Int J Software Informatics, Volume 5, Issue 3 (2011), pp. 457–474
International Journal of Software and Informatics, ISSN 1673-7288

E-mail: ijsi@iscas.ac.cn
http://www.ijsi.org
Tel: +86-10-62661040

A Process Programmer Looks at the Spiral Model: A Tribute to the Deep Insights of Barry W. Boehm

Leon J. Osterweil

(University of Massachusetts, USA)

Abstract This paper elaborates on implications of Barry W. Boehm's Spiral Model of software development. The paper notes that the Spiral Model presents a compelling view of software development, evocatively represented by a visual image that appeals strongly to intuition, and notes that the view and image have motivated and justified a range of important views of how software development should be done. This paper enhances and elaborates on the intuitions by supplementing them with a definition of the Spiral Model that is enunciated in terms of a rigorously defined language. The rigorous enunciation and accompanying alternative visual depiction are then used to provide clarification and formalization of some of the clearly-indicated elaborations of the Spiral Model. Both the Waterfall Model of software development and the Scrum agile method are presented as possible elaborations of the Spiral Model. Still other elaborations are indicated. Similarities in the visualizations of these development approaches suggest some underlying similarities in the approaches themselves, suggesting the potential value of effective process visualizations. The breadth of these elaborations is also used to suggest how the Spiral Model seems to provide a strong focus on some of the quintessential aspects of what comprises effective software development.

Key words: process programming; spiral model; software process; waterfall model; scrum process definition

Osterweil LJ. A process programmer looks at the spiral model: A tribute to the deep insights of Barry W. Boehm. *Int J Software Informatics*, **Vol.5, No.3 (2011): 457–474. http://www.ijsi.org/1673-7288/5/i96.htm**

1 Introduction

Software development continues to be one of the most challenging problems that humans have attempted to solve. It seems well-agreed that software is a real (albeit intangible) product that must conform to certain rules and structural constraints. Yet, the invisible, intangible, non-Newtonian nature of software renders it a frustratingly difficult type of product to build. While unique in some ways, software is, however, at least somewhat analogous as a product to other more conventional products. As a consequence for at least the past 50 years there has been general agreement that, as with most conventional products, software should be developed in stages, starting with

This work is supported by the U.S. National Science Foundation under Award Nos. IIS-0705772, and CCF-0820198
Corresponding author: Leon J. Osterweil, Email: ljo@cs.umass.edu
Paper received on 2011-03-08; Revised paper received on 2011-03-29; Paper accepted on 2011-03-30; Published online 2011-04-08.

requirements specification, proceeding through one or more design stages, continuing through actual implementation, and then on to evaluation and, ultimately, evolution. This view was captured diagrammatically most famously in the Waterfall Model, generally attributed to Winston Royce[10].

The most immediate problem with the classical Waterfall Model view of software development was its marginalization of the role of iteration. Experienced practitioners knew then, as now, that software development (indeed the development of any complex artifact) cannot be expected to proceed monotonically forward, but must from time to time reconsider earlier decisions and their manifestations in the context provided by previously developed artifacts, and the outcomes of previous decisions. Early attempts to modify the Waterfall Model were largely clumsy and unsatisfying. Backward pointing edges were introduced to indicate the possibility of such reconsiderations (e.g. Fig.1 in Ref.[10]), but generally with no indications of when such back-edges were to be taken, how reconsideration was to be done, and so forth. In failing to capture adequately the rationale for returning to earlier stages and artifacts, these approaches failed to grasp and illuminate the essential insight that the development of software (and most other complex systems) is inherently an iterative process.

With his brilliant enunciation of the Spiral Model[1,2], Barry Boehm made it clear that iteration is the essence of software development, and specified what it is that makes the iterations go around. In his paper, Boehm specifies that each iteration is driven by a risk analysis activity, which then leads to the development of an artifact (often a prototype) whose purpose is to evaluate and mitigate identified risks.

Boehm's original 1988 paper specifies that there be a series of three iterations, each leading to the development of a prototype that leads ultimately to the development of a requirements specification, a specification of a design, and ultimately to the generation of an implementation in code, followed by testing and analysis. In Boehm's 1988 paper a series of prototypes aimed at risk reduction is specified to come first, then followed by the successive development of requirements, design, code, and testing. Subsequent discussions of the Spiral Model then suggested that each of these artifact types was produced as the consequence of a separate iteration of the Spiral. In fact still other elaborations by Boehm and others have suggested a multitude of modifications and enhancements of the original details of the Spiral Model. Some have suggested that each stage may require multiple iterations of the spiral. Others have suggested that some spiral iterations may entail returning to earlier lifecycle stages for reconsideration. Still others have suggested that spiral iterations may indeed skip some of the traditional stages.

The real genius of the Spiral Model, however, is not these specific details of iteration. The real genius of the Spiral Model is that it elucidates the essential nature of software development as being iterative, with iterations driven by the need to understand and contain risk, and with each iteration yielding the increasingly deep and complete understandings upon which successful software development must rest. The Spiral Model makes these things clear. Although others may have had intuitions along these lines, Boehm crystallized these thoughts and underscored them with a wonderfully evocative visualization.

While the Spiral Model itself, as originally presented and subsequently elabo-

rated upon, is strongly suggestive and evocative of these various development process approaches, the informality of the Spiral Model means that it can provide only suggestions about how to perform them, and indeed about precisely what these approaches entail. And, indeed, the very form of the Spiral Model diagram, while highly evocative, does pose challenges to making clear the nature of many of these interesting and important elaborations. From the perspective of a process programmer, one who is interested in the formal specification of processes using programming language rigor, the absence of details of the different interpretations and elaborations of the Spiral Model represents a challenge and an opportunity. The purpose of this paper is to add precision and detail to the definition of the Spiral Model and to use the greater semantic power of an appropriate process definition language to develop a representative sample of some possible elaborations and interpretations. In doing so the paper will provide details of how to perform these processes, and will also provide clear demonstrations of the generality of the Spiral Model. It is hoped that in doing this, the paper may also provide a glimpse of the essential nature of software development as an iterative process of gaining knowledge, reducing risk, and maintaining intellectual control over the act of developing what is perhaps the most complex and elusive kind of product ever undertaken by humans, namely computer software.

To begin, this paper summarizes the key features of the Spiral Model. The paper then introduces a process definition language that seems to lend itself well to explorations of key ramifications of the Spiral Model. The paper then continues by using the process definition language as a vehicle for providing some precise specifications of different Spiral Model-based development processes. The paper then concludes with some summary remarks about the Spiral Model and the nature of software development.

2 The Spiral Model Summarized briefly

Figure 1 is a reproduction showing the key features of Boehm's Spiral Model, as originally published in his 1988 paper. The diagram indicates that software development proceeds through four iterations, each beginning with the development of a carefully considered prototype. Each prototype has a specific purpose, with the prototypes being aimed at providing understandings of (successively) the project's requirements, high-level design (architecture), and low-level design. A last prototype is intended to deliver essentially a blueprint for the code. These prototyping iterations are then followed by the actual development of requirements, design, and implementation, and the performance of a testing regimen.

Boehm and the larger community, however, quickly grasped the truth that the importance of the Spiral Model was not its specification of the exact number of iterations, nor the specific annotations of the purposes of the specific iterations shown in this original diagram (e.g. Refs.[2, 3, 4, 8, 11, 15]). Quickly it became clear that the Spiral Model should be taken as a representation of a number of more abstract truths. One of these truths is that software development is inevitably an iteration though a sequence of cycles. Moreover, the key to determining whether and how to proceed to a subsequent iteration is to focus on strategies for determining risk and mitigating risk. Further, the Spiral Model specifies that each iteration consists of four phases aimed successively at 1) determining objectives, alternatives and constraints,

2) evaluating alternatives and risks, 3) developing next products, and 4) planning for the next iterations. From a distance these phases of the Spiral Model are reminiscent of the four phases of the Shewhart/Deming Cycle[7] often referred to as the "Plan-Do-Check-Act" cycle, which also counsels the need to proceed carefully and iteratively in addressing complex tasks, making progress by taking carefully measured steps. But the Spiral Model adds to the Shewhart/Deming Cycle the important intuition that as the cycles proceed, the "mass" of the project grows. The Spiral Model shows this by depicting successive spiral cycles as having increasingly greater diameters, which is indicated by marking the Y-axis with the notation Cost (cumulative). Indeed the mass of a software development project certainly grows with each iteration, not only in cost but also in knowledge accumulated, risks better understood, and artifacts produced. The Spiral Model visualization makes this point graphically in a most natural and compelling way.

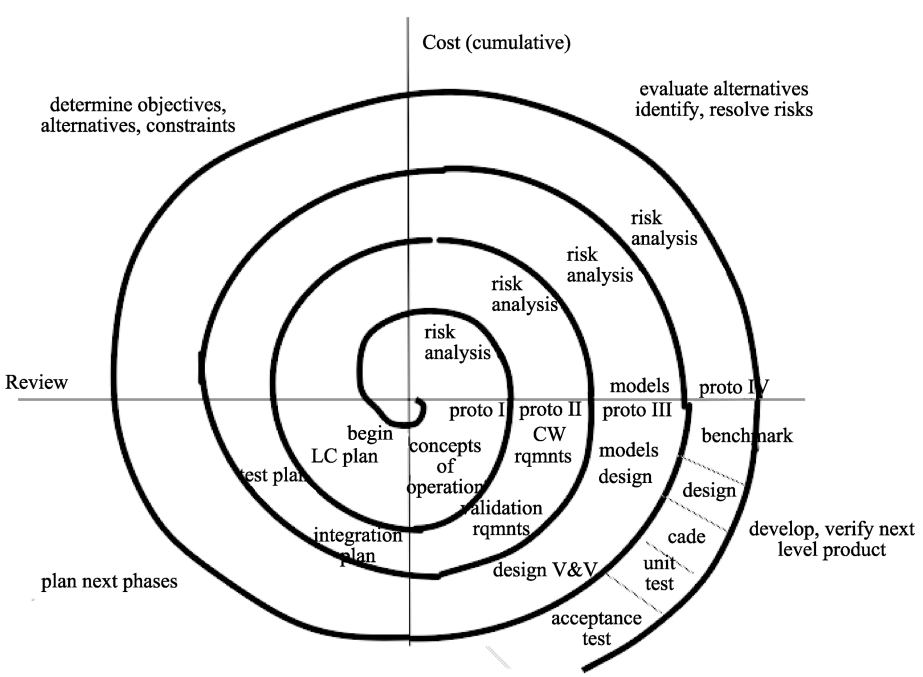

Figure 1. A rendering of the original 1988 diagram depicting the Spiral Model

The community understanding of the key messages of the Spiral Model is perhaps better captured by the rendering shown in Fig.2, in which the details of the iterations are omitted and indeed the size and number of the iterations is suggested to be indeterminate. We refer to this version as the Abstract Spiral Model and note that this is the version that has been elaborated upon variously by different thinkers, authors, and innovators. It has been used as a justification for the fundamental soundness of a number of different proposed approaches to software development. The manner in which these various authors have cited the Spiral Model as a key justification for their work is a tribute to the esteem in which Barry Boehm and his ideas are held by the community. But it also reinforces the point that the essential ideas behind the Spiral

Model are remarkably robust and fundamental.

The purpose of this paper is to indicate some of the elaborations of Figure 2 that have been suggested, noting their remarkable cogency and breadth, and also noting the ways in which they are clear elaborations of Fig.2. Using these very diverse elaborations, the paper will then attempt to distill some fundamental understandings that can be derived from how readily the diagram in Fig.2 lends itself to all of these plausible instantiations.

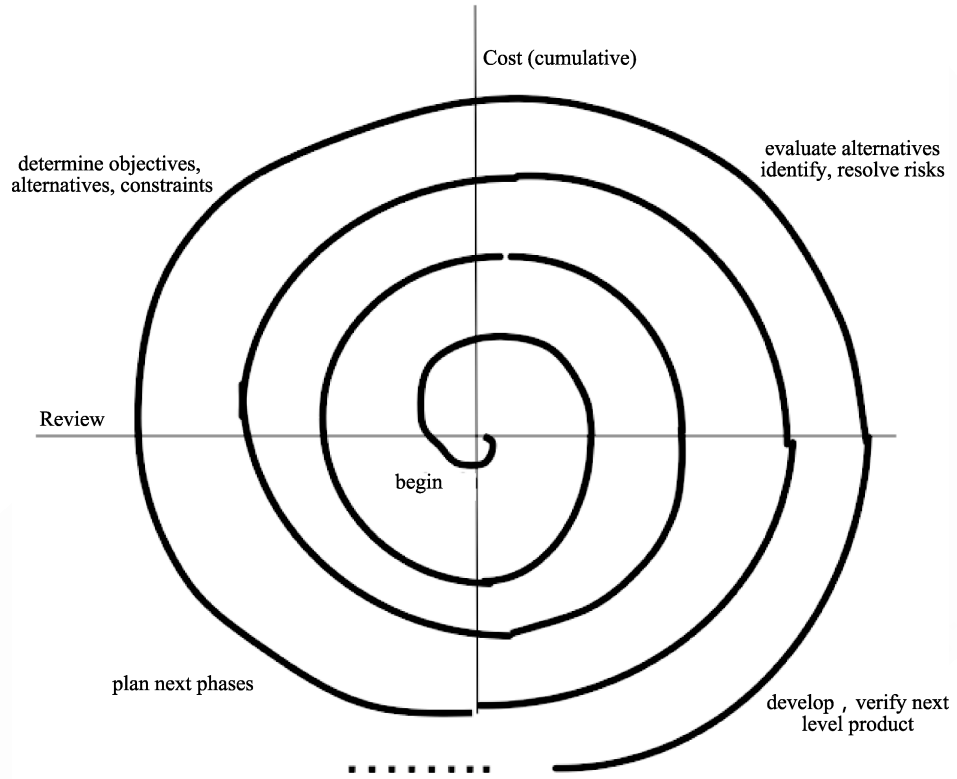

Figure 2. An abstraction showing the key underlying ideas behind the Spiral Model

3 The Little-JIL Process Definition Language

To support the goal of specifying the Spiral Model and its elaborations precisely, we will use the Little-JIL process language[5,16,17] to support the definition of various Spiral Model-based development processes.

The basic philosophy of the Little-JIL approach to process definition is that a process should be viewed as a hierarchical structure of steps. The hierarchy supports iterative decomposition of process details, and also provides a structure of scopes within which the steps reside. Within that structure, the individual steps can generally be regarded to be instances of invocations of procedures, where the step definitions have parameter lists, and arguments are passed in the course of invocations of these

steps. Upon execution of a step, the artifacts that are passed to it as arguments are bound, and upon completion of step execution output artifacts are bound to their associated calling arguments.

A Little-JIL step may incorporate a pre-requisite and/or a post-requisite. Each requisite is considered to be a step, and thus capable of definition by means of an arbitrarily deep hierarchy of substeps. But a requisite is also a predicate, and thus returns a value of either True or False. If the value is False, then an exception is thrown. The requisite, in that case, identifies the type of the exception and may also bind arguments that describe the nature of the condition causing the exception. The requisite structure that can be associated with a Little-JIL step also can provide the basis for the formulation of invariants that could be useful in supporting formal reasoning about Little-JIL process definitions.

As just noted, Little-JIL incorporates support for the handling of exceptions. Exceptions may be thrown by any step in response to any situation that requires special handling. It has already been noted that the violation of a requisite is one such situation that causes the throwing of an exception. In addition, the definer of a process may also provide the agent of a step with facilities for throwing an exception. This is done by providing a handler for the type of exception that the definer believes might be needed.

Exception handlers are substeps of a parent step. The parent step defines a scope, consisting of all of its descendants, and in doing so indicates that handlers attached to the parent step are in place to handle exceptions thrown within its scope. As noted above, exceptions in Little-JIL are typed, and thus a step may have more than one exception handler, where each different handler is in place to deal with exceptions of a different type. Each exception thrown may, moreover, be accompanied by a set of arguments that describe the nature of the exceptional situation in further detail.

The most noticeable feature of a Little-JIL process definition is its coordination structure. typically represented visually as a hierarchical structure of steps, each of which is represented as a collection of badges surrounding a black rectangular "step bar" (see Fig.3). Each step has a name that appears above its step bar, and a set of badges that surround the step bar and are imbedded within the step bar. The badges represent such step features as control flow among the step's substeps, the step's interface, which includes its parameters (i.e. the step's input and output artifacts), the exceptions the step handles, etc. A step with no substeps is called a *leaf step* and represents an activity to be performed by an agent, without any guidance from the process.

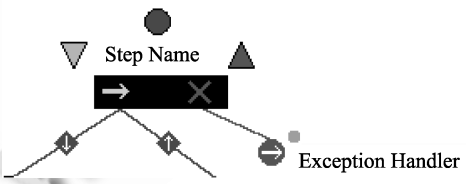

Figure 3. The iconic representation of a Little-JIL step

Substep Decomposition: Little-JIL steps may be decomposed into substeps of two different kinds, ordinary substeps and exception handlers. The *ordinary substeps* define the details of just how the step is to be executed. The substeps are connected

to the parent step by edges, which may be annotated by specifications of the artifacts that flow between parent and substep, and also by cardinality specifications. Cardinality specifications define the number of times the substep is to be instantiated and may be a fixed number, a Kleene * (for zero or more times), a Kleene + (for one or more times), or a Boolean expression (indicating whether the substep is to be instantiated or not). *Exception handlers* define the way in which exceptions thrown by the step's descendants are to be handled. The edge connecting an exception handler to its parent is annotated with the type of the exception being handled.

Step sequencing: Every non-leaf step has a *sequencing badge* (an icon embedded in the left portion of the step bar; *e.g.,* the right arrow in Fig.3), which defines the order in which its substeps execute. For example, a sequential step (right arrow) indicates that its substeps are to be executed sequentially from left to right and is only considered completed after all of its substeps have completed. A parallel step (equal sign) indicates that its substeps can be executed in any (possibly arbitrarily interleaved) order. It too is considered completed only after all of its substeps have completed. A choice step (circle slashed with a horizontal line) indicates that the agent executing the step is to make a choice among any of the step's substeps. A try step (right arrow with an X on its tail) mandates a sequence in which substeps are to be tried in order until one completes successfully.

Artifacts and artifact flows: An *artifact* is an entity that is used or produced by a step. Parameter declarations are specified as part of the interface to a step as lists of the artifacts used by the step (IN parameters) and the artifacts produced by the step (OUT parameters). The flow of artifacts between parent and child steps is (as noted above) indicated by attaching to the edges between parent and child identification of the artifacts as well as arrows indicating the direction of flow of each artifact.

Requisites: As noted above, requisites are optional and enable the checking of a specified condition either as a precondition for step execution or as a postcondition check to assure that the step execution has been completed acceptably. A downward arrowhead to the left of the step bar represents a prerequisite, and an upward arrowhead to the right of the step bar represents a post-requisite. If a requisite fails, an exception is triggered.

Exception Handling: A step in Little-JIL can define *exceptional conditions* when some aspects of the step's execution fail (*e.g.*, one of the step's requisites is violated). This violation triggers the execution of a matching exception handler associated with an ancestor of the step that throws the exception. An exception handler is represented as a step attached by an edge to an X on the right of the step bar (as shown in Fig.3).

Scoping and Recursion: The parent step and all of its descendants represent a *scope*, specifying what artifacts are considered local to that scope. Little-JIL also supports *recursive execution* of steps, which specifies the iterative application of a process step to specified inputs.

Abstraction: A Little-JIL step can be referred to from more than one location in a process definition. One such reference includes such details as the step's parameters and resource requirements, as well as any information about the step's exception handling facilities and substep decomposition details. This reference thus serves to define the step as an abstraction. Subsequent references, denoted by specifying the step's name in italics, then correspond to different instantiations of the step. Each

such reference results in the binding of the arguments presented to the step by its parent to the step's parameters, and comprises essentially an instantiation of the step as a procedural abstraction.

4 Elaborations of the Spiral Model Provide Diverse Software Development Processes

We now present a variety of elaborations of the Abstract Spiral Model (as suggested by Fig.2), using the Little-JIL process language to support precision in the descriptions of these elaborations. The breadth and diversity of these elaborations suggests that there may be more similarity than is commonly believed among the many different suggested approaches to developing software. This in turn suggests that perhaps more is understood and agreed upon about how to approach this critical and challenging problem. If so, then the Spiral Model incorporates some very deep and fundamental understandings of the essential nature of software development.

4.1 Little-JIL renderings of different Elaborations of the Abstract Spiral Model

Figure 4 depicts a Little-JIL definition of the Abstract Spiral Model, shown in Fig.2. The name of the process is **Develop Software**, but this consists of invoking **Iterate**, the step that captures the essence of software development. The very name, **Iterate**, suggests that iteration is indeed the essence of the process, and shows clearly that an iteration is the sequential execution of the four successive sequential phases, 1) **Determine Objectives, Alternatives, Constraints** 2) **Evaluate Alternatives, Identify and Resolve Risks,** 3) **Develop and Verify Next Level Product,** and 4)**Plan Next Phases**. The fourth phase, **Plan Next Phases**, is followed by a postcondition, labeled **Review**, which represents the Spiral Model mandated review of the outcome of the preceding iteration in order to determine if more iterations are needed. A failure of this **Review** throws an exception, which triggers the execution of the exception handler specified by the parent **Iterate** step. The exception handler is a recursive execution of the **Iterate** step itself. This clearly underscores that the process is iterative, and provides more detail about the nature of the iteration.

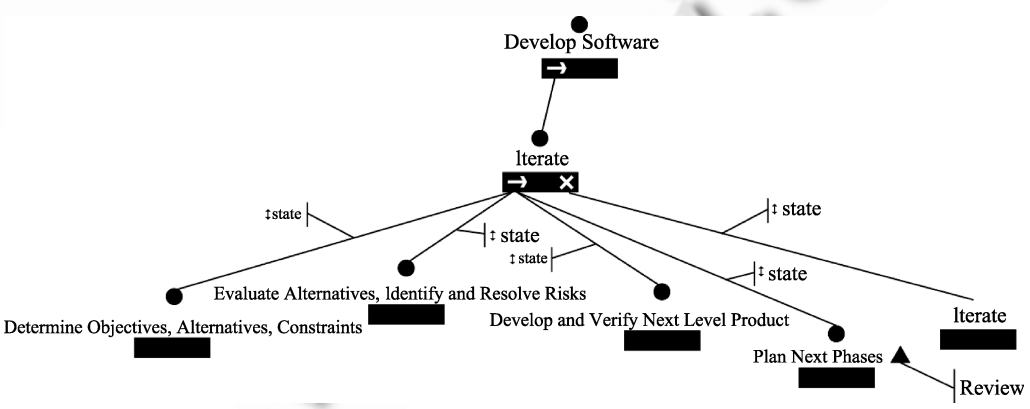

Figure 4. A representation of the Abstract Spiral Model in Little-JIL

Of particular interest is the way in which each iteration builds upon the previous iterations. Specifically note that all of the substeps of the **Iterate** step are annotated

to indicate that they receive as input the **state** artifact, which is intended to represent the evolving state of progress in developing the final software product. At the abstract level of this process representation, the **state** artifact is regarded as a kind of summary of all of the insights, accomplishments, and intermediate artifacts that have resulted from all of the preceding executions the **Iterate** step and its substeps. In subsequent processes that use **Iterate**, it will be possible to specify more fully and precisely the nature and content of **state**. The fact that **state** is denoted to be both an input to, and an output from, each of the four substeps of **Iterate** indicates that each phase is expected to build upon what has previously been done and what is now known about the software product that is being developed. The information contained in **state** is also input to the evaluation of the postrequisite that is used to determine whether or not any subsequent iteration(s) might be needed, making it explicit that the content of **state** is essential to the determination of the outcome of this review. If the review concludes that the artifact development is now complete, then this postrequisite will succeed, ending the spiral development, with **state** containing the final product. If the postrequisite fails, the current value of **state** is passed to the recursive invocation of **Iterate**, indicating that the next iteration is to build upon what has been done previously, and what has been determined previously about risks, their proposed mitigations, and the outcomes of these previous attempts at risk mitigation. This representation of the Abstract Waterfall Process makes clear the mechanism by which the "mass" of the evolving software product is continually growing, although it admittedly lacks the visual and intuitive impact that is provided by the expanding loops of the Abstract Spiral representation.

As an abstract representation of the iterative nature of software development, the process defined in Fig.4 does not yet describe in specific detail any particular software development process. But different development processes can be defined through different elaborations of different details of this process, thereby defining various specific software development processes. Thus, for example, different versions of the Waterfall Model can be constructed by adapting and combining the **Iterate** process shown in Fig.4 into different software development configurations.

More specifically, note that Fig.5 depicts a model of a Waterfall Process in which each of the four canonical software development stages, Requirements, Design, Implementation and Testing, is implemented as an instance of the **Iteration** process. The process in Fig.5 shows that software development proceeds by executing these four traditional stages sequentially, each stage implemented by the performance of the **Iteration** process. More specifically the four stages are represented by a cascade of steps called, **Requirements Stage, Design Stage, Implementation Stage,** and **Testing Stage**. Each step consists of first completing the development of an artifact and then initiating the development of the next artifact. Thus, the **Requirements Stage** step consists of first developing the **requirements** artifact (by executing the **Iteration** step, using the **requirements** artifact as its input and output), and then beginning the development of the **design** artifact (by executing the **Design Stage** step using the **requirements** artifact as input). **Iteration** is not shown here to save space, but it is identical to the **Iterate** step shown in Fig.4, except that **Iteration** as used here in Fig.5 (and in Fig.6 below) is modified by deleting the exception handler shown as a key feature of the **Iterate** step in Fig.4. Because of that modifi-

cation, when an exception is thrown by the fourth substep of **Iteration** in all of its uses in Fig.5, the exception is not handled locally by **Iteration**, but instead propagates to its parent. Thus, Figure 5 shows that an exception thrown by the execution of a **Requirements Stage** Iteration step is handled by recursively reinvoking **Requirements Stage**, but an exception thrown by the execution of a **Design Stage** Iteration step is handled by reinvoking **Design Stage**, and so forth. Thus, the failure of **Review** during **Requirements Stage** causes **Requirements Stage** to be repeated, until **Review** succeeds, the failure of **Review** during **Design Stage** causes the **Design Stage** step to be repeated, etc. Each of these development stages is now seen to be a Spiral development iteration process itself, and so the entire software development process consists essentially of four successively nested spirals.

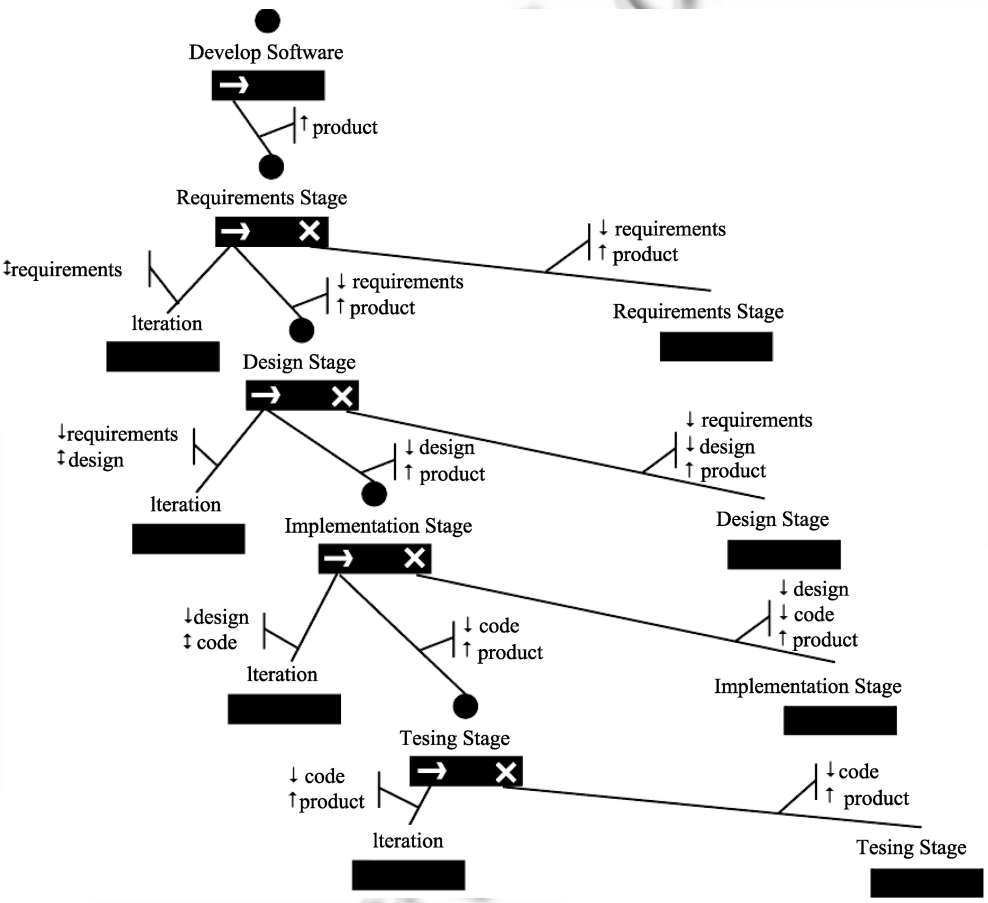

Figure 5. A rendering of a Spiral development process in which the four principal phases of software development are performed sequentially, with each being carried out as a Spiral Model iteration

Note also that the process shown in Fig.5 specifies that each of these development phases is executed using a different artifact as input. Thus, **Requirements Stage** is an **Iteration** that operates on the **requirements** artifact, which is then passed as input to **Design Stage**. **Design Stage** is an **Iteration** that operates

on the **design** artifact (using **requirements** as input), and passes **design** as input to **Implementation Stage**. This is an overly strict and narrow interpretation of the Waterfall Process that suggests that any development stage uses only the artifact developed in the previous stage as input and focuses its attention only on the artifact developed during the current stage. Thus for example the output of **Design Stage** namely the **design** artifact, would be the only input to the **Implementation Stage** step, which would disadvantage the performers of **Implementation Stage** by making **requirements** unavailable to them. Defining a Waterfall Process that enabled the performers of later stages to see all outputs of previous stages is easy to specify simply by annotating the in-edges of the different stages with the outputs of all previous stages, rather than with just the immediately previous stage. Note that the difference between these two processes is thus quite clear and apparent using this sufficiently precise process notation. The relative difficulty with which this is expressed using the more familiar spiral iconic representation seems to be one good motivator for using more precise notation to supplement the definition, and foster better understanding, of such processes.

Still more enlightened and realistic explanations of the Waterfall Process emphasize not only that each stage of software development proceeds using as input all the artifacts and understandings of all previous stages, but also that the failure of **Review** during one of the later stages might well trigger the recursive reinvocation of any previous stage, not just the immediately previous stage. A precise definition of this version of the Waterfall Process is more difficult to define and depict. This interpretation of the Waterfall Process is depicted in Fig.6.

In this process development is nominally considered to be the sequential execution of the traditional stages, **Requirements Stage, Design Stage, Implementation Stage,** and **Testing Stage**, with each of these stages being performed iteratively as specified by the process in Fig.5. But, as suggested by several authors (e.g. Boehm in Fig.1 of his 1988 paper) the **Review** carried out at the end of any development iteration might cause the reconsideration of any of the artifacts that had been developed during any prior iteration. Thus, **Review** after an **Implementation Stage** Iteration might trigger reconsideration of a previously developed **requirements** artifact (by performing **Requirements Stage**), or **design** artifact (by performing **Design Stage**), or **implementation** artifact (by performing **Implementation Stage**). Figure 6 uses Little-JIL process notation to specify this, by providing each development stage step with a different exception handler to support the possible performance of any previous stage. It is interesting to note that the process shown in Fig.6 is essentially the same as the process shown in Fig.5, except that the exception handlers for each development phase now allow any of the previous phases are to be revisited, while the handlers in the process shown in Fig.5 allow revisiting only the immediately previous stage. This software development process also emphasizes the fact that the **state** artifact, an abstract representation of all of the knowledge and artifacts previously developed, continues to grow as the process continues through the continued aggregation of newly developed and acquired information with previously acquired information and artifacts. Because the pattern of repeating previous steps is determined dynamically, however, it is impossible to be as precise about the nature of the artifacts that are passed into and out of the different stages. Thus Fig.6 indicates

that **state** is the artifact that is passed around in this process whereas it was possible to be more specific about the artifacts passed to the different stages in the process depicted in Fig.5.

Both processes also underscore the critical importance of the **Review** activity. We have noted that every Spiral iteration ends with a **Review**, aimed at determining whether or not to continue on to a next iteration, and providing evaluative information aimed at helping to guide how the next iteration is to proceed. At an abstract level, that **Review** activity is analogous in its central importance to programming language loop iterators. As such the **Review** activity would seem to merit particular attention. Thus it is not surprising that, in subsequent work, Barry Boehm has addressed the **Review** activity, making it the focus of his work on the Incremental Commitment Model (ICM). The Incremental Commitment Model advocates that the goals of each Spiral iteration be kept modest and provides guidance on how this might be done. As such the ICM provides elaborative details of how the **Review** activity, central to all Spiral Model development processes, is to proceed.

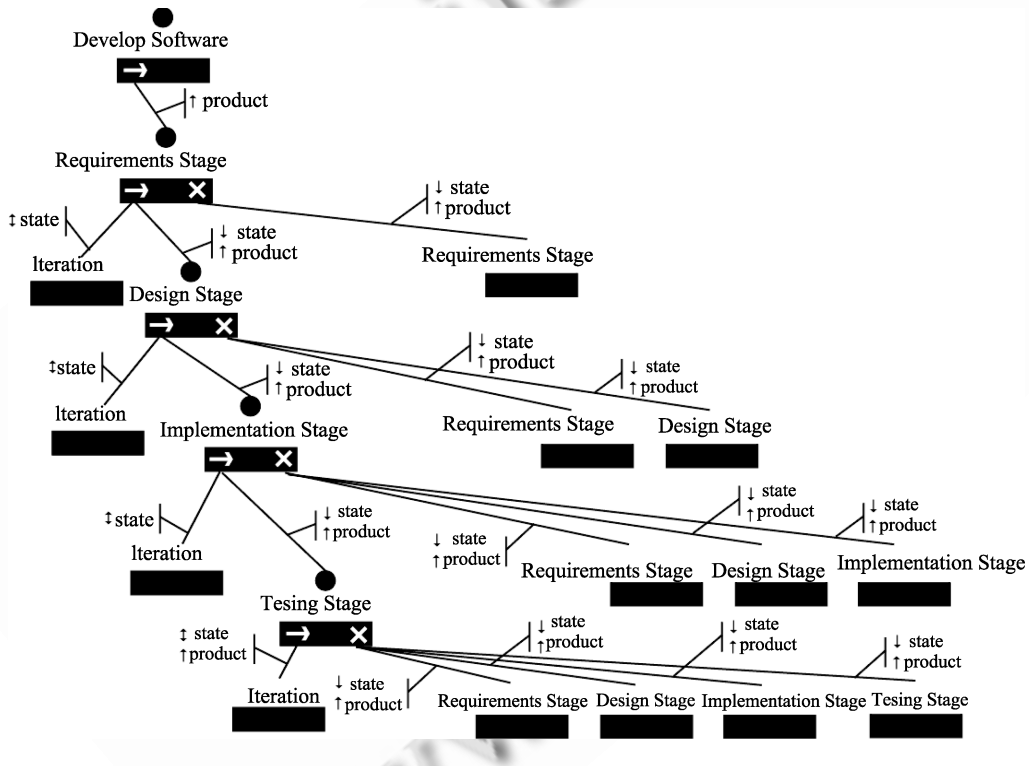

Figure 6. An interpretation of the Waterfall Process, based upon Spiral Model Iteration, in which a review of a software artifact may trigger the recursive execution of any previously developed software development stage

4.2 Other Elaborations of the Abstract Spiral Model

All of the processes elaborated in section 4.1 can be seen to be different kinds of iterations of the four Spiral Model phases. The differences between the two processes presented are entirely differences in the sequencing of the types of artifacts

considered. The process shown in Fig.5 mandates that the **Requirements Stage, Design Stage, Implementation Stage,** and **Testing Stage** are performed strictly in that order with each being performed as a sequence of iterations. Each of these four development stages is focused entirely on the development of a different type of software development artifact, guided by **state** information about the previously developed artifacts. Figure 6 presents a more liberalized, and realistic, software development process, suggesting that earlier stages and artifacts may need to be revisited and reconsidered as the development of later-stage software artifacts proceeds. But each revisitation and reconsideration is an iteration of the four stages prescribed as a Boehm-style loop around the Spiral. It is only the order in which software artifact types are considered and reconsidered that differentiates the processes shown in Figs. 5 and 6.

We now suggest that this kind of liberalization of the order in which software development artifacts and activities are undertaken opens the door to the possibility of considering many other kinds of software development processes to be elaborations of the Spiral Model. The possibilities here become still more numerous if we consider that the emphasis placed on the four different phases of a Boehm Spiral iteration will inevitably vary. Even in the processes depicted in Figs. 5 and 6, it must be expected that different developers at different times and under different circumstances will spend different amount of time and attention on the four different phases of the Boehm Spiral.

All that being the case, we now suggest that even an agile software development approach such as the Scrum[6,12,13,14] can readily be recast as an interpretation of Boehm's Spiral Model. Figure 7 suggests why that is the case.

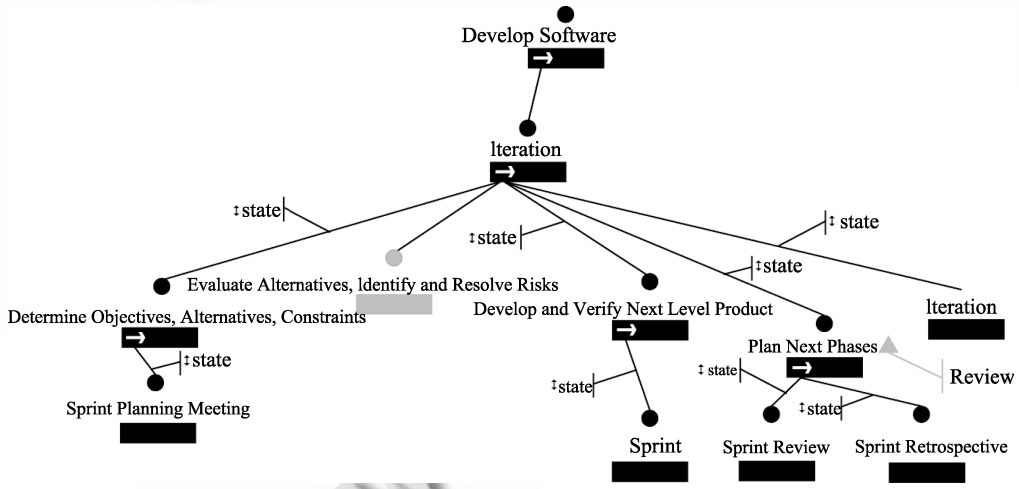

Figure 7. A Little-JIL definition of the Scrum process in which the activities that characterize the Scrum process are shown as elaborations of the phases of the Boehm Spiral Model, suggesting that the Scrum process can be thought of as a Spiral process

Figure 7 depicts the Scrum process as a sequence of iterations, each of which is called a *sprint*. Each sprint uses **state** information accumulated in the course of executing previous sprints, and adds to that body of **state** information in a way

that is little different from what was depicted in Fig.4. In addition, Figure 7 shows that a sprint is performed as a sequence of four phases, **Sprint Planning Meeting, Sprint, Sprint Review,** and **Sprint Retrospective**. The natures of these phases match closely the natures of the four phases of the Boehm Spiral Model. In particular, **Sprint Planning Meeting** seems strongly similar in goals and character to what Boehm describes as **Determine Objectives, Alternatives, Constraints**. Hence **Sprint Planning Meeting** is shown as the implementation of **Determine Objectives, Alternatives, Constraints**. It seems particularly appropriate that this identification be made, as the literature for each emphasizes that a central goal is the identification and mitigation of key product development risks. A sprint is the process of performing **Develop and Verify Next Level Product**, and thus is shown as the implementation of that Boehm Spiral phase. And the sequence of **Sprint Review** and **Sprint Retrospective** essentially comprise the Boehm activity of **Plan Next Phases**, and thus are shown as its elaboration. It is noteworthy that literature describing the Scrum process does not identify the need for the explicit performance during a sprint of activities that would comprise Boehm's **Evaluate Alternatives, Identify and Resolve Risks**. But it is unreasonable to suggest that this activity is not carried out in the course of planning and executing a sprint. Thus, this would seem to be an example of a Boehm Spiral phase that is at most simply deemphasized in this process, although it can hardly be expected that this phase and activity would not be carried out at all.

Thus, we suggest that Fig.7 is a demonstration of the way in which the Scrum process can also be viewed as an elaboration of Barry Boehm's Spiral Model. Both emphasize the centrality of iteration, both emphasize the growth of the final product as the iterative accumulation of knowledge and artifacts. Both emphasize the need for planning prior to doing, and evaluation subsequent to doing.

Having identified the way in which an agile method such as Scrum can be viewed as a series of Spiral Model-style iterations, it is now possible to conjecture that many other sorts of development processes might be synthesized by casting them as different elaborations of the Spiral Model. For example, the substitution of the Scrum process depicted in Fig.7 for the **Iteration** process used centrally in the development process depicted in Fig.6 suggests a Scrum-like approach to more traditional software development. Other mix-and-match combinations of agile and non-agile methods are also clearly possible.

5 Conclusions

Barry Boehm's enunciation of the Spiral Model presented a highly intuitive and evocative image of the nature of software development. Over the years the Spiral Model's idioms have been used widely to describe, motivate, and justify various approaches to software development. This paper supplements those intuitions about the Spiral Model with definitions of various development approaches that make their relations to the more abstract characteristics of the Spiral Model explicit. The paper suggests that a wide range of different approaches to software development can be viewed as different interpretations and instantiations of the Spiral Model.

What is perhaps most important is that the way in which this wide range of development approaches seem to be elaborations of the Spiral Model supports a view

that the Spiral Model captures some of the key essences of software development. Among those essences are that software development is quintessentially iterative, that it is a disciplined accumulation of an ever-growing body of knowledge, insights, and artifacts, that it should be driven by a continual quest to understand and mitigate risks, and that software artifact creation should be preceded by careful planning and consideration, and followed by careful evaluation and reconsideration.

These insights seem nicely supported by the diagrams presented in this paper. These diagrams themselves suggest some features of a process definition notation that seem particularly effective in making the above points clear. We noted above that Boehm's original Spiral Model diagram is intuitive and evocative. But subsequent discussions in this paper also made it clear that Boehm's original diagram was a less effective basis for supporting a clear understanding of the relations of various elaborations of the Spiral Model to each other. Thus, for example, while the processes depicted in Figures 5 and 6 are both clearly elaborations of the Spiral Model, neither is clearly enunciated by the Spiral Model diagram, which seems equally descriptive of both. Figures 5 and 6 make the differences quite clear, and also clarify their relations to each other. In particular, both figures show the cascading phases of the traditional Waterfall model in the familiar diagonal positioning on the left of the two figures. But in place of the imprecise and unclear back-edges in the Waterfall model, Figures 5 and 6 now show on the righthand side which waterfall phases can be revisited during the reworking of each waterfall stage. Figure 5 makes it clear that, for example, only Design can be revisited during the design stage, and only Coding can be revisited during the coding stage. Figure 6 makes it immediately clear, however, that in that process any prior development stage can be returned to from any later stage. While the two processes depicted in these two figures are both clearly possible elaborations of the Spiral Model, it is far harder to depict each of them clearly, in such a way as to make them distinguishable from each other, with the traditional Spiral Model visualization. Depicting the elaboration shown in Fig.6 would be particularly difficult.

Figures 5 and 6 also suggest that the essence of the processes depicted is actually recursion that makes important use of contextual information provided through the use of scoping, rather than simple iteration that makes little or no use of context. The nature of both recurrences is made quite clear by the figures, and indeed the role of scoping in indicating how the recursions work is also made clear. Recursive calls to the different development stages are clearly (and correctly) shown as different instantiations of the same process activities, but in different contexts. The hierarchical nature of Little-JIL makes this depiction particularly clear. In contrast note that the Scrum process depicted in Fig.7 is defined as an iteration, with successive sprints shown as sequential iterations. It might be argued that successive sprints do indeed take place in the context of all prior sprints, as knowledge of these prior iterations certainly resides in the heads of participants. Little-JIL would have no difficulty in using recursion to present such a depiction. But the culture and folklore surrounding the Scrum development philosophy encourages more a view that a sprint is a new undertaking, more focused on future goals and directions than on prior history. Thus, Figure 7 seems to be truer to the Scrum philosophy. Both views of the Scrum approach to software development are coherent with Boehm's Spiral Model of software development, but it would be difficult to use Boehm's visualization to distinguish be-

tween them. Little-JIL on the other hand supports the clear visual distinction of one from the other, thereby supporting a visual emphasis on the underlying philosophy of the Scrum.

The preceding discussion suggests that there are some process definition language features that seem particularly useful in defining these processes and underscoring their relations to each other. Procedural abstraction is clearly one such language feature, as it supports the specification of process step recursion in a particularly elegant way. The use of hierarchical elaboration is perhaps even more fundamental, enabling the specification of lower and lower levels of detail, but also providing a vehicle for the specification of scopes. Scoping the extent of artifacts is particularly important in supporting the effective specification of recursion, in particular.

Also of fundamental importance is the very integration of artifact flow with procedural specification. A particular weakness of Boehm's visualization of the Spiral Model is its marginalization of the specification of artifacts. As noted earlier, the increasing radii of the successive loops of the spiral suggest intuitively that each iteration has produced more knowledge and greater amounts of software artifacts. But it is hard to identify just which artifacts are produced where and how previously developed artifacts are the basis for the creation of subsequent artifacts. Language facilities for specifying artifact flow through process activities seems essential for supporting the ability to be precise and specific in this way.

And finally, it should be noted that strong support for exception management also seems to be a particularly important feature of a language to be used as the basis for defining software development processes. The examples in this paper make it clear that there must be strong capabilities for specifying what to do when, inevitably, development activities are shown to have produced flawed results. The well-known Ripple Effect is often used to describe how changing one identified software defect may incur the need to then find and fix other defects, as well as still other defects created by making changes to still other defects. A powerful exception management facility in a process language can model orderly processes for dealing with such ripple effects and for specifying how to return to nominal development flow after dealing with these kinds of cascading defects.

Thus, this work has also helped to identify process definition language semantic features that seem particularly important and adept in supporting development of precise specifications of real-world software development processes.

Acknowledgments

The Spiral Model has served as a strong and continuing inspiration for my own personal musings over the years and decades about the nature of effective software development practice. From its very first enunciation, coming almost simultaneously with my own enunciation of the idea of process programming[9], the Spiral Model has been an important test of the effectiveness and expressiveness of my ideas about software process definition. This paper seems to provide perhaps the strongest demonstration yet of the effectiveness and validity of my ideas about process programming, while simultaneously also underscoring the wisdom and robustness of Barry Boehm's ideas about the fundamental nature of software development. It is now crystal clear that Barry was right in emphasizing that software development, and indeed the de-

velopment of all large and complex systems, must be iterative, and focused on incremental acquisition of knowledge, reduction of risk, and accretion of ever-larger and better integrated bodies of software artifacts. As such the Spiral Model must stand as one of the most fundamental and far-reaching contributions of Barry Boehm to software engineering.

This paper owes a very strong debt to the ideas and contributions of Sandy Wise. The work has evolved through numerous conversations with Sandy over the years. Most recently Sandy has also made enormously important contributions to this paper by drafting, discussing, and iterating the diagrams that are the core of the contributions of this paper.

Finally, I gratefully acknowledge that research funding provided by the U.S. National Science Foundation under Award Nos. IIS-0705772, and CCF-0820198 was essential to the support of this research. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied of the U.S. National Science Foundation or the U.S. Government.

References

[1] Boehm BW. A Spiral Model for Software Development and Enhancement. IEEE Computer, May 1988. 61–72.

[2] Boehm BW, Belz F. Experience with the Spiral Model as a process model generator. Proc. of the 5th International Software Process Workshop: Experience with Software Process Models. Kennebunkport, ME, October 1989, D. Perry, Editor; IEEE Computer Society Press, 1989. 43–45.

[3] Boehm BW, Bose P. A collaborative Spiral Process Model based on Theory W. Proc. of the 1994 International Conference on the Software Process. IEEE Computer Society Press, Los Alamitos, CA, 1994. 59–68.

[4] Boehm BW, Hansen W. The Spiral Model as a tool for evolutionary acquisition. Cross Talk, May 2001. 4–11.

[5] Cass AG, Lerner BS, McCall EK, Osterweil LJ, Sutton SM Jr., Wise A. Little-JIL/Juliette: A process definition language and interpreter. 22nd International Conference on Software Engineering (ICSE 2000). Limerick, Ireland, June 2000. 754–757.

[6] Cohn M. Succeeding with Agile: Software Development Using Scrum. Pearson Education, Inc., Boston, MA, 2010.

[7] Deming WE. Out of the Crisis. MIT Press, Cambridge, MA, 1982.

[8] Hansen WJ, *et al.* Spiral Development: Building the Culture; A Report on the CSE-SEI Workshop. February, 2000, CMU/SEI-2000-SR-006, Pittsburgh PA, Software Engineering Institute, 2000.

[9] Osterweil LJ. Software Processes are Software Too. 9th International Conference on Software Engineering (ICSE 1987). Monterey, CA, March 1987. 2–13.

[10] Royce WW. Managing the development of large software systems. Proc., IEEE Wescon August 1970. 1–9. Also, Proc. of the 9th International Conference on Software Engineering, Monterey, California, United States, 1987, IEEE Computer Society Press, 1987. 328–338.

[11] Royce WE. TRW's Ada process model for incremental development of large software systems. Proc. of the 12th International Conference on Software Engineering, IEEE Computer Society Press, Los Alamitos, CA, pp. 2–11.

[12] Schwaber K, Beedle M. Agile Software Development with Scrum. Prentice Hall, Upper Saddle River, New Jersey, 2002.

[13] Schwaber K. Agile Project Management with Scrum. Microsoft Press, Redmond, WA, 2004.
[14] Scrum Alliance, Inc. http://www.scrumalliance.org/
[15] Software Productivity Consortium. Process Engineering with the Evolutionary Spiral Process Model: Version 01.00.06, Technical Report SPC-93098-CMC, Herndon, VA 1994.
[16] Wise A. Little-JIL 1.0 Language Report, Department of Computer Science, University of Massachusetts: Amherst, Amherst, MA, 1998.
[17] Wise A. Little-JIL 1.5 Language Report, University of Massachusetts Amherst, 2006.