CHANGING *the* PARADIGM *of* SOFTWARE ENGINEERING

Software evolution, iterative, and agile development represent a fundamental departure from the previous waterfall-based paradigm of software engineering.

By VÁCLAV RAJLICH

There has been much discussion as to what degree iterative and agile processes and software evolution are different from previously promoted practices based on the waterfall life cycle model. Here, I argue that the departure is fundamental and, in fact, it represents a new paradigm.

The notion "paradigm" was used by Kuhn [8] to describe a "coherent tradition of scientific research" that includes knowledge, techniques, research agenda, and so forth. Kuhn collected extensive historical data and proved that change from an old paradigm to a new one takes place when the old paradigm is in crisis and cannot explain compelling new facts. It is a genuine revolution in a particular branch of science and it represents a discontinuity in the development of that branch.

Kuhn argues that there is always a substantial investment in the old paradigm that causes a resistance to the paradigm change. This resistance is fueled by the fact that as a result of the paradigm change, knowledge accumulated up to that point may lose its significance. This makes the advantages of the new paradigm disputable, especially among those with the greatest knowledge of the old paradigm. Those who have a vested interest in the old paradigm will always attempt to extend that paradigm to accommodate new facts. Thus, the final victory of the new paradigm can only be guaranteed by a generation change.

With just a slight shift in the meaning, I use the notion of "paradigm" as "coherent tradition of software development," and argue the current movement toward software evolution and agile and iterative processes of software development represent nothing less than a paradigm change. In order to describe the scene in which this change is taking place, it helps to review the history of software technology.

Software separated in the 1950s from the underlying hardware and emerged as a distinct technology, turning out independent products and requiring a specialized set of skills. Original programmers were recruited mostly from the ranks of hardware engineers and mathematicians who used ad-hoc techniques they carried from their former fields.

The first paradigm change occurred in the late 1960s. It was precipitated by the fact that the ad-hoc techniques did not scale up to the large systems. The situation is vividly described by Brooks in [2] where the demands of the new operating system OS/360 taxed the limits of the programmers, project managers, and the resources of IBM Corp. The result of the crisis was a paradigm change that established a new academic software engineering discipline and introduced the waterfall metaphor into the production of software.

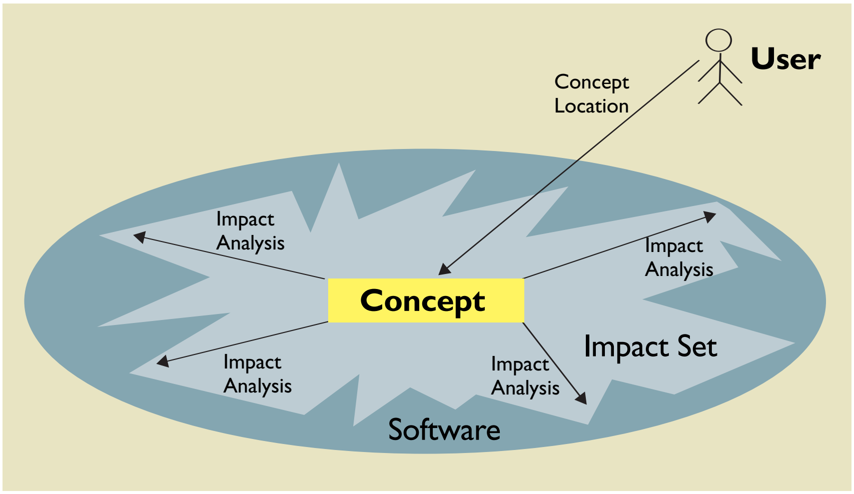
The waterfall metaphor is widely used in the construction industry and product manufacturing. It requires the developer first to collect the requirements that describe the functionality of the future product, and then to create a design that will be followed during the entire construction. When the product is finished, it is transferred to the user and any residual problems that may surface afterward are resolved through maintenance.

The waterfall metaphor is an intuitively appealing metaphor. Common sense dictates to try to avoid the expensive late rework; requirements elicitation up front and good design based on these requirements lessens the need for the expensive rework. Waterfall became the dominant paradigm through which the software development was viewed.

However, in the context of software development, waterfall is beset by a serious problem of requirements volatility. Requirements are not fully known in advance and are often added during the course of the project. For example, Cusumano and Selby [3] found that 30% of the requirements for Microsoft projects were not elicited in advance, but emerged only when the development was already under way, possibly as a result of the developer's learning. Any design based on volatile requirements must of necessity be only temporary and cannot guide developers through the entire development process.

Additional data indicated that in spite of widespread use of waterfall, there were very few successful projects [7]. In an oft-quoted report, the Standish Group revealed that in 1995, 31.1% of all software projects were canceled, 52.7% were "challenged" (completed only with great difficulty, and with large cost or time overruns), and only 16.2% could be called successful. Obviously, the waterfall metaphor did not solve the problems of software development.

Although use of waterfall is in decline in the industry, it still explicitly or implicitly drives much of the discussion and research. As in other scientific revolutions, there are attempts to extend the waterfall in order to adapt it to the new data and address the most glaring deficiencies. One such approach calls for the anticipation of the future changes and proposes to build the software in such a way that those changes will be easily implemented. This again is a commonsense idea, but it requires a crystal ball that often is not available. Examples of dramatic unanticipated events are unexpected company mergers that require companies to unify their IT efforts (example: Daimler Chrysler), unexpected changes in the law, unexpected changes in technology (arrival of the Internet) and so forth. Anticipating future changes—if at all possible—can only lessen the problem of requirements volatility, but cannot solve it.



Incremental change design.

Another approach advocates software prototyping. The prototype is a first throw-away version of the software and its purpose is to elicit the requirements. This is only a partial solution to the problem; it assumes the programmers elicit all requirements while prototyping. In reality, late new requirements emerge due to environment volatility and the developer learning that still goes on through the rest of the development. Both anticipation and prototyping lessen the problems of volatility, but they do not offer a complete solution. Indeed, a complete solution can only come from a new paradigm.

The New Paradigm is based on Software Evolution

The current revolution in software engineering is the response to the data that proved the waterfall life cycle, including its change anticipation and prototyping variants, cannot solve the problem of requirements volatility. As in other scientific revolutions, the new paradigm comes from several independent sources. It gained momentum recently and surfaced as a flood of new ideas, books, and articles.

One source is the empirical studies of actual software life cycles, which resulted in a staged model [9] that divides the software life cycle into five stages. There is initial development that creates the first version of the software, and establishes the fundamental commitments that will characterize the software through the rest of its life cycle. The next stage, evolution, consists of iterations that adapt the software to the changing requirements caused by the environment volatility and add new functionality and other new properties. The value and size of the software grows during the evolution, so sometimes the evolution is called "growth stage."

The factor that stops the evolution is either a management decision or the code decay [4] that brings the loss of architecture coherence and the loss of the knowledge about the program. The software enters the servicing stage, where the value no longer grows and changes are limited to patches and wrappers that only fix minor deficiencies. The life cycle ends with the phase-out stages, where no more servicing is provided and the users work around the known deficiencies, and close-down that discontinues the software use.

Iterative program development is another source of the new paradigm [6]. It is based on the observation that during the program development, there is a contradiction between the need of programmers to have a predictable and stable environment and the volatility of requirements. The attempt to freeze the requirements for the entire duration of the project does not work, because the accumulation of both the external and internal changes is too large and results in project failures. The opposite extreme—flooding the programmers with the constant stream of changes—also does not work because it creates project chaos. A reasonable compromise is to freeze the environment for a limited time, called iteration.

At the end of the iteration, the stakeholders (programmers, managers, users) evaluate the progress, take into account the changes in the environment, and decide the direction for the next iteration. The iterations provide feedback and lessen the risks of the surprises and failures. Fast iterations are recommended by agile processes like eXtreme Programming (XP) [1]. In terms of the staged life cycle, iterative program development extends the stage of evolution at the expense of the initial development, which becomes just one of the iterations.

The New Research Agenda

With its emphasis on software evolution, the new paradigm focuses on the modifications in the existing software. This puts a new research agenda into the spotlight. One of the increasingly important topics is *incremental change* (IC), the purpose of which is to add a new functionality or a new property to the existing software [10]. Under the old paradigm, IC was supposed to happen only rarely, because all required functionalities and properties should have been identified in advance and implemented during the initial development. In the modified old paradigm, where all changes are anticipated, all changes should be local and trivial. As the result of these views, IC was largely ignored by researchers and educators as something of little importance; in fact, current programmers must learn it on their own.

Software evolution, iterative development, and agile development are all based on repeated IC. Because of that, IC is poised to move into the center of the research interest. The questions are: can the practice of IC be summarized, modeled, formalized, improved, taught, and supported by tools? For example, many issues of the IC design are open and require additional research. IC design has been divided into two closely related activities: concept location and impact analysis (see the accompanying figure).

Concept location [10] starts with the change request and determines where in the existing software the change should be made. Programmer's knowledge might be sufficient for concept location in small and familiar programs, but it fails in large or unfamiliar ones. Ideally in that situation, there should be an external documentation that guides the programmers to the proper place in the code. However, such documentation is very rare and the programmers must rely on their skills. A crude but widely used string pattern matching technique locates identifiers or comments that indicate the presence of the concept in the code. Unix utility "grep" is the best known example of a pattern-matching tool and therefore this technique is often called grep technique. For example, if the programmers want to locate the concept "pay stub" in the code, they look for identifiers "paystub," "payStub," "paystb," and so on.

In spite of its wide use, "grep" has serious limitations as it depends on the presence of relevant comments and identifiers in the code. It frequently fails because of the use of synonyms or homonyms that lower precision and recall. Search for better concept location techniques is one of the important research issues of IC. The new concept location techniques broadly fall into static and dynamic categories, where static techniques analyze the static software source code while dynamic techniques analyze results of the program execution [12].

Impact analysis [10] is also a part of IC design and it determines full extent of the change, by finding all software components that will be affected by the IC. After the actual change in the software was made, *change propagation* [10] finds all places where the secondary changes are to be made in order to complete the change. Program dependency analysis plays an important part in both impact analysis and change propagation.

Refactoring is a change to the software that does not modify the functionality but modifies the structure. After repeated incremental changes, the architecture of the program can become disorganized and refactoring reintroduces the order. Opportunistic refactoring prepares software for a specific incremental change.

There are many different refactoring transformations, starting with simple renaming of an entity of the program, and including more sophisticated changes like refactoring a base class, refactoring design patterns, and so forth. Refactoring has been covered in numerous articles and books [5]. Many currently available software environments are equipped with tools to support at least some refactoring transformations.

Code decay [4] causes transition from the stage of evolution, where large changes in the functionality are possible, to a servicing stage where only minor patches and wrappers are possible, therefore substantially lowering the value of the software. Software managers should be aware of this phenomenon, as it can happen by accident. There has been very little research into what causes code decay and how to prevent it.

Cognitive aspects of software development and program comprehension [11] are also emphasized by the new paradigm. The software evolution presents an opportunity for the programmers to learn, and this learning will determine the success of the project. New programmers joining the project must absorb project knowledge and current documentation systems fail to capture this knowledge adequately, making the entrance of the new programmers into an old project difficult. It was speculated that the code decay is characterized by the situation where complexity of the code outruns the cognitive capabilities of the programmers [9].

Conclusion

The old waterfall paradigm tried to freeze requirements for the duration of software development and created a situation where requirements volatility caused too many project failures, while the new paradigm addresses this shortcoming by its emphasis on software evolution. The new paradigm brings a host of new topics into the forefront of software engineering research. These topics have been neglected in the past by the researchers inspired by the old paradigm, and therefore there is a backlog of research problems to be solved. C

References

1. Beck, K. *Extreme Programming Explained.* Addison Wesley, Reading, MA, 2000.
2. Brooks, F.P. *The Mythical Man-Month.* Addison-Wesley, Reading, MA, 1982.
3. Cusumano, A.M. and Selby, R.W. How Microsoft builds software. *Commun ACM 40*, 6 (June 1997), 53–61.
4. Eick, S.G., Graves, T.L., Karr, A., Marron, J.S., and Mockus, A. Does code decay? Assessing the evidence from change management data. *IEEE Trans. Software Engineering 27*, 1 (2001), 1–12.
5. Fowler, M. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, Reading, MA, 1999.
6. Jacobson, I., Booch, G. and Rumbaugh, J. *The Unified Software Development Process.* Addison Wesley, Reading, MA, 1999.
7. Johnson, J.H. *The CHAOS Report.* The Standish Group International, Inc., (1994); www.standishgroup.com/sample_research/ index.php.
8. Kuhn, T.S. *The Structure of Scientific Revolutions.* The University of Chicago Press, Chicago, 1996.
9. Rajlich, V. and Bennett, K. A staged model for the software lifecycle. *Computer 33*, 7 (July 2000) 66–71.
10. Rajlich, V. and Gosavi, P. Incremental change in object-oriented programming. *IEEE Software 21* (July/Aug. 2004), 62–69.
11. Xu, S., Rajlich, V., and Marcus, A. An empirical study of programmer learning during incremental software development. In *Proceedings of the IEEE International Conference on Cognitive Informatics* (2005), 340–349
12. Wilde, N., Buckellew, M., Page, H., Rajlich, V. and Pounds, L. A comparison of methods for locating features in legacy software. *J. Systems and Software 65,* 2 (Feb. 2003), 105–114.

This research was supported in part by grants from the National Science Foundation (CCF-0438970), the National Institute for Health (NHGRI 1R01HG003491), and by 2005 IBM Faculty Award. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSF, NIH, or IBM.

Václav T. Rajlich (rajlich@wayne.edu) is a professor and former chair in the Department of Computer Science at Wayne State University, Detroit, MI.