# Enabling Reuse-Based Software Development of Large-Scale Systems

Richard W. Selby, *Member*, *IEEE*

**Abstract**—Software reuse enables developers to leverage past accomplishments and facilitates significant improvements in software productivity and quality. Software reuse catalyzes improvements in productivity by avoiding redevelopment and improvements in quality by incorporating components whose reliability has already been established. This study addresses a pivotal research issue that underlies software reuse—what factors characterize successful software reuse in large-scale systems? The research approach is to investigate, analyze, and evaluate software reuse empirically by mining software repositories from a NASA software development environment that actively reuses software. This software environment successfully follows principles of reuse-based software development in order to achieve an average reuse of 32 percent per project, which is the average amount of software either reused or modified from previous systems. We examine the repositories for 25 software systems ranging from 3,000 to 112,000 source lines from this software environment. We analyze four classes of software modules: modules reused without revision, modules reused with slight revision (< 25 percent revision), modules reused with major revision ($\geq$ 25 percent revision), and newly developed modules. We apply nonparametric statistical models to compare numerous development variables across the 2,954 software modules in the systems. We identify two categories of factors that characterize successful reuse-based software development of large-scale systems: module design factors and module implementation factors. We also evaluate the fault rates of the reused, modified, and newly developed modules. The module design factors that characterize module reuse without revision were (after normalization by size in source lines): few calls to other system modules, many calls to utility functions, few input-output parameters, few reads and writes, and many comments. The module implementation factors that characterize module reuse without revision were small size in source lines and (after normalization by size in source lines): low development effort and many assignment statements. The modules reused without revision had the fewest faults, fewest faults per source line, and lowest fault correction effort. The modules reused with major revision had the highest fault correction effort and highest fault isolation effort as well as the most changes, most changes per source line, and highest change correction effort. In conclusion, we outline future research directions that build on these software reuse ideas and strategies.

**Index Terms**—Software reuse, software measurement, software metrics, software faults, software changes, mining software repositories, large-scale systems, experimentation, empirical study.

## 1 Introduction

Software reuse enables developers to leverage past accomplishments and facilitates significant improvements in software productivity and quality. There are several motivations for desiring software reuse, including gains in productivity by avoiding redevelopment and gains in quality by incorporating components whose reliability has already been established. Reuse-based software development emphasizes strategies, techniques, and principles that enable developers to create new systems effectively using previously developed architectures and components. Many research approaches contribute to advances in reuse-based software development, including creation of new reuse frameworks, processes, architectures, tools, and environments as well as formulation of new reuse ideas and concepts. This paper contributes to software reuse by addressing a pivotal research issue that underlies many software development approaches—what factors characterize successful software reuse in large-scale systems?

This study investigates a software development organization that successfully exemplifies reuse-based software development of large-scale systems. The research approach is to investigate, analyze, and evaluate software reuse empirically by mining software repositories from a NASA software development environment that actively reuses software. This software environment successfully follows principles of reuse-based software development in order to achieve an average reuse of 32 percent per project, which is the average amount of software either reused or modified from previous systems in this environment (Fig. 1). We identify two categories of factors that characterize successful reuse-based software development of large-scale systems: module design factors and module implementation factors. We also evaluate the fault rates of the reused, modified, and newly developed modules.

Given the attractive payoff of reusing software, there have been several efforts undertaken to discuss the topic of reusability (e.g., [17], [18], [19], [20], [45], [54], [57], [58], [59], [60], [61], [62], [67], [68]), including overviews of software reusability research directions [22], [47] and software reusability in practice [21], [23], [24], [25], [41], [43], [46], [51], [62]. Developers are adopting many of these reuse approaches, including reuse in product lines [57], [58], [61], [67], [68], design patterns [69], [70], [71], templates [60],

• *The author is with Northrop Grumman Space Technology, One Space Park, Redondo Beach, CA 90278. E-mail: rick.selby@ngc.com.*

*Manuscript received 23 Oct. 2004; revised 7 Jan. 2005; accepted 14 Jan. 2005; published online 29 June 2005.*
*Recommended for acceptance by A. Hassan and R. Holt.*
*For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-0232-1004.*

| Project ID# | Percentage (%) of modules** | | | | Number of modules |
|---|---|---|---|---|---|
| | New | Major revision | Slight revision | Complete reuse | |
| P1 | 90.07 | . | 6.51 | 3.42 | 292 |
| P2 | 68.47 | 8.24 | 6.25 | 17.05 | 352 |
| P3 | 63.76 | 4.62 | 24.27 | 7.35 | 585 |
| P4 | 18.12 | 6.12 | 7.29 | 68.47 | 425 |
| P5 | 67.92 | 1.10 | 5.01 | 25.98 | 639 |
| P6 | 66.51 | 12.56 | 12.87 | 8.06 | 645 |
| P7 | 67.06 | 6.80 | 12.90 | 13.25 | 853 |
| P8 | 98.56 | 0.48 | . | 0.96 | 418 |
| P9 | 60.87 | . | 13.04 | 26.09 | 23 |
| P10 | 92.86 | . | 2.38 | 4.76 | 42 |
| P11 | 54.46 | 6.93 | 14.85 | 23.76 | 101 |
| P12 | 91.15 | 2.65 | . | 6.19 | 113 |
| P13 | 79.73 | 5.41 | 12.16 | 2.70 | 74 |
| P14 | 48.51 | 8.01 | 16.48 | 27.00 | 437 |
| P15 | 55.41 | 5.18 | 15.09 | 24.32 | 444 |
| P16 | 75.59 | 0.39 | . | 24.02 | 254 |
| P17 | 76.57 | 3.53 | 15.37 | 4.53 | 397 |
| P18 | 91.37 | 2.40 | 6.24 | . | 417 |
| P19 | 59.39 | 0.61 | 16.36 | 23.64 | 165 |
| P20 | 66.20 | . | . | 33.80 | 71 |
| P21 | 27.27 | . | . | 72.73 | 22 |
| P22 | 87.40 | . | . | 12.60 | 246 |
| P23 | 100.00 | . | . | . | 28 |
| P24 | 77.89 | . | 2.11 | 20.00 | 95 |
| P25 | 38.00 | 2.00 | 32.00 | 28.00 | 50 |
| All | 68.07 | 4.58 | 10.27 | 17.08 | 7188 |
| Note: ** A period (.) means 0%. | | | | | |

Fig. 1. Origin of modules by individual project.

reference architectures [60], [64], [65], and advanced searching, matching, and modeling tools [63], [66]. The reuse approach in the NASA environment that we examine in this paper corresponds to a hybrid reuse approach: mature reference architectures coupled with broad populations of context-dependent component implementations. This environment's reuse process selects reference architectures and populates them with components that have simple, clearly defined interfaces. This process benefits from the maturity of the problem domain and the multi-project planning horizon of project managers and architects. Many other reuse approaches, such as product lines, design patterns, and context-independent techniques, address reuse in different ways and have also demonstrated benefits [57], [61], [67], [68], [69], [70], [71]. Even though these other approaches are not the focus of this analysis, the results from this study may yield helpful insights for those techniques as well.

Past research has defined software reuse broadly and has identified several assets associated with software as candidates for reuse: reuse of processes by which software is created and manipulated [1], reuse of technical personnel across projects [2], reuse of design objects [3], [49], [55], reuse of design histories [4], reuse of generic procedures and views [50], reuse of functions [48], and reuse of subroutine implementations [5], among others. Proposed infrastructure for reuse of software assets has been embodied in various approaches [6], [42], [44], [52], [53]. One reuse approach has been through software generation, such as report generators, complier-compilers, and language-based editors (e.g., [7], [8]). Another approach has been through the use of object-oriented programming languages, such as Smalltalk-80 [9], Flavors [10], Loops [11], CommonLoops [12], Ceyx [13], C++ [14], Eiffel [2], Object Pascal [15], and Simula [16]. A third approach has been through the use of subroutine libraries or catalogs (e.g., [5]). Some combined approaches have also been proposed, such as the MELD system that is intended to integrate the advantages of object-oriented programming and software generation [6].

This study intends to build on this past work by complementing it with empirical data from a large-scale systems development environment. We examine the repositories for 25 software systems ranging from 3,000 to 112,000 source lines from a NASA software development environment. We analyze four classes of software modules: modules reused without revision, modules reused with slight revision (< 25 percent revision), modules reused with major revision ($\geq$ 25 percent revision), and newly developed modules. We apply nonparametric statistical models to compare numerous development variables across the 2,954 software modules in the systems. The research goals for this study are to:

- characterize software reuse at the project level,
- characterize software reuse at the module design level,
- characterize software reuse at the module implementation level, and
- characterize software reuse and module faults and changes.

Section 2 defines the objectives for this study, Section 3 summarizes the NASA software environment, and Section 4 describes the data collection and analysis methods. Section 5 characterizes software reuse at the project level, and Sections 6 through 9 characterize different perspectives of software reuse at the module level. Section 10 presents the interpretations and conclusions.

## 2 Objectives

This study addresses a pivotal research issue that underlies software reuse—what factors characterize successful software reuse in large-scale systems? We apply the goal-question-metric paradigm to define the specific goals and metrics for this research [56].

**Goal 1: Characterize Software Reuse at the Project Level**

- How does the distribution of reused, modified, and newly developed modules compare across projects? What are the differences in this distribution between small and large projects?
  - Metrics: Percentage of modules reused without revision, modules reused with slight revision (< 25 percent revision), modules reused with major revision (≥ 25 percent revision), and newly developed modules; Total modules.
- How does the amount of module reuse across projects compare with module development effort and module fault rate?
  - Metrics: Percentage of modules reused without revision, modules reused with slight revision, and modules reused with major revision; Module development effort; Module fault rate.

**Goal 2: Characterize Software Reuse at the Module Design Level**

- How do reused, modified, and newly developed modules compare in terms of interfaces that a module has with other system modules?
  - Metric: Module calls per source line.
- How do reused, modified, and newly developed modules compare in terms of interfaces that a module has with utility functions?
  - Metric: Utility function calls per source line.
- How do reused, modified, and newly developed modules compare in terms of interfaces that other system modules have with a module?
  - Metric: Input-output parameters per source line.
- How do reused, modified, and newly developed modules compare in terms of interfaces that a module has with human users?
  - Metric: Read and write statements per source line.
- How do reused, modified, and newly developed modules compare in terms of documentation describing the functionality of a module?
  - Metric: Comments per source line.
- How do reused, modified, and newly developed modules compare in terms of effort spent in designing a module?
  - Metric: Percentage of development effort spent in design.

**Goal 3: Characterize Software Reuse at the Module Implementation Level**

- How do reused, modified, and newly developed modules compare in terms of size of a module?
  - Metric: Source lines of code.
- How do reused, modified, and newly developed modules compare in terms of development effort of a module?
  - Metric: Development effort per source line.
- How do reused, modified, and newly developed modules compare in terms of control flow structure in a module?
  - Metric: Cyclomatic complexity per source line.
- How do reused, modified, and newly developed modules compare in terms of assignment statements ("noncontrol flow structure") in a module?
  - Metric: Assignment statements per source line.

**Goal 4: Characterize Software Reuse and Module Faults and Changes**

- How do reused, modified, and newly developed modules compare in terms of module faults?
  - Metrics: Faults; Faults per source line.
- How do reused, modified, and newly developed modules compare in terms of module fault correction effort?
  - Metrics: Fault correction effort; Fault isolation effort.
- How do reused, modified, and newly developed modules compare in terms of module changes?
  - Metrics: Changes; Changes per source line.
- How do reused, modified, and newly developed modules compare in terms of module change correction effort?
  - Metric: Change correction effort.

For the above goals, questions, and metrics, the experimental hypotheses are that there are no differences in the characterizations of software due to modules reused without revision, modules reused with slight revision, modules reused with major revision, and newly developed modules.

## 3 The Software Environment

Twenty-five software projects have been analyzed from a NASA software development environment for this study [26], [27]. The software is ground support software for unmanned spacecraft control. These projects range in size from 3,000 to 112,000 lines of Fortran source code. They

took between 5 and 140 person-months to develop over a period of 5 to 25 months. The staff size ranged from 4 to 23 persons per project. There are 22 to 853 "modules" in each project, where the term "modules" is used to refer to the subroutines, utility functions, main programs, macros, and block data in the systems. Fig. 1 characterizes the projects according to size and distribution of module origin.

The projects vary in functionality, but the overall problem domain has an established set of algorithms and processing methods. The developers are convinced of the payoffs from reuse and have created a successful environment for reuse-based software development. When working on a new project, they select one of their reference architectures to meet the needs of the system requirements. They populate the architecture by utilizing their repositories to determine which existing modules can be reused or modified. For new capabilities, they will create new modules and ensure that the new modules will be designed for reusability on future projects.

## 4 Data Collection and Analysis Method

A variety of information was collected about each of the software projects and their constituent modules. This section describes the data collection, validation, and analysis approach as well as defines the specific metrics used in the analysis.

### 4.1 Data Collection and Validation Approach

First, we outlined the metrics needed for this research based on the research goals and questions defined in Section 2. Second, we collected the metric data from mining repositories that support the software environment at NASA Goddard Space Flight Center in Greenbelt, Maryland. Software personnel at the NASA environment record information about their development processes and products into these repositories on an ongoing basis using a set of data collection forms and tools [26]. Data collection occurs during project startup, continues on a daily and weekly basis throughout development, captures project completion information, and incorporates any postdelivery changes. For example, effort, fault, and change data are collected using manual forms while source code versions are analyzed automatically using configuration management and static analysis tools. NASA personnel have used these data collection forms and tools continually since the 1970s, and the data included in this study include projects from the 1970s through the 1990s. Third, the data were validated through a series of steps including extensive training, interviews, independent crosschecks, tool instrumentation, and reviews of results by objective internal and external personnel. Fourth, the validated data were organized into a relational database for investigation, analysis, and evaluation by NASA personnel as well as outside researchers [28], [29], [40]. Fifth, we developed queries and custom analysis programs to extract the relevant data from the databases for analysis in this study.

### 4.2 Software Metric Definitions

The research goals and questions defined in Section 2 identified the following metrics for use in this analysis. The term "modules" is used to refer to the subroutines, utility functions, main programs, macros, and block data in the systems.

#### 4.2.1 Software Reuse Metrics

The NASA software developers classified each module into one of four mutually exclusive and exhaustive categories based on its degree of reuse or modification from previous projects. The four categories are:

- Modules reused without revision—These modules were completely developed for a previous project, and they were selected for incorporation into a new project. Some effort may be spent on the modules during development of a new project, such as for design reviews, integration testing, or documentation updates. No functionality revisions were made to them except possibly fault corrections.
- Modules reused with slight revision—These modules were completely developed for a previous project, and they were selected for incorporation into a new project. Functionality revisions were made to the modules for the new project, and less than 25 percent of the source lines of code were revised.
- Modules reused with major revision—Same as "modules that are reused with slight revision," except that greater than or equal to 25 percent of the source lines of code were revised.
- Newly developed modules—Modules that are newly created for a new project.

#### 4.2.2 Project Metrics

The project metrics aggregate various module level metrics into project level summaries, including the following:

- Total modules—The total number of modules in a project.
- Percentage of modules reused without revision—Modules reused without revision divided by total modules for a project.
- Percentage of modules reused with slight revision—Modules reused with slight revision divided by total modules for a project.
- Percentage of modules reused with major revision—Modules reused with major revision divided by total modules for a project.
- Percentage of newly developed modules—Newly developed modules divided by total modules for a project.
- Average module development effort—Average of the module development effort for all modules on a project.
- Average module fault rate—Average of the module fault rate for all modules on a project, where the "module fault rate" is the number of faults in a module.

#### 4.2.3 Module Design Metrics

Most of the module design metrics were collected using configuration management tools and a static analysis tool called SAP that can be executed against design-level pseudocode or complete source code. NASA software developers collected the design effort metrics using manual data collection forms on a daily and weekly basis during the projects. The module design metrics are:

- Module calls per source line—The number of procedure and function calls to other modules, excluding calls to utility functions, divided by the number of source lines of code.
- Utility function calls per source line—The number of function calls to utility functions, divided by the number of source lines of code.
- Input-output parameters per source line—The number of input and output parameters including any global data referenced, divided by the number of source lines of code.
- Read and write statements per source line—The number of read and write statements, divided by the number of source lines of code.
- Comments per source line—The number of comments, divided by the number of source lines of code.
- Percentage of development effort spent in design—The effort spent during module design activities divided by total module development effort. Any modules that were reused without revision and had zero module development effort are excluded from this metric calculation.

#### 4.2.4 Module Implementation Metrics

Analogous to the module design metrics, most of the module implementation metrics were collected using configuration management tools and a static analysis tool called SAP that is executed against complete source code implementations. NASA software developers collected the development effort metrics using manual data collection forms on a daily and weekly basis during the projects. The module implementation metrics are:

- Source lines of code—The number of source lines of code in the implementation.
- Development effort per source line—The development effort from design specification through acceptance testing in tenths of hours, divided by the number of source lines of code. The development effort includes all design, code, and test effort as well as fault correction effort and change correction effort. Any postdelivery effort is also included. For modules reused with no, slight, or major revision, the development effort includes only the effort expended on the new project; the effort required to create the original module is not included.
- Cyclomatic complexity per source line—The cyclomatic complexity of the implementation, divided by the number of source lines of code.
- Assignment statements per source line—The number of assignment statements, divided by the number of source lines of code.

#### 4.2.5 Module Fault and Change Metrics

NASA software developers collected the fault and change metrics using manual data collection forms on a continuous basis during the projects. After a module had been released into configuration management, a formal modification request form needed to be completed for modifications of any type. Based on information in the project software repositories, module modifications were categorized into the following types: error correction; planned enhancement; implement requirements change; improve clarity; improve user service, insertion or deletion of debug code, optimization of time, space, or accuracy; or adaptation to environmental change. A modification of type "error correction" is called an error in this study. An error may affect more than one module. Each module affected by an error is counted as having a "fault." For example, if an error affected three modules, each module would be counted as having one fault. For each module, we then totaled their faults. A nonerror modification is simply called a modification. Each module affected by a nonerror modification is counted as having a "change." In the counting method, errors are analogous to modifications and faults are analogous to changes. NASA software developers also collected fault correction effort and change correction effort. Fault correction effort was further divided to be equal to the sum of fault isolation effort and fault implementation effort. The module fault and change metrics are:

- Faults—The number of faults.
- Faults per source line—The number of faults, divided by the number of source lines of code.
- Fault correction effort—The fault correction effort in tenths of hours.
- Fault isolation effort—The fault isolation effort in tenths of hours.
- Changes—The number of changes.
- Changes per source line—The number of changes, divided by the number of source lines of code.
- Change correction effort—The change correction effort in tenths of hours.

### 4.3 Data Analysis Approach

The preliminary analysis of the empirical data was done with scatter plots and histograms. The preliminary analysis showed that several of the dependent variables, such as development effort per source line and faults per source line, were not normally distributed in this environment, which is consistent with earlier studies [30], [31]. Therefore, the primary method for further analysis was through nonparametric analysis of variance (ANOVA) models using ranked data [32]. ANOVA models enable the assessment of the potential contributions of a wide range of factors simultaneously. Such models also enable the interactions of the factors to be detected, not only their individual contributions. The specific ANOVA factors and levels analyzed are outlined below.

### 4.4 Project Data Analysis

For the analysis of the project level data, the factor of "project size" was analyzed in a nonparametric ANOVA model. The classification levels given below (e.g., "larger"

versus "smaller" project) are relative and specific for this particular environment. This factor was selected because studies have indicated the importance of project size in the analysis of software development data [33], [34], [35], [36], [37]. The dependent variables evaluated in the ANOVA model were the project metrics defined earlier in Section 4.2.2. The project ANOVA factor and levels were:

- Project size
  - Larger (greater than 20,000 source lines),
  - Smaller (less than or equal to 20,000 source lines).

### 4.5 Module Design, Implementation, Fault, and Change Data Analysis

For the analysis of the module-level data for designs, implementations, faults, and changes, the factors of "module origin," "module size," "project containing the module," and "interaction of module origin with module size" were analyzed in a nonparametric ANOVA model. As stated previously, the classification levels given below are relative and specific for this particular environment. These factors were selected because the focus of this analysis is on module origin and since studies have indicated the importance of module size and individual project differences in the analysis of software development data [33], [34], [35], [36], [37], [28]. The dependent variables evaluated in the ANOVA model were the module metrics defined earlier in Sections 4.2.3, 4.2.4, and 4.2.5. The module ANOVA factors and levels were:

- Module origin
  - Complete reuse without revision,
  - Reuse with slight revision (< 25 percent revision),
  - Reuse with major revision ($\geq$ 25 percent revision),
  - Complete new development.
- Module size
  - Larger (greater than 140 source lines),
  - Smaller (less than or equal to 140 source lines).
- Project containing the module
  - One level for each of the individual projects,
- Interaction of module origin with module size.

The results presented in this paper focus on the statistically significant differences in the dependent variables due to the effect of module origin.

## 5 Characterizing Software Reuse at the Project Level

This section focuses on the characterization of several aspects of software reuse at the project level:

- Distribution of reused, modified, and newly developed modules; and
- Module reuse, module development effort, and module fault rate.

### 5.1 Distribution of Reused, Modified, and Newly Developed Modules

Fig. 1 presents the distribution of module origin according to project size and individual project. Approximately one-third (31.9 percent) of the modules in this environment were either reused or modified from previous projects. Of the 7,188 total modules, 17.1 percent were reused without revision from previous systems, 10.3 percent were reused with slight revision, and 4.6 percent were reused with major revision. The amount of reuse varied across the projects: in project P4 81.9 percent of the modules were either reused or modified, while in project P23 0 percent of the modules were either reused or modified. Fig. 2 graphically depicts the percentage of modules reused or modified in the projects.

The project-level analysis of variance model described earlier was applied to evaluate any statistically significant differences in the module reuse across the projects. Projects of larger size had a higher percentage of modules that were reused with major revision ($\alpha < .02$). However, project size had no statistically significant effect on the percentages of module reuse without revision or module reuse with slight revision ($\alpha > .05$). Fig. 3 displays the relationship between project size, in terms of number of modules, and the amount of reuse.

### 5.2 Module Reuse, Module Development Effort, and Module Fault Rate

Fig. 4 displays the relationship between the amount of reuse in a project, in terms of the percentage of modules reused with no, slight, or major revision, and module development effort. The projects with more reuse did not require higher module development effort in most cases—in general, the trend is level or decreasing average module development effort with more reuse. Fig. 5 displays the relationship between the amount of reuse in a project, in terms of the percentage of modules reused with no, slight, or major revision, and module fault rate. There is a cluster of four projects with approximately 50 percent reuse that had higher module fault rates.

The data analysis at the project level does not seem to differentiate cleanly among the projects. This may be due to the variation in functionality and development factors that tends to occur across individual projects. For example, we did preliminary analysis of an additional factor, project start date, and it also did not indicate a statistically significant effect on the amount of reuse ($\alpha > .05$). The next steps in the data analysis characterize software reuse at finer levels of granularity: the module design level and the module implementation level.

## 6 Analyzing Module Level Data

Of the 7,188 modules appearing in Fig. 1, 2,954 modules were Fortran subroutines with complete data collected on their development. The remainder of the modules were either: 1) not Fortran subroutines (e.g., they were assembler macros, utility functions, main programs, or block data); 2) Fortran subroutines with incomplete development effort

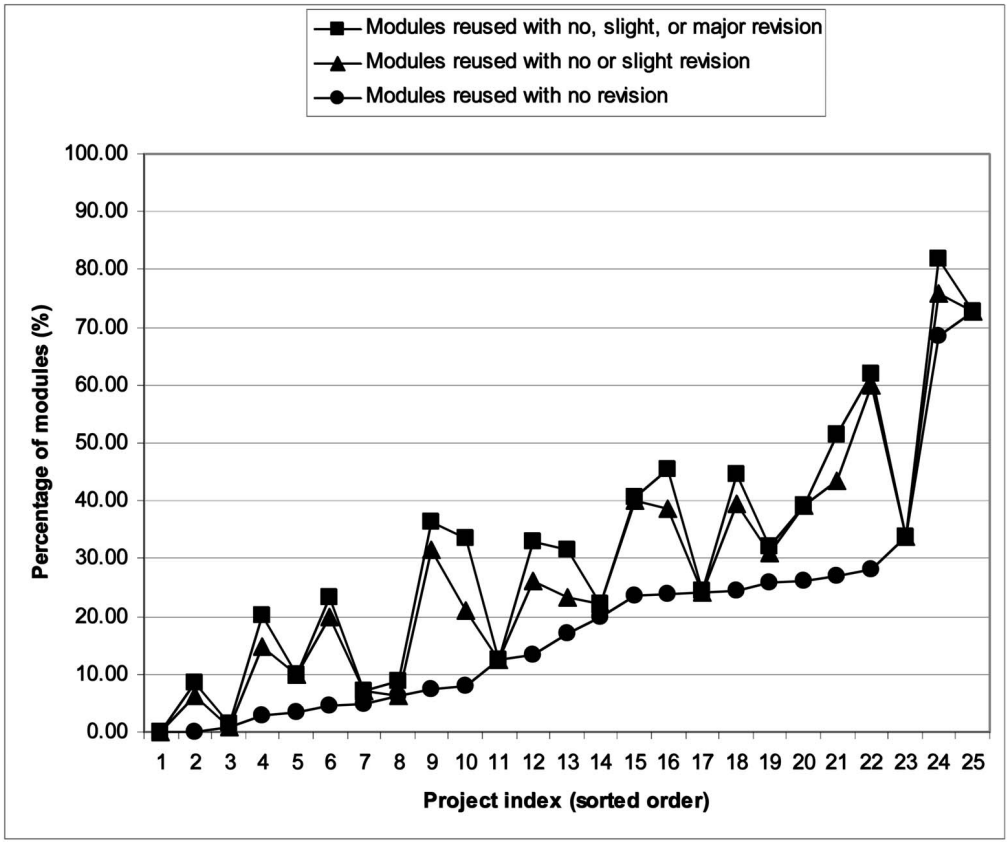



Fig. 2. Origin of modules in the 25 projects. Projects are sorted according to the percentage of their modules reused without revision. There is one index for each project: the higher the percentage of modules reused without revision, the higher the index. (These project indexes have no relation to the project ID#s in Fig. 1.)

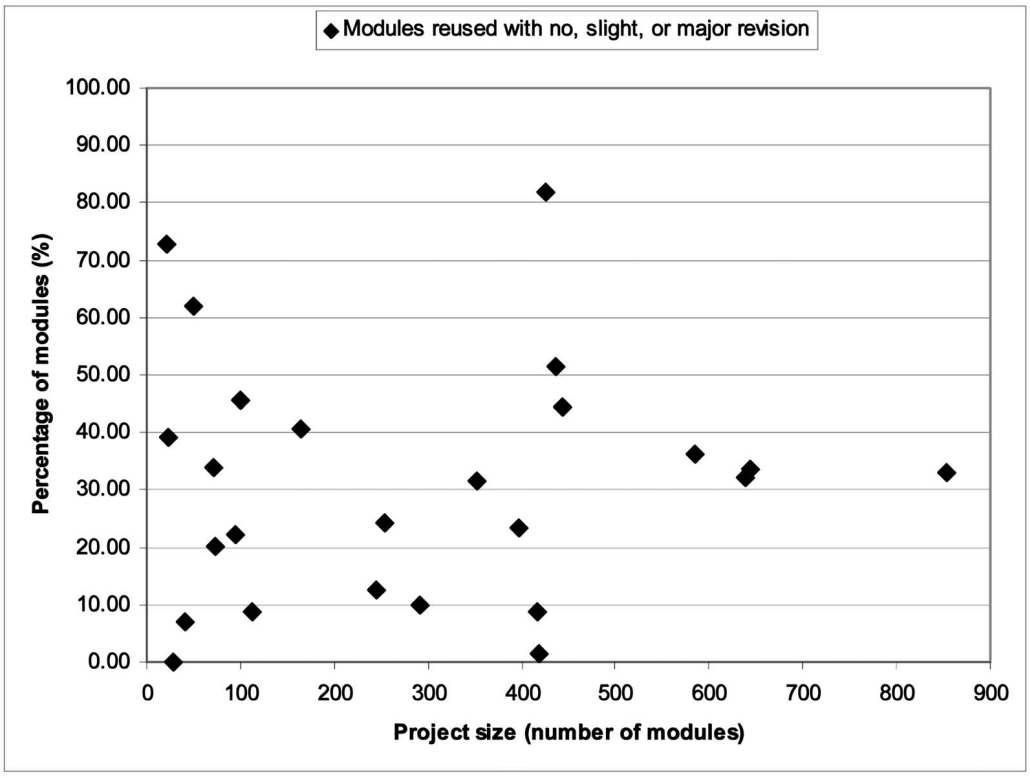



Fig. 3. Project size versus the percentage of project modules that were reused with either no, slight, or major revision.

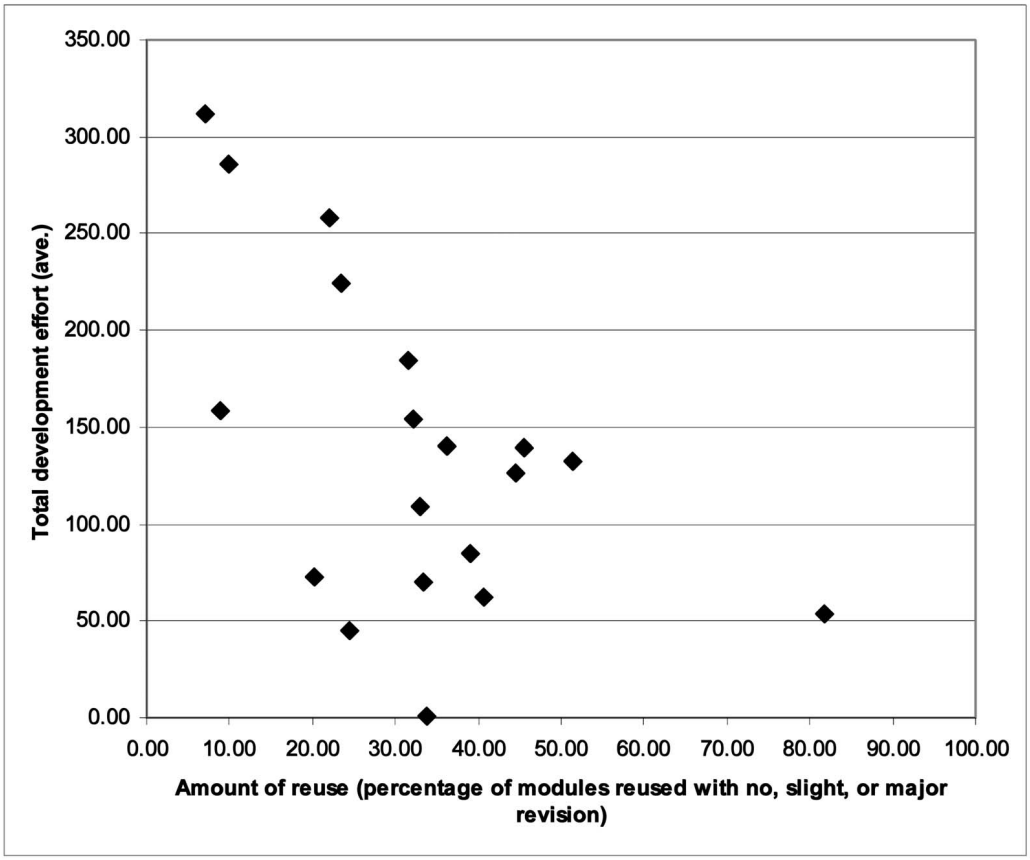



Fig. 4. Amount of reuse in a project versus module development effort (in tenths of hours).

data; or 3) Fortran subroutines that were not analyzed using the same SAP static analysis tool mentioned earlier. The distribution of the 2,954 modules according to module origin is given in Fig. 6. The overall distribution profile is similar to the one in Fig. 1. The module-level analysis in this section and later sections is based on applying the module-level analysis of variance model described earlier to data from these 2,954 modules. All comparisons among module origins cited in this analysis were evaluated using Tukey's multiple comparisons test [38]. In general, the discussion focuses on only those pairwise comparisons of module origins that were statistically significant.

Studies in general have indicated that module attributes, such as development effort and various static measures, tend to correlate with module size [35]. In particular, an earlier study showed this relationship to be true for data from this environment [28]. Therefore, most aspects of this analysis use only module attributes that have been normalized by the final module implementation size measured in source lines.

## 7 Characterizing Software Reuse at the Module Design Level

Several researchers have advocated the merits of software reuse at the design level (e.g., [6], [22]). Software design information tends to be applicable across a variety of problems, while specific implementations may tend to embody information customized to individual circumstances. This section focuses on the characterization of software reuse at the module design level. Several aspects of software design are considered:

- interfaces that a module has with other system modules,
- interfaces that a module has with utility functions,
- interfaces that other system modules have with a module,
- interfaces that a module has with human users,
- documentation describing the functionality of a module, and
- effort spent in designing a module.

### 7.1 Module's Interfaces with Other Modules and Utility Functions

Two interpretations are considered for capturing the interfaces between a given module and other modules. The first interpretation is the number of calls that a module makes to other system modules, where utility functions are not counted since they are relatively low level. The second interpretation is the total number of calls that a module makes just to utility functions. These two views are intended to capture the amount of potential interaction that a module has with other system modules and with support modules, respectively. These are two forms of module "fan-out." Figs. 7a and 7b present these two interpretations. Modules with major revision had the most

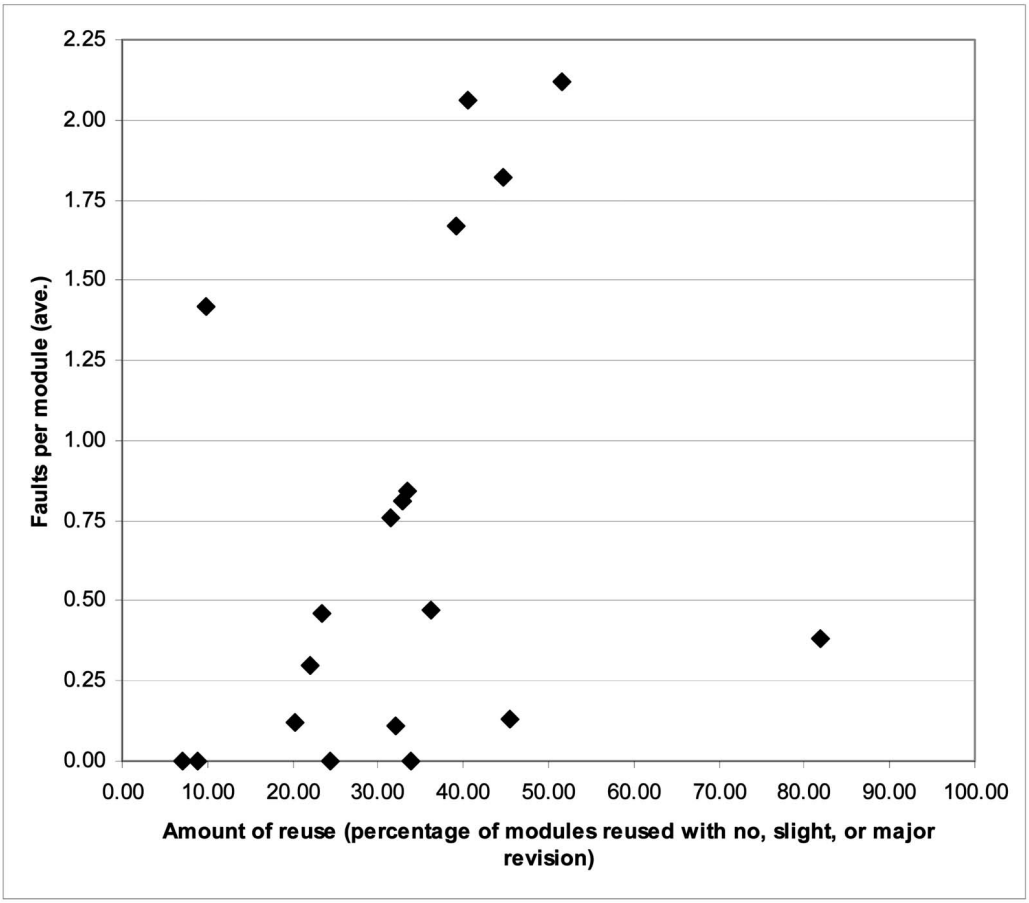



Fig. 5. Amount of reuse in a project versus module fault rate.

| | Module origin | | | | |
|---|---|---|---|---|---|
| Distribution of | New | Major revision | Slight revision | Complete reuse | All |
| Number | 1629 | 205 | 300 | 820 | 2954 |
| Percentage | 55.15 | 6.94 | 10.16 | 27.76 | 100.00 |

Fig. 6. Origin of modules for those Fortran subroutines with complete data collected on their development.

| Module attribute | | | Class of software modules | | | | |
|---|---|---|---|---|---|---|---|
| Factor | Metric | Statistic | New development | Major revision | Slight revision | Complete reuse | All modules |
| a) Interface with other system modules | Module calls per source line | Mean | 0.0315 | 0.0353 | 0.0307 | 0.0186 | 0.0281 |
| | | Std. dev. | 0.0329 | 0.0303 | 0.0384 | 0.0375 | 0.0352 |
| | | Median | 0.0246 | 0.0283 | 0.0204 | 0.0000 | 0.0195 |
| b) Interface with utility functions | Utility function calls per source line | Mean | 0.0146 | 0.0142 | 0.0144 | 0.0328 | 0.0196 |
| | | Std. dev. | 0.0317 | 0.0230 | 0.0266 | 0.0810 | 0.0505 |
| | | Median | 0.0000 | 0.0040 | 0.0000 | 0.0055 | 0.0000 |
| c) Interface provided to other modules | Input-output parameters per source line | Mean | 0.096 | 0.118 | 0.103 | 0.081 | 0.094 |
| | | Std. dev. | 0.063 | 0.072 | 0.090 | 0.084 | 0.073 |
| | | Median | 0.086 | 0.104 | 0.088 | 0.066 | 0.083 |

Fig. 7. Module design attributes that characterize the classes of module origin: interfaces with system modules, interfaces with utility functions, and interfaces provided to other modules. In the first attribute ("a"), calls to utility functions are not counted. Overall differences for the three attributes are statistically significant at: $\alpha < .0001$, $\alpha < .0001$, and $\alpha < .0001$, respectively. Number of modules in each category is: 1,629, 205, 300, 820, and 2,954, respectively.

| Module attribute | | | Class of software modules | | | | |
|---|---|---|---|---|---|---|---|
| Factor | Metric | Statistic | New development | Major revision | Slight revision | Complete reuse | All modules |
| a) Interface with human users | Read and write statements per source line | Mean | 0.0178 | 0.0156 | 0.0187 | 0.0058 | 0.0144 |
| | | Std. dev. | 0.0213 | 0.0163 | 0.0189 | 0.0134 | 0.0196 |
| | | Median | 0.0120 | 0.0106 | 0.0141 | 0.0000 | 0.0072 |
| b) Documentation | Comments per source line | Mean | 0.546 | 0.524 | 0.543 | 0.585 | 0.555 |
| | | Std. dev. | 0.141 | 0.128 | 0.146 | 0.197 | 0.159 |
| | | Median | 0.554 | 0.538 | 0.565 | 0.621 | 0.566 |
| c) Design effort | % of development effort spent in design | Mean | 19.8 | 15.7 | 12.3 | 14.6 | 18.1 |
| | | Std. dev. | 24.7 | 24.6 | 22.2 | 30.3 | 24.9 |
| | | Median | 9.5 | 0.0 | 0.0 | 0.0 | 5.4 |

Fig. 8. Module design attributes that characterize the classes of module origin: interface with users, documentation, and design effort. Overall differences for the three attributes are statistically significant at: $\alpha < .0001$, $\alpha < .0001$, and $\alpha < .0001$, respectively. Number of modules in each category for the first two attributes ("a" and 'b") is: 1,629, 205, 300, 820, and 2,954, respectively. Number of modules in each category for the third attribute ("c") is: 1629, 205, 300, 124, and 2,258, respectively (since 696 completely reused modules had zero hours development effort).

calls to other system modules, newly developed or slightly revised modules had the second most, and completely reused ones had the fewest (Fig. 7a; simultaneous $\alpha < .05$). Completely reused modules had more calls to utility functions than did either newly developed or slightly revised modules (Fig. 7b, simultaneous $\alpha < .05$).

### 7.2 Interfaces Other System Modules Have with a Module

A straightforward measure is considered for characterizing the interface between other system modules and a given module. The measure is the number of input and output parameters in a module, including any global data referenced. Fig. 7c displays the module averages for the number of input and output parameters. Modules with major revision had the most parameters, those that were either newly developed or slightly revised had the second most, and those that were completely reused had the fewest (simultaneous $\alpha < .05$).

### 7.3 Module's Interfaces with Human Users

Modules have interfaces not only with other modules, but also with human users. One perspective of a module's interface with human users is captured by the number of its input and output statements, which is the number of reads and writes. Fig. 8a presents the module averages for read and write statements. The modules that were completely reused had the fewest read and write statements (simultaneous $\alpha < .05$).

### 7.4 Documentation of a Module's Functionality

In the environment being examined, a description of the intended functionality of a module is written in text as comments. This description is included with the final implementation of a module as a set of comments. One may argue that a lengthy description enables a clear understanding of a module's functionality. One may also argue that a lengthy description indicates a complicated specification that may be difficult to understand or implement. An approximation for a module's ratio of commentary to functionality is the number of comments per source line in the final implementation. The distribution across module origin for this measure is given in Fig. 8b. Modules that were completely reused had a higher commentary to source line ratio than did all other modules (simultaneous $\alpha < .05$).

### 7.5 Module Design Effort

Fig. 8c displays the average percentage of module development effort spent in module design. Modules that were newly developed had a higher percentage of effort spent in module design than did all other modules (simultaneous $\alpha < .05$).

## 8 Characterizing Software Reuse at the Module Implementation Level

This section focuses on the characterization of software reuse at the module implementation level. Several aspects of software implementation are considered:

- size of a module,
- development effort of a module,
- control flow structure in a module, and
- assignment statements ("noncontrol flow structure") in a module.

### 8.1 Module Size

The size and development effort of the modules analyzed provide an initial characterization of their implementation. Fig. 9a displays the module averages for final implementation size in source lines. Modules reused with major revision were the largest, newly developed modules were the second largest, those reused with slight revision were the third largest, and those completely reused without revision were the smallest (simultaneous $\alpha < .05$).

### 8.2 Module Development Effort

Fig. 9b displays the module averages for total development effort. Modules that either were newly developed or had major revision had the most development effort, while slightly revised modules had the second most and those completely reused had the least (simultaneous $\alpha < .05$). There was no statistically significant difference in development effort when newly developed modules and modules with major revision were compared ($\alpha > .05$). The development effort for a module with major revision, slight revision, or complete reuse is the effort spent on an existing

| Module attribute | | | Class of software modules | | | | |
|---|---|---|---|---|---|---|---|
| | | | New | Major | Slight | Complete | All |
| Factor | Metric | Statistic | development | revision | revision | reuse | modules |
| a) Final implementation size | Source lines | Mean | 226.7 | 324.1 | 192.8 | 146.2 | 207.7 |
| | | Std. dev. | 158.1 | 221.3 | 125.7 | 152.3 | 165.5 |
| | | Median | 185.0 | 262.0 | 164.5 | 92.0 | 165.0 |
| b) Total development effort | Tenths of human hours | Mean | 213.5 | 193.6 | 86.2 | 5.8 | 141.5 |
| | | Std. dev. | 378.0 | 284.8 | 144.6 | 28.0 | 308.5 |
| | | Median | 110.0 | 100.0 | 60.0 | 0.0 | 60.0 |
| c) Total development effort | Tenths of human hours per source line | Mean | 1.089 | 0.758 | 0.601 | 0.047 | 0.727 |
| | | Std. dev. | 3.575 | 1.976 | 1.861 | 0.207 | 2.808 |
| | | Median | 0.616 | 0.421 | 0.345 | 0.000 | 0.346 |

Fig. 9. Module implementation attributes that characterize the classes of module origin: size and development effort. Overall differences for the three attributes are statistically significant at: $\alpha < .0001$, $\alpha < .0001$, and $\alpha < .0001$, respectively. Number of modules in each category is: 1,629, 205, 300, 820, and 2,954, respectively.

module to evaluate and/or modify it for a current project; the effort required for its original development is not included. Although some of the completely reused modules required a design inspection or some testing, 696 (84.9 percent) of them required zero hours of development effort. Fig. 9c displays the module averages for total development effort per source line. Newly developed modules had the most effort per source line, modules with major or slight revision had the second most, and completely reused modules had the least (simultaneous $\alpha < .05$).

### 8.3 Module Control Flow

One characteristic of an implementation for a module is its control flow structure. The cyclomatic complexity metric is based on the control flow graph for a module and has provided the foundations for various software testing methods [39]. Fig. 10a gives the module averages for cyclomatic complexity divided by the number of source lines. The overall difference is not statistically significant ($\alpha > .05$).

### 8.4 Module Assignment Statements

One aspect of the "noncontrol flow structure" in a module is its assignment statements. Fig. 10b gives the module averages for the number of assignment statements divided by the number of source lines. Modules that either were completely reused or had major revision had more assignment statements than did newly developed modules (simultaneous $\alpha < .05$).

## 9 Characterizing Software Reuse and Module Faults and Changes

This section focuses on the characterization of software reuse and several aspects of module faults and changes:

- module faults,
- module fault correction effort,
- module changes, and
- module change correction effort.

### 9.1 Module Faults

Figs. 11 and 12 display the total faults and fault isolation effort for the modules, broken down by module origin. The modules reused without revision had the fewest faults, fewest faults per source line, and lowest fault correction effort (simultaneous $\alpha < .05$). The newly developed modules and those reused with major revision were not statistically different in terms of faults or faults per source line ($\alpha > .05$). However, the modules reused with major revision had the highest fault correction effort and highest fault isolation effort (simultaneous $\alpha < .05$).

### 9.2 Module Changes

Fig. 13 displays the changes for the modules, broken down by module origin. Modules reused with major revision had the most changes, most changes per source line, and highest change correction effort (simultaneous $\alpha < .05$). Note that changes are sometimes made to "modules reused without revision" because of improved documentation or other changes that do not modify functionality.

| Module attribute | | | Class of software modules | | | | |
|---|---|---|---|---|---|---|---|
| | | | New | Major | Slight | Complete | All |
| Factor | Metric | Statistic | development | revision | revision | reuse | modules |
| a) Implementation style | Cyclomatic complexity per source line | Mean | 0.0795 | 0.0902 | 0.0882 | 0.0806 | 0.0814 |
| | | Std. dev. | 0.0512 | 0.0475 | 0.0554 | 0.0696 | 0.0572 |
| | | Median | 0.0709 | 0.0849 | 0.0827 | 0.0645 | 0.0710 |
| b) Implementation style | Assignment statements per source line | Mean | 0.111 | 0.134 | 0.117 | 0.141 | 0.121 |
| | | Std. dev. | 0.080 | 0.087 | 0.089 | 0.123 | 0.096 |
| | | Median | 0.099 | 0.123 | 0.098 | 0.118 | 0.105 |

Fig. 10. Module implementation attributes that characterize the classes of module origin: implementation styles. Overall differences for the second attribute ("b") are statistically significant at: $\alpha < .0001$. The overall differences for the first attribute ("a") are not statistically significant: $\alpha > .05$. Number of modules in each category is: 1,629, 205, 300, 820, and 2,954, respectively.

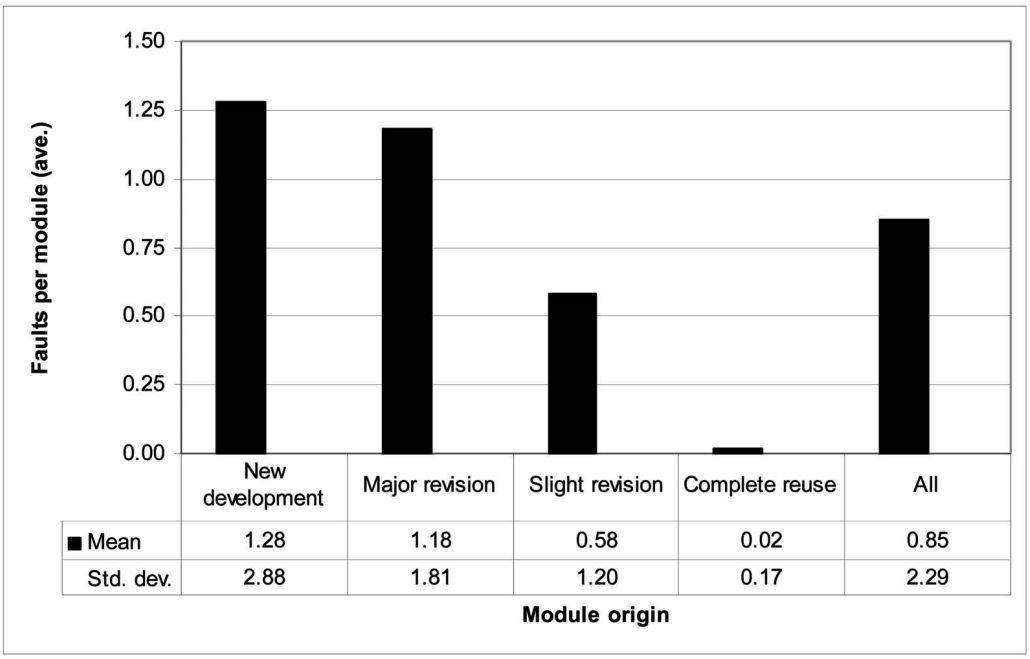



Fig. 11. Module averages (and standard deviations) for total faults, broken down by module origin. Overall difference is statistically significant ($\alpha < .0001$). Number of modules in each category is: 1,629, 205, 300, 820, and 2,954, respectively.

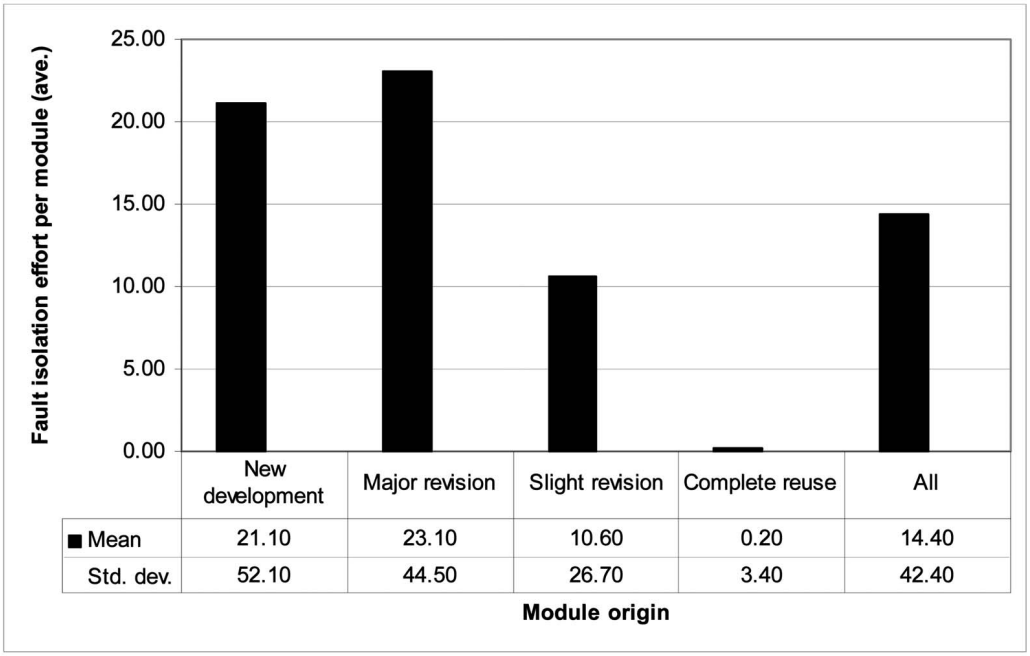



Fig. 12. Module averages (and standard deviations) for total fault isolation effort (in tenths of hours), broken down by module origin. Overall difference is statistically significant ($\alpha < .0001$). Number of modules in each category is: 1,629, 205, 300, 820, and 2,954, respectively.

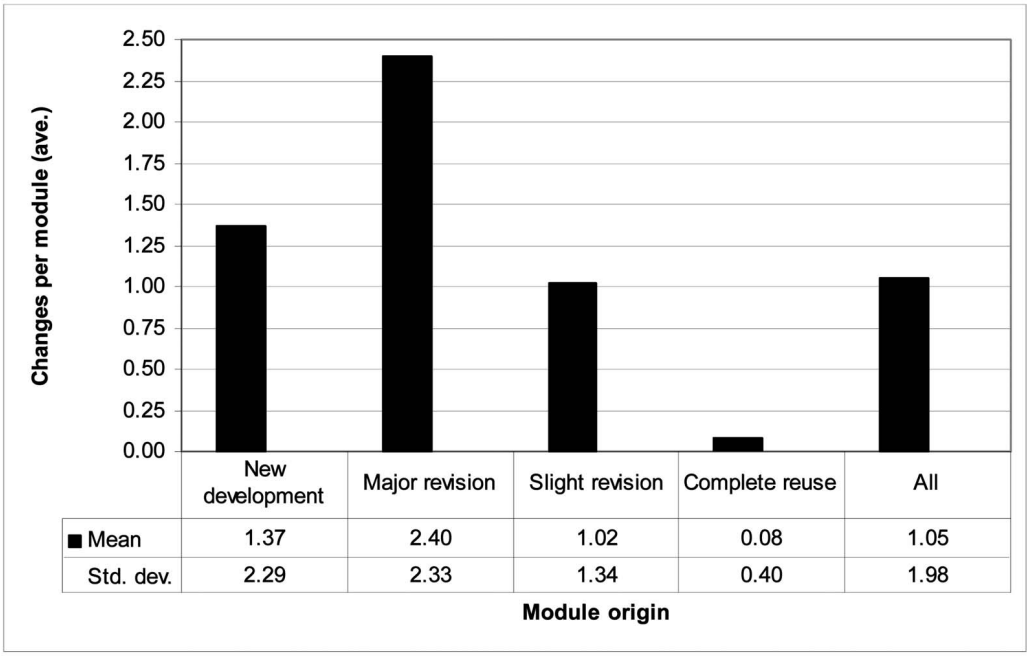
## 10 Interpretations and Conclusions

The purpose of this study is to characterize reuse-based software development empirically by mining software repositories from one environment that actively reuses software. This study intends to yield insights into factors that characterize successful software reuse in large-scale systems as well as enabling technologies that facilitate software reuse. Nonparametric analysis of variance models were applied to examine a wide range of development variables across the software modules in the systems.

### 10.1 Summary of Empirical Results

The experimental hypotheses that there are no differences in the characterizations of software due to modules reused without revision, modules reused with slight revision, modules reused with major revision, and newly developed modules were incorrect in terms of several different perspectives, as summarized below.

**Characterize software reuse at the project level**. When compared to smaller projects, projects of larger size had:




| | New development | Major revision | Slight revision | Complete reuse | All |
|---|---|---|---|---|---|
| Mean | 1.37 | 2.40 | 1.02 | 0.08 | 1.05 |
| Std. dev. | 2.29 | 2.33 | 1.34 | 0.40 | 1.98 |

Fig. 13. Module averages (and standard deviations) for total changes (excluding faults), broken down by module origin. Overall difference is statistically significant ($\alpha < .0001$). Number of modules in each category is: 1,629, 205, 300, 820, and 2,954, respectively.

- a higher percentage of modules that were reused with major revision.

**Characterize software reuse at the module design level**. When compared to modules that were newly developed, had major revision, or had slight revision, modules that were completely reused without revision had:

- less interaction with other system modules, in terms of module calls per source line,
- simpler interfaces, in terms of input-output parameters per source line,
- less interaction with human users, in terms of read and write statements per source line, and
- higher ratios of commentary to implementation size, in terms of comments per source line.

When compared to newly developed modules, modules that were completely reused without revision had:

- more interaction with utility functions, in terms of utility function calls per source line and
- a lower percentage of development effort spent in design activities.

**Characterize software reuse at the module implementation level**. When compared to modules that were newly developed, had major revision, or had slight revision, modules that were completely reused without revision had:

- smaller size, in terms of source lines of code and
- less total development effort, in terms of either total development hours or total development hours per source line.

When compared to newly developed modules, modules that were completely reused without revision had:

- more assignment statements, in terms of the number of assignment statements per source line.

**Characterize software reuse and module faults and changes**. When compared to modules that were newly developed, had major revision, or had slight revision, modules that were completely reused without revision had:

- fewer faults, in terms of either total faults or total faults per source line, and
- less fault correction effort, in terms of total fault correction hours.

When compared to modules reused with major revision, modules that were completely reused without revision had:

- less fault isolation effort, in terms of total fault isolation hours,
- fewer changes, in terms of either total changes or total changes per source line, and
- less change correction effort, in terms of total change correction hours.

**Graphical Profile of Modules**. Fig. 14 summarizes graphically the characterization of module attributes for the four module classes:

- modules reused without revision,
- modules reused with slight revision (< 25 percent revision),
- modules reused with major revision ($\geq$ 25 percent revision), and
- newly developed modules.

Fig. 14 is interpreted as follows: A value of -20 percent for some attribute (e.g., size) means that 30 percent (= -20 percent + 50 percent) of the modules reused without revision had size above the median size value for all modules, and 70 percent of the modules reused without

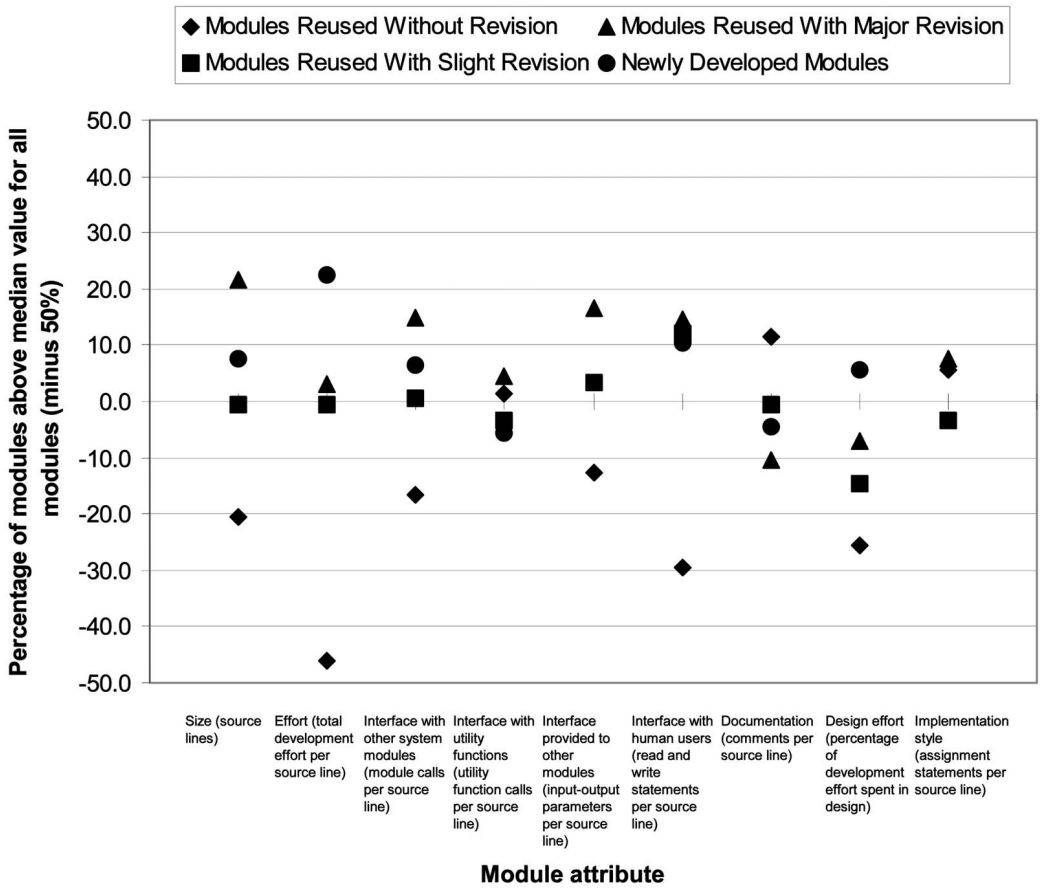



Fig. 14. Summary of nine module attributes, broken down by module origin. Number of modules newly developed, reused with major revision, reused with slight revision, and reused without revision is 1,629, 205, 300, and 820, respectively. There are 2,954 total modules.

revision had size below the median size value for all modules. A value of +10 percent for some attribute (e.g., documentation) means that 60 percent (= 10 percent + 50 percent) of the modules reused without revision had documentation above the median documentation value for all modules, and 40 percent of the modules reused without revision had documentation below the median documentation value for all modules. If modules reused without revision were representative of all modules in terms of some attribute, then it is likely that the attribute would be very close to 0 percent on this figure; this would mean that approximately 50 percent of the modules reused without revision are above the median value and approximately 50 percent are below the median value for all modules.

### 10.2 Interpretations

Analyzing the data from this NASA environment highlights that software reuse rates of 32 percent per project are both achievable and sustainable in large-scale systems. Many strategies, such as product lines [57], [61], [67], [68], design patterns [69], [70], [71], and architecture-based code generation, can be applied to achieve reuse. This particular environment uses a hybrid approach of mature reference architectures and broad populations of context-dependent component implementations. In this environment, the characterization of module design factors and module implementation factors clearly distinguished the reused modules from the others. The modules reused without revision tended to be small, well-documented modules with simple interfaces and little input-output processing. It also seems that these modules tended to be "terminal nodes" in the module invocation hierarchies, because they had less interaction with other system modules but more interaction with utility functions.

Analyzing these data also demonstrated that significant economic benefits can be realized through the systematic application of reuse. When a new project incorporated modules reused without revision, the modules required little additional development effort and they had few faults, few faults per source line, and low fault correction effort. Therefore, projects in this environment can achieve development effort savings in relative proportion to their degree of reuse. When comparing the development effort that was expended, a lower percentage was spent in design activities. This is because the design of a module from scratch requires the creation and evaluation of a new design, while the reuse of a module may require only a walkthrough of an existing design. When the developers were working on larger projects, they may have been even more motivated than usual to reuse modules because of the project scale. Consequently, a higher percentage of modules on the larger projects ended up reused with major revision.

From a strategic viewpoint, this study helps substantiate the following results:

- From the perspective of software reuse, applying module design and implementation factors can help enable reuse-based software development of large-scale systems.
- From the perspective of software design, integrating flexible combinations of architectures and components can yield sustainable, multiproject benefits in well-defined problem domains.
- From the perspective of software modeling, analysis, and evaluation, reusing software can produce significant simultaneous gains in productivity and quality.
- From the perspective of data analysis, combining disciplined data collection with mining software repositories can be very fruitful in quantifying relationships, trends, and trade-offs.

### 10.3 Future Directions

This research intends to yield insights that accelerate broad and deep adoption of reuse-based software development principles, processes, and infrastructure. Other factors beyond those investigated in this study may influence the degree and success of reuse, such as system requirements, functionality, environment, and organizational factors. This paper focuses on a set of results from an empirical study from mining software repositories from one environment. This work helps advance ideas that span a variety of software development areas:

- lifecycle models for large-scale systems,
- reuse-based development principles, methods, and techniques, and
- processes, tools, and infrastructure to support reuse.

A continuing theme in this work is to analyze more than just the reuse of source code, including in particular, information that relates to software reuse at the architecture and component levels. Further analysis of software reuse in the projects as well as other data sets is in progress, including even broader investigations and evaluations of project, module, and process attributes.

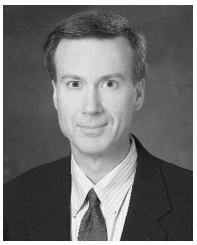
## ACKNOWLEDGMENTS

The author greatly appreciate the collaboration with the NASA software engineering laboratory. He would also like to thank the editors and anonymous referees whose insightful suggestions were very helpful.

## REFERENCES

[1] L.J. Osterweil, "Software Processes are Software Too," *Proc. Ninth Int'l Conf. Software Eng.,* pp. 2-13, Mar. 1987.
[2] B. Meyer, "Reusability: The Case for Object-Oriented Design," *IEEE Software,* vol. 4, no. 2, pp. 50-64, Mar. 1987.
[3] G. Booch, "Object-Oriented Development," *IEEE Trans. Software Eng.,* vol. 12, no. 2, pp. 211-221, Feb. 1986.
[4] J. Neighbors, "The Draco Approach to Constructing Software from Reusable Components," *IEEE Trans. Software Eng.,* vol. 10, no. 5, pp. 564-573, Sept. 1984.
[5] R. Conn, "An Overview of the DoD Ada Software Repository," *Dr. Dobbs J.,* pp. 60-61, 86-91, Feb. 1986.
[6] G.E. Kaiser and D. Garlan, "Melding Software Systems from Reusable Building Blocks," *IEEE Software,* vol. 4, no. 4, pp. 17-24, July 1987.
[7] T. Reps, *Generating Language-Based Environments.* MIT Press, 1983.
[8] A.N. Habermann and D. Notkin, "Gandalf: Software Development Environments," *IEEE Trans. Software Eng.,* vol. 12, no. 12, Dec. 1986.
[9] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.
[10] J.I. Cannon, "Flavors," *MIT Artificial Intelligence Laboratory,* technical report, MIT, Cambridge, Mass., 1980.
[11] D.G. Bobrow and M.J. Stefik, "Loops: An Objected-Oriented Programming System for Interlisp," technical report, Xerox PARC, Palo Alto, Calif., 1982.
[12] D. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel, "CommonLoops: Merging Lisp and Object-Oriented Programming," *Proc. OOPSLA 86: Object Oriented Programming Systems, Languages, and Applications,* pp. 17-29, 1986.
[13] J.-H. Hulot, "Ceyx, Version 15:1—Une Initiation," Technical Report no. 44, INRIA, France, 1984.
[14] B. Stroustrup, *The C++ Programming Language.* Addison-Wesley, 1986.
[15] L. Tesler, "Object Pascal Report," *Structured Language World,* 1985.
[16] G. Birstwistle et al., *Simula Begin.* Berlin: Studentliteratur and Auerbach, 1973.
[17] "Special Issue on Software Reusability," *IEEE Trans. Software Eng.,* T. Biggerstaff and A. Perlis, eds., vol. 10, no. 5, Sept. 1984.
[18] *Proc. Workshop Reusability in Programming,* T. Biggerstaff, ed., 1983.
[19] "Special Issue on Software Reusability," *IEEE Software,* W. Tracz, ed., vol. 4, no. 4, July 1987.
[20] T. Biggerstaff and A. Perlis, *Software Reusability.* Addison-Wesley, 1989.
[21] W. Tracz, "Ada Reusability Efforts: A Survey of the State of the Practice," *Proc. Fifth Nat'l Conf. Ada Technology,* Mar. 1987.
[22] T. Biggerstaff and C. Richter, "Reusability Framework, Assessment, and Directions," *IEEE Software,* vol. 4, no. 2, pp. 41-49, Mar. 1987.
[23] R.W. Selby, "Analyzing Software Reuse at the Project and Module Design Levels," *Proc. First European Software Eng. Conf.,* pp. 227-235, Sept. 1987.
[24] R.W. Selby, "Empirically Analyzing Software Reuse in a Production Environment," *Software Reuse—Emerging Technologies,* W. Tracz, ed. Sept. 1988.
[25] R.W. Selby, "Quantitative Studies of Software Reuse," *Software Reusability,* T. Biggerstaff and A. Perlis, eds., Addison-Wesley, 1989.
[26] "Software Eng. Laboratory (SEL): Database Organization and User's Guide, Revision 1," Technical Report no. SEL-81-102, Software Eng. Laboratory, NASA/Goddard Space Flight Center, Greenbelt, Md., July 1983.
[27] "Collected Software Eng. Papers: Vol. 1," Technical Report no. SEL-82-004, Software Eng. Laboratory, NASA/Goddard Space Flight Center, Greenbelt, Md., July 1982.
[28] V.R. Basili, R.W. Selby, and T.Y. Phillips, "Metric Analysis and Data Validation Across Fortran Projects," *IEEE Trans. Software Eng.,* vol. 9, no. 6, pp. 652-663, Nov. 1983.
[29] D.M. Weiss and V.R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory," *IEEE Trans. Software Eng.,* vol. 11, no. 2, pp. 157-168, Feb. 1985.
[30] D.N. Card, G.T. Page, and F.E. McGarry, "Criteria for Software Modularization," *Proc. Eighth Int'l Conf. Software Eng.,* pp. 372-377, Aug. 1985.
[31] D.N. Card, V.E. Church, and W.W. Agresti, "An Empirical Study of Software Design Practices," *IEEE Trans. Software Eng.,* vol. 12, no. 2, pp. 264-271, Feb. 1986.
[32] H. Scheffe, *The Analysis of Variance.* John Wiley and Sons, 1959.
[33] F.P. Brooks, *The Mythical Man-Month.* Addison-Wesley, 1975.
[34] C.E. Walston and C.P. Felix, "A Method of Programming Measurement and Estimation," *IBM Systems J.,* vol. 16, no. 1, pp. 54-73, 1977.
[35] B.W. Boehm, *Software Eng. Economics.* Prentice-Hall, 1981.
[36] W.D. Brooks, "Software Technology Payoff: Some Statistical Evidence," *J. Systems and Software,* vol. 2, pp. 3-9, 1981.
[37] J. Vosburgh, B. Curtis, R. Wolverton, B. Albert, H. Malec, S. Hoben, and Y. Liu, "Productivity Factors and Programming Environments," *Proc. Seventh Int'l Conf. Software Eng.,* pp. 143-152, 1984.

[38] W.G. Cochran and G.M. Cox, *Experimental Designs.* John Wiley and Sons, 1950.
[39] T.J. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.,* vol. 2, no. 4, pp. 308-320, Dec. 1976.
[40] R.W. Selby and A.A. Porter, "Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis," *IEEE Trans. Software Eng.,* vol. 14, no. 12, pp. 1743-1757, Dec. 1988.
[41] A. Tomer, L. Goldin, T. Kuflik, E. Kimchi, and S.R. Schach, "Evaluating Software Reuse Alternatives: A Model and its Application to an Industrial Case Study," *IEEE Trans. Software Eng.,* vol. 30, no. 9, pp. 601-612, Sept. 2004.
[42] B. Morel and P. Alexander, "SPARTACAS: Automating Component Reuse and Adaptation," *IEEE Trans. Software Eng.,* vol. 30, no. 9, pp. 587-600, Sept. 2004.
[43] M.A. Rothenberger, K.J. Dooley, U.R. Kulkarni, and N. Nada, "Strategies for Software Reuse: A Principal Component Analysis of Reuse Practices," *IEEE Trans. Software Eng.,* vol. 29, no. 9, pp. 825-837, Sept. 2003.
[44] T. Menzies and J.S. Di Stefano, "More Success and Failure Factors in Software Reuse," *IEEE Trans. Software Eng.,* vol. 29, no. 5, pp. 474-477, May 2003.
[45] M. Morisio, M. Ezran, and C. Tully, "Success and Failure Factors in Software Reuse," *IEEE Trans. Software Eng.,* vol. 28, no. 4, pp. 340-357, Apr. 2002.
[46] G. Succi, L. Benedicenti, and T. Vernazza, "Analysis of the Effects of Software Reuse on Customer Satisfaction in an RPG Environment," *IEEE Trans. Software Eng.,* vol. 27, no. 5, pp. 473-479, May 2001.
[47] P.T. Devanbu, D.E. Perry, and J.S. Poulin, "Guest Editors Introduction: Next Generation Software Reuse," *IEEE Trans. Software Eng.,* vol. 26, no. 5, pp. 423-424, May 2000.
[48] F. Lanubile and G. Visaggio, "Extracting Reusable Functions by Flow Graph Based Program Slicing," *IEEE Trans. Software Eng.,* vol. 23, no. 4, pp. 246-259, Apr. 1997.
[49] A. Sen, "The Role of Opportunism in the Software Design Reuse Process," *IEEE Trans. Software Eng.,* vol. 23, no. 7, pp. 418-436, July 1997.
[50] G.S. Novak, "Software Reuse by Specialization of Generic Procedures Through Views," *IEEE Trans. Software Eng.,* vol. 23, no. 7, pp. 401-417, July 1997.
[51] N.Y. Lee and C.R. Litecky, "An Empirical Study of Software Reuse with Special Attention to Ada," *IEEE Trans. Software Eng.,* vol. 23, no. 9, pp. 537-549, Sept. 1997.
[52] M.J. Harrold and G. Rothermel, "Separate Computation of Alias Information for Reuse," *IEEE Trans. Software Eng.,* vol. 22, no. 7, pp. 442-460, July 1996.
[53] T. Isakowitz and R.J. Kauffman, "Supporting Search for Reusable Software Objects," *IEEE Trans. Software Eng.,* vol. 22, no. 6, pp. 407-423, June 1996.
[54] W.B. Frakes and C.J. Fox, "Quality Improvement Using a Software Reuse Failure Modes Model," *IEEE Trans. Software Eng.,* vol. 22, no. 4, pp. 274-279, Apr. 1996.
[55] V. Rajlich and J.H. Silva, "Evolution and Reuse of Orthogonal Architecture," *IEEE Trans. Software Eng.,* vol. 22, no. 2, pp. 153-157, Feb. 1996.
[56] V.R. Basili, R.W. Selby, and D.H. Hutchens, "Experimentation in Software Eng.," *IEEE Trans. Software Eng.,* vol. 12, no. 7, pp. 733-743, July 1986.
[57] G. Bockle, P. Clements, J.D. McGregor, D. Muthig, and K. Schmid, "Calculating ROI for Software Product Lines," *IEEE Software,* vol. 21, no. 3, pp. 23-31, May-June 2004.
[58] S. Deelstra, M. Sinnema, J. Nijhuis, and J. Bosch, "COSVAM: A Technique for Assessing Software Variability in Software Product Families," *Proc. 20th IEEE Int'l Conf. Software Maintenance,* pp. 458-462, Sept. 2004.
[59] J. Bosch and N. Juristo, "Designing Software Architectures for Usability," *Proc. 25th Int'l Conf. Software Eng.,* pp. 757-758, May 2003.
[60] J. Bosch, "Architecture-Centric Software Eng.," *Proc. 24th Int'l Conf. Software Eng.,* pp. 681-682, May 2002.
[61] J. Bosch, "Software Product Lines: Organizational Alternatives," *Proc. 23rd Int'l Conf. Software Eng.,* pp. 91-100, May 2001.
[62] J. Bosch, "Design and Use of Industrial Software Architectures," *Proc. Conf. Technology of Object-Oriented Languages and Systems,* pp. 404-404, June 1999.
[63] H. Gomaa, L. Kerschberg, V. Sugumaran, C. Bosch, and I. Tavakoli, "A Prototype Domain Modeling Environment for Reusable Software Architectures," *Proc. Third Int'l Conf. Software Reuse: Advances in Software Reusability,* pp. 74-83, Nov. 1994.
[64] P. Clements, D. Garlan, R. Little, R. Nord, and J. Stafford, "Documenting Software Architectures: Views and Beyond," *Proc. 25th Int'l Conf. Software Eng.,* pp. 740-741, May 2003.
[65] M. Shaw and P. Clements, "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems," *Proc. 21st Ann. Int'l Computer Software and Applications Conf.,* pp. 6-13, Aug. 1997.
[66] R. Kazman, G. Abowd, L. Bass, and P. Clements, "Scenario-Based Analysis of Software Architecture," *IEEE Software,* vol. 13, no. 6, pp. 47-55, Nov. 1996.
[67] J. Coplien, D. Hoffman, and D. Weiss, "Commonality and Variability in Software Engineering," *IEEE Software,* vol. 15, no. 6, pp. 37-45, Nov.-Dec. 1998.
[68] J. Klein, B. Price, and D. Weiss, "Industrial-Strength Software Product-Line Engineering," *Proc. 25th Int'l Conf. Software Eng.,* pp. 751-752, May 2003.
[69] N. Soundarajan and J.O. Hallstrom, "Responsibilities and Rewards: Specifying Design Patterns," *Proc. 26th Int'l Conf. Software Eng.,* pp. 666-675, May 2004.
[70] J.K.H. Mak, C.S.T. Choy, and D.P.K. Lun, "Precise Modeling of Design Patterns in UML," *Proc. 26th Int'l Conf. Software Eng.,* pp. 252-261, May 2004.
[71] J. Gustafsson, J. Paakki, L. Nenonen, and A.I. Verkamo, "Architecture-Centric Software Evolution by Software Metrics and Design Patterns," *Proc. Sixth European Conf. Software Maintenance and Re-Eng.,* pp. 108-115, Mar. 2002.



**Richard W. Selby** received the PhD and MS degrees in computer science from the University of Maryland, College Park, in 1985 and 1983, respectively. He received the BA degree in mathematics from St. Olaf College, Northfield, Minnesota in 1981. He is the head of software products at Northrop Grumman Space Technology in Redondo Beach, California. He manages a 250-person software organization and has served in this position since 2001. Previously, he was the chief technology officer and senior vice president at Pacific Investment Management Company (PIMCO) in Newport Beach, California, where he managed a 105-person organization for three years. From 1985-1998, he was a full professor of information and computer science (with tenure) at the University of California in Irvine. Since 2004, he has held an adjunct faculty position at the USC Computer Science Department in Los Angeles. In 1993, he held visiting faculty positions at the MIT Laboratory for Computer Science and MIT Sloan School of Management in Cambridge, Massachusetts and in 1992, he held a visiting faculty position at the Osaka University Department of Computer Science in Osaka, Japan. His research focuses on development and management of large-scale systems, software, and processes. He has authored more than 95 refereed publications and given over 190 invited presentations at professional meetings. At Northrop, he led the three billion dollar company to a successful enterprise-wide rating of Capability Maturity Model Integration (CMMI) Level 5 for Software. At PIMCO, he led the one billion dollar company to be ranked as the fourth most innovative technology organization in financial services, according to *Wall Street & Technology*. At UC Irvine, he coauthored an international best-selling book that analyzed Microsoft's technology, strategy, and management that was entitled *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*. The book, written with Michael Cusumano, has been translated into 12 languages, has 150,000 copies in print, and was ranked as a top-six best-seller in *Business Week*. He is a member of the IEEE.