
ALAN M. DAVIS, SENIOR MEMBER, IEEE, EDWARD H. BERSOFF, SENIOR MEMBER, IEEE, AND EDWARD R. COMER, MEMBER, IEEE

Abstract—The classic waterfall model of software engineering is used throughout the production software development community. The escalating costs associated with software development and the unsatisfactory reliability, performance, and functionality of the resulting software have motivated software engineers to develop new alternate models of software development including prototyping, software synthesis, and reusable software. It is difficult to compare and contrast these new models of software development because their disciples often use different terminology, and the models often have little in common except their beginnings (marked by a recognition that a problem exists) and ends (marked by the existence of a software solution). This paper provides a framework which can serve 1) as a basis for analyzing the similarities and differences among alternate life cycle models; 2) as a tool for software engineering researchers to help describe the probable impacts of a new life cycle model; and 3) as a means to help software practitioners decide on an appropriate life cycle model to utilize on a particular project or in a particular application area.

Index Terms—Reusable software, software development life cycles, software prototyping, software synthesis, waterfall model.

INTRODUCTION

The classic waterfall model (see Fig. 1) was defined as early as 1970 by Royce [1] and later refined by Boehm [2] in 1976 to help cope with the growing complexity of the software projects being tackled. The use of such a model:

- encourages one to specify what the system is supposed to do (i.e., to define the requirements) before building the system (i.e., designing);
- encourages one to plan how components are going to interact (i.e., designing) before building the components (i.e., coding);
- enables project managers to track progress more accurately and to uncover possible slippages early;
- demands that the development process generate a series of documents which can later be utilized to test and maintain the system;
- reduces development and maintenance costs due to all of the above reasons; and
- enables the organization that will develop the system to be more structured and manageable.

Most "standard" methodologies for commercial corporations, government contractors, and governmental entities follow some basic variation of the waterfall model, although there are a variety of different names for each of the stages. Thus, the requirements stages are often called user needs analysis, system analysis, or specifications; the preliminary design stage is often called high-level design, top-level design, software architectural definition, or specifications; the detailed design stage is often called program design, module design, lower-level design, algorithmic design, or just plain design, etc. For the most part, however, all these methodologies are equivalent.

During the past five to ten years, alternative, radically different methodologies have appeared, including rapid throwaway prototypes, incremental development, evolutionary prototypes, reusable software, and automated software synthesis. The developers of these techniques had a variety of motivations, but primarily they were looking for ways of rescuing the software industry from what appears to be a dilemma: software is almost always more expensive and...
delivered later than expected, and to make matters worse, it is often unreliable and fails to meet the ultimate users' needs. Project managers often bypass stages or take shortcuts in order to solve the problem, but these unplanned and extemporaneous alterations of the life cycle just make the software even more expensive, later, and more unreliable.

The rapid throwaway prototyping approach (made popular by Gomaa [3]) addresses the issue of ensuring that the software product being proposed really meets the users' needs. The approach is to construct a "quick and dirty" partial implementation of the system prior to (or during) the requirements stage. The potential users utilize this prototype for a period of time and supply feedback to the developers concerning its strengths and weaknesses. This feedback is then used to modify the software requirements specification to reflect the real user needs. At this point, the developers can proceed with the actual system design and implementation with confidence that they are building the "right" system (except in those cases where the user needs evolve). An extension of this approach uses a series of throwaway prototypes [4], culminating in full-scale development.

Incremental development [5] is the process of constructing a partial implementation of a total system and slowly adding increased functionality or performance. This approach reduces the costs incurred before an initial capability is achieved. It also produces an operational system more quickly, and it thus reduces the possibility that the user needs will change during the development process.

In evolutionary prototyping (established briefly at the end of [6]), the developers construct a partial implementation of the system which meets known requirements. The prototype is then used by its intended users in order to understand the full requirements better. Whereas incremental development implies that we understand most of our requirements up front and simply choose to implement it in subsets of increasing capability, evolutionary prototyping implies that we do not know up front all of our requirements, but need to experiment with an operational system in order to learn them. Note that in the case of throwaway prototypes we are likely to implement only those aspects of the system that are poorly understood, but that in the case of evolutionary prototypes we are more likely to start with those system aspects that are best understood and thus build upon our strengths. For complex applications, it is not reasonable at this time to expect this application of prototypes to be particularly "rapid" because reliability, adaptability, maintainability, and performance (RAMP) are major forces behind making such system developments expensive and time-consuming. Since the technology is not yet available to retrofit RAMP requirements, they would have to be implemented up front, thus forcing software development costs high and schedules to their limit. Evolutionary prototypes will become more practical in the future as techniques for retrofitting RAMP requirements are developed.

Whereas prototyping attempts to reduce development costs through partial implementations, reusable software [7] is the discipline of attempting to reduce development costs by incorporating previously proven designs and code in new software products. The software industry is guilty of continuously reinventing the wheel. This is primarily because few tools are available to help reuse software designs or code from previous projects. Clearly, what is needed are techniques to create reusable components, techniques and tools to store and retrieve reusable components, and component specification techniques to help catalog and locate relevant components. The net effect of reusing components would be shorter development schedules (by using wheels rather than reinventing them) and more reliable software (by using components that have been previously "shaken down").

Automated software synthesis is a term used to describe the transformation of requirements or high-level design specifications into operational code. The transformation process may be directed by algorithmic [8] or knowledge-based [9] techniques. As Parnas [10] points out, each generation of software engineering researchers applies the term "software synthesis" to one language "higher" than the one currently used for programming. Thus, when machine language was used, software synthesis referred to the "automatic" translation of assembly language into machine code (now called assembly). Later, it referred to the translation of a high-level language into machine code (now called compilation). Now, it refers to the translation of very-high-level languages (VHLL’s) into machine code. Since lines of code produced by person-month are relatively independent of the implementation language, it becomes clear that the higher the programming language used becomes, the more true productivity (as measured by the amount of functionality implemented per person-month) increases.

The purpose of this paper is to describe a paradigm which can be used to compare and contrast each of the above alternative life cycle models with the more conventional waterfall model (and with each other) in the face of constantly evolving user needs.

The Paradigm

For every application beyond the trivial, user needs are constantly evolving. Thus, the system being constructed is always aiming at a moving target. This is a primary reason for delayed schedules (caused by trying to make the software meet a new requirement it was not designed to meet) and software that fails to meet customer expectations (because the developers "froze" the requirements and failed to acknowledge the inevitable changes).

Fig. 2 shows graphically how users' needs evolve over time. It is recognized that the function shown is neither linear nor continuous in reality. Please note that 1) the scale on the x-axis is not shown (the units can be either

2The term "paradigm" is used in this paper to mean "metamodel," i.e., a model to describe software life cycle models.
months or years), but could be assumed to be nonuniform, containing areas of compression and decompression, and 2) the units of the scale on the y-axis are not shown, but are assumed to be some measure of the amount of functionality (such as DeMarco's "Bangs for the Buck" [11]). However, none of the observations made in this paper is dependent on either the uniformity of the axes or the linearity or continuity of the curve shown in Fig. 2.

Fig. 3 shows what happens during a conventional software development. At time $t_0$, a need for a software system is recognized and a development effort commences with relatively incomplete knowledge of the real user needs at time $t_0$. At time $t_1$, the development effort has produced an operational product, but not only does it not satisfy the current $t_1$ needs, it does not even satisfy the old $t_0$ needs because of a poor understanding of those needs in the first place. The product now undergoes a series of enhancements (between times $t_1$ and $t_3$), which eventually enable it to satisfy the original requirements (at $t_3$) and then some. At some later point in time $t_4$, the cost of enhancement is so great that the decision is made to build a new system (once again based on poorly understood requirements), development of the product is completed at time $t_4$, and the cycle repeats itself.

A number of useful metrics can now be defined based on the paradigm defined above. These metrics can later be used to compare and contrast sets of alternative life cycle approaches. These metrics are portrayed graphically in Fig. 4 and are described below.

1) A **shortfall** is a measure of how far the operational system, at any time $t$, is from meeting the actual requirements at time $t$. This is the attribute that most people are referring to when they ask "Does this system meet my needs?"

2) **Lateness** is a measure of the time that elapses between the appearance of a new requirement and its satisfaction. Of course, recognizing that new requirements are not necessarily implemented in the order in which they appear, lateness actually measures the time delay associated with achievement of a level of functionality.

3) **The adaptability** is the rate at which the software solution can adapt to new requirements, as measured by the slope of the solution curve.

4) **The longevity** is the time a system solution is adaptable to change and remains viable, i.e., the time from system creation through the time it is replaced.

5) **Inappropriateness** is the shaded area between the user needs and the solution curves in Fig. 5 and thus captures the behavior of shortfall over time. The ultimately "appropriate" model would exhibit a zero area, meaning that new requirements are instantly satisfied.

Each of the alternative life cycle models defined earlier is now analyzed with respect to the paradigm described above.

---

**Fig. 2.** Constantly evolving user needs.

**Fig. 3.** Software products fall short of meeting all current user needs.
Rapid Throwaway Prototypes

The use of a rapid throwaway prototype early in the development life cycle increases the likelihood that customers and developers will have a better understanding of the real user needs that existed at time $t_0$. Thus, its use does not radically affect the life cycle model per se, but does increase the impact of the resulting system. This is shown in Fig. 5, where the vertical line (i.e., the increase in functionality provided by the system upon deployment) at time $t_i$ is longer than in the conventional approach. Fig. 5 also shows the rapid prototype itself as a short vertical line providing limited and experimental capability soon after time $t_0$. There is no reason to believe that the length of time during which the product can be efficiently enhanced without replacement is any different than with the conventional approach. Therefore, this period of time for the rapid prototype-based development (i.e., $t_3 - t_1$) is shown in Fig. 5 the same as for the conventionally developed product.

Incremental Development

When using incremental development, software is deliberately built to satisfy fewer requirements initially, but is constructed in such a way as to facilitate the incorporation of new requirements and thus achieve higher adaptability. This approach has two effects: 1) the initial development time is reduced because of the reduced level of functionality, and 2) the software can be enhanced more easily and for a longer period of time. Fig. 6 shows how this approach compares to the conventional approach. Note that the initial development time is less than for the conventional approach, that the initial functionality (A) is less than for the conventional approach (B), and that the increased adaptability is indicated by a higher slope of the curve A-C than for the conventional approach (line B-D). The stair step aspect of the graph indicates a series of well-defined, planned, discrete builds of the system.

Evolutionary Prototypes

This approach is an extension of the incremental development. Here, the number and frequency of operational prototypes increases. The emphasis is on evolving toward a solution in a more continuous fashion, instead of by a discrete number of system builds.

With such an approach, an initial prototype emerges rapidly, presumably demonstrating functionality where the requirements are well understood (in contrast to the throwaway prototypes, where one usually implements the
poorly understood aspects first) and providing an overall framework for the software. Each successive prototype explores a new area of user need, while refining the previous functions. As a result, the solution evolves closer and closer to the user needs (see Fig. 7). In time, it too will have to be redone or undergo major restructuring in order to continue to evolve.

As with the incremental development approach, the slope (line A–C) is steeper than in the conventional approach (line B–D) because the evolvable prototype was designed to be far more adaptable. Also, the line A–C in Fig. 7 is not stepped like line A–C in Fig. 6 because of the replacement of well-defined and well-planned system "builds" with a continuous influx of new, and perhaps experimental, functionality.

**Reusable Software**

Reuse of existing software components has the potential to decrease the initial development time for software significantly. Fig. 8 shows how this approach compares to conventional development. No parameters are changed, except for the development times.

**Automated Software Synthesis**

In the ultimate application of this approach, as an engineer recognizes the requirements, these are specified in some type of VHLL and the system is automatically synthesized. This approach has two dramatic effects: 1) the development time is greatly reduced, and 2) the development costs are reduced so much that adapting "old" systems is rarely more meritorious than resynthesizing the entire system. Thus, the longevity of any version is low, and the result is a stair-step graph, as shown in Fig. 9, where the horizontal segments represent the time the system is utilized and the time needed to upgrade the requirements. The vertical segments represent the additional functionality offered by each new generation.

**Summary**

Note that all five approaches reduce the area between the user need graph and actual system functionality graph when compared to conventional development. That is, all five approaches decrease shortfall, lateness, and inappropriateness to varying degrees. It is for this very reason that these alternative life cycle approaches were developed by their inventors.
Other Models

The paradigm discussed in this paper makes it apparent that alternative life cycle models improve product development. It also provides insight into how we might modify the conventional life cycle model to improve our situation. For example, it is apparent by looking at all the models described above that the evolution of requirements is fundamentally ignored during development. In actuality, as important new requirements become apparent, they are reviewed by a configuration control board and either discarded (or perhaps deferred for a later release) or approved [12]. Such an approval may result in relatively expensive modifications to all existing documentation and code under development. If techniques could be developed which would make the intermediate results of the software life cycle (e.g., software requirements specifications, high-level design specifications, detailed design specifications, and code) highly adaptable, then the software development process itself could result in systems that more closely met current requirements. If such a scheme were possible, the result would be as shown in Fig. 10. Note that although the development time remains unchanged, the system that appears at time $t_1$ only meets all the requirements of time $t_0$, but most of the additional $t_1$ requirements as well.

Productivity Analysis

Suppose that at a particular point in time a project manager needed to meet a set of new requirements. He/she could select one of a number of choices:

1) modify the existing program,
2) build a new system from scratch using conventional software development practices, or
3) build a new system from scratch using any of the new alternative approaches.

Currently, many project managers make this selection based on fuzzy perceptions and past experiences. On the other hand, some project managers might analyze the project's aspects which affect the choice. For example, aspects of the application that might affect the selection include

- requirements volatility (i.e., the likelihood that the requirements will change);
- the "shape" of requirements volatility (e.g., discrete leaps, based on brand new threats; or gradual changes, as with a need to do things faster);
Comparing Software Life Cycle Models

Fig. 10. Adaptable development versus conventional.

Fig. 11. The cost dimension.

Fig. 12. Possible cost analysis for conventional development.

Fig. 13. Possible cost analysis for evolutionary prototype development.

- the longevity of the application; and
- the availability of resources to develop or effect changes (i.e., it may be easier to get resources up front than to devote significant resources for enhancements).

Wise project managers must make the appropriate life cycle model selection based not only on maximum functionality in minimum elapsed time at minimum cost, but also on the above four factors. The remainder of this section describes how the paradigm can be used as a potential tool for project managers who wish to consider all the factors in making their life cycle model selections.

Comparisons of life cycle model alternatives based solely on functionality and time can be made using the graphs shown earlier in this paper (i.e., Figs. 5-10). To add cost considerations, a third dimension can be added, as shown in Fig. 11. While the cost over time is certainly projected and tracked today, the cost per unit of functionality is not. Quantifying this three-dimensional model for various development approaches would have three significant outcomes.

1) Comparisons of different life cycle model alternatives could consider not only the functionality needs of the application, but also the funding and funding profile constraints. For example, at a particular point in time, a project manager may need to make a choice between a conventional life cycle and an evolutionary prototype life cycle. He/she could plot the projected costs on a graph like that in Fig. 11 for the two approaches. The resulting plots, shown in Figs. 12 and 13, respectively, graphically show that (for this particular project) the evolutionary prototype approach costs less up front and costs more over the product's life time, but more closely meets user needs than the conventional approach.

2) Future upgrades could be weighed in importance according to the cost impact of the additional functionality.

3) Productivity would be redefined as the functionality provided (i.e., the "bang" per DeMarco [11]) per hour of labor.

Productivity today is usually measured in terms of the characteristics of the solution (e.g., lines of code per hour). This is valid only where the selected characteristic of the solution, for example, lines of code, is a valid reflection of the size and complexity of the problem, which is generally true for manually developed software. This
measure of productivity is, however, inappropriate for development models where lines of code are not a valid reflection of the problem space. This is one of the primary reasons that comparison of the various development models is so difficult today.

**Future Research**

While the paradigm presented in this paper is useful for visualizing conceptual differences between development models, additional research is needed to allow precise tradeoffs to occur. Future research must tackle

- quantifying and measuring the functionality axis (i.e., the bang);
- developing the correct productivity metric in terms of effort per bang;
- understanding and measuring real user needs (in many cases, the gap between the user need and the operational system is an unquantified feeling of dissatisfaction on the user's part); and
- measuring the actual behavior (in terms of functionality, time, and cost) of the solution curves for the various models (i.e., generating techniques to arrive at accurate versions of Figs. 12 and 13 for any particular project).

Based on the paradigm presented in this paper, it follows that the "ultimate" model is one that is predictive of the user needs (i.e., the functionality is available before the user needs it) and exhibits the lowest possible cost. If the user need line could be projected with some degree of accuracy (which requires a mature application domain), then methods could be developed to provide functionality at or ahead of the need.

**Summary**

A paradigm has been given which helps one compare and contrast alternative software development life cycle models in terms of their abilities to satisfy user needs and reduce life cycle costs. Five "new" models were compared to the conventional model with respect to their abilities to produce software which meets user needs.

**Acknowledgement**

The authors would like to thank E. Koene for participating in early discussions on the subject of this paper.

**References**


**Edward H. Bersoff** (M'75–SM'78) received the B.S., M.S., and Ph.D. degrees in mathematics from New York University, New York.

He is President and Founder of BTG, Inc., a high-technology systems analysis and engineering firm based in the Washington, DC, area. BTG specializes in the application of modern systems engineering principles to the computer-based system development process. At BTG he has been actively involved in the FAA's Advanced Automation Program, where he is focusing on software management and software configuration management issues in this extremely complex program. He also participates in the company's activities within the national intelligence community, providing software engineering services to a wide variety of national and major military system development programs. He is also a member of the FAA's Advisory Panel, and a past member of the editorial board of IEEE Software.

**Alan M. Davis** (M'80–SM'80) received the B.S. degree in mathematics from the State University of New York, Albany, in 1970 and the M.S. and Ph.D. degrees in computer science from the University of Illinois, Champaign-Urbana, in 1973 and 1975, respectively.

He is Vice President and Director of Engineering Services at BTG, Inc., Vienna, VA, where he is responsible for all service-related aspects of BTG's business, including product assurance, SETA training, and systems analysis. Prior to that, he was Director of Research at BTG, responsible for all aspects of corporate research, and Director of Engineering Technology at BTG, responsible for all software and system engineering consulting and software and system engineering training. Prior to that, he was a Director of Research and Development for GTE Communication Systems, Phoenix, AZ, Director of the Software Technology Center at GTE Laboratories, Inc., Waltham, MA, and an Assistant Professor of Computer Science at the University of Tennessee, Knoxville.

He is the Chairman of the IEEE Computer Society's Working Group on Software Requirements Standards. He is currently an Associate Editor of Journal of Systems and Software, a member of the ACM Computing Practices Advisory Panel, and a past member of the editorial board of IEEE Software. He was an IEEE International Distinguished Lecturer in 1983 and in that role toured South America delivering lectures on software requirements specification techniques. He is a member of the Association for Computing Machinery, Sigma Xi, the AFCEA, and the AMA.
at universities in Boston, New York, and Washington, DC. His technical contributions to the fields of software requirements and design range from early publications in computer architecture, reliability, and programming languages to more recent publications on software quality and configuration management, including a textbook entitled *Software Configuration Management* (Prentice Hall).

Dr. Bersoff is a member of the AFCEA, the American Management Association, the Air Traffic Control Association, and Mensa.

Edward R. Comer (M'79) received the B.S. and M.S. degrees in mathematics and the M.S. degree in computer science from the Florida Institute of Technology, Melbourne.

He is President and Founder of Software Productivity Solutions, Inc. (SPS), Melbourne, a software technology company specializing in the development and application of advanced Ada-based methodologies and tools. He has been a primary technical contributor in the development of a number of strategic software technology plans including the research and development plans for the Software Productivity Consortium in the areas of reusable software, software prototyping, and knowledge-based systems for software development; Ada transition plans for a number of corporate clients; and SPS's internal research and development plan for Ada reusability. Prior to founding SPS, he led the Software Technology Section at Harris Government Information Systems, Melbourne. In this role he was responsible for the research, development, and transfer of advanced software techniques and tools for military software. He provided the major technical input and guidance for Harris's integrated software methodology. He was on the original planning group for the Software Technology Project of the Microelectronics and Computer Technology Corporation (MCC). He has developed a number of original techniques for systems requirements definition, Ada software design, interface specification, database design, and structured systems modeling. He also held positions at Harris in modeling and simulation and systems engineering, where he participated in the development of mission-critical systems for command and control, communications, intelligence data handling, avionics, data acquisition, and physical security. He has taught mathematics and computer science at the university level and has taught software and systems engineering courses in a variety of subjects for national seminars and for corporate and government clients.

Mr. Comer has received a Distinguished Engineering Award. He was an author of the first published *Ada Design Language Guide* in industry and participated in the IEEE Ada as a PDL Working Group. He has published numerous papers in database design, Ada design languages, software techniques and environments, communications, and simulation. He was twice the Proceedings Editor of the Annual Simulation Symposium and has served on the program committees of COMSAC '86 and the Ada Applications and Environments Conference. He is a member of the Association for Computing Machinery (SIGAda, SIGSOFT, SIGMOD, and SIGART).