books were investigated. These texts indicated early, foundational articles as well as works that were not considered to be iterative. The main exception to this analysis was the ISD approaches (socio-technical and sensemaking). The references for these two approaches came from the Hirschheim et al (1995) review, where the authors used these as representative of the respective streams of ISD literature. In the next section, these five approaches will be briefly addressed and their treatment of iteration will be explored.

Traditional Approach – Waterfall and SDLC

There are many system development life-cycle methodologies (SDLC). The life-cycle approach is based on the assumption that all development projects should follow a series of phases that center on definition, development and implementation, with varying sub-steps based on methodology (Davis 1974). The life-cycle structure emphasizes the documents and approvals

that must be completed during each step. Iterations typically follow review processes that occur between steps and determine whether the project can go on to the next step, or whether a step must be performed again. Iterations, although inevitable, are generally a bad thing, and imply rework, error correction, and additional cost.

The Waterfall model is the most well known a life cycle methodology and is often characterized as top-down, unidirectional, and non-iterative (Royce 1970). Contrary to this popular claim about the waterfall process, even in its earliest manifestation Royce suggested that unwanted changes and their resulting iterations are inevitable, and he recommended a number of practices to address such unanticipated problems, including piloting any sizable software project with a “preliminary program design” (Royce 1970, p.331). This concept was later popularized by Brooks when he stressed to “plan to throw one away; you will, anyhow” (Brooks 1995, p.116). Royce also suggested iterative maintenance of design documentation. He understood that requirements change over time as the developer learns from the design process, and therefore the requirements should evolve through a series of at least five documents to the final documentation of the design “as built.” Updates to design documentation occur for two primary reasons: to guide or to track development.

Iterating Artifact	Description	Reason for Iteration	Roles Involved	Timing of Iteration	Value or Cost
Conceptual					
Stages - formal	Phase is repeated due to discovery of a mistake	Modify Solution	Designer	Coding stage	Cost
Stages - cycle	Activity within and between phases is linear	-			
Agreements	Rational organizational goals drive activity	-			
Problem understanding	Problem apparent, communication unproblematic	-			
Representational					
Requirements	Can be defined up-front, then frozen	-			
Specifications	Unproblematic interpretation of requirements	-			
Documentation	Program must be documented as it is created	Capture Solution	Designer	Coding stage	Neutral
Process tracking	Document progress within structured process	Capture Solution	Designer	Entire process	Neutral
Instantiation					
Software code	Changed when problem is discovered	Modify Solution	Designer	Coding stage	Cost
Object system	Users should accept, resistance is bad	-			
Methodological					
Between projects major	Method is universally applied, does not change	-			
Between projects minor	Method is universally applied, does not change	-			
Within project	Method is universally applied, does not change	-			

Table 2. Traditional approach iterating artifacts

Evolutionary / Agile Approach

Iterative development practices emerged soon after Waterfall. The idea of “stepwise refinement” involved a blunt, top-down design of the main system, then a phased decomposition and modular improvement of the code – largely to increase system performance (Wirth 1971). Stepwise refinement was criticized for requiring “the problem and solution to be well understood,” and not taking into consideration that “design flaws often do not show up until the implementation is well underway so that correcting the problems can require major effort” (Basili & Turner 1975, p.390). To address these issues, Basili and Turner recommended an “iterative enhancement” method of software development. They suggested that designers start small and simple, by coding a “skeletal subproblem of the project.” The team should develop a “project control list” that details all of the expected tasks that the system is expected to achieve. Then developers incrementally add functionality by iteratively extending and modifying the code, using the control list as a guide, until all items on the list have been addressed. Each iteration involves design, implementation (coding & debugging), and analysis of the software. When analysis finds a given iteration to be acceptable, the control list is modified accordingly, and the developers are on to the next task on the list. In this idea of iterative enhancement was the foundation for evolutionary prototyping and many modern agile development methods.

As developers implemented linear life-cycle processes, occasionally complemented with iterative enhancement or refinement practices, “software crisis” continued, where increasingly complex

programs were implemented with unending cost overruns, bugs, and problems (Dijkstra 1972, Brooks 1987). It became widely realized that software “frozen” prematurely and “modified reluctantly” in rigid waterfall-based methodologies – including incremental enhancement and refinement – was not tenable in software development due to constant and inevitable changes (McCracken & Jackson 1982). The basic sources of these changes were user requirements: “Requirements cannot ever be stated fully in advance, not even in principle, because the user doesn’t know them in advance” (McCracken & Jackson 1982, p. 31). One project manager summed up the benefits of prototyping in gaining user input as follows: “The users are extremely capable of criticizing an existing system, but not too good at articulating or anticipating their needs” (Alavi 1984 p.557). Early prototyping of systems offered users a better tool for understanding the system and enabled them to give better feedback. Specification documents do not necessarily need to be written, as the prototype evolves into the final product and essentially is the specification, tested in the user environment (McCracken & Jackson 1982). In addition to requirements elicitation, evolutionary development also addressed problem variety, political, and learning issues (Keen & Scott Morton 1978).

Anticipated benefits of evolutionary development are many. By growing the design in such a matter, software can be developed more quickly (Brooks 1987). Beyond speed, evolutionary development (or prototyping as a methodology), enabled a “more realistic validation of user requirements,” the surfacing of “second-order impacts,” and increased the possibility of comparing several alternatives (Boehm 1981, p. 656). Prototyping demonstrates technical feasibility, determines efficiency of part of the system, aids in design / specification communication, and structures implementation decisions (Floyd 1984).

Prototyping is thought to mitigate requirements uncertainty (Davis 1982), aid in innovation and increase participation (Hardgrave & Wilson 1999), reduce project risk (Matthiassen et al 1995; Boehm 1988; Lyytinen et al. 1996), and lead to generally more successful outcomes (Larman & Basili 2003). Because developers generate code rather than plan and document, they are expected to be more productive (Basili & Turner 1975, Beck 2002, Larman 2004). Therefore projects using evolutionary prototyping can be expected to cost less (Basili & Turner 1975, Larman & Basili 2003, Cockburn 2002, Beck 2002).

One problem with strict evolutionary design, however, is the lack of “iterative” planning for each prototype. Starting with a poor initial prototype could turn users away; prototyping can contribute to a short-term, myopic focus for the project; and “developing a suboptimal system” could necessitate a great deal of rework in later phases of development (Boehm 1981). Exhaustive design documentation will still be required even if prototyping is the primary process (Humphrey 1989). Also, the output of evolutionary development often resembles unmanageable “spaghetti code” that is difficult to maintain and integrate, similar to the “code and fix” problems that waterfall was originally intended to correct (Boehm 1988). Many continuing problems associated with evolutionary development include “ad hoc requirements management; ambiguous and imprecise communication; brittle architectures; overwhelming complexity; undetected inconsistencies in requirements, designs, and implementation; insufficient testing; subjective assessment of project status; failure to attack risk; uncontrolled change propagation; insufficient automation” (Kruchten 2000 ch.1).

Modern manifestations of evolutionary development practices are often labeled “agile,” or lightweight methodologies (Cockburn 2002). Agile methodologies are based on the assumption that communication is necessarily imperfect (Cockburn 2002), and that software development is a social activity among multiple developers and future users of the system. According to proponents of agile methods, increased documentation is not necessarily the answer to the weaknesses of evolutionary development practices. Rather, certain complementary activities must be in place to augment evolutionary development.

The most popular agile software development methodology is extreme programming, or XP, which promotes a variety of innovative development practices such as pair programming, test-first development, and refactoring (Beck 2002). The structure of XP is almost identical to the early evolutionary design practices, where limited functionality is initially developed, then incrementally expanded. However, XP can take advantage of a number of innovations that were not available to early evolutionary developers. Toolsets are now available that enable unit testing, efficient refactoring, and immediate feedback, and object-oriented development environments allow for modular assembly of significant portions of code. Also, process innovations such as testing-first, time-boxing, collocation, story cards, pair programming, shared single code base, and daily deployment mitigate the problems found in early evolutionary processes.

Iterating Artifact	Description	Reason for Iteration	Roles Involved	Timing of Iteration	Value or Cost
Conceptual					
Stages - formal	No phases	-			
Stages - cycle	Organic cycles or time-boxing	Modify & Capture	Designer User	Entire Process	Value
Agreements	Represented in the code	-			
Problem understanding	Communicated live	-			
Representational					
Requirements	Can be defined up-front, then frozen	-			
Specifications	Unproblematic interpretation of requirements	-			
Documentation	Program must be documented as it is created	-			
Process tracking	Time-boxing and status updates	Capture Solution	Designer	Entire Process	Cost
Instantiation					
Software code	Changed based on feedback as well as problems	Modify & Capture	Designer User	Entire Process	Value
Object system	Users and designers learn about OS implicitly				
Methodological					
Between projects major	Method can change based on contingencies	Modify Solution	Designer	Between Proc	Value
Between projects minor	Method changes incrementally with learning	Modify Solution	Designer	Between Proc	Value
Within project	Plan evolves during development	Capture Solution	Designer User	Entire Process	Value

Table 3. Evolutionary approach iterating artifacts

Many are cautious, however, that evolutionary development practices are not suited to every situation as such methods make some unrealistic assumptions. Evolutionary methods assume that projects can be structured according to short-term iterations, face-to-face interaction is always tenable and superior to formal documentation, and the cost of change remains constant over the project (Turk et al, 2005). Issues such as scaling, criticality, and developer talent will often require hybrid methodologies – or some combination of evolutionary prototypes with more formal methods (Cockburn 2002, Boehm 2002, Lindvall et al 2003). Also, evolutionary development often requires complementary innovations to succeed (Boehm 1981, Beck 2002).

Heavyweight / Hybrid Approach

While the bulk of the software engineering discipline understands the power of prototyping, many do not believe strict evolutionary development to be a universal answer to the software crisis. Rather, they greater formal planning, documentation, and discipline to be the answer, and thus developed “heavyweight” methodologies.

Boehm (1981) described his COCOMO process as the waterfall model using incremental development. Boehm’s process involved three explicit “increments of functionality”: the first of which provided “a basic capability to operate and gain experience with the model”; the second is intended to add “valuable capabilities”; and the third added “nice to have features” (Boehm 1981 p. 42). Within each increment, however, development was expected to proceed linearly with distinct phase start and endpoints. Boehm was detailed about the number of documents that were expected to be produced in each phase, and they were generally fixed after they were produced within incremental trajectories, but modified between increments. One modern example of the adapted waterfall method is McConnell’s “survival” process, which recommends essentially a repackaged waterfall process with prototyping in the requirements definition phase, and staged delivery and testing in the implementation phase (McConnell 1998).

Boehm’s spiral model (Boehm 1988) involves a great deal of planning and formal reviews, but it is based rounds of development and allows for a number of prototypes. Rather than three clear increments (Boehm 1981), now there can be any number of increments, which are guided by a hierarchy of risks. Early in development, when user-interface risks were particularly salient, evolutionary rounds of prototyping may occur until these specific risks are reduced. Later rounds, when interface-control risks might dominate, the rounds might entail a series of mini-waterfall processes. A modern descendent of the spiral model is the rational unified process, or RUP (Kruchten 2000). RUP consists of four primary phases: inception, elaboration, construction, and transition. Each phase has recursively one or more “iterations,” each of which has its own inception, elaboration, construction, and transition phases. To reduce costs, RUP is based on modular “component-based architectures” and involves rigorous quality testing of each iteration. There are also detailed document requirements with some flexibility, as project management can choose which artifacts are required for each project. Chosen artifacts are subject to strict code and document change controls. Early in the development process there is more of a focus on planning, architecture, and design. Late in the process there is more of a focus on implementation and testing.

Iterating Artifact	Description	Reason for Iteration	Roles Involved	Timing of Iteration	Value or Cost
Conceptual					
Stages - formal	Phase is repeated for enhancement or mistake	Modify Solution	Designer	Entire Process	Value
Stages - cycle	Entire cycle is repeated for enhancement	Modify Solution	Designer User	Entire / Portion	Value
Agreements	Rational organizational goals drive activity	-			
Problem understanding	Problem apparent, communication unproblematic	-			
Representational					
Requirements	Requirements can iterate before frozen	Modify Solution	Designer User	Entire / Portion	Value
Specifications	Specifications can iterate before frozen	Modify Solution	Designer	Entire / Portion	Neutral
Documentation	Program must be documented as it is created	Capture Solution	Designer	Coding stage	Neutral
Process tracking	Document progress within structured process	Capture Solution	Designer	Entire process	Neutral
Instantiation					
Software code	Changed based on feedback as well as problems	Modify & Capture	Designer User	Entire Process	Value
Object system	Users and designers learn about OS implicitly				
Methodological					
Between projects major	Method is universally applied, does not change	-			
Between projects minor	Method is universally applied, does not change	-			
Within project	Particular contingencies change within method	Capture Solution	Designer User	Entire Process	Value

Table 4. Heavyweight approach iterating artifacts

A noteworthy software engineering practice that stresses high formality is the capability maturity model of software development, or CMM (Humphrey 1989). The fundamental premise of CMM is that only statistically measured processes can be improved to reach consistency with cost, performance, and schedule expectations. In order to accomplish accurate measurement that is essential for consistency, the process must be mature. Different development teams have different levels of maturity. The failure of immature teams (levels 1&2) are not typically because of technical failure, but rather the “confused and incoherent process” due to lack of consistent management is typically at fault (Humphrey 1989 p.28). More mature teams have orderly, measured processes, actively developed standards, advanced tools and languages, etc. CMM is a top-down philosophy that stresses continuous process improvement and repeatable processes. The process itself is incrementally (iteratively) improved through the five CMM levels, with well-defined assessment methods between each level. Process reassessments are recommended annually.

A fundamental assumption of each formal methodology is that certain aspects of the development process can be frozen, documented (and therefore communicated) adequately, and built-on with subsequent phases. The spiral model’s rounds build upon frozen previous rounds. Although Humphrey warns not to freeze requirements too soon, he indicates that requirements should in fact be frozen at an appropriate time. By the final construction phases, RUP assumes the bulk of modeling, architecture and design work is completed.

Participatory & Socio-technical Approach

Many information development researchers tended to be more far more concerned with system design as a social process, which involves communication, participation, and power issues. The multidimensionality of information solutions needs to be addressed. Although they are certainly technical systems, information systems are “more primarily, organizational and social communication systems” (Iivari & Koskela 1987). A view of information systems as socio-technical systems, rife with a multiplicity of perspectives, and as potential vehicles for oppression or emancipation (Hirschheim, Klein, & Lyytinen 1995) is not addressed in much of the software engineering literature. In the cases where such issues find their way into software engineering literature, for example, the treatment can often be superficial or hypocritical (Beath & Orlikowski 1994), and is not fully internalized by the practitioner community.

A growing segment of the information system development research and practitioner community believe that “passive” user participation is not sufficient to gain an adequate fit between the process and the IS, and that continuous user involvement in the development process and the resulting knowledge of the IS will lead to improved system design and a more successful implementation (Oppelland & Kolf 1980). The socio-technical approaches, also known as participatory design approaches, not only involve users in the development process, but give them control of the development process. System designers act more like consultants to user teams who develop systems. The presence of ‘neutral’ facilitators is also required to manage the diversity and inevitable conflict in a participatory exercise involving varying interests. Socio-technical development approaches look to create a balance between social and technical aspects of the development system. (Mumford 2003; Hirschheim et al 1995).

ETHICS is a popular socio-technical, participatory methodology which resembles the life-cycle approaches in that it is primarily linear and looks to have clear understandings of the problems, objectives, etc. before the start of the design. However, within each stage, a great deal of organic, discussion-based interaction takes place to define the steps, multiple alternatives are encouraged, adding a level of iteration reminiscent of early evolutionary approaches (i.e. Keen & Scott-Morton 1978), but it is not evolution of the design itself, rather, evolution of interpersonal agreement about the design and the design process. Ideas of large groups of people, discussion, diverse goals, and conflict add much to illustrate the interpersonal tensions associated with a system design. If such cooperation and conflict does not occur early, the result would be a potentially ineffective solution – with less cooperation and more conflict after implementation when it is more difficult to make changes to the system. Agreements on the design and its process are documented and change iteratively throughout the development process.

Based on ETHICS, as well as a number of other participatory methodologies from the Scandinavian school of software development (Floyd et al 1989), STEPS was developed as a sort of culmination of various participatory methods (Floyd 1989). One important contribution of STEPS is formal documentation of expected work practices resulting from implementation of the system, and changes in these planned work practices aligning with changes in the evolutionary development of the system (Floyd 1993, 1989). Participatory methods often resemble the highly formalized hybrids of linear SDLC methodologies with prototyping principles, however, they can add unique theoretical twist to their prescriptive methodologies. For example, the PICO methodology (Iivari & Koskela 1987) requires iterative problem

solving within levels of abstraction in software design. Rather than freezing portions of the design through predetermined linear phases, developers should allow for explicit non-linear iterative activity throughout the design, and then freeze the design at specific levels of abstraction before tackling subsequent levels of abstraction.

Conceptual					
Stages - formal	Phase is repeated for enhancement or mistake	Modify Solution	Designer User	Entire Process	Cost
Stages - cycle	Activity between phases is linear	-			
Agreements	Multiple perspectives, cooperation & conflict	Capture Solution	Designer User	Early Stages	Value
Problem understanding	Problem can be understood	-			
Representational					
Requirements	Can be defined up-front, then frozen	-			
Specifications	Unproblematic interpretation of requirements	-			
Documentation	Program must be documented as it is created	Capture Solution	Designer	Coding stage	Neutral
Process tracking	Document progress within structured process	Capture Solution	Designer User	Entire process	Value
Instantiation					
Software code	Changed based on feedback as well as problems	Modify & Capture	Designer User	Entire Process	Value
Object system	Users and designers learn about OS explicitly	Modify & Capture	Designer User	Early Stages	Value
Methodological					
Between projects major	Method is universally applied, does not change	-			
Between projects minor	Particular contingencies change between projects	Capture Solution	Designer User	Entire Process	Value
Within project	Particular contingencies change within method	Capture Solution	Designer User	Entire Process	Value

Table 5. Participatory approach iterating artifacts

Sensemaking Approach

Sensemaking approaches to information system design emerged around issues concerning problem formulation. Problems, or requirements, do not exist objectively on their own in the world, but rather are thought to be “constructed between various stakeholders adhering to various perspectives” (Hirschheim et al 1995, p.37). Such approaches criticized other methods for taking simplistic, unrealistic view of meaning, and a mechanistic view of organizations (Lyytinen 1986). The most well-known sensemaking approach is the soft systems approach (SSA), developed by Peter Checkland (1981).

The soft systems approach conceives of organizations as dynamic and socially-oriented. Different people make a variety of judgments about reality and values. Common goals and rational decision making do not guide action, rather peoples’ judgments and interpretations of meaning guide action. This social action, in turn, influences the way in which people make judgments and meaning (Checkland 1994). Checkland found the traditional goal-oriented methods of problem definition and problem solution to be limiting when applied to complex, ill-defined “wicked” problems (Checkland & Scholes 1999). The soft systems approach downplays

the technical aspects of information system development and conceives of it as “mostly a social process” (Hirscheim et al 1995).

The soft systems approach is concerned with improving a problem situation, focuses on the “planning process” (Checkland & Scholes 1999). This process involves refinement of a designer’s conceptual model of the problem through multiple cycles of Checkland’s seven steps. These conceptual models are based on a designer’s understanding of the salient relationships, perceptions of the problem, political dimension, and social roles, norms and values. To document this understanding, Checkland recommends conceptual representations called “rich pictures,” “holons,” and “root definitions.” The designer’s conceptual models are linked together and extended to resolve the problem in an iterative fashion - between conceptual models and the real world. The analyst can cycle through all seven stages, among a few stages, or within a stage a number of times throughout the analysis. To understand a given process, the analyst iterates cognitively between perceptions of the social world external to him, his internal ideas, various representations, and the methodology of the analysis (Checkland and Scholes 1999).

Conceptual					
Stages – formal	Not specified	-			
Stages – cycle	Unlimited cycling between stems	Modify & Capture	Designer User	Early Stages	Value
Agreements	Designer wields all power	-			
Problem understanding	Conceptual iterations to gain understanding	Modify & Capture	Designer	Early Stages	Value
Representational					
Requirements	Object system representations iterated	Modify & Capture	Designer	Early Stages	Value
Specifications	Not specified	-			
Documentation	Not specified	-			
Process tracking	Not specified	-			
Instantiation					
Software code	Not specified	-			
Object system	Active explicit interaction with the object system	Capture Solution	Designer User	Early Stages	Value
Methodological					
Between projects major	Method is universally applied, does not change	-			
Between projects minor	Method is universally applied, does not change	-			
Within project	Designer reflects and adjusts method accordingly	Modify & Capture	Designer	Early Stages	Value

Table 6. Sensemaking approach iterating artifacts

Although this form of inquiry is likely to offer the richest view of any social and cognitive phenomena, it is apparent that there is little focus on overall cost and rework associated with iterations during the design beyond what can be articulated through the perceptions and interests of individuals within the process. Also, due to the cyclical nature of the methodology, no formal “universal” steps are addressed to provide a sense of progress. Rather, this process of inquiry resembles the evolutionary methods, where convergence is the goal rather than fit into neatly planned steps.

Discussion of Iterating Artifacts

From above analysis, it is apparent that different methodologies vary in the iterations they address, and the reasons for iteration. For example, to address changing user requirements, traditional approaches iterate representational artifacts such as documentation, whereas evolutionary approaches iterate the instantiation. Both methods iterate to capture requirements, but they do so using different artifacts.

The philosophy associated with this iteration is also distinct across approaches. For example, traditional, heavyweight, and participatory methods target towards “freezing” requirements, which is expected to put an end to iterations. Evolutionary and sensemaking approaches question whether freezing can ever really occur. Table 7 summarizes the iterating artifacts that are addressed in the different approaches, and their expected impacts.

Empirical Impacts of Iteration

In the spirit of natural science as an informing source for design science, the empirical literature on the impacts of iteration on software development should be reviewed and causal connections between specific design artifacts and interventions and outcomes evaluated. There are problems associated with assessing the impacts of iteration, however. Empirical research on iterative software development is as scarce while the prescriptive research is plentiful (Gordon & Biemen 1995; Lindvall et al 2002; Wynekoop & Russo 1997). Also, with the absence of any theoretical treatment of iterations other than the system code itself or its prototypes, the empirical research has primarily focused on software prototyping and related methods (Alavi 1984, Boehm et al 1984, etc.). Nevertheless despite the sorry state, we will assess the state of empirical research across each of the four levels of iterating artifacts: concepts, representations, instantiations, and methods.

Approach	Iterating Artifacts Type	Description	Expected Impacts
Traditional	Conceptual: Stages - formal	Phases in linear process	Discourage costly & inevitable problem fixes
	Representational: Documentation	Capture solutions to problems	Track changes to code
	Representational: Process Tracking	Capture progress of project	Monitor project through phases
	Instantiation: Software Code	Fix problems according to phases	Fix problems
Evolutionary	Conceptual Stages - cycle	Scheduled and informal iterations	Enables enhancement of code
	Representational: Process tracking	Capture progress of project	Monitor project over time
	Instantiation: Software code	Enhance code, fix problems	Better code, less costly, stronger user-related outcomes
	Method: Between projects major	Change method	Contingency alignment
	Method: Between projects minor	Improve method	Learning and experience
	Method: Within project	Refine method	Learning and experience
Heavyweight	Conceptual: Stages - formal	Phase within cycle is repeated	Enables disciplined enhancement / problem fixes
	Conceptual: Stages - cycle	Scheduled repetition of entire cycle	Enables enhancement of code
	Representational: Requirements	Requirements can iterate before frozen	Guide specifications
	Representational: Specifications	Specifications can iterate before frozen	Guide development of code
	Representational: Documentation	Code documentation	Track development of code
	Representational: Process tracking	Formal process documentation	Monitor project through phases
	Instantiation: Software code	Enhance code, fix problems	Better code, stronger user-related outcomes
	Method: Within project	Adapt process within method	Better fit to support code development
Participatory	Conceptual: Stages - formal	Phases in linear process	Discourage costly & inevitable problem fixes
	Conceptual: Agreements	Document cooperation/conflict results	Accommodate multiple perspectives
	Representational: Documentation	Capture solutions to problems	Track changes to code
	Representational: Process Tracking	Capture progress of project	Monitor project through phases
	Instantiation: Software Code	Fix problems according to phases	Better code, stronger user-related outcomes
	Instantiation: Object System	Address object system context	Accommodate multiple perspectives
	Method: Between projects minor	Improve method	Accommodate multiple perspectives
	Method: Within project	Refine method	Accommodate multiple perspectives
Sensemaking	Conceptual: Stages - cycle	cycling between designer and context	Enables the progressive improvement of understanding
	Conceptual: Problem understanding	iterations btwn reps and world	Refine understanding
	Representational: Requirements	Object system representations iterated	Tool for improvement of understanding
	Instantiation: Object system	Active explicit interaction with object system	Improve and refine understanding
	Method: Within project	Change method based on understanding	Supports designer understanding

Table 7. Iterating artifacts of each approach and expected impacts

Conceptual Iterations - Empirical Research

Some empirical research outside information systems has been conducted on iterative cognitive processes of designers (Simon 1996, Malhotra et al 1980, Cross 1989, etc.). Some empirical research has been conducted that is also specific to software developer cognition, as well (Curtis, Krasner, & Iscoe 1988; Boland & Day 1989). Of the empirical research on cognitive aspects of

software design, none significantly addresses the iterative cognitive processes of a developer and relates them to design outcomes. Most observe challenges related to iteration. However, if we view conceptual iterations to be the sharing of concepts, perspectives, and understandings during the design process, there is some research that focuses primarily on iterative interactions between developers and users. An early study compared traditional unidirectional flow of problem information from user to developer during requirements definition with an alternative and more iterative dialog, where both the user and developer prepared their suggestions and offered feedback. The iterative method generated greater mutual understanding, system design quality, and system implementability (Boland 1978).

Representational Iterations - Empirical Research

Representational artifacts include all of the documents, data models, and other physical representations of the software, including artifacts such as user-interface mock-ups and “throw-away” prototypes. Documentation, data models, and other representations are addressed quite extensively in the prescriptive literature, but iterations of representations and the effect those iterations on design outcomes is absent from empirical development literature. The primary exception to this is the research on “throw-away” prototypes. The problem with empirical research on such prototypes is that they are usually lumped in with prototyping in general, and empirical research that actually distinguishes between evolutionary and “throw-away” prototypes finds that there is no significant difference in many of the outcomes (Gordon & Biemen 1995, 1993). Therefore, all forms of prototyping will be addressed as instantiations of the system itself.

Instantiation Iterations - Empirical Research

The notion most commonly associated with iterative systems development is evolutionary prototyping, where either a subset (in the case of enhancement) or a blunt version (in the case of refinement) of the final code is initially generated, and then it iteratively evolves over the course of the project. Evolutionary prototyping, or the common label of “iterative development,” is fundamental to modern agile methods (Cockburn 2002; Beck 2002; Highsmith 2002), but is also important to evolutionary methodologies (Basili & Turner 1975; Wirth 1971; Boehm 1987; Alavi 1984). Although many researchers distinguish between prototypes that occur at different stages of the design cycle, and are used for different purposes (Floyd 1984; Janson & Smith 1985; Benyon-Davies et al 1999; Gordon & Biemen 1995, 1993), the empirical literature does not typically highlight a distinction between types of prototypes and their outcomes.

The fundamental reason Basili & Turner advocated iterative development is that problems and solutions are not well understood at the outset of a project, and even if they were “it is difficult to achieve a good design for a new system on a first try” (1975 p.390). Subsequent empirical research found prototyping to be an excellent method for users and developers together to learn about the requirements of the new system (Naumann & Jenkins 1982; Alavi 1984; Boehm, Gray, & Seewaldt 1984; Necco, Gordon, Tsai 1987). Prototyping has been found to support communication and problem solving between users and developers (Mahmood 1987; Deephouse et al 1996), and led to greater user involvement in the design process (Naumann & Jenkins 1982; Alavi 1984; Gordon & Bieman 1995). Improved aspects of user participation are often credited

with better user satisfaction (Naumann & Jenkins 1982; Necco, Gordon, Tsai 1987), designer satisfaction (Mahmood 1987), ease of use (Gordon & Bieman 1993; Boehm, et al 1984), and greater use of the system (Alavi 1984; Mahmood 1987). Research on the effects of prototyping on system performance is generally mixed (Gordon & Bieman 1993). Some found prototyping to be positively related to higher system performance (Alavi 1984, Larman 2004), but others found that prototyping might create less robust, less functional systems, with potentially less coherent designs (Boehm et al 1984), and may call for “negotiable” quality requirements (Baskerville & Pries-Heje 2004).

Whilst they advocated iterative development, Basili & Turner (1975) indicated that software created through evolutionary prototypes can require less “time and effort” than traditional methods, and the “development of a final product which is easily modified is a by-product of the iterative way in which the product is developed” (1975, p.395). A large number of subsequent studies indicate that prototyping can shorten lead times for projects and/or less effort, typically measured by fewer man-hours (Naumann & Jenkins 1982; Boehm, Gray, & Seewaldt 1984; Necco, Gordon, Tsai 1987; Gordon & Bieman 1995; Subramanian & Zarnich 1996; Baskerville & Pries-Heje 2004). A number of studies also support the assertion that modular evolutionary prototyping results in more maintainable systems (Boehm, Gray, & Seewaldt 1984; Gordon & Bieman 1993).

In most empirical studies, iteration is treated as an independent variable that affects important outcomes (Gordon & Bieman 1993, 1995). Moderators are often introduced, for example, prototyping must be combined with factors such as powerful development tools (Naumann & Jenkins 1982; Alavi 1984), a standardized architecture (Baskerville & Pries-Heje 2004), greater developer expertise (Gordon & Bieman 1995), a complementary culture (Lindvall et al 2002; Beynon-Davies et al 2000), and “low technology” artifacts and processes for scheduling and monitoring (Beynon-Davies, Mackay, & Tudhope 2000). Prototyping can also be seen as a dependent variable. For example, researchers found that prototyping may pose challenges for management and planning (Alavi 1984; Boehm et al 1984; Mahmood 1987).

In recent years there has been a dearth of rigorous research on the effects of prototyping on system development, and much of the empirical literature on modern agile methods is anecdotal (Lindvall, et al 2002). Although studies in the past have typically compared prototype-based processes to specification or plan-based processes, modern empirical research will likely assess varying combinations of iterative and specification-based processes (e.g. Matthiassen et al 1995), or compare agile methods. When pursuing either of these research avenues, it would make sense to adopt a more granular and refined view of iteration and also define more carefully the dependent outcome variables.

Methodological Iterations - Empirical Research

The methodology is in a constant state of evolution and reification, and this dynamic activity is not necessarily the result of iteration. When groups together settle on a given methodology, then they must work to explicitly update this methodology during the design process. This would entail an in-process methodological iteration. Empirical literature focuses on what effects may occur if methodologies do not change or iterate (Lyytinen & Robey 1999), benefits of moving to

a given methodology (see prototype review, for example), and discuss problems associated with adopting new methodologies (Beynon-Davies, Carne, et al 1999). However, research does not empirically address the effect of methodological iterations within and across projects. The only exception is recent methods engineering research that shows how a given approach yielded greater developer satisfaction and reduced overhead resource usage in comparison to previous techniques, but even this study was based on cases with single iterations (Tolvanen 1998).

Discussion

Information systems development process can be essentially defined as a change process of an object system to achieve certain objectives (Hirshheim et al 1995). In this sense, designers are addressing issues of the current system and identifying and mobilizing resources that will enable change from the current socio-technical system to a new one of the future. Due to uncertainty and ambiguity this is by necessity iterative. All design can be thought to be the design of tools (Gargarian 1998), and thus information system development can be thought of as the design of tools, or tangible artifacts, that enable the creation of a desired socio-technical system.

It is important to understand, however, that information systems are dynamic (Orlikowski & Iacono 2001). They are always evolving, under revision, and behaving differently in unique contexts. We assert that there is no single entity that *is* the system in a given development process, rather we view the system to be a concept that can only be approximated through representations (Lyytinen 1987). Early in the design process, the information system may be little more than an idea generated by a handful of people whose only tangible artifact is a vague requirements idea. Later in the process the information system may be represented by lines of incomplete code, dozens of use cases, and a great number of varying expectations of the system's utility. Yet throughout the process, individuals can all discuss the information system as if it were a single, discrete entity, although all individuals only have partial views (Turner 1987).

Much of design science research considers system development to be essentially a complex problem solving exercise (Hevner et al 2004). As such, the design process can be thought of a "nested series" of generate-test cycles (Simon 1996 p.129), where representations of designs are created then tested against a pre-defined schema. Simon uses the idea of a "space" to illustrate the evolution of a design. The design takes place in both a cognitive space and a physical, representational space. Since both spaces are "artificial," or generated by humans, the complexion of these spaces change and are affected by the conceptions of the humans. For example, Simon indicates that the process chosen for a design will affect the style of the design; a given conceptualization of the problem will align the resulting design with that conceptualization; and a design focused on certain resource constraints will differ significantly from a design that addresses other resource constraints. However, the initial conceptions and goals of the design are apt to change over the course of the design, as designers continue to learn about both the problem and the solution as "each step of implementation created a new situation; and the new situation provided a starting point for fresh design activity" (Simon 1996 p.163). Therefore, due to its very nature, systems development will by necessity involve iterating representational artifacts. System development is a social exercise where communication and

cognition is necessarily supported by representational artifacts that evolve through cycles of learning, problem solving, negotiation, etc.

Since all system development is iterative, one might question the actual difference between modern “iterative” development and traditional methodologies. The answer appears to be in the way that the different methodologies view the idea of iteration, and in the audience for a given iteration. The first distinction lies in the view, or attitude toward iteration. Traditional methodologies attempt to avoid or minimize iteration, viewing it as an inevitable evil that can only be managed through better planning. Evolutionary, or iterative, development processes embrace iteration as the mechanism by which developers can reach better project outcomes. Traditional methodologies have attempted to deal with iteration as a reaction to an error or problem, whereas modern methodologies proactively seek the value of planned iteration. The other difference between iterative and traditional development is the audience for the iteration. Iterative development creates iterations specifically for use and feedback from some portion of the clientele, whereas traditional development keeps its iterations within the developer community until a complete release is made.

The first distinction, the proactive view of iteration, is addressed by foundational researchers through notions such as iterative enhancement (Basili & Turner 1975) and evolutionary prototyping (Floyd 1984; Alavi 1984), and has now been embraced by agile methodologies (Cockburn 2002; Beck 2002; Larman 2004). The second distinction, which is the visibility of iteration has not been addressed, and can only be understood through a fundamental understanding of the artifacts used in a design process. In this paper we have attempted to tease out iterations associated with system development by identifying sets of artifacts that are described in the extant literature. By establishing a nuanced understanding the information that each artifact makes visible, the role of the various artifacts, and the relationships between these artifacts, better methodologies can be designed.

The four artifacts we have identified are located at different levels of abstraction and they often overlap. For example, methodology is a particularly important artifact in the development process, since the methodology frames the conception of the design and therefore the design’s outcomes, as mentioned above (Simon 1996). A given methodology, however, can be considered a function of cognitive understanding (Brinkkemper 1990; Rossi et al 2004, Cockburn 2002), and might therefore be considered a conceptual artifact which iterates. Likewise, all representational artifacts typically represent conceptual, or methodological facets, or instantiations of the system itself. Instantiations can be self-evident (evolutionary prototypes), or they can be characterized by representational artifacts (such as throw-away prototypes), or they can be mental constructs of the designers.

Although separation into these four artifacts is expedient, future research may look to establish abstracted relationships between these artifacts such as those presented in figure 1: methodological artifacts form a subset of conceptual artifacts, which are manifested through representational artifacts. Design instantiations are made evident through themselves, their representations, and conceptions of the designers. The “realm of tangibility” reminds us that once there is an attempt to communicate about an artifact, it becomes a representation of that concept.

Our conceptualization of the relationship of these four artifacts follows:

- Conceptual artifacts are representations of constructs that support the design process; such as stages in the process, or representations of a methodology
- Representational artifacts are representations of the system-in-development, or software code
- Instantiation artifacts are manifestations of the code itself
- Methodological artifacts are conceptual artifacts relating specifically to the process from a meta level

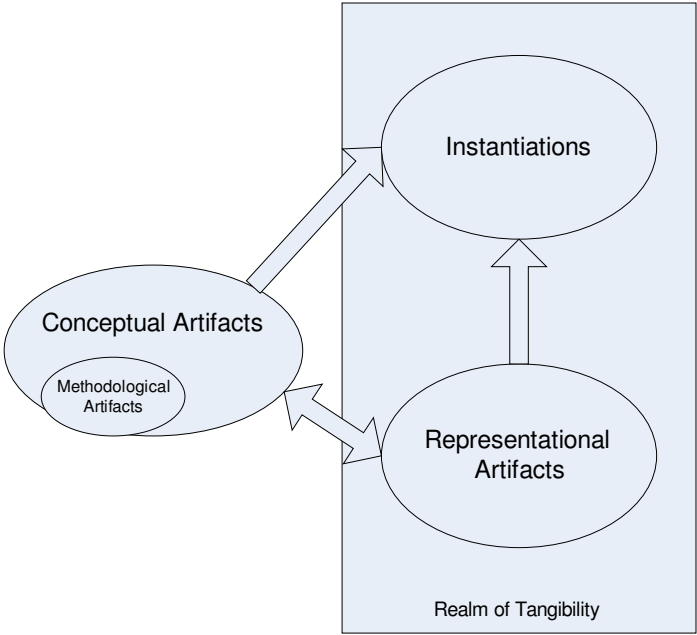


Figure 1. Relationship among design artifacts

Given this extension of the March & Smith (1995) framework, we identified a number of artifacts in the software engineering and information systems development literatures (Table 8), and it is apparent that empirical literature is essentially void of any content other than evolutionary iteration of code itself (with the exception of throw-away prototyping in many cases).

Table 8 indicates that research on the impact of iteration and the artifacts of the system development process is primarily focused on the instantiations of the code itself. While this is arguably the central aspect of software development, there are a good number of other iterating artifacts that are either prescribed by researchers, or observed in practice. Yet, the literature is conspicuously void of any empirical research beyond iteration of the instantiations themselves.

	<u>Expected Impact</u>	<u>Empirical Research</u>
Concepts		
<i>Stages - formal</i>	inevitable problem fixing	-
<i>Stages - cycles</i>	incremental enhancement	-
<i>Agreements</i>	improve system use / user outcomes	-
<i>Problem & context</i>	improve developer understanding	Boland 1978
Representations		
<i>Requirements</i>	accommodate changing requirements	-
<i>Specifications</i>	accommodate learning - guide & track	-
<i>Throw away prototypes</i>	enhance requirements determination	-
<i>Documentation</i>	record "as built" information	-
Instantiations		
<i>Evolutionary</i>	communication, learning, productivity	Naumann & Jenkins 1982; Alavi 1984; Boehm, Gray, & Seewaldt 1984; Necco, Gordon, Tsai 1987, Mahmood 1987; Gordon & Biemen 1995, 1993; Subramanian & Zarnich 1996; Deephouse et al 1996; Beynon-Davies, Mackay, & Tudhope 2000; Lindvall et al 2002; Larman 2004; Baskerville & Pries-Heje 2004
<i>Refinement</i>	performance	-
Methodologies		
<i>Between projects major</i>	overall process improvement	-
<i>Between projects minor</i>	incremental process refinement	-
<i>Within projects</i>	adjustment to changes	-

Table 8. Empirical research on iterating artifacts

Conclusion

In this essay, we have identified a number of iterating artifacts that are present in the system design process, and have distinguished among them based on the type of artifact and the expected and empirically investigated impacts of different forms of iteration. Our findings show that, although many different types of iteration are prescribed in the development process, only evolutionary prototyping has been subject to significant rigorous empirical research.

Our sparse empirical findings may come as a surprise, given the large volume of empirical research that is evident in the literature. However, when one looks to empirical literature to

specifically tease out the implications of iterative activity, there is little to find. For example, even literature that investigates the impact of modern agile methods such as XP focuses enablers of iteration such as pair programming (Williams et al 2000), while taking for granted the beneficial aspects of iteration.

This paper reminds researchers and practitioners that iteration forms the first principle of *any* system development project, regardless of methodology. Simon (1996) shows us that design cannot exist without iteration, as an activity without iteration cannot rightly be called design. The only questions become then: What is iterated? Why is it iterated? How long is it allowed to iterate? For whom and by whom it is iterated? Our primary contribution has been to lay foundations for establishing a better defined construct of iteration. This construct (or conceptual artifact) can be leveraged in future to aid in the design of future methodologies and tools supporting these methodologies. By distinguishing different forms of iterating artifacts in the system development process we offer a baseline for future treatments of these artifacts across methodological approaches and empirical investigations.

References

- Alavi, M., (1984) "An Assessment of the Prototyping Approach to Information Systems Development," *Communications of the ACM* 27(6) 1984
- Anton, AI (1996) "Goal-Based Requirements Analysis," *Proceedings of ICRE '96*
- Auer, Ken; Meade, Erik; and Reeves, Gareth; (2003) "The Rules of the Game," in Maurer, Frank, & Wells, Don, eds, *Extreme Programming and Agile Methods – XP/Agile Universe 2002, Lecture Notes in Computer Science 2753*, August 2003
- Barrett, Barry, Chan, Demmel, Donato, et al. (1994) *Templates for the Solution of Linear Systems - Building Blocks for Iterative Methods* SIAM 1994, <http://www.netlib.org/templates/Templates.html>
- Basili & Turner, (1975) "Iterative Enhancement: A Practical Technique for Software Development". *IEEE Transactions on Software Engineering*. v.~SE-1, n.~4, December 1975, pp.390--396.
- Baskerville, RL; Stage, J; (1996) "Controlling prototype development through risk analysis," *MIS Quarterly*, 1996
- Baskerville, R; Pries-Heje, J; (2004) "Short cycle time systems development," *Information Systems Journal*, 2004
- Beath, C.M., and Orlikowski, W.J. (1994) "The Contradictory Structure of Systems Development Methodologies: Deconstructing the IS-User Relationship in Information Engineering," *Information Systems Research*, 5,4, 1994, 350-377.
- Beck, K. (2002). *Extreme Programming Explained: Embrace Change*. The Agile Software Development Series, ed Cockburn & Highsmith, 2002. Addison-Wesley.

- Bergman, M., King, J.L., and Lyytinen, K. (2002) "Large Scale Requirements Analysis as Heterogeneous Engineering", *Scandinavian Journal of Information Systems*, vol. 14, no. 1, pp. 37-55.
- Beynon-Davies, P., D. Tudhope, Mackay. (1999) "Information Systems Prototyping in Practice," *Journal of Information Technology*. 14(1), 107-120.
- Beynon-Davies, P., C. Carne, et al. (1999). "Rapid Application Development: an empirical review," *European Journal of Information Systems*. 8(2), 211-223.
- Beynon-Davies, P., Mackay, H., Tudhope, D. (2000) "'It's lots of bits of paper and ticks and post-it notes and things...': A Case Study of a Rapid Application Development Project," *Journal of Information Systems*. 10(3). 195-216.
- Blair, D.C., (2005) *Wittgenstein, Language, and Information: Back to the Rough Ground*, Springer, December 2005.
- Boehm, B., (1981) *Software Engineering Economics* Prentice-Hall.
- Boehm, B; Gray, TE; Seewaldt, T; (1984) "Prototyping vs. Specification: A MultiProject Experiment," *IEEE Transactions on Software Engineering*.
- Boehm, B, (1988) "The Spiral Model of Software Development and Enhancement," *Computer* 21(5), May 1988
- Boland, RJ (1978) "The Process and Product of System Design," *Management Science*, 24(9) May 1978
- Boland, RJ; Day, WF; (1989) "The experience of system design: a hermeneutic of organizational action," *Scandinavian Journal of Management*, 1989
- Boland, RJ; Tenkasi, RV; Te'eni, D; (1994) "Designing information technology to support distributed cognition," *Organization Science*, 1994
- Boland, RJ; Tenkasi, RV; (1995) "Perspective making and perspective taking in communities of knowing," *Organization Science*, 1995
- Brinkkemper, S, (1996) "Method engineering: Engineering of information systems development methods and tools," *Information and Software Technology*, 1996
- Brooks, Frederick P. Jr. (1987) "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, April 1987.
- Brooks, Frederick, *The Mythical Man Month: Essays on Software Engineering*, Addison-Wesley Publishing Company, Anniversary Edition 1995
- Carlile, P. R. (2002). A pragmatic view of knowledge and boundaries: Boundary objects in new product development, *Organization Science* (Vol. 13, pp. 442-455).
- Cazanescu, V.E.; & Stefanescu, Gh.; (1994) "Feedback, Iteration and Repetition," *Mathematical aspects of natural and formal languages*, Pages: 43 - 61
- Checkland, P. (1981). *Systems Thinking, Systems Practice*, John Wiley & Sons 1981
- Checkland, P. (1994). "Systems theory and management thinking," *American Behavioral Scientist*, 38(1).

- Checkland, P. & Scholes, J. (1999). *Soft Systems Methodology in Action*, John Wiley, New York, NY 1999
- Churchman (1971), *The Design of Inquiring Systems*, 1971, Basic Books, Inc.
- Cockburn, A. (2002). *Agile Software Development*, The Agile Software Development Series, ed Cockburn & Highsmith, 2002. Addison-Wesley.
- Coopriider, JG; Henderson, JC; (1991) "Technology-Process Fit: Perspectives on Achieving Prototyping Effectiveness," *J. Management Information Systems*, 7, 3
- Cross, Nigel, (1989) *Engineering Design Methods*, John Wiley & Sons.
- Curtis, B; Krasner, H; & Iscoe, N; (1988) "A field study of the software design process for large systems," *Communications of the ACM*, 1988
- Davis, (1974) *Management Information Systems: Conceptual Foundations, Structure, and Development*, McGraw Hill, 1974
- Davis, GB; (1982) "Strategies for information requirements determination," *IBM Systems Journal*, 1982
- Deephouse, C.; Mukhopadhyay, T.; Goldenson, D.; & Kellner, M. (1996) "Software Processes and Project Performance." *Journal of Management Information Systems* 12, 3 (Winter 1995-96), 187-205.
- Dijkstra, E. W. (1972) "Notes on structured programming." In *Structured Programming*. Academic Press.
- Floyd, C.; (1984) "A Systematic Look at Prototyping," in Budde et al *Approaches to Prototyping*, Springer-Verlag 1984
- Floyd, C.; Mel, WM; Reisin, FM; Schmidt, G; Wolf, G; (1989) "Out of Scandanavia: Alternative Approaches to Software Design and System Development," *Human-Computer Interaction*, Volume 4, p. 253-350.
- Floyd, C.; (1993) "STEPS - a methodical approach to PD," *Communications of the ACM*, 1993
- Gargarian, Gregory, (1996) "The Art of Design," in Eds: Kafai, Yasmin; & Resnick, Michael, *Constructionism in Practice: designing, thinking, and learning in a digital world*, Lawrence Erlbaum Associates 1996
- Gordon, V.S.; Biemen, J.M.; (1993) "Reported effects of rapid prototyping on industrial software quality," *Software Quality Journal*, 1993
- Gordon, V.S.; Biemen, J.M.; (1995) "Rapid Prototyping: Lessons Learned," *IEEE Software*, 1995
- Hardgrave, B; Wilson, R; Eastman, K; (1999) "Toward a contingency model for selecting an information system prototyping strategy," *Journal of Management Information Systems*, 1999
- Hevner, A.R.; March, S.T.; Park, J.; & Ram, S. (2004) "Design Science in Information Systems Research," *MISQ* 28(1) March 2004
- Highsmith (2002) *Agile Software Development Ecosystems*, The Agile Software Development Series, ed Cockburn & Highsmith, 2002. Addison-Wesley.

- Hirschheim, R.; Klein, H.; & Lyytinen, K. (1995) *Information Systems Development and Data Modeling: Conceptual and Philosophical Foundations*, Cambridge University Press 1995
- Hutchins, E, (1995) *Cognition in the Wild*, MIT Press.
- Humphrey (1989) *Managing the Software Process*, Addison-Wesley.
- Iivari, Juhani; Koskela, Erkki; (1987) "The PICO Model for Information Systems Design," *MIS Quarterly*, September 1987
- Janson & Smith (1985) "Prototyping for Systems Development: A Critical Appraisal," *MIS Quarterly* December 1985
- Keen & Scott Morton (1978) *Decision Support Systems: An Organizational Perspective*, Addison Welsley Publishing Co 1978
- Kruchten, P, (2000) *The Rational Unified Process An Introduction*, Second Edition, Addison-Wesley-Longman.
- Larman, C; (2004) *Agile and Iterative Development, A Manager's Guide*, Pearson Education
- Larman, C; Basili, V; (2003) "Iterative and incremental development: a brief history," *Computer*, 2003
- Lindvall, M; Basili, V; Boehm, B; et al (2003) "Empirical Findings in Agile Methods," in Maurer, Frank, & Wells, Don, eds, *Extreme Programming and Agile Methods – XP/Agile Universe 2002, Lecture Notes in Computer Science 2753*, August 2003
- Lyytinen, K. (1986) *Information Systems Development as Social Action: Framework and Critical Implications*, Dissertation, Jyvaskyla, Jyvaskyla yliopisto, 1986 (Jyvaskyla Studies in Computer Science, Economics and Statistics ISBN 0357-9921; 8).
- Lyytinen, K. (1987) "A Taxonomic Perspective of Information Systems Development: Theoretical Constructs and Recommendations," in Boland and Hirschheim, *Critical Issues in Information Systems Research*, 1987 John Wiley & Sons Ltd.
- Lyytinen, K; Robey, D; (1999) "Learning failure in information systems development," *Information Systems Journal*, Volume 9 Issue 2 Page 85 - April 1999
- Lyytinen, K; Mathiassen, L; Ropponen, J; (1998) "Attention Shaping and Software Risk-A Categorical Analysis of Four Classical Risk Management Approaches," *Information Systems Research*, 9(3) March 1998
- Mahmood, MA, (1987) "System development methods—a comparative investigation," *MIS Quarterly*, 11(3) 1987
- Malhotra, A; Thomas, JC; Carroll, JM; Miller, L; (1980) "Cognitive Processes in Design," *Journal of Man-Machine Studies*, 12, 119-140.
- March, S. & Smith, D. (1995) "Design and Natural Science Research on Information Technology," *Decision Support Systems*, 15 1995
- Matthiassen, L; Seewaldt, T; Stage, J; (1995) "Prototyping and Specifying: Principles and Practices of a Mixed Approach," *Scandinavian Journal of Information Systems*, 1995, 7(1): 55-72.
- McConnell, S, (1998) *Software Project Survival Guide*, Microsoft Press.

- McCracken, D.D., "A maverick approach to systems analysis and design", pp. 446-451, in Cotterman, W.W, et al (ed.), *Systems Analysis and Design. A foundation for the 1980's*, New York, North-Holland, 1981.
- Mumford (2003) *Redesigning Human Systems*, Idea Group Inc. 2003
- Nauman, J.D.; Jenkins, M. (1982) "Prototyping: The New Paradigm for Systems Development," *MIS Quarterly*, 6, 3, 29-44.
- Necco, CR; Gordon, CL; Tsai, NW; (1987) "Systems analysis and design: current practices," *MIS Quarterly*, 1987
- Oppelland, H.J. & Kolf, F., (1980) "Participative Development of Information Systems: Methodological Aspects and Empirical Experience," in Lucas et al *The Information Systems Environment, [IFIP 1979]* North Holland Publishing Company 1980
- Orlikowski, W; Iacono, S; (2001) "Desperately Seeking the .IT. in IT Research: A Call to Theorizing the IT Artifact," *Information Systems Research* (12:2), 2001, pp. 121-124.
- Parnas & Clements "A Rational Design Process: How and why to fake it," *IEEE Transactions on Software Engineering*, SE-12, 2 (Feb. 1986)
- Peirce, C.S.; (1992) *Reasoning and the Logic of Things*, Harvard University Press.
- Potts, C; Takahashi, K; & Anton, AI; (1994) "Inquiry-Based Requirements Analysis," *IEEE Software*, 1994
- Rossi, M; Ramesh, B; Lyytinen, K; Tolvanen, JP; (2005) "Managing Evolutionary Method Engineering by Method Rationale," *Journal of the AIS*, 5(9), September, 2004.
- Royce (1970) "Managing the Development of Large-Scale Software Systems," *Proceedings of IEEE WESCON*, pp 1-9, August 1970
- Simon, Herbert (1996) *The Sciences of the Artificial*, third edition, 1996 The MIT Press
- Subramanian GH, Zarnich GE (1996) "An examination of some software development effort and productivity determinants in ICASE tool projects," *Journal of Management Information Systems* 1996; 12: 143-160
- Tolvanen, JP (1998) *Incremental Method Engineering with Modeling Tools: Theoretical Principles and Empirical Evidence* (Ph.D. thesis), Jyväskylä Studies in Computer Science, Economics and Statistics, Jyväskylä: University of Jyväskylä
- Tolvanen, JP; & Lyytinen, K.; (1993) "Flexible method adaptation in CASE. The Metamodeling Approach," *Scandinavian Journal of Information Systems*, 1993
- Turk, D.; France, R.; & Rumpe, B.; (2005) "Assumptions Underlying Agile Software Development Processes," *Journal of Database Management*, 16(4), October-December 2005.
- Turner, Jon (1987) "Understanding the Elements of System Design" in Boland and Hirschheim, *Critical Issues in Information Systems Research*, 1987 John Wiley & Sons Ltd.
- Wikipedia, 2005 <http://en.wikipedia.org/wiki/Iteration>
- Williams, L.; Kessler, R.; Cunningham, W.; & Jeffries, R.; (2000) "Strengthening the case for pair programming," *IEEE Software*, 2000

Wirth, Niklaus (1971) "Program development by stepwise refinement," *Communications of the ACM*, v.14 n.4, p.221-227, April 1971

Wynekoop, JL; & Russo, NL (1997) "Studying system development methodologies: an examination of research methods," *Information Systems Journal*, 7, 47-65.