Dimensions of Software Evolution

Dewayne E. Perry

Professor and Motorola Regents Chair in Software Engineering
Electrical and Computer Engineering
The University of Texas at Austin

perry@ece.utexas.edu
www.ece.utexas.edu/ perry/

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. Each dimension is discussed and provided with examples to illustrate its various aspects and to summarize how evolution in that dimension affects system evolution. This wholistic approach provides a deep understanding of evolution and how evolution might be effectively managed.

1. Introduction

The evolution of software systems is usually thought of in terms of the kinds of changes that are made. While the overall motivation of evolution is adaptation, software changes are usually partitioned 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.

This approach is too limiting and does not consider important sources of evolution that affect how systems evolve. To understand software evolution properly, one must take a wholistic view — that is, one must consider *everything* that is involved in well-engineered software systems. There are 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 a deep understanding of software system evolution can be reached. With this understanding, the evolution of our systems can be managed more effectively.

In the subsequent sections each of these dimensions will be discussed in turn. While precise definitions of these dimensions will not be given, a number of examples will be provided for each to illustrate their various aspects. Finally, a summary will be given of 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 the model and specification of the system, and various theoretical subdomains which provide foundational underpinnings for the system.

In the subsequent subsections, the real world, the model of the real world, the specification of the system derived from the model, and foundational theories and algorithms will be discussed. How these interact with

each other and how each of these elements evolves and affects the evolution of the software system will be discussed.

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. The first attempts to introduce specific software systems are usually those systems that imitate what already exists in the real world. This imitation is the starting point from which the system evolves.

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

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

By its very nature, the model is an abstraction of the real world. The real world provides an uncountable number of observations — that is, one can always make more observations. A subset of these observations is used as the basis for the system model. Over time, further observations of the world are taken and as a result often changes what is considered to be relevant. These changes provide some of the stimulus to change the model.

The real world also provides a richer set of objects than is needed for the model. To keep the model within some reasonable bounds, one 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 1st Law, and as most of these kinds of changes are additive, Lehman's 6th Law as well [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 the model or even affect the operational system. If the 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 the theory much change to mirror the change in the real world. If the system depends on parts of the real world as its context and they change (as for example hardware often does), then the 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 the model of the world is now out of step with the actual world [LB80]. By its very nature, the 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 the 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 the model of the real world, one uses abstraction to initially derive the specification of the system to be build. This specification is then reified through the software development processes into an operational system which then becomes part of the real world [Lehman84]. How this cycle is an intrinsic source of evolution both in the real world and in our model of that world has been discussed above.

Given the intrinsic evolutionary nature of the model that is the source of the 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 are discussed 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, one appeals 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]. There are parser generators of various sorts (for example, `yak`) to automatically produce parsers from a language specification. Hence, one no longer worries 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. There are the beginnings of such a theory and it has been used 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. There are only the beginnings of this theory and there is much yet to discover and establish.

There is yet a third category of theoretical underpinnings: those aspects of the model or system for which there is 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 one experiences substantial, and seemingly arbitrary, changes. There is very little guidance and thus it is very difficult to find suitable theories to serve in the implementation and evolution of the operational system.

Closely allied to the various 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, there is a well established set of sorting algorithms [Knuth73] that have well-known properties so that they can be used both efficiently and appropriately where and when needed in software systems. Alternatively, there are algorithms that are known to be optimal. In either case, there is no need for improvement, and hence, no need for evolution, provided the appropriate algorithm is used.

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, there are 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 there are algorithms that may not terminate — that is, they may not find a solution. Or in the cases where there is little or no theory, such as in banking, one makes some approximations and see how well they work [Turski81]. In all of these cases, there is a constant search for circumstances in which one can improve the performance of the algorithms, in which one can find sub-cases for which there are workable algorithms, or in which one can move from approximate to definitive algorithms.

Thus, independent of the real world and the specification, there are theories and algorithms which evolve and which can be used to reify the specifications into an operational system. The benefits of this evolution are germane to the implemented 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.

One gains experience in a number of different ways: some through various forms of feedback, some from various forms of experimentation, and some with the accumulation of knowledge about various aspects relevant to the system. Each of these forms of experience are discussed in turn.

3.1 Feedback

Feedback is, of course, one of the primary results of introducing the implemented software system into the real world. There is an immediate response to the system from those affected by it. However, there are various other important forms of feedback as well, both internal and external, 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 there are explicitly planned feedback paths. That is the purpose of various forms of validation: 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, one plans carefully controlled use to provide user feedback. Typically, one controls the feedback loop by limiting the number of people exposed to the system. For example, there are alpha and beta tests of the system for this reason. In both tests, one limits the population to ‘‘friendly’’ users to optimize the amount of useful feedback — that is, feedback that will result in improvements — and minimize the amount of useless 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 remove 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 does one learn facts about various artifacts in evolving the operational system, one learns facts about the methods and techniques we use in evolving those artifacts.

3.2 Experimentation

Whereas feedback provides information as a byproduct of 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 for 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.

Experiments are divided 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 there are well-designed experiments in which one has 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], the Perpich et al work about on-line inspections [PPPVM02], and the design studies of Guindon [Guindon90] are examples of these kinds of experiments. These approaches exemplify basic experimental science. One increases ones understanding by means of the experiment and generates new hypotheses because of that increased experience and understanding.

In statistical experiments, there is a set of data about which one makes assumptions. Those assumptions are then evaluated by means of statistical analysis. In these cases, one is experimenting with ideas — that is, one performs conceptual experiments. Votta's work on inspections [Votta93], the Leszak et al work on root causes [LPS02], and Lehman and Belady's work on evolution [LB85] are examples of these kinds of experiments. Knowledge is increased by analyzing existing sets of data and extracting useful information from them.

In engineering experiments, one generally builds something to see how useful it is or whether it exhibits a desired property. This form of experiment is usually called ''prototyping''. In a real sense, it is a miniature version of the full evolution process or operational system, depending on whether one is experimenting with aspects of the process or the system. For example, the database community made effective use of this approach over about a decade or so in the realization of relational databases as practical systems. Here one finds 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, there are various forms of experimentation that provide us with focused knowledge about both software processes and software systems. The evolution of this knowledge is a source of evolution for both software systems and our software processes.

3.3 Understanding

Thus, there are a number of important ways in which to expand knowledge by means of experience: of knowledge of the real world and the 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 knowledge expands, it is understanding that evolves. It is the combination of experience and 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 one makes in building and evolving software systems. And, as understanding evolves, some of those assumptions and choices may be invalidated .

Thus, the evolution of understanding and judgment is a fundamental source of evolution of 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 processs. And, organizations bound, support or 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 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. One learns as much from doing 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, in the Inscape Environment [Perry89], the underlying principle is that once one has

constructed the interface of a code fragment, one need not worry about the internal structure of that fragment. The interfaces have the property of referential transparency — that is, one only needs 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 may be lost when assignment occurs: whatever 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 which Inscape rests cannot be maintained. For example, in the following case some facts are lost about the variable `a`.

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

The first statement does not cause any problems as one must only maintain that no information is lost 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 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 software systems are evolved.

4.2 Technology

The tools used in implementing software systems 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 used are congruent with the prescribed processes.

For example, the tools used for compiling and linking `C` programs require 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 [Organick72].

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 the program has to be completely coded, or has 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 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 one builds and evolves software and hence may affect the way that the software itself evolves.

4.3 Organization

Organizations provide the structure and culture within which processes are executed and software systems are evolved. 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 as well — albeit relatively slowly. This evolution too affects the way systems evolve by changing the implicit biases, and eventually, the processes and products.

Not only do organizations establish an overall structure, they establish the structure of the projects, the structure of the processes, and, inevitably, the structure of products [HG99]. Given that there is such a direct influence on these product 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, one must look at the dimensions of the context in which these systems evolve: the domains that are relevant to these systems, the experience gained from building, evolving and using these systems, and the processes used in building and evolving these systems. Taking this wholistic view, one gains insight into the 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 opportunities for changes to the evolving 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 the understanding of the software system and its related domains. This experience provides an evolving basis for judgment.

- Feedback provides 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 the software systems and processes. The resulting insights enable one to improve and enhance the systems.
- The accumulation of knowledge by means of feedback, experimentation, and learning is of little use if it does not evolve the 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, tools, and technologies — by which systems are built and evolved. These processes offer an indirect source of system evolution: as processes evolve they change the way one thinks 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 used in building and evolving software systems are based on experience or theory, they change the way one thinks about and evolves those systems. They shape perceptions about the system and about ways in which it may evolve.
- Tools and software development environments embody processes within themselves. As in methods and techniques, they both limit and amplify the way things are done and thus the way software systems are evolved. As tools, methods and techniques 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 direct sources of evolution, but indirect sources as well.

One will be able to effectively understand and manage the evolutions of our systems only when there is 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 ACM*, 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.

[HG99] James D. Herbsleb and Rebecca E. Grinter, ‘‘Splitting the Organization and Integrating the Code: Conway’s Law Revisited’’, *21st International Conference on Software Engineering*, Los Angeles, May 1999, ACM Press.

[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). 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.

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

[Leveson94] Nancy Leveson, *Safeware: System Safety for Computer-Based Systems*, Addison-Wesley, 1995.

[MPS02] Marek Leszak, Dewayne E. Perry and Dieter Stoll. ‘‘Classification and Evaluation of Defects in a Project Retrospective’’. *Journal od Systems and Software*, 2002:16.

[Ogranick72] Elliot I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press, 1972.

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

[PPPVW02] J.E. Perpich, D.E. Perry, A.A. Porter, L.G.Votta, and M.W. Wade, ‘‘Studies in Code INspection Interval Reductions in Large-Scale Software Development’’, *IEEE Transactions on Software Engineering*, 28:7 (July 2002).

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

[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.

[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*, 9:6 (1981).

[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.