Research Issues in Characterizing the Performance of Reusable Software Components*

Jeffrey G. Gray
Computer Science Department, Vanderbilt University, Box 1679, Station B, Nashville, TN, 37235
e-mail: jgray@vuse.vanderbilt.edu
WWW: http://www.vuse.vanderbilt.edu/~jgray

Abstract

The software reuse practices of many organizations could be described as "ad hoc," at best. Some reuse efforts do not consider even basic concepts such as completely specifying or correctly verifying reusable code. Those organizations that do attempt to specify formally their reusable assets often do so at the expense of neglecting issues concerned with performance. The area of Software Performance Engineering (SPE) provides various measures for determining the *responsiveness* of the software aspects of a computing system. Applying SPE methods to reusable software would aid future clients in assuring that performance objectives are met. This paper presents some of the basic issues associated with the need for characterizing the performance of reusable components.

I. Introduction

Historically, traditional methods of software development have focused primarily on the *functionality* of a software system. For example, throughout most life-cycle models, the successful completion of a particular stage is based upon the culmination of a deliverable which describes the functionality of the system at that phase. Verification and validation are carried out at the completion of each stage to ensure that the desired functionality is preserved as development proceeds.

Performance considerations, on the other hand, are often ignored within the general context of software development. Software engineering textbooks, such as [Pressman 90], [Sommerville 92], and [Schach 96], devote less than a handful of pages to an overview of performance concerns. Object-oriented gurus have stated that "Changes in the system architecture to improve performance should as a rule be postponed until the system is being (partly) built" [Jacobson 92, pg. 197]. Furthermore, some have forwarded "principles" of software engineering which encourage one to postpone performance issues until after the initial coding is complete; e.g., "Principle 10" in [Davis 94]. It is this attitude of viewing performance as an afterthought that has contributed to the labeling of performance analysis as being a "fix-it-later" approach [Smith 90].

Various forms of this paper were presented at:
- The Carnegie Mellon University/Software Engineering Institute *Second Annual Disciplined Engineering Workshop*, Pittsburgh, PA, June 1995.
- *OOPSLA '95: Workshop on the Design and Construction of Large-Scale Components*, Austin, TX, October 1995.

The problems of the "fix-it-later" approach are at the heart of why performance issues deem further consideration in the reuse community. The following paragraph offers a poignant, but occasionally true, scenario that is a consequence of ignoring performance issues:
During the requirements engineering phase, the client and developer determine the proposed functionality of a particular system[1]. Regrettably, the requirements document is signed off and the remaining stages of the development process are explored (i.e., specification, planning, design, implementation, and testing) without heeding the complete instructions found in the requirements document; that is, both functional *and* performance requirements. As the process moves along and when implementation and integration testing are complete, the developers then enter into product testing. It is at this point when severe problems are sometimes discovered with respect to performance. Stress and volume testing will often reveal the fact that degraded performance occurs during peak loads. After the panic has subsided, a usual response is to simply add more hardware resources to the underlying computing system (a practice similar to placing a "Band-Aid" on a fatal wound). In the event that increased hardware does not solve the problem, the entire architecture of the software system may need to be reconsidered - at an obvious expense to both time and money as well as contributing to reduced maintainability. Of course, all of these dilemmas could have been ameliorated at an earlier stage of development if only the traditional software process models focused more attention on performance issues.

There are various circumstances in which the ability to quantify the performance of a software system is important. An obvious concern is to ensure that the response time of a user query is satisfied within a "satisfactory" amount of time. Users may become disgruntled if they are forced to wait an exceedingly long amount of time for the system to respond. Such considerations are especially important for commercial vendors of software. In fact, the main motivation for the work presented in [Allen 83] was to produce a commercial product which had better performance than a competing product providing the same functionality. Other reasons for being able to quantify the performance of software have more severe implications. Real-time systems with hard deadlines need to ensure that their deadlines can be met. Failure to do so may result in loss of life or property and suggests why such systems are often described as "mission critical."

This paper is the result of an observation that the SPE community, while focusing on the performance of a new system as a whole, has overlooked the importance of reusable components in particular[2]. Likewise, it appears that the reuse community has neglected to investigate in detail the implications of non-functional issues like performance (although recently the disregard for performance documentation has been identified as a cause for the lack of widespread reuse at the component level [Pancake 95]). Therefore, the paper attempts to unite these two areas by asserting the need for an increased awareness of the performance of reusable software.

As one reuse pundit recently opined, "Like Caesar's wife, reusable components must be above reproach" [Berard 95]. Further research into the topics mentioned in this paper should help in advancing the support that reusable components offer to the software engineer who recognizes

---

[1] Fortunately, this phase of the software life-cycle does address some performance concerns This could be attributed to the role that prototypes play in arriving at a mutual agreement of the description of the desired system. A rapid prototype could capture some of the response requirements that the user will come to expect in the final product.

[2] In order to expose reuse issues to the performance community, an earlier draft of this paper served as a position statement at the recent Software Engineering Institute's (SEI) *Disciplined Engineering Workshop: Effective Practice in Performance Engineering*.

the need for examining performance concerns prior to implementation. Such support will also aid toward improving the credibility of reuse repositories.

The format of the rest of this paper is as follows. The motivation for performance characterization of reusable components is given in section II. In section III, a brief overview of the history of Software Performance Engineering is presented along with several of the techniques that have been developed. Following such an introduction, a framework for discussing the performance and architecture of reusable components is expounded upon in section IV. A short description of the importance of performance verification issues can be found in section V. As usual, the final section offers some concluding remarks.

II. Characterizing Performance Engineered Reusable Components

Within the past fifteen years, the area of software performance engineering has received much attention. The focus of study has been on analyzing/predicting *complete* systems which were built from scratch. Little work seems to have been done in investigating the benefits of characterizing the performance of reusable software. In this section, justifications are presented for some of the benefits that can be accrued by a heightened awareness of the performance of reusable software.

At the 1968 NATO conference on Software Engineering, Doug McIlroy presented what is considered to be one of the first references to a component-based view of software reuse [McIlroy 69]. Although over a quarter century has passed since this seminal paper was written, the current state of software reuse is far from reaching McIlroy's initial goals. The original concept was to construct an industry that created a library of software components that could be retrieved and classified as a "condensed document like the Sears-Roebuck catalogue." In the abstract for his paper, McIlroy states, "...yet software production in the large would be enormously helped by the availability of spectra of high quality routines, quite as mechanical design is abetted by the existence of families of structural shapes, screws, or resistors." This view implies the use of methods similar to those found in the electronic or mechanical components industry[3] (although this view has been criticized, e.g., "Myth #7" in [Tracz 95, pg. 118]). The concept was to assemble new systems by obtaining "off-the shelf" components that have already been constructed and tested. For example, one could walk into a local software shop and purchase, in either source or binary form, many of the operational components needed to create the functionality of a complete system. Furthermore, these components could be retrieved from some standard component catalog that is indexed by both functional and performance criteria.

One might argue that the general state of reuse could be found in an examination of several government funded software repositories (e.g., NASA's ELSA (formerly AdaNet), DOD's CARDS, and STARS ASSET). What a client often finds when accessing these repositories is, among other artifacts, a collection of components ranging from reusable design documents to source code. In some cases a client may find that the specification of the functionality of source components is either ambiguous or non-existent[4]. This state of affairs could point to the need for formal specification of components, a topic beyond the scope of this paper (for additional information, see [Jeng 93]). Aside from the functional documentation problems that sometimes

[3]There are efforts to revive the notion (e.g., the "Software-IC" idea proposed in [Cox 86] and [Cox 90]).
[4]This is not an attack on the efficacy of such organizations. The situation is a consequence of the fact that the outside authors who submitted the artifacts did not have these issues in mind.

exist with software repositories[5], the concept of characterizing the specific performance of components is almost always overlooked. Furthermore, practically every commercial set of reusable components fails to address hard performance concerns (although they do a better job of representing functional characteristics). For example, the famous "Booch components" [Booch 87] fail to incorporate any notion concerning the performance of software like that provided by software performance engineering.

Using the electronic components industry as an analogy, suppose that one were to walk into a local Radio Shack and ask the store clerk to "Please give me something that you think resembles a resistor." This describes the current approach adopted by many software reuse efforts. As an improvement, an electronics enthusiast would like to ask the store clerk for a component with exact functionality and specific performance (e.g., "Please give me a 20Ω resistor"). In terms of performance considerations, this type of retrieval does not exemplify current reuse efforts.

To move closer to this idealized view of software reuse, three pivotal research issues are identified:

- Performance Modeling

  Mechanisms for determining the performance of modular components need to be devised. In section III, the area of software performance engineering is presented as a foundation for this issue.

- Performance Specification Frameworks

  Section IV inspects another problem that needs to be examined: classification and retrieval frameworks which incorporate performance information. That is, what is the taxonomy used to classify components based on performance and what are some of the techniques employed to allow efficient access to these components?

- Verification of Performance Properties

  After investigating solutions for incorporating performance into the classification and retrieval problem, further research is needed in the area of performance verification. This will be presented briefly in section V as one of the last obstacles to be undertaken.

III. Overview of Software Performance Engineering

Software Performance Engineering (SPE) has been defined by many authors as a method for constructing software systems to meet performance objectives [Marciniak 94], [Smith 90]. Some define the area as an intersection between software engineering and performance evaluation [Allen 83]. The goal of SPE is to provide a method for discussing the performance requirements of a system throughout the *entire* spectrum of the software life-cycle [Smith 83]. Thus, the necessity of the "fix-it later" approach is minimized.

There are three basic evolutions that have taken place in the area of performance to bring SPE to its current state. The first evolutionary stage produced the various models developed for

[5]Ed Berard has recently stated that "All too often, software reuse repositories more closely resemble 'software landfills,' i.e., any and all software -- regardless of quality -- is dumped into the repository" [Berard 95].

performance prediction. The queuing network models (QNM) proposed in the early seventies are an example. Early work with QNMs focused on the modeling of hardware resources and provided little benefit to an overall analysis of software. A second stage of evolution involved the construction of numerous tools and algorithms to facilitate performance studies of software. Software execution models, like those described in [Beizer 84], [Booth 80], and [Smith 79], allow one to determine measurements pertaining to software. The third evolutionary advance is a synergy of QNMs and software execution models to produce system execution models. This allows for a complete analysis of software and the utilization of hardware resources. This permits a more detailed examination of systems while they are still under development.

Early work in software performance concentrated on the development of specific languages that allow one to determine the efficiency of a program without actually executing any code. For example, [Cohen 74] describes two incremental languages which resemble Algol-60 and can be used for determining efficiency. Translators for the languages allow a designer to compare different performance alternatives. Another early attempt, described in [Graham 73], provides a description of the problems associated with the "fix-it later" approach. The authors propose a single high-level language (named DES - Design and Evaluation System) that is used to describe the system at all stages of development. The programs coded in DES can be used as direct input into performance analysis and simulation routines. A common problem with these early attempts, however, can be seen in the fact that implementation details are required during early stages of development.

The bulk of research contributing to SPE appears to have begun in the late 1970s and early 1980s [Beizer 78], [Browne 83], and [Smith 79]. Most of these efforts belong in the second evolutionary stage mentioned above. A model that seems to have gained the most popularity as a representation for software execution models is the execution graph. Unfortunately, the literature describing execution graphs often provides only trivial examples of the method which makes it difficult to comprehend fully the complete process (e.g., those examples provided in [Browne 83]). The notation for execution graphs allows software modules to be represented as nodes in a graph [Smith 79] and [Smith 90]. The basic operations permitted on these graphs are concerned with iteration, branching, and sequential execution. Additional operations provide for probabilistic transfers of control, locking, and other features. After modeling the architecture of the software system using execution graphs, graph analysis and reduction algorithms are applied. After the analysis of the *software* execution model is completed, parameters are obtained that can then be fed into the next phase: the *system* execution model.

The system execution model represents the results of research performed during the third evolutionary stage. The standard representation for the system execution model has been the information processing graph (IPG). An IPG could be thought of as an intermediate representation of a QNM that combines the information obtained from the software execution model with the topology of the resources contained in the computer system. In fact, much of its notation has been derived from the commonly used notations associated with QNMs [Allen 83]. The final analysis of an IPG is often obtained by supplying the IPG as input to some of the solution packages that are commercially available [Smith 90]. For a detailed case study, which illustrates the application of software and system execution models to a real-time system, see [Smith 93].

As individuals experiment with the SPE process, the intuition and wisdom gained from the experience often becomes formalized so that others can receive benefits. A summary of seven

basic SPE principles is described in [Marciniak 94] and [Smith 90]. Likewise, [Allen 83] and [Lampson 84] offer various hints to help in the design and implementation of future systems.

The first impediment to this effort involves the additional time that is required to ascertain the performance of a component. A problem with many reuse efforts is that it is often difficult to encourage developers to specify the functionality of a component correctly, not to mention the extra effort that is needed to analyze its performance. A commitment from management is needed to ensure that incentives are in place to encourage this extra effort. The additional time spent on performance analysis may incur a large expense initially, but, as the component is made available for retrieval, this cost is amortized over each use.

Another impediment focuses on concerns with the fact that many SPE solutions produce mean-value results [Sarkar 89]. The problem with mean-value results occurs if the system has periodic problems or unusual execution characteristics [Marciniak 94]. When these problems exist, the characterization of the performance of a system may not be adequately represented. There exist, however, bounding techniques that can be used in lieu of mean-value results [Smith 90].

There are still many open issues within SPE, as noted in [Marciniak 94]. A lack of specific case studies should point to an area of future work to help in documenting explicit problems encountered when applying the SPE methods. The case studies that have been presented to date are either vague in their description (e.g., [Allen 83]) or are the results of work done on small to medium sized systems (e.g., [Nixon 93] and [Smith 93]). The scalability of SPE methods to larger systems needs to be demonstrated.

This section has served as a brief overview of the topics found in the area of SPE. For a more detailed account, the Fall 1985 edition of *CMG Transactions* contains a special issue on SPE. Additionally, the more recent and thorough account found in [Smith 90] can be consulted.

IV. A Framework for Discussing Reusable Software

Existing reuse repositories often contain thousands[6] of different components. There may even be many reusable components that implement the same functionality but have different timing and space characteristics. For example, a taxonomy similar to that proposed in [Booch 87] would allow for many different components that implement the functionality of a stack. Some stacks may be represented in a bounded form using arrays while others may be unbounded using a pointer representation. Many other different classifications could exist (based upon space/time tradeoffs) that are variants of what one would consider to be a basic LIFO stack. The majority of current classification and retrieval methods address solely the functional characteristics of a desired component. This poses a problem for the client who needs to choose among several alternative implementations (which have different levels of performance) that serve the same functionality.

To begin addressing this point, Murali Sitaraman has argued that frameworks for representing the performance requirements of reusable components must be decomposed into three layers [Sitaraman 94]; to wit:

[6]Actually, the size and granularity of repositories vary considerably. For example, the number of work products/modules examined in [Lim 94] and [Prieo-Diaz 91] are 685 and 128, respectively; on the other hand, [Lanergan 84] reports on 3,200 modules while [Selby 89] inspected thousands of components. The problems of large reuse libraries has been presented in [Biggerstaff 1994].

- implementation *independent* specifications which describe the functionality of a component

- implementation *dependent* specifications which describe the performance and other non-functional requirements

- the implementation itself

The idea has also been adapted to the constraints facility offered in the RESOLVE specification language [RESOLVE 94].

A useful model for discussing some of the topics related to software reuse can be found in the 3C model [Tracz 89]. This model, when applied to the underlying structure of a software system, can be used to illustrate the relationship among concepts (specifications), contents (implementations), and the context (environment) in which they occur. Conventions of the model state that concepts are to be represented by circles while contents are indicated by rectangles. As an illustration, the 3C model will be used to describe the possible structure of a prime number generator.

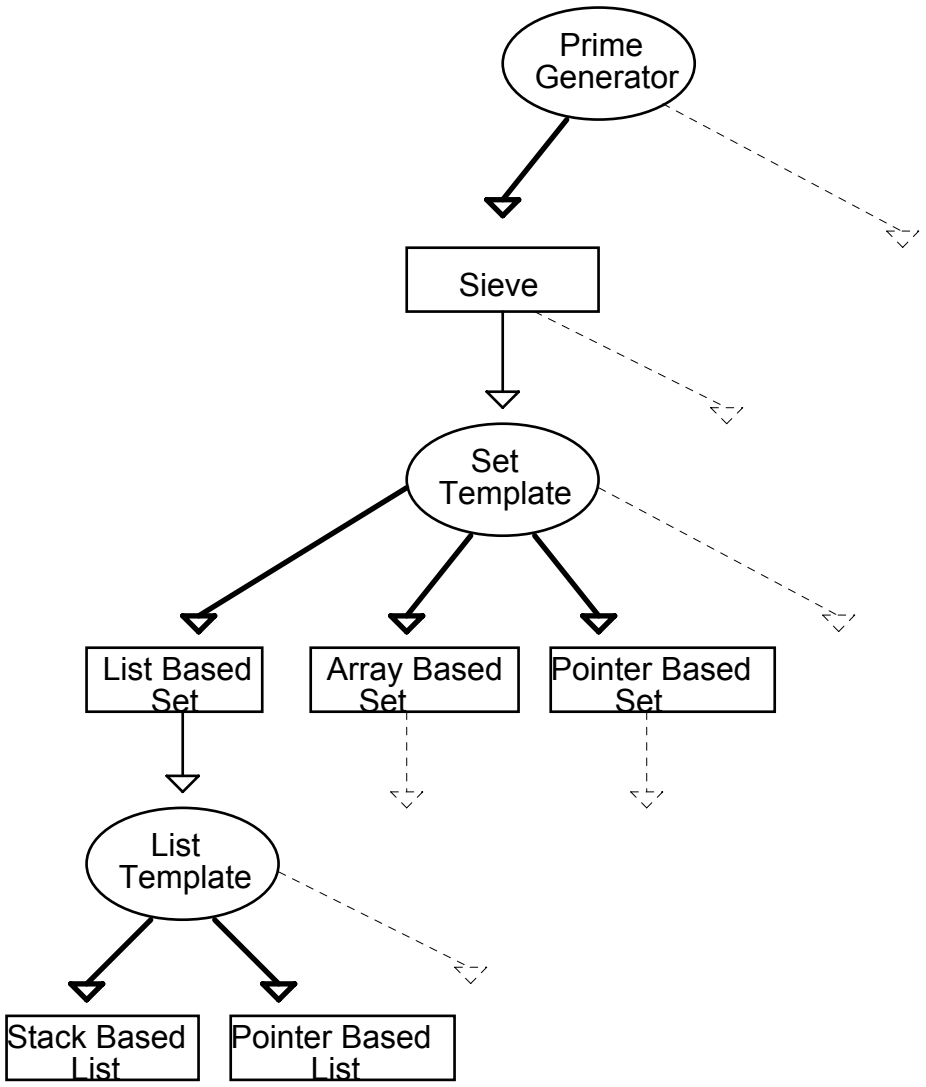
In Figure 1, the representation of a prime number generator is given which is based upon the use of sets. In the figure, an arrow from a concept to a content indicates that the content implements the concept. An arrow from a content to a concept means that the content uses the facilities provided by the concept. Dotted arrows which point to empty spaces indicate additional concepts/contents which were not included in order to keep the figure as simple as possible. For example, the content List_Based_Set can be used to implement the concept Set_Template. Likewise, the concept List_Template is used by the content List_Based_Set. It is obvious from the illustration that the model allows a concept to be realized by multiple contents, a useful attribute that is needed in the description of reusable software.

In [Krone 93], a proposed extension to the 3C model is given where additional constraints are used to indicate performance requirements. In terms of notation, "tombstones" are inserted between the arrows of the concept-content relationship[7]. The constraints contained in the tombstones are then formally specified. The overall purpose of such extensions apply to new proof rules for the verification of performance expressions. The incorporation of such extensions into the 3C model will allow the architecture of a system to be described in a way which highlights performance alternatives of multiple implementations.

V. Verification

After a client has successfully obtained a component, it is desirable to formally prove that the new component functions correctly within the overall system. Formal methods for proving performance are also needed to allow the client to certify and locally reason about the component. Recent efforts have been increasing in the area of formally verifying timing properties of software through the use of specification languages. In particular, [Krone 93] presents an approach for reasoning about the performance of reusable components and also provides a series of proof rules needed to automate the approach.

[7] Diagrammatically, this is the same as the proposed extensions in [Sitaraman 94].



The 3C Model Representation of a Prime Number Generator Using Sets
Figure 1

In related work done by Murali Sitaraman, a set of language extensions are proposed for Ada that allow a programmer to parameterize the performance of a system [Sitaraman 92]. This allows for the concept of "plugging in" reusable components.

VI. Conclusion

The overall theme of this paper has been to assert the need for an increased awareness of performance issues in the reuse community. Such an awareness will push reuse efforts closer to the original goals proposed more than twenty-five years ago with respect to an industry rivaling that found in the manufacturing and classification of electronic components.

As pointed out previously, an advantage of determining the performance of reusable components can be found in a more aggrandized retrieval facility. That is, users of software repositories can more effectively select those components that are desired based not only on the functionality of a component but also its performance. Perhaps the hardest problem in addressing this issue is the

lack of a generalized retrieval facility to be used by various users on myriad hardware platforms. In addition to containing reusable components, software repositories also contain, in some sense, *portable* components. Some clients may be retrieving components to be incorporated into a PC application while others might be obtaining components for a Sun application. Therefore, a component cannot be classified specifically by the performance that is observed on a particular development platform.

A future area of study would investigate how a *generalized* performance expression is converted into a *specific* measure once the platform of the client is ascertained. As a starting point, [Booth 80] offers some help in arriving at cost expressions that may be useful in finding generalized performance expressions. Booth first noted the distinction of separate parts to handle the functional and performance descriptions of an abstract data type (ADT). He presented the following three aspects as considerations for analyzing the performance of ADTs [Booth 80]:

- the algorithms and data representation selected to represent the data type
- the particular machine for which the software system is designed
- the statistical properties of the actual data

More recently, Opdahl and Solvberg have initiated an investigation into what they call *target platform modeling*, whereby a target model "abstracts the target platform and operating system software of a computer installation" [Opdahl 92]. Their overall model, then, consists of the high-level application model integrated with the target platform model. The notion of the term "platform," however, will need to be expanded to include not only the hardware and operating system but also the particular compiler which is used. This was pointed out by Saavedra and Smith within the context of the different performance characteristics of optimizing compilers [Saavedra 95]. Perhaps the Java language may offer some solutions to this problem [Sun 95]. Java generates an architecture-neutral object file format. The bytecodes of the Java-compiled applet are then interpreted by the specific Java run-time system for each architecture. Thus, a generalized performance characteristic can be given for each application, based upon the generated bytecodes, which can then be combined with the specific performance of each bytecode once the architecture is determined.

It could be argued that the goal of an industry which mass produces software components (to be used by clients with varying requirements), like that envisioned by McIlroy, will be hardpressed to see fruition until performance issues are further investigated. The first obstacle of analyzing performance has already been addressed by researchers in the field of SPE. Frameworks for extending the classification/retrieval problem to handle performance have just begun to be explored. After this basic framework is in place, a more mature industry can begin to tackle some of the more demanding problems that may be involved with performance verification. On the positive side, a recent proposal for a systematic methodology for creating a reusable library has identified the following as one of five core principles: "Execution times of components are included as part of the specification. A component's assumptions about its environment are specified explicitly and systematically" [Gall 95].

Acknowledgments

I would like to thank Professors Larry Dowdy and Steve Schach of Vanderbilt University for reviewing earlier drafts of this paper and providing many helpful suggestions.

References

[Allen 83] Allen, Karl S., and Allan Levy, "A Case Study in Software Performance Engineering," *Proceedings of the Fifth International Conference on Computer Capacity Management*, 1983, pp. 255-266.

[Anderson 84] Anderson, Gordon, "The Coordinated Use of Five Performance Evaluation Methodologies," *Communications of the ACM*, February 1984, pp. 119-125.

[Beizer 78] Beizer, Boris, *Micro-Analysis of Computer System Performance*, Van Nostrand Reinhold, 1978.

[Beizer 84] Beizer, Boris, "Software Performance," in C.R. Vick and C.V. Ramamoorthy, eds., *Handbook of Software Engineering*, Van Nostrand Reinhold, 1984, pp. 413-436.

[Berard 95] Berard, Ed, "Subject: Trust and Software Reuse," posting to the `comp.object` and `comp.software-eng` newsgroups, September 23, 1995. Message-ID: <4411bi$7mc@news3.digex.net>

[Biggerstaff 94] Biggerstaff, Ted, "The Library Scaling Problem and the Limits of Concrete Component Reuse," *Third International Conference on Software Reuse: Advances in Software Reusability*, IEEE Press, 1994, pp. 102-109.

[Booch 87] Booch, G., *Software Components with Ada*, Benjamin/Cummings, 1987.

[Booth 80] Booth, Taylor, and Cheryl Wiecek, "Performance Abstract Data Types as a Tool in Software Performance Analysis and Design," *IEEE Transactions on Software Engineering*, March 1980, pp. 138-151.

[Browne 83] Browne, J.C., and D.M. Neuse, "Graphical Tools for Software System Performance Engineering," *Proceedings of CMG Conference XIV*, December 1983, pp. 353-355.

[Cohen 74] Cohen, Jacques, and Carl Zuckerman, "Two Languages for Estimating Program Efficiency," *Communications of the ACM*, June 1974, pp. 301-308.

[Cox 86] Cox, Brad J., *Object-oriented Programming: An Evolutionary Approach*, Addison-Wesley, 1986.

[Cox 90] Cox, Brad J., "There is a Silver Bullet," *BYTE*, October 1990, pp. 209-218.

[Davis 94] Davis, Alan M., "Fifteen Principles of Software Engineering," *IEEE Software*, November 1994, pp. 94-101.

[Gall 95] Gall, Harald, Mehdi Jazayeri, and Rene Klosch, "Research Directions in Software Reuse: Where to go from here?" *Proceedings of the Symposium on Software Reuse '95,* Seattle, WA (special edition of ACM SIGSOFT *Software Engineering Notes,* August 1995).

[Graham 73] Graham, Robert, Gerald Clancy, and David DeVaney, "A Software Design and Evaluation System," *Communications of the ACM*, February 1973, pp. 110-116.

[Jacobson 92] Jacobson, Ivar, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard, *Object-Oriented Software Engineering - A Use case Driven Approach*, Addison-Wesley Publishing Company, Inc. 1992.

[Jeng 93] Jeng, Jun-Jang, and Betty Cheng, "Using Formal Methods to Construct a Software Component Library," *Proceedings of the Fourth European Software Engineering Conference*, LNCS 717, Springer-Verlag, 1993, pp. 397-417.

[Krone 93] Krone, Joan, and Murali Sitaraman, "A Modular System for Verification of Functionality and Performance Correctness of Software Components," Tech. Report TR 93-5, Department of Statistics and Computer Science, West Virginia University, 1993, pp. 1-28.

[Lampson 84] Lampson, B.W., "Hints for Computer System Design," *IEEE Software*, February 1984, pp. 11-28.

[Lanergan 84] Lanergan, R.G., and C.A. Grasso, "Software Engineering with Reusable Designs and Code," *IEEE Transactions on Software Engineering*, September 1984, pp. 498-501.

[Lim 94] Lim, Wayne C., "Effects of Reuse on Quality, Productivity, and Economics," *IEEE Software*, September 1994, pp. 23-30.

[Marciniak 94] "Performance Engineering," John Marciniak, ed., *The Encyclopedia of Software Engineering*, John Wiley, 1994, pp. 794-810.

[McIlroy 69] McIlroy, Doug, "Mass Produced Software Components," *Software Engineering Concepts and Techniques: Proceedings of the NATO Conferences*, J.M. Buxton, P. Naur, and B. Randell, eds., Petrocelli/Charter, 1969, pp. 88-98.

[Nixon 93] Nixon, Brian, "Dealing with Performance Requirements During the Development of Information Systems," *IEEE International Symposium on Requirements Engineering*, IEEE Press, 1993, pp. 42-49.

[Opdahl 92] Opdahl, Andreas, and Arne Solvberg, "A Framework for Performance Engineering During Information System Development," *Fourth*

International Conference on Advanced Information Systems Engineering (CAiSE), 1992, pp. 65-87.

[Pancake 95] Pancake, Cherri, "The Promise and the Cost of Object Technology: A Five Year Forecast," *Communications of the ACM*, October 1995, pp. 32-49.

[Pressman 90] Pressman, R. S., *Software Engineering: A Practitioners Approach*, McGraw-Hill, Inc., 1990.

[Prieto-Diaz 91] Prieto-Diaz, Ruben, "Implementing Faceted Classification for Software Reuse," *Communications of the ACM*, May 1991, pp. 88-97.

[RESOLVE 94] Special Feature on RESOLVE. *ACM Software Engineering Notes*, October 1994.

[Saavedra 95] Saavedra, Rafeal H., and Alan Jay Smith, "Performance Characterizations of Optimizing Compilers," *IEEE Transactions on Software Engineering*, July 1995, pp. 615-627.

[Sarkar 89] Sarkar, Vivek, "Determining Average Program Execution Times and their Variance," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1989, pp. 298-312.

[Schach 96] Schach, Stephen R., *Classical and Object-Oriented Software Engineering*, 3rd ed., Richard D. Irwin, 1996.

[Selby 89] Selby, R.W., "Quantitative Studies of Software Reuse," in *Software Reusability. Volume II: Applications and Experience*, T.J. Biggertaff and A.J. Perlis (eds.), ACM Press, 1989, pp. 213-233.

[Sitaraman 92] Sitaraman, Murali, "Performance-Parameterized Reusable Software Components," *International Journal of Software Engineering and Knowledge Engineering*, vol. 2, no. 4, 1992, pp. 567-587.

[Sitaraman 94] Sitaraman, Murali, "On Tight Performance Specification of Object-Oriented Software Components," *Third International Conference on Software Reuse: Advances in Software Reusability*, IEEE Press, 1994, pp. 149-156.

[Smith 79] Smith, Connie, and J.C. Browne, "Modeling Software Systems for Performance Predictions," *CMG Transactions*, Winter 1979, pp. 321-340.

[Smith 83] Smith, Connie, and J.C. Browne, "Performance Engineering of Software Systems: A Design-Based Approach," *Proceedings of CMG Conference XIV*, December 1983, pp. 466-467.

[Smith 90] Smith, Connie, *Performance Engineering of Software Systems*, Addison-Wesley, 1990.

[Smith 93] Smith, Connie, and Lloyd Williams, "Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives," *IEEE Transactions on Software Engineering*, July 1993, pp. 720-741.

[Sommerville 92] Sommerville, Ian, *Software Engineering*, Addison-Wesley, 1992.

[Sun 95] *The Java Language: A White Paper*, Copyright 1994, 1995 by Sun Microsystems (available at http://www.java.sun.com).

[Tracz 88] Tracz, Will, "Software Reuse Myths," *ACM SIGSOFT Software Engineering Notes*, vol. 13, no. 1, January 1988, pp. 17-21.

[Tracz 89] Tracz, Will, and Steve Edwards, "Implementation Working Group Report," *Reuse in Practice Workshop*, Pittsburgh, PA, 1989.

[Tracz 95] Tracz, Will, *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*, Addison-Wesley Publishing Company, 1995. (Note: the original list of myths appeared in [Tracz 88].)