PART II

METHODOLOGY FOUNDATION
AND EVOLUTION OF SYSTEMS
ANALYSIS AND DESIGN
CHAPTER 4

ITERATION IN SYSTEMS ANALYSIS AND DESIGN

Cognitive Processes and Representational Artifacts

NICHOLAS BERENTE AND KALLE LYYTINEN

Abstract: The concept of iteration is fundamental to systems analysis and design practice and methods. In this chapter we explore the notion of iteration, and distinguish two domains of iteration: iterations associated with cognitive processes that take place during design and iterations over representational artifacts about the design. Cognitive iterations can be concerned with the design: the design process; or stages within the design process. Representational artifacts can take the form of descriptive documentation of the system or the executable code itself. We discuss the claimed impacts of “iterative development” and compare these impacts to empirical findings on the effects of iterative methods. The findings are generally consistent with expected outcomes. We conclude with the observation that the differences between “iterative” or “agile” development and traditional methodologies lies not in the presence or absence of iteration, but in the locus of visibility and control, and the associated timing and granularity of what is being iterated.

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

INTRODUCTION

Recent agile methods recognize “iterative development” as a fundamental design principle (Cockburn, 2002). Yet, the idea of iteration is not new—system analysis and design has always been iterative! From the earliest development methodologies, the concept of iteration has been inherent in discussions about system design, though not always explicitly. Therefore, for those researching and developing systems it is important to understand what is iterated during design, why it is iterated, and what the impacts of this iteration are.

In this chapter we explore the concept of iteration and how it has been applied to systems analysis and design. We distinguish between two domains of iteration: iterations inherent in cognitive processes during design; and iterations over representational artifacts about designs. Cognitive iterations are concerned with the design object itself, the design process, or stages in the design process. Iterations of representational artifacts take place across descriptive documentation associated with the system and its components or the executable code. Table 4.1 depicts the framework. Our goal is to review how past research on systems analysis and design has treated iteration within these different domains, what we know about these iterations, and how these iterations have been shown to affect design outcomes. We do this by surveying the main streams of the systems analy-
sis and design literature and soliciting the main findings through a literature review. We aim to offer readers an understanding of how iteration has been defined and treated in both prescriptive and empirical studies of design in order to determine what we know and do not know about the impacts of different types of iterations under different design contingencies.

The remainder of the chapter is organized as follows. First, we provide a description of our sampling of the theoretical literature associated with this framework, and then review the sparse empirical body of research on the effects of iteration. We observe that empirical research on iteration focuses almost entirely on one type of iterating artifact: the evolutionary prototype. The findings associated with evolutionary prototyping are generally consistent with expected outcomes.

We conclude the chapter with a new perspective on iteration in systems analysis and design. As iteration forms a fundamental property of all systems analysis and design, then we must ask what, exactly, is the difference between iterative or agile, and traditional, “noniterative” development? If it is not the presence or absence of iteration we need to have a more refined vocabulary to analyze differences among iterations and the criteria that can be used to spell out those differences. We accordingly suggest that these differences lie in the criteria that define the content and outcomes of iterative behavior as defined by notions of (a) iteration visibility—who can observe it? (b) control—who can control it? (c) granularity—what is being iterated and at what level of detail? and (d) timing—when do the iterations occur?

This insight challenges researchers to be mindful of the perspectives that designers and other stakeholders assign to various forms of documentation and to the executable code itself. The timing and level of detail where evolving artifacts are made visible affect the perspectives of the various stakeholders associated with the project, and these perspectives, in turn, affect project outcomes. We stress that iteration must be understood in terms of multidimensional, dynamic behaviors that are central to design, not as an unproblematic “thing” that either exists or does not.

ITERATION DEFINED

We need to carefully explore the concept of iteration because it underpins all systems development practices. The term “iteration” is common in a variety of disciplines. It can be 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 with repetition in general (Wikipedia, 2007). The “iterative method” describes a problem-solving methodology in many fields, including computer science and mathematics. These 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 successive iterations.

It is no wonder the term “iteration” is not used consistently to refer to the same aspect of system design in the extant literature. For example, for software designers, iteration commonly refers to the cyclical generation and testing of increasingly functional software code (Beck, 2002), but it can also describe the repetition of a phase of development due to rework (Davis 1974), or successive subphases within a main phase (Iivari and Koskela, 1987). Less common applications also abound. For example, Checkland and Scholes (1999) indicate that the cyclical comparison of conceptual models to the real world represents a form of iteration. Iterative activities also often go by different names such as “prototyping” when designers iteratively elicit user input (Alavi, 1984), “rounds” when designers iteratively search for a design solution to reduce functional or implementation risk (Boehm, 1988), or even a “dance” of interactions among designers and users toward increased mutual understanding (Boland, 1978).

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 equating iteration with mere repetition does not capture the most salient aspect in its common usage for systems analysis and design, computer science, or mathematics. For us, use of the term “iteration” implies a progression toward an objective, whereas repetition has no such implication. Software development activity accordingly involves work toward closure, which is the delivery of a product. Even if repeated activities bear a strong resemblance to each other, some learning in the development project can be reasonably assumed to take place within each step while the same development activity is carried out many times. Yet, no formal, single definition of the term “iteration” in systems analysis will be presented here. Rather, echoing the spirit of its many uses we suggest that the key facets of iteration are: (1) looping operations through repeated activities, and (2) a progression toward a convergence or closure.

Systems design occurs within the minds of individual developers, among developers, and between developers and other groups. Consequently, iterations take place cognitively, within the minds of developers, and socially or communicatively across individuals. Cognitive iterations imply repeated mental activity as a designer converges on a solution that is deemed adequate—the perfecting of the design idea. Likewise, any object, or artifact, can be iterated during design while it evolves in discrete steps toward some notion of completion as recognized by the rules of the genre that define its completeness. As noted above, we suggest that there are two fundamental forms of iteration during systems analysis and design process: (1) iterative cognitive processes in the minds of the developers; and (2) iterations over representational artifacts that are used and
shared by designers and other stakeholders during the design (see Table 4.1). These representa-
tions include instances of the executable code.

To understand cognitive iteration, it is important to explore how the minds of designers work. This task is not unproblematic due to the intangibility and nonobservability of cognitive activity. Representational artifacts, however, are tangible objects representing something about the design, and can be identified, discussed, and tracked in a relatively straightforward manner. Below, we analyze theoretical views of cognitive iteration in design, and then examine how cognitive processes are reflected in changes in representational artifacts.

COGNITIVE ITERATION IN DESIGN

In a sense, 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 advances iteratively, where some forms of mental looping take place to guide the design. A substantiation of this simple observation, beyond a mere statement, demands, however, that we open ourselves to the vast cognitive science literature, as well as to the wide array of treatments of cognitive phenomena in psychology, design, computer science, and information systems research—complete with accompanying rival epistemologies and ontological assumptions. Rather than attempting in this chapter to establish any distinct ontological stance, we broadly review what we characterize as the “rationalistic” school of cognition. We also address an alternative tradition as represented in critiques of artificial intelligence and ethnographic analyses of design. We then offer examples of these two traditions in their treatment of cognitive iteration in software design. The goal of this section is thus to illustrate the common thread of cognitive iteration that permeates all perspectives on system design, and to highlight the importance of representational artifacts in iteration from an individual designer’s standpoint.

Views of Designer Cognition

The mainstream view of designer’s cognition falls squarely within what computer scientists refer to as the “symbol system hypothesis” of cognition (Newell and 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” (ibid., p. 125).

Two concepts that are associated with designer’s cognition in this view are: abductive reasoning (Peirce, 1992) and mental models (Johnson-Laird, 1980). The reasoning process of a designer is described as abductive (or retroductive) inference, which is different from and should be contrasted with inductive and deductive inference, which are well-known modes of inference in scientific study (Peirce, 1992). Abduction generates a design hypothesis (a mapping between a problem space and a solutions space), often a “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 workable—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 of design as problem solving, where designers engage in a “heuristic search” for alternatives, and then choose a satisficing design to go forward. When the alternative is found not to be the proper course, a new cycle of heuristic search begins. During design activity, designers engage in iterative learning about both the problem space and the solution space (Cross, 1989; Simon, 1996).

Another critical aspect of a designer’s cognition involves the mental models that represent
both the problem spaces and the solutions spaces, which designers manipulate in order to connect
the solution space with the problem space. “Mental model” here becomes a generic term that is
used to describe (meta)concepts that organize representations of problems and solutions and their
connections. This includes representational metamodels such as frames, schemas, causal models,
situational models, and so on (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 expected to leverage a new
“episodic” mental model during design (Brewer, 1987).

These ideas underpinning design cognition form the essence of the “rationalistic” tradition of
cognition. Yet, alternatives exist that criticize some of the fundamental assumptions of rational
models (Bruner, 1990; Hutchins, 1995; Suchman, 1987; Weick, 1979; Winograd and Flores,
1986; and others). Any attempt to reconcile these critiques with the rationalistic tradition would
be problematic, as rationalist theories address issues such as “meaning” in a simplistic manner,
whereas many of the other traditions view the meaning of “meaning” as highly 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 family of alternatives that are par-
ticularly salient to research on design cognition can be called the “situated action” perspective,
which calls attention to “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, and Stefik, 1994, p. 170).

The situated action view does not focus exclusively on what happens 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 and Flores, 1986). An example of an
iterative cognitive activity in this tradition would be the idea of a hermeneutic circle of interpreta-
tion where the individual leverages his “pre-understanding” to understand something within its
context and forms a new “pre-understanding” (Winograd and Flores, 1986). Each hermeneutic
circle can be considered a 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 a key to understanding design (Bucciarelli, 1994; Clark and Fujimoto, 1991). In this
stream, the idea of cognitive iteration is not the neat, temporally ordered and fully formed mental
model of a design in an individual’s mind. Rather, it is a messy, partially formed object and process
of dialogue, laden with meaning and interests and evolving through hermeneutic cycles. In the situ-
ated action view, the notion of a discrete and individual cognitive iteration loses its vividness.

To summarize, we must first become aware of the assumptions of each tradition, as each
tradition offers an alternative view of iteration. The rationalistic tradition assumes fully formed
and well-organized mental models that emerge and are manipulated during design, whereas the
situated action perspective assumes partial, evolving understandings of the design as realized in
dialogue. Either way, both these cognitive iterations share three facets:

1. steps or stages within the design (e.g., generate-test cycles/hermeneutic circles);
2. the design process as a gradual movement of the “mental” object (mental model/under-
standing); and
3. the design object (the representation/the text).
The bulk of system design research treats cognitive iteration in accordance with the rationalistic tradition and normally seeks to map a designer’s mental operations into a set of corresponding operations on the artifacts. Since the mid-1990s, however, there has been a growing amount of research that draws upon the situated action perspective (Bergman, King, and Lyytinen, 2002; Boland and Tenkasi, 1995; Cockburn, 2002; Hazzan, 2002; and others). In this alternative tradition, design is an ongoing dialogue that is always open to reinterpretation. Next we review ways in which the information systems literature has addressed cognitive iteration and its three aspects of design activity, as well as prescriptive and descriptive accounts of systems design.

**Cognitive Iterations Within Design**

Surprisingly, cognitive iterations as such have gone largely unaddressed in the systems design literature. Although the design literature draws extensively upon the systems approach (Churchman, 1968), the mainstream of the systems design research rarely accounts for the iterating cognitive process inherent in design (Churchman, 1971). In contrast, systems development literature has focused mainly on the cognitive iterations in the form of operations associated with steps in the design process, and less so with the design process itself, or cognitive iterations about the design. Table 4.2 offers examples of each form of cognitive iteration as recognized in the literature. We emphasize that this is not an exhaustive list, but rather is intended as an illustration.

**Cognitive Iteration of Stages in the Design Process**

In the systems design tradition, cognitive activity is assumed to coincide with formal stages of the design—the moments at which a given aspect of the software crystallizes and becomes “frozen.” The most common conceptual iteration observed in systems design is that of the step, stage, or phase. Stages are iterated as they are repeated during the design. Such iterations have traditionally been considered inevitable, necessary evils (Davis, 1974; Royce, 1970), but are now more commonly thought to enhance system quality (Basili and Turner, 1975; Beck, 2002; Boehm, 1981; Brooks, 1995; Cockburn, 2002; Floyd, 1984; Keen and Scott Morton, 1978; Larman and Basili, 2003; McCracken and Jackson, 1982). Such stages can be formal, such as the requirements determination phase that results in “frozen” requirements (Davis, 1982), or they can be fairly indeterminate, such as “time-boxed” steps in agile methods (Auer, Meade, and Reeves, 2003; Beck, 2002; Beynon-Davies, Tudhope, and Mackay, 1999). Stages, phases, rounds, or iterations of the process are prescribed by a methodology but are not directly related to the status of the design or the code (Beck, 2002; Boehm, 1988; Kruchten, 2000; Larman, 2004). The rationalistic tradition within systems design thus tends to equate cognitive iterations with the formal procedural iterations.

**Cognitive Iteration About the Design Process**

Cognitive iterations associated with system development are not necessarily limited to those within the process, but can also relate to the designer’s conceptions about the process itself. If we follow the idea that a method is in itself a formal design model, this model can iterate during the design process much the same as conceptualizations of the design object itself. This idea is prominent in the concept of method engineering (Brinkkemper, 1996; Rossi et al., 2004; Tolvanen and Lyytinen, 1993). Method-engineering advocates claim that formal methodologies cannot specify a priori all design tasks to be completed, as problems and solutions spaces change. Therefore designers...
<table>
<thead>
<tr>
<th>Cognitive iterations</th>
<th>Description</th>
<th>Method</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stages in the design process</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Phase</td>
<td>Iteration between definition, design, and implementation processes</td>
<td>Life cycle</td>
<td>Davis (1974)</td>
</tr>
<tr>
<td>Round</td>
<td>Iterations of plans, prototypes, risk analyses together</td>
<td>Spiral model</td>
<td>Boehm (1988)</td>
</tr>
<tr>
<td>Iteration</td>
<td>Inception, elaboration, construction, and transition cycle</td>
<td>Rational</td>
<td>Kruchten (2000)</td>
</tr>
<tr>
<td>Time-box</td>
<td>Time period within which planned iteration of running, tested code</td>
<td>eXtreme Programming</td>
<td>Beck (2002)</td>
</tr>
<tr>
<td>Design process</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model</td>
<td>Method engineering, method as model</td>
<td>n/a</td>
<td>Brinkkemper (1996)</td>
</tr>
<tr>
<td>Maturity</td>
<td>Formal assessment of the designer and design process maturity</td>
<td>Capability maturity model</td>
<td>Humphrey (1989)</td>
</tr>
<tr>
<td>Iteration</td>
<td>Reflection on methodology, cycle between artifact and representation</td>
<td>Soft systems</td>
<td>Checkland (1981)</td>
</tr>
<tr>
<td>Design</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Learning</td>
<td>Iteratively learning about the problem and the design together</td>
<td>n/a</td>
<td>Alavi (1984)</td>
</tr>
<tr>
<td>Object system</td>
<td>Conceptualization of the anticipated sociotechnical work system</td>
<td>n/a</td>
<td>Lyytinen (1987)</td>
</tr>
<tr>
<td>Hermeneutic</td>
<td>Cycle of comparison between artifact, context, and understanding</td>
<td>Soft systems</td>
<td>Checkland (1981)</td>
</tr>
<tr>
<td>Dialog</td>
<td>Cycles of cooperation and conflict between developers and users</td>
<td>ETHICS</td>
<td>Mumford (2003)</td>
</tr>
</tbody>
</table>
must be reflective (Checkland, 1981; Hazzan, 2002). Through this reflection, designers learn and continuously revise their practices and iterate over the design objects and design processes they will engage in (Rossi et al., 2004). During design activity, designers thus learn by iterating over their cognitive models of the method (second loop learning), and they can also capture the rationale for these iterations in order to facilitate the continued evolution of methods and associated mental models (Rossi et al., 2004).

Cognitive Iterations of the Design

The situated action tradition frequently ventures beyond stages and models and draws attention to other forms of cognitive iteration. For example, systems design has been likened to a hermeneutic circle (Boland and 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 hermeneutic cycles between the representations, personal judgments, and understandings of reality that will progressively refine his underlying design 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 systems development to dialectic cycles (Churchman, 1971). Such cycles are evident in participatory approaches to design that encourage dialogues between system developers and the user community (Floyd et al, 1989; Mumford, 2003). These dialogues can result in a series of explicit agreements concerning system functionality, the anticipated environment, or appropriate methodologies (Mumford, 2003). They also 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 iterations. For example, the early PIOCO methodology (Iivari and Koskela, 1987) goes beyond sequential stages and formulates iterative problem-solving processes within multiple levels of abstraction. Rather than freezing portions of the design into predetermined linear phases, development follows a nonlinear iterative (recursive) progression that is explicitly allowed throughout the design. The design can be frozen at specific levels of abstraction before tackling subsequent, lower 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 interactions with representational artifacts, the social context, and managerial concerns. Cognitive iterations are not discrete, fully formed views of the information system and its design, but rather form incomplete perspectives about the design object and the design process. They instantiate representations of the system on three levels: technical (computer system, such as code), symbolic (data and inferences, such as data models), and the organizational level (tasks supported, such as anticipated sociotechnical work scenarios) (Iivari and Koskela, 1987; Lyytinen, 1987).

Little empirical research has been conducted on software developers’ cognition (Boland and Day, 1989; Curtis, Krasner, and Iscoe, 1988; Jeffries et al., 1981; B). Most studies observe cognitive challenges related to designs that demand iteration, but do not test or compare iterative versus noniterative cognitive practices. Exceptions do exist, however. For example, an early study compared the traditional unidirectional flow of problem information from a user to a developer during requirements definition with an iterative dialogue, where both the user and developer prepared their suggestions and offered feedback in a “dance.” The iterative method generated greater mu-
tual understanding, better design quality, and enhanced system implementability (Boland, 1978). In another study, researchers found that novice developers benefited from sequential processes in database design, whereas experts leveraged iterative behaviors to improve design outcomes (Prietula and March, 1991).

ITERATIONS OVER REPRESENTATIONAL ARTIFACTS

Whatever the content of a designer’s cognitive activity, it relies on representations that act as tools by which designers extend their cognition (Bucciarelli, 1994; Hutchins, 1995; Simon, 1996). 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 through them, and in reflecting on artifacts, design ideas crystallize and change, or new ideas emerge. Representational artifacts can take the form of documentation such as data models or system requirements documents, or the executable code itself (code, components, database schemata, etc). Table 4.3 describes a number of iterating representational artifacts identified in the literature. Again, this is intended as an illustration of the wide range of representational iterations that are prescribed in methodologies.

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 and Iacono, 2001). Systems evolve, get revised, and behave differently in different contexts. In fact, there is no single entity or thing that is the system. Rather the system is a shared, ambiguous and ambivalent conception about a slice of reality that can only be more or less accurately approximated through representations (Lyytinen, 1987). 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). 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 or note. Later in the process the system may become represented by lines of incomplete code, dozens of use cases, and a great number of varying rationales of the system’s utility.

In the following sections we analyze how representational artifacts that are deemed pivotal in the information systems development and software engineering literature iterate. There are two broad categories of these artifacts: the descriptive documents associated with the design object and the executable code. We treat them differently because ultimately only the code inscribes new behaviors in the target system—the technical and sociotechnical system. The system descriptions are needed to make the code inscribe behaviors that are intended, acceptable, and correct. We then address the idea of “iterative development” as reflected in the ways in which artifacts iterate, as well as what we know about the impacts of “iterative development” on design outcomes.

Iterating Documents

Early representations of the system center on descriptions of system requirements. 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 requirements with downstream documentation and ultimately with the executable code, no development methodology can overlook iteration across documents entirely, although some, such as XP (Beck, 2002), aspire to remove a majority of documentation from the critical design path.

Traditional system development life-cycle and “heavy-weight” methodologies are thought to
Table 4.3

Iterating Representational Artifacts

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Description</th>
<th>Method</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterating document examples</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Requirements</td>
<td>Specify project purpose and customer needs</td>
<td>Waterfall</td>
<td>Royce (1970)</td>
</tr>
<tr>
<td>Project control list</td>
<td>Tasks that the system is expected to achieve</td>
<td>Iterative enhancement</td>
<td>Basili and Turner (1975)</td>
</tr>
<tr>
<td>Data model</td>
<td>Model to support inferences from anticipated use</td>
<td>n/a</td>
<td>Hirschheim, Klein, and Lytinen (1995)</td>
</tr>
<tr>
<td>Agreements</td>
<td>Written contracts between users and developers</td>
<td>ETHICS</td>
<td>Mumford (2003)</td>
</tr>
<tr>
<td>Risk analysis</td>
<td>Simplify documentation to crucial requirements/specs</td>
<td>Spiral model</td>
<td>Boehm (1988)</td>
</tr>
<tr>
<td>Process assessments</td>
<td>Annual analysis of process and team to gauge maturity, and so on</td>
<td>CMM</td>
<td>Humphrey (1989)</td>
</tr>
<tr>
<td>Iteration plan</td>
<td>Plan for four-nested rational process phases</td>
<td>Rational</td>
<td>Kruchten (2000)</td>
</tr>
<tr>
<td>Inter-team specs</td>
<td>Specifications of interface between object-oriented teams</td>
<td>Crystal Orange</td>
<td>Cockburn (1998)</td>
</tr>
<tr>
<td>Iterating software code examples</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pilot</td>
<td>First version of the code that is “thrown away”</td>
<td>Waterfall</td>
<td>Royce (1970)</td>
</tr>
<tr>
<td>Version</td>
<td>Output of a development process, to be followed by another</td>
<td>Life cycle</td>
<td>Davis (1974)</td>
</tr>
<tr>
<td>Refinement</td>
<td>Step-by-step elaboration on initial blunt “complete” code</td>
<td>Stepwise refinement</td>
<td>Wirth (1971)</td>
</tr>
<tr>
<td>Enhancement</td>
<td>Portion of the code evolves into “complete” code</td>
<td>Iterative enhancement</td>
<td>Basili and Turner (1975)</td>
</tr>
<tr>
<td>Prototype</td>
<td>Exploratory, experimental, and evolutionary types</td>
<td>n/a</td>
<td>Floyd (1984)</td>
</tr>
<tr>
<td>Refactored code</td>
<td>Iteration of entire code made to work on a daily basis</td>
<td>eXtreme Programming</td>
<td>Beck (2002)</td>
</tr>
</tbody>
</table>
focus on documentation and its iterations. They encourage “freezing” documentation upstream in order to move in a disciplined manner to the next steps in the design. This conceptualization is not, however, always the case, as most major methodologies allow for iteration of upstream documents at least to some extent (Boehm, 1981, 1988; Humphrey, 1989; Kruchten, 2000).

The waterfall model (Royce, 1970) is the best-known life-cycle methodology and is often characterized as top-down, unidirectional, and noniterative. Contrary to this claim, even in its earliest manifestation Royce suggested that unwanted changes and associated 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 “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, and therefore the requirements should evolve through a series of at least five classes of 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 iterations related to upstream system representations—such as requirements determinations (Davis, 1982), data models (Hirschheim, Klein, and Lyytinen, 1995), and the wide array of technical and social documentations associated with formal methodologies (Boehm, 1981, 1988; Davis, 1974; Humphrey, 1989; Kruchten, 2000; Mumford, 2003; and others). Although most of the literature addresses changing documents throughout the design, the value of these changes is not elaborated beyond guiding and tracking. Even the more nuanced views of documentation that treat its creation as problematic and argue its content to be flawed (Parnas and Clements, 1986), have made no distinction between the value and cost of iterations across representations. There are some exceptions to this, however. For example, the “inquiry cycle model” (Potts, Takahashi, and 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 original 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 and Stage, 1996), prototypes that evolve into a final system, or maintenance of different versions of a system. The common usage of “iterative development” normally refers to software design that proceeds through “self-contained mini-projects” where each produces partially complete software (Larman, 2004). This has traditionally been referred to as evolutionary prototyping (Alavi, 1984; Beynon-Davies et al., 1999; Floyd, 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 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 and 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 recent agile methods. Agile methods 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. A popular agile methodology—extreme programming (XP)—promotes a variety of iterative 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 software 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 software tools that were not available to early software developers. Powerful toolsets are now available that enable unit testing, efficient refactoring, and immediate feedback, while object-oriented environments allow for modular assembly of significant portions of the 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. Users and developers do not know what they need until they see something—similar to Weick’s (1979) illustration of organizational sensemaking: “how can I know what I think till I see what I say?” Thus generating prototypes assists communication better than traditional abstract upstream documentation and thereby supports mutual learning (Alavi, 1984; Basili and Turner, 1975; Beck, 2002; Boehm, 1981; Brooks, 1995; Cockburn, 2002; Floyd, 1984; Keen and Scott Morton, 1978; Larman and Basili, 2003; McCracken and Jackson, 1982; and others). We now review some of the anticipated outcomes associated with iterative development.

Anticipated benefits of evolutionary, or “iterative development” are many. By “growing” the design, 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 a greater possibility of comparing 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 and Eastman, 1999), reduce project risk (Boehm, 1988; Lyytinen, Mathiassen, and Ropponen, 1998; Mathiassen, Seewaldt, and Stage, 1995), and lead to more successful outcomes (Larman and Basili, 2003). Because developers generate code rather than plan and document, they are expected to be more productive (Basili and Turner, 1975; Beck, 2002; Larman, 2004). Therefore, projects using evolutionary prototyping can be expected to cost less (Basili and Turner, 1975; Beck, 2002; Cockburn, 2002; Larman and Basili, 2003).

A problem often associated with strict evolutionary development, however, is the lack of maintaining “iterative” process plans. 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 is not suited to every situation as the idea of continuous iteration makes 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, France, and Rumpe, 2005). Issues such as scaling, criticality, and developer talent will often require hybrid methodologies—or a combination of evolutionary prototypes with formal and control-oriented methods (Boehm, 2002; Cockburn, 2002; Lindvall et al., 2002). Also, evolutionary development demands other complementary assets like smart designers or the availability of enlightened users in order to succeed (Beck, 2002; Boehm, 1981).

Empirical Impacts of “Iterative Development”

Empirical research on “iterative development” is as scarce as the prescriptive research is plentiful (Gordon and Biemen, 1995; Lindvall et al., 2002; Wynekoop and Russo, 1997). The empirical research that does exist focuses on the effects of prototyping on project success (e.g., Alavi, 1984; Boehm, Gray, and Seewaldt, 1984; and others), while neglecting the impact and role of other iterating representations. Nevertheless, in what follows, we assess the state of empirical research on iterations over representational artifacts.

Representational artifacts include the documents, data models, and other 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 effects of those iterations on design outcomes are notably absent. The primary exception is the research on “throw-away” prototypes. Although many researchers distinguish between prototypes that occur at different stages and are used for different purposes (Beynon-Davies et al., 1999; Floyd, 1984; Janson and Smith, 1985), the empirical literature does not underscore distinctions between these types of prototypes and their outcomes, and when they see a distinction, there is no significant difference in the outcomes (Gordon and Bieman, 1993, 1995).

As indicated earlier, the notion most commonly associated with “iterative development” is evolutionary prototyping. Table 4.4 summarizes the expected impacts of evolutionary prototyping and compares them with empirical findings. It is important to note that a good number of researchers have found empirical evidence to be inconclusive, and these data are not reported in our review. Furthermore, many expectations highlight the drawbacks of the evolutionary method, but these criticisms focus on design outcomes, which are addressed below.

The fundamental reason Basili and Turner advocated iterative enhancement is that problems and solutions are not well understood, 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 to learn about the requirements together (Alavi, 1984; Boehm, Gray, and Seewaldt, 1984; Naumann and Jenkins, 1982; Necco, Gordon, and Tsai, 1987). Prototyping has been found to support communication and problem solving between users and developers (Deephouse et al., 1995 Mahmood, 1987), and has led to greater user involvement (Alavi, 1984; Gordon and Bieman, 1995; Naumann and Jenkins, 1982). Improved user participa-
<table>
<thead>
<tr>
<th>Promise of iterative development</th>
<th>Source</th>
<th>Conclusion</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Supports mutual learning between users and developers</td>
<td><em>Supported</em>? <em>Yes</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Learning about the problem and solution; addresses requirements uncertainty; more realistic validation of requirements; demonstrates technical feasibility</td>
<td>Alavi (1984); Basili and Turner (1975); Beck (2002); Boehm (1981); Brooks (1995); Cockburn (2002); Davis (1982); Floyd (1984); Keen and Scott Morton (1978); Larman and Basili (2003); McCracken and Jackson (1982)</td>
<td>Learn about requirements; support communication and problem solving</td>
<td>Alavi (1984); Boehm, Gray, and Seewaldt (1984); Deephouse et al. (1995); Mahmood (1987); Naumann and Jenkins (1982); Necco, Gordon, and Tsai (1987)</td>
</tr>
<tr>
<td>2. Improves user-related outcomes</td>
<td><em>Supported</em>?</td>
<td>Increase participation; more successful system use</td>
<td>Hardgrave and Wilson 1999; Larman and Basili 2003</td>
</tr>
<tr>
<td></td>
<td><em>Yes</em></td>
<td>Greater user involvement; better user satisfaction; ease of use; greater system use</td>
<td>Alavi (1984); Boehm, Gray, and Seewaldt (1984); Gordon and Bieman (1993, 1995); Mahmood (1987); Naumann and Jenkins (1982); Necco, Gordon, and Tsai (1987)</td>
</tr>
<tr>
<td>3. Improves design process</td>
<td><em>Supported</em>? <em>Yes</em></td>
<td>Shorten lead times for projects and/or less effort; designer satisfaction</td>
<td>Basavire and Pries-Heje (2004); Boehm, Gray, and Seewaldt (1984); Gordon and Bieman (1995); Naumann and Jenkins (1982); Necco, Gordon, and Tsai (1987); Mahmood (1987); Subramanian and Zarnich (1996)</td>
</tr>
<tr>
<td>Software developed more quickly; designers more productive; projects cost less; reduce risk</td>
<td>Basili and Turner (1975); Beck (2002); Boehm (1988); Brooks (1987); Cockburn (2002); Larman (2004); Larman and Basili (2003); Lyytinen et al., (1998); Mathiassen, Seewaldt, and Stage (1995)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Improves design outcomes</td>
<td><em>Supported</em>? <em>Mixed</em></td>
<td>Positively related to higher system performance; more maintainable code</td>
<td>Alavi (1984); Boehm, Gray, and Seewaldt (1984); Gordon and Bieman (1993); Larman (2004)</td>
</tr>
<tr>
<td>Better code with more successful outcomes; results in code that is easily modified and maintained; increased innovativeness</td>
<td>Basili and Turner (1975); Larman and Basili (2003); Hardgrave et al., (1999)</td>
<td>Positively related to higher system performance; more maintainable code Less functional systems, with potentially less coherent designs; “negotiable” quality requirements</td>
<td>Baskerville and Pries-Heje (2004); Boehm, Gray, and Seewaldt (1984)</td>
</tr>
<tr>
<td></td>
<td><em>Supported</em></td>
<td>Not supported</td>
<td>Baskerville and Pries-Heje (2004); Boehm, Gray, and Seewaldt (1984)</td>
</tr>
<tr>
<td>5. Requires complementary practices</td>
<td><em>Supported</em>? <em>Yes</em></td>
<td>Prototyping must be combined with other factors, such as tools, standards, expertise, and so on</td>
<td>Alavi (1984); Baskerville and Pries-Heje (2004); Beynon-Davies, Mackay, and Tudhope (2000); Gordon and Bieman (1995); Lichter et al. (1994); Naumann and Jenkins (1982)</td>
</tr>
<tr>
<td>Requires complementary assets/practices; or more formal structure</td>
<td>Beck (2002); Boehm (1981, 2002); Cockburn (2002); Lindvall et al. (2003)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
inclusion is often credited with better user satisfaction (Naumann and Jenkins, 1982; Necco, Gordon, and Tsai, 1987), designer satisfaction (Mahmood, 1987), ease of use (Boehm, Gray, and Seewaldt, 1984; Gordon and Bieman, 1993), and greater use of the system (Alavi, 1984; Mahmood, 1987). Research on the effects of prototyping on system performance is generally mixed (Gordon and 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, Gray, and Seewaldt, 1984), and may call for “negotiable” quality requirements (Baskerville and Pries-Heje, 2004).

While Basili and Turner (1975) advocate iterative enhancement, they indicate that software created through evolutionary prototypes can require less “time and effort,” 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 require less effort, typically measured by fewer man-hours (Baskerville and Pries-Heje, 2004; Boehm, Gray, and Seewaldt, 1984; Gordon and Bieman, 1995; Naumann and Jenkins, 1982; Necco, Gordon, and Tsai, 1987; Subramanian and Zarnich, 1996). A number of studies also support the claim that evolutionary prototyping results in more maintainable code (Boehm, Gray, and Seewaldt, 1984; Gordon and Bieman, 1993).

In most empirical studies, iteration is treated as an independent variable that affects outcomes. Additional moderators are sometimes introduced, but not in a systematic manner. For example, prototyping must be combined with other factors such as powerful development tools (Alavi, 1984; Naumann and Jenkins, 1982), a standardized architecture (Baskerville and Pries-Heje, 2004), greater developer expertise (Gordon and Bieman, 1995), a complementary culture (Beynon-Davies, Mackay, and Tudhope, 2000; Lindvall et al., 2002), and “low technology” artifacts and processes for scheduling and monitoring (Beynon-Davies, Mackay, and Tudhope, 2000). Also, if users are not involved, prototype-based outcomes can suffer (Lichter, Schneider-Hufschmidt, and Zullighoven, 1994). Prototyping can also be seen as a dependent variable. For example, researchers find that prototyping may pose challenges for management and planning (Alavi, 1984; Boehm, Gray, and Seewaldt, 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 past studies have typically compared prototype-based processes with specification or plan-based processes, current empirical research will likely suggest combinations of iterative and specification-based processes (e.g., Mathiassen, Seewaldt, and Stage, 1995) or compare variations in agile practices. When pursuing either of these research avenues, it would make sense to adopt a more granular and refined view of iteration and define the dependent variables carefully.

DISCUSSION

“Iterative development” has been both advocated and contested as a systems analysis and design principle. Yet it has remained a fundamental building block of system analysis and design methodologies. In this chapter, we have consistently enclosed the term in quotation marks because literally all systems development is iterative. Both cognitive and, consequently, representational iterations are essential to every design practice. This begs the question: What is the difference, then, between “iterative” practices of today, and the traditional “noniterative” practices? The answer is not the presence or absence of iteration, as both types exhibit iteration.

One explanation of the difference can be the way in which the two types approach iteration.
Both modern and traditional practices focus on iteration as reactive fixes or improvements based on new information, or uncovered problems. Many modern methods, however, also anticipate the need for and inevitability of new information, and proactively seek it. Thus, the difference is not the presence of iteration, but, rather, the timing and visibility of it. With earlier visibility of iteration needs, designers invite user input and thus relinquish a certain amount of control over iteration decisions. 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 systems development life-cycle model (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. Any time 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 subsequent iteration, but only sees the code that is presented at key milestones. The bricolage of everyday workarounds, failures, and changes is 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 has become 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 control over the process. Risk analysis (Boehm, 1988; Lyytinen, Mathiassen, and Ropponen, 1998) that focuses on continued risk mitigation—rather than overly 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 ongoing 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 the level of iteration 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, Schneider-Hufschmidt, and Zullighoven, 1994);
• the success of any development, iterative or not, depends more on developer experience than anything else (Boehm, 2002); and
• for “iterative development” to succeed, the complementary practices such as co-location, pair programming, and so on are essential (Beck, 2002).

Therefore, it is not the presence of iteration that primarily determines the outcomes of systems analysis and design activity. Rather, these outcomes are determined by the activities that the types and forms of iterations can enable and constrain. The black box of iteration should be opened to understand structures and affordances of 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 applicable for organizations to ask what level of granularity, visibility, and control over changes and decisions about design objects and processes are appropriate at different times, and for different purposes of the design.
Implications for Practice

We have indicated that the essential difference between what is widely considered to be “iterative development” and traditional software development is the audience for design activity. “Iterative development” allows visibility to other developers or 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 may be a distraction from the issues that drive better results. The iterative processes by which key concerns arise throughout the development process are essential for 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. Rather than attempting to implement new methodologies blindly, software developers would be better served in first determining who needs design input, at what level of granularity, and at what stage of the design process.

Implications for Research

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 tackled. Instead the technical artifact itself is the central concern. Typically, the actual cognitive practices within development are not dealt with, 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, and the nature, content, and scope of these representations is seldom fully explicated in how they support the cognitive activities of design groups or their iterations (with some exceptions, of course; e.g., Checkland, 1981; Hirschheim, Klein, and Lyytinen, 1995).

This review of the literature related to iteration points to two broad areas of future research in systems analysis and design. First, cognitive iterations of software developers and other stakeholders need to be better understood. Perspectives of individuals associated with the design process do impact the design process, and understanding the evolution of these perspectives is imperative to improving outcomes. Second, future research should involve opening the black box of iteration over code and other representational artifacts to understand outcomes associated with particular design practices. By opening this black box, we encourage researchers not to treat the term “iteration” as an undifferentiated construct, but rather to look at the degrees of visibility, granularity, timing, and control associated with evolutionary changes of the code, various forms of documentation, and the perspectives of the various stakeholders.

CONCLUSION

The contribution of this chapter is to illustrate the multidimensional nature of the concept of iteration. Iteration is often characterized in the literature as a straightforward concept: either a development process is iterative or it is not. We have shown that this characterization is too simplistic, as all development practices contain significant levels of iteration. We also identified two fundamental dimensions of iteration: cognitive and representational. Cognitive processes of
developers and others involved in a 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. In addition, cognitive iterations involve iterative engagement with representations acting as both extensions to cognition and mediation between individuals.

We identify two primary forms of representation: the descriptive documents associated with analysis and design activity; and the executable code itself. Although many representational artifacts for both types are prescribed by advocates of particular methodologies, the empirical literature is limited solely to examining iterations over the software code as evolutionary prototyping. Furthermore, recent “iterative development” identifies entirely with the centrality of the iterations associated with the code itself. In this chapter we provide a starting point for unpacking the notion of iteration to expand the discussion beyond iterating code to other artifacts and associated cognitive processes.

REFERENCES


