4-10-2008

Iteration in Systems Analysis and Design: Cognitive Processes and Representational Artifacts

Nicholas Berente
Case Western Reserve University, nxb41@case.edu

Kalle Lyytinen
Case Western Reserve University, kalle@case.edu

芽|Sprouts Working Papers on Information Systems ISSN 1535-6078

Iteration in Systems Analysis and Design: Cognitive Processes and Representational Artifacts

Nicholas Berente
Case Western Reserve University, USA

Kalle Lyytinen
Case Western Reserve University, USA

Abstract
The idea of iteration is inherent to systems analysis and design methodologies and practices. In this essay we explore the notion of iteration, and distinguish two dimensions of iteration: iterations inherent in cognitive processes and iterations over representational artifacts. Cognitive iterations can be concerned with the design; the design process; or stages within the design process. Representational artifacts can take the form of documentation or the software code itself. We identify and discuss the promise of â iterative developmentâ and compare this promise to empirical findings on the effects of iterative methods. The findings are generally consistent with expected outcomes. We conclude with an observation that the difference between â iterative developmentâ and more traditional methodologies lies not in the presence of iteration, but in the locus of visibility and control, and the associated timing and granularity.

Keywords: Iterative Development, Design Iteration, Evolutionary Prototyping, Evolutionary Enhancement, Software Prototyping, Agile Methodologies, Rapid Application Development

Permanent URL: http://sprouts.aisnet.org/5-23

Reference: Berente, N., Lyytinen, K. (2005). "Iteration in Systems Analysis and Design: Cognitive Processes and Representational Artifacts," Case Western Reserve University, USA . *Sprouts: Working Papers on Information Systems*, 5(23). http://sprouts.aisnet.org/5-23

Introduction

System analysis and design has always been an iterative process. Recent agile methods recognize iterative development as a fundamental principle of design (Cockburn 2002), but the idea of iteration is not new. From the earliest development methodologies and practices, the idea of iteration has been inherent in discussions among researchers and practitioners, although not always explicitly. Therefore, for those researching and managing systems development it is important to understand what can iterate, why iterations occur, and the implications of iteration on design outcomes.

In this essay we will explore the notion of iteration, and how it applies to systems analysis and design. We distinguish two dimensions of iteration: iterations inherent in cognitive processes and iterations over representational artifacts. We identify and discuss genres of iterated representational artifacts that are prescribed by mainstream system development methodologies. We then review the sparse empirical body of research on the effects of iteration and observe that empirical research on iteration focuses almost entirely on one form of iterating artifact: the evolutionary prototype. The findings associated with evolutionary prototypes are generally consistent with expected outcomes.

We conclude with a provocation. If iteration forms a fundamental property of all systems analysis and design methodologies, then what, exactly, is the difference between iterative and traditional, "non-iterative" development practices? It certainly is not the existence or non-existence of iteration, as in this regard there is less of a difference than one might suppose. Rather, differences lie in the conditions that define iterative behavior and content - in notions of visibility and control.

Iteration Defined

We need to carefully address the iteration concept because it underpins most development practices. Yet, the term "iteration" is not always used to address the same aspect of design. For example, iteration commonly refers to the cyclical generation of functional software code and its testing (Beck 2002), but it also describes repetition of a phase of development due to rework (Davis 1974), or successive sub-phases within a main phase (Iivari and Koskela 1987). Less common applications of the word can be found. For example, Checkland and Scholes (1999) indicate that the cyclical comparison of conceptual models to the real world as a form of iteration. Iterative activities go often 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 toward increased mutual understanding (Boland 1978).

The term "iteration" is used in a variety of disciplines and in different contexts of 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). The "iterative method" describes a problem-solving methodology in many fields, including computer science and mathematics. These 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.

Although all of these uses bear a Wittgensteinian family resemblance (Blair 2005), the fundamental aspect of iteration relates to a question of whether iteration is goal-driven or mere repetition. Dowson illustrates the difference vividly when speaking of a choice between Sisyphus and Heraclitus while modeling software processes:

> The Greek mythic hero Sisyphus was condemned to repeatedly roll a rock up, a hill, never to quite achieve his objective; the Greek philosopher Heraclitus maintained that "You can never step in the same river twice". That is, do we see iteration as repetition of the same (or similar) activities, or does iteration take us to somewhere quite new? (Dowson 1987, p.37)

Here we contend that simply equating iteration with mere repetition does not capture what is the most salient aspect in its common usage. Use of the term "iteration" implies an objective and the progression towards that objective, whereas repetition has no such implication. Software development activity necessarily involves work towards closure, which is delivering a product. Even if repeated activities bear a strong resemblance to each other, some learning within an individual or progress in the development project can be reasonably assumed to take place as steps or operations in the development process are carried out many times. Therefore, 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 or repeated activities, and 2) a progression toward convergence or closure, that is, development and implementation of an information system.

Systems analysis and design occurs within the minds of individual developers, among developers, and between developers and other groups. Iterations take place both cognitively, within the mind of a developer, and socially, across individuals. An object, or artifact, can also be iterated as it evolves in discrete steps toward some notion of completion as recognized by the rules of the genre which define its completeness. We contend that there are two fundamental forms of iteration in the systems analysis and design process: (1) iterating cognitive processes which take place in the minds of the developers, often through interactions with representations; and (2) iterations over representational artifacts that are used by designers and other people during the design. These include instantiations of the software code itself. To understand cognitive iteration, it is important to explore how minds of designers work. This task is not unproblematic due to the intangibility and non-observability of cognitive activity. Representational artifacts, however, are tangible objects representing something about the design, and can be identified, discussed, and tracked in a straightforward manner. Therefore, in the following we will analyze theoretical views of cognitive iteration in design and then examine how these cognitive processes are reflected in changes in representational artifacts.

Cognitive Iteration in Design

From one angle all systems analysis and design depends on what goes on in the heads of designers. It is a commonly held belief that this cognitive activity occurs in an iterative fashion, where some form of mental looping operations take place to guide the design. A substantiation of this simple observation, beyond a mere statement, demands that we open ourselves to the vast cognitive science literature, as well as the wide array of treatments of cognitive phenomena in

psychology, design, computer science, and information systems research – complete with accompanying often rival epistemologies and ontological assumptions. Rather than attempting to justify any distinct ontological stance in this essay, we will next broadly review what we characterize as the "rationalistic" view of cognition. We also address an alternative tradition as represented in some critiques of artificial intelligence and ethnographic analyses of design work. We will then offer examples from these two traditions in their treatment of cognitive iteration in software design. The goal of this section is thus to illustrate commons thread of iterative activity that permeate all perspectives on software design, and then to highlight the importance of representational artifacts in iteration from an individual designer's standpoint.

Views of Designer's Cognition

The mainstream view of designer's cognition falls squarely within what computer scientists refer to as the "symbol system hypothesis" of cognitive activity (Newell & Simon 1976). This hypothesis claims that cognitive activity is essentially comprised of "patterns and processes, the latter being capable of producing, modifying, and destroying the former. The most important property of these patterns is that they designate objects, processes, or other patterns, and that, when they designate processes, they can be interpreted." (Newell & Simon 1976, p. 125)

Two concepts that are associated with designer's cognition in this view: abductive reasoning (Peirce 1992) and mental models (Johnson-Laird 1980). The reasoning process of a designer is described as abductive (retroductive) inference (a chain of operations), in contrast to it being inductive or deductive inference, which are well known modes of inference in scientific studies (Peirce 1992). Abduction generates a design hypothesis (a mapping between a problem space and a solutions space), often a good guess by the designer in the face of an uncertain situation, to a given problem and then works with this hypothesis until it is no longer deemed practical – at which time another hypothesis is generated. Simon (1996) describes this form of cognitive activity as nested "generate-test cycles" and argues that they are fundamental to design. He conceives design as problem solving, where designers engage in a "heuristic search" of design alternatives, and then choose (decide) a satisficing design to go forward. When the alternative is shown not to be the proper course, a new cycle of heuristic search begins. During the design process, designers engage in iterative learning about both the problem space and the solution space (Simon 1996, Cross 1989).

Another critical aspect in viewing designer cognition is by enlisting the aid of mental models that represent both the problem spaces and the solutions spaces and which show how specific mental operations manipulate them and their connections during the design. "Mental model" here becomes a generic term which is used to describe (meta) concepts that organize representations of problems and solutions. This includes terms such as frames, schemas, causal mental models, situational models, etc. (Brewer 1987). This notion was popularized by Johnson-Laird (1980) to refer to cognitive representations that are constructed as required to assist human cognition. Mental models are not images of problems or solutions, but can lead to such images. Specific, localized mental models are expected to both draw from and contribute to a global schema of "generic knowledge structures" within the individual that can later be leveraged to form new "episodic" mental models during design (Brewer 1987).

These central ideas underpinning design cognition have been characterized to form the essence of the "rationalistic tradition" of cognition. Yet alternatives exist to this approach, and they criticize some of its fundamental assumptions, including critiques of artificial intelligence

(Winograd & Flores 1986, Suchman 1987) and theories of cognition in social psychology (Weick 1979, Bruner 1990; Hutchins 1995; etc.). Attempts at unifying the common thread of a wide range of rationalistic theories are seen problematic, as they address issues such as "meaning" in a simplistic and objective manner, though the meaning of meaning is more nuanced and situated (Suchman 1994). In the rationalistic tradition "the machinery of the mind has taken precedence in theory building, insofar as mental representations and logical operations are taken as the wellspring for cognition" (Suchman 1994 p.188). A group of alternatives to this tradition that are particularly salient to research on design cognition can be called the "situated action" perspective, which exposes "the socially constructed nature of knowledge, meaning, and designs… no objective representations of reality are possible; indeed, intelligence is not based exclusively on manipulating representations" (Clancey, Smoliar, & Stefik 1994, p.170).

The situated action view does not focus exclusively within an individual's mind. Rather, it looks at the interactions between social and contextual phenomena within the ongoing activity of a designer (Suchman 1987, Winograd & Flores 1986). An example of an iterative cognitive activity in this tradition would be the idea of a hermeneutic circle of interpretation where the individual leverages his "pre-understanding" to understand something within its context and forms a new "pre-understanding" (Winograd & Flores 1986). Each hermeneutic circle can be considered a single cognitive iteration.

Although mainstream management and design research generally aligns with the rationalistic tradition, there is an increasing amount of research that emphasizes interpersonal negotiation and dialog as key to the design process (Bucciarelli 1994; Clark & Fujimoto 1991). In this view the idea of cognitive iteration is not the neat, temporally-ordered and fully-formed hypothesis or mental model of a design within an individual. Rather, it is a messy, partially-formed object of dialogue that is created in negotiation and laden with meaning and interests and evolves through hermeneutic cycles of dialogue. In the situated action view, the notion of a discrete cognitive iteration thus loses its vividness.

To summarize cognitive design iterations, we must first be aware of the assumptions of each of the two traditions, as each tradition offers a different view of iteration. The rationalistic tradition will assume fully-formed and well organized mental models or hypotheses during design, whereas the situated action perspective will assume iterative, partial understandings of the design as realized in action and dialogue. Either way, both these cognitive iterations address three aspects:

(1) the design object;
(2) the design process as a gradual movement of the "mental" object; and
(3) the steps or stages within the design process.

Mainstream system design research typically assumes cognitive iteration in the form as prescribed in the rationalistic tradition and seeks to map the mental operations into a set of operations in the artifacts. Since the mid 1990s, however, there has been a growing amount of research that draws upon the situated action perspective (Cockburn 2002; Bergman et al 2002; Hazzan 2002; Boland & Tenkasi 1995; etc.). Next we review ways in which the information systems literature has addressed cognitive iteration and its three aspects of design activity in prescriptive and descriptive accounts of systems design activity.

Cognitive Iterations within Design

Surprisingly, cognitive iterations have gone largely unaddressed in systems design literature. Although the design literature draws upon the systems approach (Churchman 1968), the mainstream systems design research rarely accounts for the iterating, dialectical cognitive process inherent in design (Churchman 1971). In contrast, systems development literature has focused mainly on the cognitive iterations associated with steps or stages in the design process; less so with the design process itself, or cognitive iterations about the design. Table 1 offers examples of each form of cognitive iteration as recognized in the literature. This is not an exhaustive list. Rather, it is intended as an illustration.

Cognitive Iterations	Description	Method	Source
Stages in the Design Process			
Phase	Iteration between definition, design & implementation processes	Life cycle	Davis 1975
Round	Iterations of plans, prototypes, risk analyses together	Spiral Model	Boehm 1988
Iteration	Inception, elaboration, construction and transition cycle	Rational	Kruchten 2000
Time-box	Time period within which planned iteration of running, tested code	eXtreme Programming	Beck 2002
Design Process			
Model	Method engineering, method as model	n/a	Brinkkemper 1996
Maturity	Formal assessment of the designer and design process maturity	Capability maturity	Humphrey 1989
Iteration	Reflection on methodology, cycle between artifact & representation	Soft Systems	Checkland 1981
Design			
Learning	Iteratively learning about the problem and the design together	n/a	Alavi 1984
Object system	Conceptualization of the anticipated socio-technical work-system	n/a	Lyytinen 1987
Hermeneutic	Cycle of comparison between artifact, context and understanding	Soft Systems	Checkland 1981
Dialog	Cycles of cooperation and conflict between developers and users	ETHICS	Mumford 2003

Table 1. Cognitive Iterations

Cognitive Iteration of Stages in the Design Process. In the system design tradition, cognitive activity is implicitly assumed to coincide with formal stages of the design – the moments at which a given aspect of the software crystallizes and is "frozen." The most common conceptual iteration observed in systems design is that of the step, stage, or phase of the design. Stages are iterated when they are repeated during the design. Such iterations have traditionally been considered inevitable, necessary evils in system development (Royce 1970, Davis 1974), 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 (Beck 2002; Auer, Meade, & Reeves, 2003; Beynon-Davies et al 1999). Stages and phases of the process are prescribed by a methodology but are not directly related to the status of designs or the code itself. Other terms for such repeating steps in the methodology are "rounds" (Boehm 1988) and "iterations" (Kruchten 2000; Larman 2004; Beck 2002). The rationalistic tradition within system design tends to equate (at least implicitly) cognitive iterations of the design with the formal procedural iterations.

Cognitive Iteration about the Design. Cognitive iterations associated with the design process are not necessarily limited to those *within* the process, but can also relate to the designer's conception *about* the process. For example, using the idea of a method as a formal model, iteration is recognized in the method engineering literature (Brinkkemper 1996; Rossi et al 2005; Tolvanen & Lyytinen 1993). Formal methodologies cannot specify all that tasks to be completed during design, as problems change, so designers must reflect on their actions in order to be successful (Checkland 1981). Through this reflection, designers learn and continuously expand their practices (Rossi et al 2005). As practices evolve and designers learn by iterating over their cognitive models of the method, they capture the rationale for method-related iterations which reduces design errors and facilitates the evolution of methods and associated mental models (Rossi et al 2005).

Cognitive Iteration of the Design. The situated action tradition frequently ventures beyond stages and models of the process and draws attention to other forms of cognitive iteration inherent in design. For example, systems design has been likened to a hermeneutic circle (Boland & Day 1989), where a designer iteratively compares an artifact with its context to understand its meaning. Checkland (1981) recommends specific representations, such as rich pictures and holons, to guide a system developer in iterative cognitive, or hermeneutic, cycles between the representations, personal judgments, and understandings of reality that progressively refine his underlying conception. 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).

Researchers have also likened forms of system development to dialectic cycles (Churchman 1971). Such cycles are evident in participatory approaches to design that encourage dialogs between system developers and the user community (Floyd 1989, 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 iterations of cooperation and conflict that are intended to improve user-related outcomes such as user satisfaction or system use.

Other approaches consistent with the situated action perspective offer radically alternative cognitive iterations. For example, the PIOCO methodology (Iivari & Koskela 1987) goes beyond sequential stages in a process and requires iterative problem solving within levels of abstraction. Rather than freezing portions of the design into predetermined linear phases, development following a non-linear iterative (recursive) activity is explicitly allowed throughout the design. The design can be frozen, however, at specific levels of abstraction before tackling subsequent levels of abstraction.

In all of these examples, the cognitive iteration does not stand on its own, but is intimately involved with the designer's interaction with the representational artifact, social context, or managerial environment. Cognitive iterations are not discrete, fully-formed views of the information system and its design, but rather, incomplete perspectives about the design and the design process can therefore be seen to instantiate representations of the system on three levels: technical (computer system, such as code), symbolic (data & inferences, such as data models), and the organizational level (tasks supported, such as anticipated socio-technical work scenarios) (Lyytinen 1987; Iivari & Koskela 1987).

Little empirical research has been conducted about developer's cognition (Curtis, Krasner, & Iscoe 1988; Jeffries et al 1981; Boland & Day 1989). Most observe cognitive

challenges related to design that demand iteration, but do not test iterative versus non-iterative cognitive practices. Exceptions do exist, however. For example, an early study compared traditional unidirectional flow of problem information from user to developer during requirements definition with a more iterative dialogue, where both the user and developer prepared their suggestions and offered feedback. The iterative method generated greater mutual understanding, better system design quality, and enhanced system implementability (Boland 1978). In another study, researchers found that novice developers benefited from sequential processes in database design, whereas expert developers leveraged on iterative behaviors to improve design outcomes (Prietula & March 1991).

Iterations over Representational Artifacts

Whatever the content of a designer's cognitive activity, it integrates representations as tools by which designers extend their cognition (Simon 1996; Bucciarelli 1994; Hutchins 1995). A representation is a "way in which a human thought process can be amplified" (Churchman 1968, p.61). Designers represent their designs, the design process, and other associated information using symbolic and physical artifacts. In the making of these artifacts, in manipulating and navigating them, and in reflecting on artifacts, design ideas crystallize and change, and new ideas emerge. Representational artifacts can take the form of documentation such as models or requirements, or the software code itself. Table 2 describes a number of iterating representational artifacts described in the literature, and is intended to offer an illustration of the wide range of representational iterations that are prescribed by different methodologies – not an exhaustive list.

Artifact	Description	Method	Source
Iterating Document Examples			
Requirements	Specify project purpose and customer needs	Waterfall	Royce 1970
Project Control List	Tasks that the system is expected to achieve	Iterative enhancement	Basili & Turner 1975
Data Model	Model to support inferences from anticipated use	n/a	Hirshheim et al 1995
Agreeements	Written contracts between users and developers	ETHICS	Mumford 2003
Risk Analysis	Simplify documentation to crucial requirments/specs	Spiral Model	Boehm 1988
Process Assessment	Annual analysis of process & team to gage maturity, etc.	Capability maturity	Humphrey 1989
Iteration Plan	Plan for four nested Rational process phases	Rational	Kruchten 2000
Inter-Team Specs	Specifications of interface between object-oriented teams	Crystal Orange	Cockburn 1998
Iterating Software Code Examples			
Pilot	First version of the code that is "thrown away"	Waterfall	Royce 1970
Version	Output of a development process, to be followed by another	Life-cycle	Davis 1974
Refinement	Step-by-step elaboration on initial blunt "complete" code	Stepwise refinement	Wirth 1971
Enhancement	subset of the final code is developed to evolve into final	Iterative enhancement	Basili & Turner 1975
Prototype	Exploratory, experimental, and evolutionary types	n/a	Floyd 1984
Refactored Code	Iteration of entire code made to work on a daily basis	eXtreme Programming	Beck 2000

Table 2. Iterating Representational Artifacts

To appreciate the nature and role of representational artifacts in systems design, it is important to view an information system as a dynamic entity (Orlikowski & Iacono 2001).

Systems evolve, get revised, and behave differently in unique contexts. We assert that there is no single entity that *is* the system in a development process rather the system is a shared ambiguous concept about a slice of reality that can only be more or less accurately approximated through representations (Lyytinen 1987). Early in the design process, the information system may be little more than an idea invented by a handful of people whose only tangible artifact is a vague requirements memo. 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 rationales of the system's utility. Yet throughout the process, individuals often discuss the information system as if it were a single, discrete entity, although all individuals only have partial views (Turner 1987) of this boundary object.

In the following sections we will discuss how representational artifacts iterate and are iterated that are regarded pivotal in the information systems development and software engineering literature: the documents associated with the design and the software code. Then we will address the idea of "iterative development" as reflected in specific ways in which artifacts are iterated as well as recognized empirical impacts of "iterative development" on design outcomes.

Iterating Documents

Early representations of the system during the design center on requirements definitions associated system specifications. Over the course of the design these representations change regularly, and often evolve into other representations, such as "as built" software documentation. Because of this need for connecting with downstream documentation like code, no software development methodology can overlook iteration across documents entirely, although some, such as XP (Beck 2002), want to remove documentation from the critical path of system development.

Traditional system development life-cycle and "heavy-weight" methodologies are popularly thought to focus on documentation and its iterations downstream, and they thus encourage "freezing" such documentation upstream in order to move on to the next step in the design. This popular conceptualization is not, in fact, the case, as every major methodology allows for iteration of such documents at least to some extent (Boehm 1981, 1988; Kruchten 2000; Humphrey 1989).

The waterfall model (Royce 1970) is the most well known life-cycle methodology and is often characterized as top-down, unidirectional, and non-iterative. Contrary to this claim, even in its earliest manifestation Royce suggested that unwanted changes and following iterations are inevitable, and he recommended a number of practices to address such 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 as the developer learns from the design, 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.

The extant literature addresses various forms of iteration related to upstream system 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 1974; Mumford

2003; etc.). Although most of the literature addresses changing documents 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 representations. 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 is expected to be efficient, since many goals can be eliminated, refined, or consolidated before entering the design step (Anton 1996).

Iterating Software Code

The code evolves through multiple instantiations in many development approaches including "throw-away" prototypes (Baskerville & Stage 1996), prototypes that evolve into a final system, or maintenance of different versions of a system. The common usage of "iterative development" refers normally to software design 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, Beynon-Davies et al 1999, Alavi 1984). Such iterative development practices emerged soon after waterfall was made the "orthodox" model. 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" where designers start small and simple, by coding a "skeletal sub-problem of the project." Then developers incrementally add functionality by iteratively extending and modifying the code, using a project 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.

This idea of iterative enhancement forms the foundation of evolutionary prototyping and many recent agile methods. Agile methodologies are based on the assumption that design communication is necessarily imperfect (Cockburn 2002), and that software design is a social activity among developers and users. The most popular agile methodology - extreme programming, or XP - promotes a variety of iterative development practices such as pair programming (cognitive iteration during each design step through dialogue), test-first development (generating test information that guides subsequent iteration), and refactoring (iterating the artifact during each cycle) (Beck 2002). The structure of XP is almost identical to the early evolutionary design, where limited functionality is first developed, and then incrementally expanded. However, XP can take advantage of a number of tool innovations that were not available for early software developers. Toolsets are now available that enable unit testing, efficient refactoring, and immediate feedback, and object-oriented environments allow for modular assembly of significant portions of system. Also, process innovations such as testing-first, time-boxing, collocation, story cards, pair programming, shared single code base, and daily deployment mitigate the communication problems found in earlier evolutionary processes.

The Promise of "Iterative Development"

The justification of evolutionary prototyping, or more commonly "iterative development," centers on trial and error learning about both the problem and solution during design. Users and developers do not know what they need until they see something. Thus generating prototypes (mock-ups) assists communication better than traditional abstract upstream documentation and thereby supports 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.). In the following we will review some of the anticipated outcomes associated with iterative development in the information systems development and software engineering literature.

Anticipated benefits of evolutionary, or "iterative development" (or prototyping as methodology) are many. By growing the design in such a matter, software can be developed more quickly (Brooks 1987). Beyond speed, evolutionary development enables 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 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).

A problem often associated with strict evolutionary development, however, is the lack of maintaining "iterative" plans for each prototype. Starting with a poor initial prototype could turn users away; prototyping can contribute to a short-term, myopic focus, and "developing a suboptimal system" can necessitate rework in later phases (Boehm 1981). Exhaustive design documentation will still be required even if prototyping forms the primary process (Humphrey 1989). Also, the output of evolutionary development often resembles unmanageable "spaghetti code" that is difficult to maintain and integrate. These are similar to the "code and fix" problems that waterfall was originally intended to correct (Boehm 1988). Many 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).

Not surprisingly, many caution that evolutionary development practices are not suited to every situation as the idea of continuous iteration makes 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 upstream 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 demands complementary assets to succeed (Boehm 1981, Beck 2002).

Empirical Impacts of "Iterative Development"

Empirical research on "iterative development" is as scarce as the prescriptive research is plentiful (Gordon & Biemen 1995; Lindvall et al 2002; Wynekoop & Russo 1997). The empirical research that does exist has primarily focused on the effects of prototyping on project success (Alavi 1984, Boehm et al 1984, etc.), while neglecting the impact and role of other iterating representations in project outcomes. Nevertheless, in the following we assess the state of empirical research on iterations over representational artifacts.

Promise of Iterative Development	Source	Supported?	Conclusion	Source
1. Supports mutual learning between users and developers		**Yes**		
learning about the problem & solution; addresses requirements uncertainty; more realistic validation of requirements; demonstrates technical feasibility	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, Davis 1982		learn about requirements; support communication & problem solving	Naumann & Jenkins 1982; Alavi 1984; Boehm, Gray, & Seewaldt 1984; Necco, Gordon, Tsai 1987; Mahmood 1987; Deephouse et al 1996
2. Improves user-related outcomes		**Yes**		
increase participation; more successful system use	Hardgrave & Wilson 1999; Larman & Basili 2003		greater user involvement; better user satisfaction; ease-of-use; greater system use	Naumann & Jenkins 1982; Alavi 1984; Gordon & Bieman 1993, 1995; Necco et al 1987; Boehm et al 1984; Mahmood 1987
3. Improves design process		**Yes**		
software developed more quickly; designers more productive; projects cost less; reduce risk	Brooks 1987; Basili & Turner 1975, Beck 2002, Larman 2004; Larman & Basili 2003, Cockburn 2002; Matthiassen et al 1995, Boehm 1988, Lyytinen et al 1996		shorten lead times for projects and/or less effort; designer satisfaction	Naumann & Jenkins 1982; Boehm, Gray, & Seewaldt 1984; Necco, Gordon, Tsai 1987; Gordon & Bieman 1995; Subramanian & Zarnich 1996; Baskerville & Pries-Heje 2004; Mahmood 1987
4. Improves design outcomes		**Mixed**		
better code with more successful outcomes; results in code that is easily modified / maintained; increased innovativeness	Larman & Basili 2003; Basili & Turner 1975; Hardgrave & Wilson 1999	supported	positively related to higher system performance; more maintainable code	Alavi 1984, Larman 2004; Boehm, Gray, & Seewaldt 1984; Gordon & Bieman 1993
		not supported	less functional systems, with potentially less coherent designs; "negotiable" quality requirements	Boehm et al 1984; Baskerville & Pries-Heje 2004
5. Requires complementary practices		**Yes**		
requires complementary assets / practices; or more formal structure	Boehm 1981; Beck 2002; Cockburn 2002, Boehm 2002, Lindvall et al 2003		prototyping must be combined with other factors, such as tools, standards, expertise, etc.	Naumann & Jenkins 1982; Alavi 1984; Baskerville & Pries-Heje 2004; Gordon & Bieman 1995; Beynon-Davies et al 2000; Lichter et al 1993

Table 3. Testing the Promise of "Iterative Development"

Representational artifacts include the documents, data models, and other physical representations of the software, including artifacts such as user-interface mock-ups and "throw-

away" prototypes. These representations are addressed quite extensively in the prescriptive literature, but the iteration of these representations, and the effect those iterations have on design outcomes is notably absent. The primary exception to this is the research on "throw-away" prototypes. 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; Beynon-Davies et al 1999), the empirical literature does not typically highlight a distinction between these types of prototypes and their outcomes, and when there is a distinction, there is no significant difference in the outcomes (Gordon & Biemen 1995, 1993).

As indicated earlier, the notion most commonly associated with "iterative development" is evolutionary prototyping, and this will be the focus of our review. Table 3 summarizes the expected impacts of evolutionary prototyping, and broadly compares them to empirical findings. It is important to note that a good number of researchers have found empirical evidence to be inconclusive on many accounts, and this data is not reported in our review. Also, many expectations highlight the drawbacks of the evolutionary method, but these criticisms tend to focus on design outcomes and they are addressed in the fourth item of our review.

The fundamental reason Basili & Turner advocated iterative enhancement 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 (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 (Naumann & Jenkins 1982; Alavi 1984; Gordon & Bieman 1995). Improved user participation is 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).

While they advocate iterative enhancement, Basili & Turner (1975) indicate that software created through modular 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 code (Boehm, Gray, & Seewaldt 1984; Gordon & Bieman 1993).

In most empirical studies, iteration is treated as an independent variable that affects outcomes. Moderators are often introduced, but not in a systematic manner. For example, prototyping must be combined with other 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). Also, if users are not involved, prototype-based project outcomes can suffer (Lichter et al 1994). Prototyping can also bee 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. Most of the empirical literature on the impacts of 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, current empirical research will likely assess varying combinations of iterative and specification-based processes (eg Matthiassen et al 1995), or compare variations in agile practices (what types of iterations and by whom counts). When pursuing either of these research avenues, it would make sense to adopt a more granular and refined view of iteration and also define the dependent outcome variables more carefully.

Discussion

"Iterative development" has been both advocated and contested as a fundamental systems analysis and design principle. Yet it has remained a fundamental building block to most modern design methodologies. In this essay, we have consistently kept the term in quotes because literally all systems development is iterative. Both cognitive and consequently representational iterations are fundamental to every design practice. This begs the question, what is the difference, then, between "iterative" practices of today, and the "non-iterative" traditional practices? The answer is *not* the presence of iteration, as both types exhibit iteration in abundance.

One explanation of the difference can be the way in which the two types of methodology approach iteration. Both modern and traditional practices focus on iteration as *reactive* fixes or improvements based on new information or uncovered problems. Modern methods, however, tend to anticipate the need for and inevitability of new information, and *proactively* seek it. The difference thus is not the presence of iteration, but, rather, the timing and visibility. With earlier visibility of iteration needs, designers are inviting user input and thus relinquishing a certain amount of control over iterations. Because this visibility is staged earlier, its granularity with regard to foundational details and assumptions of the system development is also greater. Fundamentally "iterative development" is not necessarily more iterative. But it is likely to be more open, and the control over iterations is shared and at a much more detailed level.

Consider the code as an iterating artifact, for example. All application software iterates over its life even if its design methodology is the life-cycle model (SDLC; Davis 1974). Each version of a software system can be considered an iteration. As bugs are fixed or enhancements added to code – even if consistent with the linear life-cycle method - any new instantiation of code can be considered an iteration. When all or some portions of the code are compiled, the result is an iteration of compiled code. Anytime a designer replaces or adds to any part of working code, he has iterated over that code.

In the traditional life-cycle method, however, the user is not highly involved beyond listing requirements. Management is not aware of each iteration, they only see the code that is presented at key milestones. The bricolage of everyday work-arounds, failures, changes, etc., is often neatly hidden from everyone except the designer himself – as are micro-level assumptions

and decisions that can have disproportionately large impacts on path dependent future designs. As systems development became more "iterative," the veil hiding this practice has been progressively lifted. Prototyping invited other developers, users, and managers into discussions at a more granular level of detail sooner during development. When participating in this activity, those parties adopted more detailed control over the process. Risk analysis (Boehm 1988) that focuses on system risk and mitigation (rather than over-detailed requirements that draw no real distinction of risks), exposes the key requirements of design to scrutiny outside of developers. Pair programming (Beck 2002) opens on-going moment-by-moment deliberations of an individual developer to observation and demands a dialog with a fellow developer. This observation indicates that the key contingency for distinguishing iterations between development practices is not whether one engages in evolutionary prototyping or not. Observations such as the following indicate that a focus on iteration as such may be misplaced:

- user involvement is a more important determinant of project outcomes than presence of iterative development (Lichter et al 1994);
- the success of any development, iterative or not, depends more on developer experience than anything else (Boehm 2002);
- for iterative development to succeed, the complementary practices such as co-location, pair programming, etc., are essential (Beck 2002).

Therefore, it is not the presence of iteration that primarily determines the outcomes of systems analysis and design activity. Rather, it is the activities that specific iterations enable or constrain. The black box of iteration should be opened to understand structures and affordances of certain prescribed iterations and complementary processes, and their effect on design process and its outcomes. Rather than asking whether an organization should adopt iterative development, it is more salient for organizations to ask what level of granularity, visibility, and control over iteration are appropriate at different times and for different purposes of the design.

Conclusion

The contribution of this essay is to illustrate the multi-dimensionality of iteration. Iteration is often characterized in the literature as an unproblematic concept - either a development process is iterative or it is not. We have shown that this emasculated concept is too simplistic, as all development practices contain significant levels of iteration. Also, we identified two fundamental dimensions of iteration: cognitive and representational. Cognitive processes of developers and others involved in the design are necessarily iterative, but this can mean different things depending on whether the rationalistic tradition or the situated action perspective of human cognition is adopted. Also, cognitive iterations often involve iterative engagement with representations acting as both extensions to cognition and mediation between individuals.

In systems analysis and design literature, cognitive iterations are addressed (usually implicitly) through the iterative treatment of representational artifacts. The perspectives and meanings that designers ascribe to artifacts are rarely addressed. Instead the technical artifact itself is the central concern. Typically, the actual cognitive practices within development are not addressed, but rather the formal steps and stages of the methodology as reflected in representational outcomes are treated at length. Genres of representations are typically

advocated and designed to enable communication and human interaction at specific steps and junction points and such artifacts are expected to change iteratively. This communication is not always seen as unproblematic, but seldom is the nature of these representations addressed in how they support the cognitive activities of design groups or their iterations.

We identify two primary forms of representation: the documents associated with the analysis and design; and the software code itself. Although there are many representational artifacts for both types which are prescribed by advocates of particular methodologies, the empirical literature is limited to the examination of iterations over the software code as evolutionary prototyping and its impacts. Recent "iterative development" identifies entirely with the centrality of iterations being associated with the code.

We have indicated that the essential difference between what is widely considered as "iterative development" and traditional software development is the audience for the iteration. Iterative development creates iterations specifically for visibility to some portion of the managerial and user community earlier in the process, and at a more granular level. With such activity, developers are also relinquishing a degree of control. Because of the dual nature of software code – acting as a representational artifact of the system as well as a fundamental physical structure within the task system – analysis of iterations solely on the basis of the presence of evolutionary prototyping appears problematic. The iterative processes by which key concerns arise throughout the development process are essential to understanding success. These processes can be facilitated by evolutionary prototyping, but also by the creative use of other representational artifacts, generative language and dialogue, or other collaborative mechanisms. Also, all prototypes are generally characterized as equal. Opening the black box of "iterations" over code and other representational artifacts is essential to understanding better outcomes associated with different design practices.

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 Journa*l, 2004

Beck, K. (2002). *Extreme Programming Explained: Embrace Change*. 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*, 1984.

Boehm, B, (1988) "The Spiral Model of Software Development and Enhancement," *Computer* 21(5), May 1988

Boehm, B, (2002) "Get Ready for Agile Methods, with Care," *IEEE Computer*, January 2002

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; (1995) "Perspective Making and Perspective Taking in Communities of Knowing," *Organization Science* 5(4) July-August 1995

Brewer, W. (1987) "Schemas Versus Mental Models in Human Memory," In Morris, I.P., *Modeling Cognition*, John Wiley and Sons, 1987.

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

Bruner, J., (1990) *Acts of Meaning*, Harvard University Press, 1990.

Bucciarelli, L.L. (1994) *Designing Engineers*, The MIT Press, 1994.

Checkland, P. (1981). *Systems Thinking, Systems Practice*, John Wiley & Sons 1981

Checkland, P. & Scholes, J. (1999) *Soft Systems Methodology in Action.* John Wiley.

Churchman (1968), *The Systems Approach*, Dell Publishing Co., 1968

Churchman (1971), *The Design of Inquiring Systems*, 1971, Basic Books, Inc.

Clancey, W.J.; Smoliar, S.W.; & Stefik, M.J. (1994) *Contemplating Minds: A Forum for Artificial Intelligence*, The MIT Press, 1994.

Clark & Fujimoto (1991) *Product Development Performance : strategy, organization, and management in the world auto industry*, Harvard Business School Press 1991

Cockburn, A. (1998) *Surviving Object-Oriented Projects*, Addison-Wesley.

Cockburn, A. (2002) *Agile Software Development*, Addison-Wesley.

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.

Dowson, M. (1987) "Iteration in the software process; review of the 3rd International Software Process Workshop," *ICSE 1987*, Proceedings of the 9th international conference on Software Engineering, Monterey California 1986.

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.

Gordon, VS; Biemen, JM; (1993) "Reported effects of rapid prototyping on industrial software quality," *Software Quality Journal*, 1993

Gordon, VS; Biemen, JM; (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

Hazzan, O., (2002) "The Reflective Practitioner Perspective in Software Engineering Education," *The Journal of Systems and Software* 63, 2002.

Hirschheim, Klein, Lyytinen (1995) *Information Systems Development and Data Modeling: Conceptual and Philosophical Foundations*, Cambridge University Press 1995

Humphrey (1989) *Managing the Software Process*, Addison-Wesley.

Hutchins, E, (1995) *Cognition in the Wild*, MIT Press.

Iivari, J. & Koskela, E. (1987) "The PIOCO Model for Information Systems Design," *MIS Quarterly*, September 1987.

Janson & Smith (1985) "Prototyping for Systems Development: A Critical Appraisal," *MIS Quarterly* December 1985

Jeffries, R.; Turner, A.A.; Polson, P.G.; & Atwood, M.E.; (1981) *The Processes Involved in Designing Software*, Lawrence Erlbaum Associates, 1981

Johnson-Laird, P.N. (1980) "Mental models in Cognitive Science," *Cognitive Science*, Volume 4, p.71-115, 1980.

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

Lichter, H.; Schneider-Hufschmidt, M.; & Sullighoven, H.; (1993) "Prototyping in Industrial Software Projects – Bridging the Gap Between Theory and Practice," *IEEE Transactions on Software Engineering*, November 1994

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. (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; 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

Matthiassen, L; Seewaldt, T; Stage, J; (1995) "Prototyping and Specifying: Principles and Practices of a Mixed Approach," *Scandanavian Journal of Information Systems*, 1995, 7(1): 55-72.

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 Newell & Simon 1976

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

Prietula, M.J., & March, S.T., (1991) "Form and Substance in Physical Database Design: An Empirical Study," *Information Systems Research* 2:4 1991.

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

Suchman, Lucy A., (1987) *Plans and Situated Actions: The Problem of Human-Machine Communication*, Cambridge University Press

Suchman, Lucy A., (1994) Review of Winograd & Flores: *Understanding Computers and Cognition*, in Clancey, W.J.; Smoliar, S.W.; & Stefik, M.J. (1994) *Contemplating Minds: A Forum for Artificial Intelligence*, The MIT Press, 1994.

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.

Weick, Karl, (1979) *The Social Psychology of Organizing*, McGraw-Hill, 1979

Wikipedia, 2005 http://en.wikipedia.org/wiki/Iteration

Winograd, T., & Flores, F. (1986) *Understanding Computers and Cognition: A New Foundation for Design*, Ablex Publishing Corporation, 1986.

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.

芽|Sprouts

Working Papers on Information Systems | ISSN 1535-6078

Office:
Sprouts
University of Amsterdam
Roetersstraat 11, Room E 2.74
1018 WB Amsterdam, Netherlands
Email: admin@sprouts.aisnet.org

芽|Sprouts