Dimensions of Software Evolution

Dewayne E. Perry

Software and Systems Research Center
AT&T Bell Laboratories
Murray Hill NJ 07974
dep@research.att.com

Abstract

Software evolution is usually considered in terms of corrections, improvements and enhancements. While helpful, this approach does not take into account the fundamental dimensions of well-engineered software systems (the domains, experience, and process) and how they themselves evolve and affect the evolution of systems for which they are the context. I discuss each dimension, provide examples to illustrate its various aspects and summarize how evolution in that dimension affects system evolution. Only by taking this wholistic approach to evolution can we understand evolution and effectively manage it.

1. Introduction

We usually think about the evolution of software systems in terms of the kinds of changes that are made. While the overall motivation of evolution is adaptation, we usually partition these changes into three general classes: corrections, improvements and enhancements. Corrections tend to be fixes of coding errors, but may also range over design, architecture and requirements errors. Improvements tend to be things like increases in performance, usability, maintainability and so forth. Enhancements are new features or functions that are generally visible to the users of the system.

I think that this approach is too limiting and does not consider important sources of evolution that affect how our systems evolve. To understand software evolution properly, we need to take a wholistic view — that is, consider *everything* that is involved in well-engineered software systems. I claim that three interrelated ingredients are required to for well-(software)-engineered systems:

- the domains,
- experience, and
- process.

Moreover, these three ingredients are the sources, or dimensions, of software evolution. The critical issue is that each of these dimensions evolves, sometimes independently, sometimes synergistically with other dimensions. It is only by understanding these dimensions and how they evolve that we can reach a deep understanding of software system evolution. With this understanding, we can then manage the evolution of our systems more effectively.

In the subsequent sections I will discuss each of these dimensions in turn. While I will not give precise definitions of these dimensions, I will provide a number of examples for each to illustrate their various aspects. Finally, I will summarize what I consider to be the important lessons to be learned from each dimension about software evolution.

2. The Domains

In building and evolving software systems there are a number of domains that are pertinent: the ‘‘real world’’ which provides the context domain for our model and specification of the system, and various theoretical subdomains which provide foundational underpinnings for the system.

In the subsequent subsections I will discuss the real world, our model of the real world, the specification of the system we derive from our model, and foundational theories and algorithms. I will also discuss how these interact with each other and how each of these elements evolves and affects of the evolution of the software system.

2.1 The Real World and Its Model

The real world is of primary importance in any software system. It is, ultimately, the originating point of the system. Our first attempts to introduce software systems are such that the systems usually imitate what already exists in the real world. This imitation is the starting point from which the system evolves.

In the real world we have objects and processes. From this basis, we derive our model of the application domain for our system complete with the selected objects and their associated theories. It is this model that is the abstraction basis for our system specification which then becomes reified into an operational system [Lehman84].

Thus, we have both the real world and our model of it, with the latter obviously tied closely to the former. It is at this point that we note the following sources of evolution: changes in the real world and changes in our model.

By its very nature, our model is an abstraction of the real world. The real world provides an uncountable number of observations — that is, we can always make more observations. We use a subset of these observations as the basis for our model. Over time, we make further observations of the world and as a result often change what we consider to be relevant. These changes provide some of the stimulus to change our model.

The real world also provides a richer set of objects than we want to use for our model. To keep our model within some reasonable bounds, we must select objects in the world for inclusion in the model and, hence, leave some objects out of the model. It is these excluded objects that become bottlenecks and irritants in the operational system, and thus cause the model to change (Lehman's 6th Law [Lehman91])

The real world evolves in two important distinct ways: independently of the system and as a consequence of the system being operational in that real world. In the first case, these changes may affect our model or even affect our operational system. If our system model has an object ''bus conductor'' and its associated theory of behavior, and that object in the real world changes (that is, bus conductors behave differently), then our theory much change to mirror the change in the real world. If our system depends on parts of the real world as its context and they change (as for example hardware often does), then our system evolves as well.

The more fundamental of the two kinds of real world evolution is that set of changes which happen in the real world as a result of introducing the operational system into it. This introduction inherently perturbs the world and changes it, so that our model of the world is now out of step with the actual world [LB80]. By its very nature, our model must include a model of the system itself as part of the world. This is inherently unstable and is an intrinsic source of evolution. This closed-loop source of evolution is more interesting than the open-loop (that is, independent changes) source and more difficult to understand and manage.

The real world and our abstracted application model of the real world are fundamental sources of system evolution because they are intrinsically evolving themselves.

2.2 The Model and The Derived Specification

From our model of the real world, we use abstraction to initially derive the specification of the system we wish to build. This specification is then reified through our software development processes into an operational system which then becomes part of the real world [Lehman84]. We have discussed above how this cycle is an intrinsic source of evolution both in the real world and in our model of that world.

Given the intrinsic evolutionary nature of the model that is the source of our system specification, it will come as no surprise that the specification will evolve as well. The various objects and their behavior which evolve in the model will have to evolve in the specification.

Equally obvious is the fact that the operational system must evolve to accommodate the evolving specification, since the fundamental relationship between the operational system and the specification is one of correct implementation. It is not a matter at this point of whether the theory of the system — the specification — is right or wrong, but whether the operational system implements that theory.

The relationship between the specification and the implementation is the best understood of the various ingredients that I discuss in this paper: when the specification evolves, the system must evolve as well.

2.3 Theory

In the reification of the specification into an operational system, we appeal to a number of different theoretical

domains that are relevant either because of the domain of the real world or because of the various architectural and design domains used to reify the specification. It is likely that these underlying theories will evolve independently throughout the life of the system.

Some theories are stable — that is, they have reached a point where they are well understood and defined. For example, the language theory used as the basis of programming language parsing [HU79] is well understood and our understanding of how to use that theory is well-established [AU72]. We have parser generators of various sorts (for example `yak`) to automatically produce parsers from a language specification. Hence, we no longer worry about how to produce the front-ends of compilers.

However, some theories are not as well-established and are much less stable. For example, the theory of machine representation is not so well-understood. We do have the beginnings of such a theory and have used it as the basis of generating compiler back-ends [PQCC80]. The introduction of this kind of theory had the initial effect of evolving the standard architecture of a compiler as well as evolving the way we both think about, describe and build compilers.

More practical examples of theory stability and evolution are those of structured programming [DDH72] and process improvement [Humphrey89]. The theory of structured programming is now well understood. It has been stable for more than a decade (though, unfortunately, there still seem to be many that do not understand the theory or its practice). Still in its evolutionary phase is the theory of process improvement. We have only the beginnings of this theory and have much yet to discover and establish.

There is yet a third category of theoretical underpinnings: those aspects of the model or system for which we have only very weak theory or no theory at all. Many of the real world domains have, at best, only weak theories and many have none other than what is established in the specification. It is in this category that we experience substantial, and seemingly arbitrary, changes. We have very little to guide us and so we cast about as best we can to find suitable theories to serve in the implementation and evolution of the operational system.

Closely allied to our theories are the algorithms that perform various transformations on the domains or determine various facts about those domains. As with their attendant theories, some of these algorithms are stable. For example, we have a well established set of sorting algorithms [Knuth73] that have well-known properties so that they can be used both efficiently and appropriately in our systems. Alternatively, we have algorithms that are know to be optimal. In either case, there is no need for improvement, and hence, no need for evolution.

Analogous to theories that are still evolving, are algorithms that are evolving as well. This usually means that the complexity bounds are improving either by reducing that complexity in the worst case or in the average case [HS78].

In some cases, we have domains which are very hard and must be satisfied with algorithms that are at best approximations [HS78]. In other cases, problems are inherently undecidable as in, for example, various forms of logic. For these problems we have algorithms that may not terminate — that is, they may not find a solution. Or in the cases where have little or no theory, such as in banking, we make some approximations and see how well they work [Turski81]. In all of these cases, we constantly look for circumstances in which we can improve the performance of our algorithms or in which we can move from approximate to definitive algorithms.

Thus, independent of the real world and our specification, we have theories and algorithms which evolve and which we use to reify the specification into an operational system. The benefits of this evolution are germane to our system.

3. Experience

Of fundamental and critical importance in the enterprise of building and evolving a software system is judgment. While some aspects of the abstraction and reification process proceed from logical necessity, most of this process depends on judgment. Unfortunately, good judgment is only gained by insight into a rich set of experience.

We gain experience in a number of different ways: some through various forms of feedback, some from various forms of experimentation, and some from the accumulation of knowledge about various aspects relevant to the system. I discuss each of these forms of experience in turn.

3.1 Feedback

Feedback is, of course, one of the primary results of introducing the software system into the real world. There is an immediate response to the system from those affected by it. However, we have various other important forms of feedback as well, both internal and external, and planned and unplanned.

A major form of unplanned feedback is gotten from the modelers, specifiers and reifyers of the system. For example, in the process of evolving the model by abstracting essential objects and behaviors from the real world, there are various paths of feedback between the people evolving that model. This is the interaction typical of group design efforts. Similarly these interacting feedback loops exist when defining the specification and reifying that specification into an operational system.

At the various transformation points from one representation to another there are various paths of feedback from one group of people to another, For example, in going from the model to the specification, there is feedback about both the abstractions and the abstraction process from those trying to understand the specification. In going from the specification to the operational system, there is feedback about both the specification and the specification process.

At the various validation points of the system we have explicitly planned feedback paths. That is the purpose of the validations: to provide specific feedback about the validated portion of the system representation (model, specification, or reification).

Prior to delivering the operational system into general use, we plan carefully controlled use to provide user feedback. Typically, we control the feedback loop by limiting the number of people exposed to the system. For example, we do alpha and beta testing of the system for this reason. In both tests we limit the population to ‘‘friendly’’ users to optimize the amount of useful positive feedback — that is, feedback that will result in useful comments — and minimize the amount of negative feedback — that is, feedback that is essentially noise.

The difference between alpha and beta testing is the number of users involved. The focus of the alpha test is to removal of as many of the remaining problems as possible by means of a small population of users. The focus of the beta testing is the removal of a much smaller set of problems that usually require a much larger set of users to find. Once a certain threshold has been reach, the system is then provided to the complete set of users.

Thus, feedback provides a major source of experience about modeling, specifying and reifying software systems. Some of that feedback is immediate, some of it is delayed. In all cases, this set of feedback is one of the major source of corrections, improvements, and enhancements to the system.

Feedback also provides us with experience about the system evolution process itself. Not only do we learn facts about various artifacts in evolving the operational system, we learn facts about the methods and techniques we use in evolving those artifacts.

3.2 Experimentation

Where feedback provides information as a byproduct of our normal work, experimentation seeks to provide information by focusing on specific aspects of either the system or the process. The purpose of experimentation is to create information of the sake of understanding, insight, and judgment. The purpose of feedback is to provide corrective action. They both are concerned about understanding and corrective action, but their emphases are complimentary.

We divide experiments into three classes: scientific experiments, statistical experiments, and engineering experiments. Each has a different set of goals and each provides us with a different class of experience.

In scientific experiments we have well-designed experiments in which we have a specific set of hypotheses to test and a set of variables to control. The time and motion studies of Perry, Staudenmayer, and Votta [PSV94] and the design studies of Guindon [Guindon90] are examples of these kinds of experiments. These approaches exemplify basic experimental science. We increase our understanding by means of the experiment and generate new hypotheses because of that increased experience and understanding.

In statistical experiments, we have a set of data about which we make assumptions. We then test those assumptions by means of statistical analysis. In these cases, we are experimenting with ideas — that is, we perform conceptual experiments. Votta’s work on inspections [Votta93], Perry and Evangelist’s work on interface errors [PE85, PE87], and Lehman and Belady’s

work on evolution [LB85] are example of these kinds of experiments. We increase our knowledge by analyzing existing sets of data and extracting useful information from them.

In engineering experiments, we generally build something to see how useful it is or whether it exhibits a desired property. We often call this form of experiment ‘‘prototyping’’. In a real sense, it is a miniature version of the full evolution process or operational system, depending on whether we are experimenting with aspects of the process or the system. For example, the database community has been making effective use of this approach over the past decade or so in the realization of relational databases as practical systems. Here we have an interesting interaction between theory and experiment. Codd [Codd70] initially defined relational theory. While clean and elegant, it was the general wisdom that it would never be practical. However, a decade of engineering experimentation in storage and retrieval structures [GR93] in conjunction with advances in theories of query optimization have resulted in practical relational databases in more or less ubiquitous use today.

Thus, our various forms of experimentation provide us with focused knowledge about both software processes and software systems. The evolution of this knowledge is a source of evolution for both our software systems and our software processes.

3.3 Understanding

We have discussed a number of important ways in which we expand our knowledge by means of experience: of our knowledge of the real world and our model of it, of the supporting theoretical domains, of the system specification, of the software systems, its structure and representation, and of the software evolution process (see also the section on process below).

However, knowledge itself is valueless without understanding. While our knowledge expands, it is our understanding that evolves. It is the combination of our experience and our understanding of that experience that forms the basis of judgment and rejudgment. It is judgment that is the source of both the assumptions and the choices that we make in building and evolving our software systems. And, as our understanding evolves, we may invalidate some of those assumptions and choices.

Thus, the evolution of our understanding and judgment is a fundamental source of evolution of our software systems and processes.

4. Process

Process, in a general sense, is composed of three interrelated and interacting ingredients: methods, technologies and organizations. Methods embody the wisdom of theory and experience. Technology provides automation of various parts of the process, of process fragments. And, organizations bound, support and hinder effective processes. In some sense this a virtual decomposition, as it becomes very hard to separate organizational culture, or practices, from methods. Technology is somewhat easier to separate, though what is done manually in one organization may be automated in another.

4.1 Methods

Some of our methods find their basis in experience. For example, in Leveson’s method for designing safety critical systems [Leveson94], the principle ‘‘always fail with the systems off’’ is derived from various disasters where the failure occurred with the systems on. We learn as much from how we do things when they turn out to be wrong as when they turn out to be right.

Some of our methods are the result of theoretical concerns. For example, on the Inscape Environment [Perry89], the underlying principle is that once you have constructed the interface of a code fragment, you need not worry about the internal structure of that fragment. The interfaces have the property of referential transparency — that is, you only need to know what is reported at the interface boundaries.

A serious problem arises when trying to maintain this principle in the presence of assignment. Assignment is destructive — that is, assignment does not maintain referential transparency. Knowledge is lost when assignment occurs: the properties the assignee had before the assignment are lost after the assignment. Thus, if multiple assignments are made to the same variable, knowledge is lost that is important if that variable is visible at the code fragment interface. If multiple assignment is allowed, the fundamental principle upon Inscape rests cannot be maintained. For example, in the following case we lose some facts about the variable `a`.

```
a := b;
a := a + c;
a := a * q
```

The first statement does not cause any problems as we only have to maintain that we lose no information at the interface boundaries. Here `a` assumes a new value that would be visible at the interface. However, with the second and third statements, `a` assumes a new value and the properties of the previous assignments are lost — and so is the referential transparency that is required by Inscape.

The solution to this problem is provided by a method that requires the use of distinct variables for each assignment. Thus the previous example should use another variable name for the first and second assignment since it is the value of the third assignment that is to be seen at the interface.

```
v1 := b;
v2 := v1 + c;
a := v2 * q
```

In this way, referential transparency is preserved: there are no intermediate facts hidden from the interface that might interfere with the propagation of preconditions, postconditions or obligations in the context in which the code fragment interface is used.

Thus our methods evolve, not only as a result of experience and of theoretical considerations, but also because of technology and organizations. In any of these cases, their evolution affects how we evolve our software systems.

4.2 Technology

The tools that we use embody fragments of process within them and because of this induce some processes and inhibit others. Because of this fact, it is important that the tools and technology we use are congruent with our prescribed processes.

For example, the tools used for compiling and linking `C` programs requires that all names be resolved at linking time. This induces a particular coding and debugging process that is quite different from that possible within the Multics environment.

In the UNIX environment, the name resolution requirement means that every name referenced in a program has to have a resolvable reference for the linking process to complete, and hence for the user to be able to debug a program. That means you have to have the program completely coded, or you have to have stubs for those parts that have not been completed. Thus, while debugging incomplete programs is possible, it requires extra scaffolding that must be built and ultimately thrown away.

In the Multics environment, because segmentation faults are used to resolve name references, one may incrementally debug incomplete programs as long as the part that is does not yet exist is not referenced. This is a much more flexible and easier way to incrementally build and debug programs.

New tools and changes in the environment all cause changes in the processes by which we build and evolve our software and hence may affect the way that the software itself evolves.

4.3 Organization

Organizations provide the structure and culture within which we execute our processes and evolve our software systems. The organizational culture establishes an implicit bias towards certain classes of processes and modes of work. However, the organizational culture does not remain static, but evolves albeit relatively slowly. This evolution too affects the way we evolve our systems by changing the implicit biases, and eventually, our processes and products.

Not only do organizations establish an overall structure, they establish the structure of our projects, the structure of our processes, and, inevitably, the structure of our products. Given that there is such a direct influence on these structures, it is disturbing that organizations seem to be in such a constant state of flux. This organizational chaos can only have adverse affects on the evolution of the software system.

Someone at IBM stated that ‘‘The structure of OS360 is the structure of IBM’’ [LB85]. This is not an observation only about IBM but is true of large projects everywhere. (It is also true of the software processes used: the process structure reflects the structure of the organization.) Moreover, as a system ages, inertia sets in and the system can no longer adapt. When this happens, the system and the organization get out of step and the system can no longer adapt to needs of the organization. This happened with OS360: the system could no longer adapt to the organization and it fractured along geographical lines into VS1 and VS2 for the US and

Europe, respectively [LB85].

Not only does the way an organization evolves affect the way software systems evolve, but the way that organizations and systems interact has serious consequences for the way that a system may evolve.

5. Summary

To understand the evolution of software systems properly, we must look at the dimensions of the context in which we evolve these systems: the domains that are relevant to these system, the experience we gain from building, evolving and using these systems, and the processes we use in building and evolving these systems. Taking this wholistic view, we gain insight into sources of evolution not only of the software systems themselves, but of their software evolution processes as well.

The domains needed to build software systems are a fundamental and direct source of system evolution. They are the subject matter of the system. Changes to the domains often require corresponding changes to the software system.

- The real world intrinsically evolves as a result of introducing and evolving the software system. The context of the system in the real world also changes independently.
- The application model of the real world evolves first because it is inherently unstable (because it must contain a model of itself) and second because our assumptions and judgments about the real world change over time.
- As the model changes, the specification changes and forces changes in its reification (the operational system).
- While some of the supporting theory may be stable, many of the subdomains have either evolving theory, weak theory, or no theory at all (apart from that embodied in the model and specification). Improvements in the supporting theories offer improvements in our systems.

Experience is also a fundamental source of system evolution, not because of changes in the subject matter, but because of changes it brings to our understanding of the software system and its related domains. This experience provides an evolving basis for our judgment.

- Feedback provides us insight into the modeling, specification, and reification of the operational system. It is a major source of corrections, improvements, and enhancements.
- Scientific, statistical, and engineering experiments supply focused knowledge about various aspects of our software systems and processes. The resulting insights enable us to improve and enhance our systems
- The accumulation of knowledge by means of feedback, experimentation, and learning is of little use if it does not evolve our understanding of the system. This evolution of understanding and judgment is a critical element in the evolution of software systems.

Experience is also a major source of process evolution. It provides insight and understanding into the processes — the methods, techniques, and tools — by which we build and evolve our systems. These processes offer an indirect source of system evolution: as our process evolve they change the way we think about building and evolving software systems. This change in thinking results in changes in the systems themselves — changes in processes bring about a second order source of system evolution.

- Whether the evolution of the methods and techniques we use in building and evolving our systems are based on experience or theory, they change the way we think about and evolve our systems. They shape our perception about the system and about ways in which it may evolve.
- Tools and software development environments embody process within themselves. As in methods and techniques, they both limit and amplify the way we do things and thus the way we evolve software systems. As they evolve, the way they limit and amplify evolves as well.
- Organizations provide the contextual culture and structure for software systems and processes. While one tends to think of them as providing third order effects on evolution, they do have direct, fundamental, and pervasive effects both on the evolution of the systems and on the evolution of the processes.

These three dimensions of evolution provide a wide variety of sources of evolution for software systems.

They are interrelated in various ways and interact with each other in a number of surprising ways as well. Not only do they provide first order sources of evolution, but second and third order sources as well.

We will be able to effectively understand and manage the evolutions of our systems only when we have a deep understanding of these dimensions, the ways in which they interact with each other, and the ways in which they influence and direct system evolution.

Acknowledgements

This paper would not be possible without the foundational work of Professor Manny Lehman. Moreover, much in the current paper is a result of discussions with Manny in the context of the FEAST project.

References

[AU72] Alfred V. Aho and Jeffrey D. Ullman, *The Theory of Parsing, Translation and Compiling*, 2 Volumes, Prentice-Hall, 1972.

[Codd70] E. F. Codd, ‘‘A relational model for large shared data banks’’, *Communications of the CAM*, 13:6, pp 337-387.

[DDH72] O-J. Dahl, E. W. Dijkstra and C. A. R. Hoare, *Structured Programming*, Academic Press, 1972.

[GR93] Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kauffman, 1993.

[Guindon90] R. Guindon, ‘‘Designing the Design Process: Exploiting Opportunistic Thoughts’’, *Human-Computer Interaction, Vol 5*, 1990, pp. 305-344.

[HU79] John E. Hopcroft and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[HS78] Ellis Horowitz and Sartaj Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, 1978.

[Humphrey89] Watts S. Humphrey. *Managing the Software Process*, Addison-Wesley, 1989.

[Knuth73] Donald E. Knuth, *The Art of Computer Programming: Sorting and Searching*, Volume 3, Addison-Wesley, 1973.

[LB80] M. M. Lehman and L. A. Belady, ‘‘Programs, Life Cycles and Laws of Software Evolution’’ *Proceedings of the IEEE*, 68:9 (September 1980), pp. 1060-1076. Reprinted in [LB85].

[LB85] M. M. Lehman and L. A. Belady, *Program Evolution. Process of Software Change*, Academic Press, 1985.

[Lehman84] M. M. Lehman, ‘‘A Further Model of Coherent Programming Processes’’, *Proceedings of the Software Process Workshop*, Surrey UK, February 1984, pp 27-33.

[Lehman91] M. M. Lehman, ‘‘Software Engineering, The Software Process and their Support’’, *The Software Engineering Journal*, September 1991, pp 243-258.

[Leveson94] Nancy Leveson, *Safeware: System Safety for Computer-Based Systems*, Addison-Wesley, to appear end of 1994.

[Perry89] Dewayne E. Perry. ‘‘The Inscape Environment’’. *The Proceedings of the Eleventh International Conference on Software Engineering*, May 1989, Pittsburgh, PA.

[PE85] Dewayne E. Perry and W. Michael Evangelist. ‘‘An Empirical Study of Software Interface Errors’’, *Proceedings of the International Symposium on New Directions in Computing*, IEEE Computer Society, August 1985, Trondheim, Norway, pages 32-38.

[PE87] Dewayne E. Perry and W. Michael Evangelist. ‘‘An Empirical Study of Software Interface Faults — An Update’’, *Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences*, January 1987, Volume II, pages 113-126.

[PSV94] Dewayne E. Perry, Nancy A. Staudenmayer, and Lawrence G. Votta, ‘‘People, Organizations, and Process Improvement’’, *IEEE Software*, 11:4 (July 1994), pp 36-45.

[PQCC80] Bruce W. Leverett, Roderic G.G. Cattell, Steven O. Hobbs, Joseph M. Newcomer, Andrew H. Reiner, Bruce R. Schatz, William A. Wulf, ‘‘An Overview of the Production-Quality Compiler-Compiler Project’’, *Computer*, August 1980, pp 38-49.

[Turski81] W. M. Turski, ‘‘Specification as a theory with models in the computer world and in the real world’’, *Info Tech State of the Art Report*, 1981, 9:6, pp 363-377.

[Votta93] Lawrence G. Votta, ‘‘Does Every Inspection Need a Meeting’’, *Foundations of Software Engineering*, December 1993, Redondo Beach, CA. *ACM SIGSOFT Software Engineering Notes*, December 1993.