SOFTWARE ENGINEERING LABORATORY SERIES

SEL-95-003

COLLECTED SOFTWARE ENGINEERING PAPERS: VOLUME XIII

NOVEMBER 1995

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

SOFTWARE ENGINEERING LABORATORY SERIES

SEL-95-003

COLLECTED SOFTWARE ENGINEERING PAPERS: VOLUME XIII

NOVEMBER 1995

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

FOREWORD

The Software Engineering Laboratory (SEL) is an organization sponsored by the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) and created to investigate the effectiveness of software engineering technologies when applied to the development of application software. The SEL was created in 1976 and has three primary organizational members:

NASA/GSFC, Software Engineering Branch

University of Maryland, Department of Computer Science

Computer Sciences Corporation, Software Engineering Operation

The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effect of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document.

Single copies of this document can be obtained by writing to

Software Engineering Branch
Code 552
Goddard Space Flight Center
Greenbelt, Maryland, U.S.A. 20771

Page intentionally left blank

TABLE OF CONTENTS

omit

SECTION 1—INTRODUCTION

This document is a collection of selected technical papers produced by participants in the Software Engineering Laboratory (SEL) from September 1994 through November 1995. The purpose of the document is to make available, in one reference, some results of SEL research that originally appeared in a number of different forums. This is the 13th such volume of technical papers produced by the SEL. Although these papers cover several topics related to software engineering, they do not encompass the entire scope of SEL activities and interests. Additional information about the SEL and its research efforts may be obtained from the sources listed in the bibliography at the end of this document.

For the convenience of this presentation, the nine papers contained here are grouped into four major sections:

- Software Engineering Laboratory (Section 2)
- Software Models (Section 3)
- Software Measurement (Section 4)
- Technology Evaluations (Section 5)

Section 2 includes several papers and articles that describe the SEL's process improvement program and the Experience Factory and it's relationship to other improvement approaches. Section 3 contains a case study that uses the Actor-Dependency Model to analyze and assess a large software maintenance organization. Section 4 includes four papers. The first describes a rigorous and disciplined approach to defining product metrics, and the second evaluates property-based metrics defined using this approach. The third paper in Section 4 gives a study that uses error data to better understand and evaluate an evolving reuse process, and the fourth paper presents an experimental investigation of a suite of object-oriented design metrics. Finally, Section 5 contains an experience report that describes using domain analysis to create a library of highly reusable components that are able to be configured within a standard architecture to produce low-cost systems.

The SEL is actively working to understand and improve the software development process at the Goddard Space Flight Center (GSFC). Future efforts will be documented in additional volumes of the *Collected Software Engineering Papers* and other SEL publications.

omit

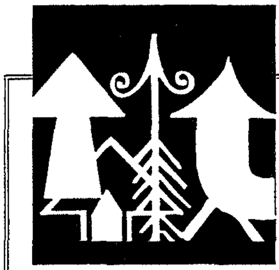
SECTION 2—THE SOFTWARE ENGINEERING LABORATORY

The technical papers included in this section were originally prepared as indicated below.

- "SEL's Software Process-Improvement Program," V. Basili, M. Zelkowitz, F. McGarry, G. Page, S. Waligora, and R. Pajerski, *IEEE Software*, vol. 12, no. 6, November 1995, pp. 83–87
- *The Experience Factory Strategy and Practice*, V. R. Basili and G. Caldiera, University of Maryland, Computer Science Technical Report, CS-TR-3483, UMIACS-TR-95-67, May 1995
- "The Experience Factory and Its Relationship to Other Quality Approaches," V. R. Basili, *Advances in Computers*, vol. 41, Academic Press, Incorporated, 1995

Page intentionally left blank

51-61
415776
361054
6P



SEL'S SOFTWARE PROCESS-IMPROVEMENT PROGRAM

In 1993, the IEEE Computer Society and the Software Engineering Institute jointly established the Software Process Achievement Award to recognize outstanding improvement accomplishments. This award is to be given annually if suitable nominations are received by the SEI before November 1 each year. The nominations are reviewed by an award committee of Barry Boehm, Manny Lehman, Bill Riddle, myself, and Vic Basili (who did not participate in this award decision because of his involvement in the Software Engineering Laboratory).

It is particularly fitting that the SEL was selected as the first winner for this award. They started their pioneering work nearly a decade before the Software Engineering Institute was founded, and their work has been both a guide and an inspiration to all of us who have attempted to follow in their footsteps.

— Watts Humphrey

VICTOR BASILI
and MARVIN ZELKOWITZ
University of Maryland

FRANK McGARRY,
JERRY PAGE,
and SHARON WALIGORA
Computer Sciences Corporation

ROSE PAJERSKI
NASA Goddard Space
Flight Center

For nearly 20 years, the Software Engineering Laboratory has worked to understand, assess, and improve software and the software-development process within the production environment of the Flight Dynamics Division of NASA's Goddard Space Flight Center. We have conducted experiments on about 125 FDD projects, applying, measuring, and analyzing numerous software-process changes. As a result, the SEL has adopted and tailored processes — based on FDD goals and experience — to significantly improve software production.

The SEL is a cooperative effort of NASA/ Goddard's FDD, the University of Maryland Department of Computer Science, and Computer Sciences Corporation's Flight Dynamics Technology Group. It was established in 1976 with the goal of reducing

♦ the defect rate of delivered software,

♦ the cost of software to support flight projects, and

♦ the average time to produce mission-support software.

Our work has yielded an extensive set of empirical studies that has guided the evolution of standards, management practices, technologies, and training within the organization. The result has been a 75 percent reduction in defects, a 50 percent reduction in cost, and a 25 percent reduction in cycle time. Over time, the goals of SEL have matured. We now strive to:

♦ *Understand* baseline processes and product characteristics, such as cost, reliability, software size, reuse levels, and error classes. By characterizing a production environment, we can gain better insight into the software process and its products.

♦ *Assess* improvements that have been incorporated into development projects. By measuring the impact of available technologies on the software process, we can determine which technologies are beneficial to the environment and — most importantly — how the technologies should be refined to best match the process with the environment.

♦ *Package and infuse* improvements into the standard SEL process and update and refine standards, handbooks, training materials, and development-support tools.[1-3] By identifying process improvements, we can package the technology so it can be applied in the production environment.

As Figure 1 shows, these goals are pursued in a sequential, iterative process that has been formalized by Basili as the Quality Improvement Paradigm[4] and its use within the SEL formalized as the Experience Factory.[5]

IMPROVING THE PROCESS

We select candidates

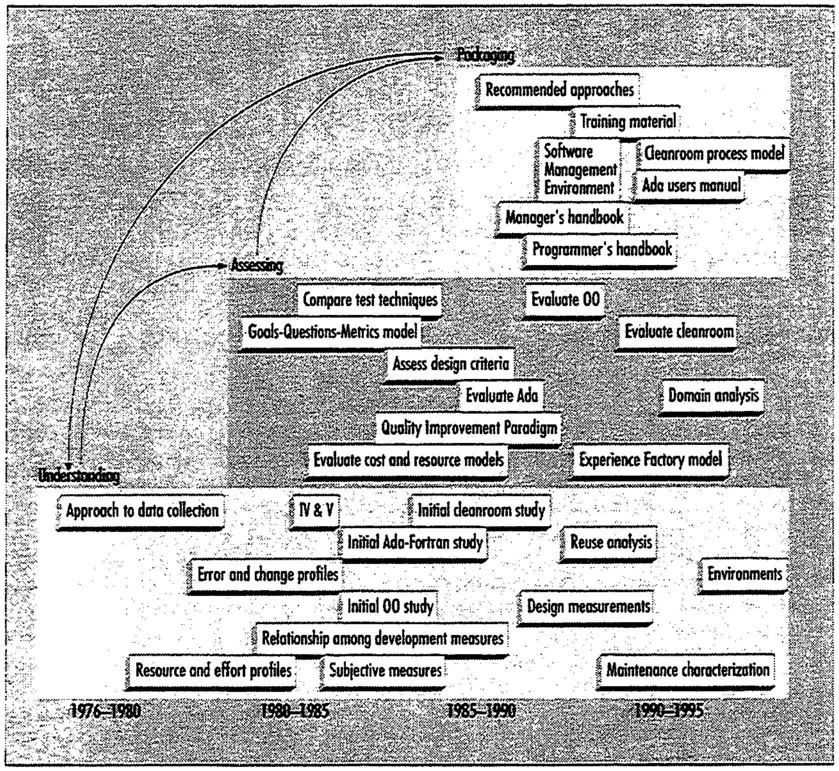


Figure 1. The SEL goals are pursued in a sequential, iterative fashion. The diagram includes some of the many SEL studies that have been conducted over the years, including those of Cleanroom, Ada, and Fortran.

for process change on the basis of quantified SEL experiences (such as the most significant causes of errors) and clearly defined goals for the software (such as to decrease error rates). After we select the changes, we provide training and formulate experiment plans. We then apply the new process to one or more production projects and take detailed measurements. We assess a process's success by comparing these measures with the continually evolving baseline. Based upon the results of the analysis, we adopt, discard, or revise the process.

Process improvement applies to individual projects, experiments (the observation of two or three projects), as well as the overall organization (the observation of trends over many years). In the early years, the SEL emphasized building a clear understanding of the process and products within the environment. This led us to develop models, relations, and general characteristics of the SEL environment. Most of our process changes consisted of studying specific, focused techniques (such as program-description language, structure charts, and reading techniques), but the major enhancement was the infusion of measurement, process-improvement concepts, and the realization of the significance of process in the software culture.

SEL OPERATIONS

The SEL has collected and archived data on more than 125 of its software-development projects. We use the data to build typical-project profiles against which we compare and evaluate ongoing projects. The SEL provides its managers with tools for monitoring and assessing project status. The FDD typically runs six to 10 projects simultaneously, each of which is considered an experiment within the SEL.

For each project, we collect a basic set of information (such as effort and error data). From there, the data we collect may vary according to the experiment or be modified as changes are made to specific processes (such as the use of Ada). As the information is collected, it is validated and placed in a central database. We then use this data with other information — such as the subjective lessons learned — to analyze the impact of a specific software process and to measure and feed back results to both ongoing and follow-on projects.

We also use the data to build predictive models and to provide a rationale for refining current software processes. As we analyze the data, we generate papers and reports that reflect the results of numerous studies. We also package the results as standards, policies, training materials, and management tools.

PROCESS AND PRODUCT ANALYSIS

The FDD is responsible for the development and maintenance of flight-dynamics ground-support software for all Goddard flight projects. Typical FDD projects range in size from 100,000 to 300,000 lines of code. Several projects exceed a million lines of code; others are as small as 10,000 lines of code. (At SEL, reused code is not "free"; it is counted as 20 percent of new Fortran code and 30 percent of new Ada code.) The SEL improvement goal is to demonstrate continual improvement of the software process within the FDD environment by carrying out analysis, measurement, and feedback to projects within this environment.

Understanding. Understanding what an organization does and how it operates is fundamental to any attempt to plan, manage, or improve the software process. This is especially true for software-development organizations. The SEL supports this understanding in several ways, including, for example, the study of effort distribution and error-detection rate.

♦ Effort distribution identifies which phases of the life cycle consume which portion of development effort. Figure 2 presents the effort distribution of 11 Fortran projects by life-cycle phase and activity. Understanding these distributions helps us plan new efforts, evaluate new technologies, and assess the similarities and differences within an ongoing project.

♦ Error-detection rate provides the absolute error rate expected in each phase. At SEL, we collected information on software errors and built a model of the expected errors in each life-cycle phase. For 1,000 lines of code, we found about four errors during implementation; two during system test; one during acceptance test; and one-half during operation and maintenance. The trend we derive from this model is that error detection rates fall by 50 percent in each subsequent phase. This pattern seems to be independent of the actual error rates; it is true even in recent projects, in which the overall error rates are declining. We use this model of error rates, as well as other similar types of models, to better predict, manage, and assess change on newly developed projects.

Assessing and refining. We consider each SEL project to be an experiment, in which we study some software method in detail. Generally, the subject of the study is a specific modification to the standard process — a process that obviously comprises numerous software methods.

For example, the Cleanroom software methodology[6] has been applied on four projects within the SEL, three of which have been analyzed thus far. Each project gave us additional insight into the Cleanroom process and helped us refine the method for use in the FDD environment. After training teams in the Cleanroom methodology, we defined a modified set of Cleanroom-specific data to be collected. The teams studied the projects to assess the impact that Cleanroom had on the process, as well as on measures such as productivity and reliability. Figure 3 shows the results of the three analyzed projects.

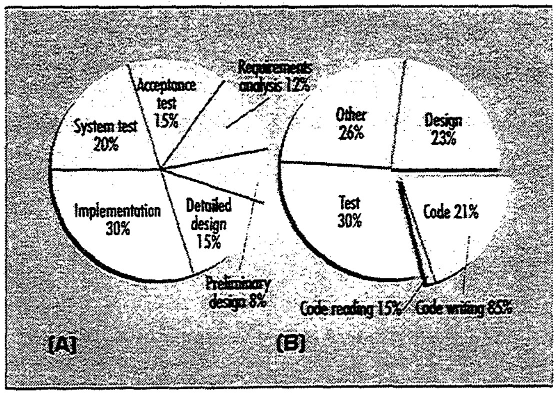


Figure 2. Effort distribution by (A) life-cycle phase and (B) activity. Phase data counts hours charged to a project during each calendar phase. Activity data counts hours attributed to a particular activity (as reported by the programmer), regardless of when in the life cycle the activity occurred.

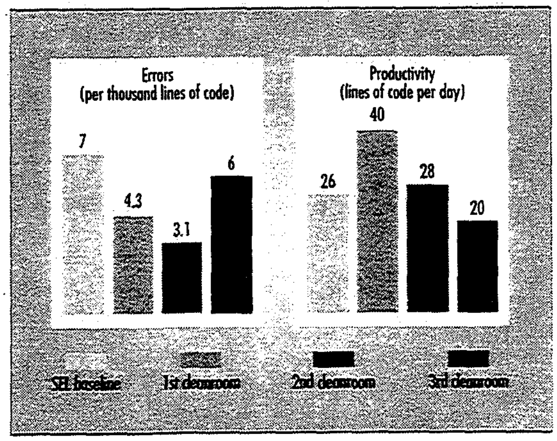


Figure 3. Results of three completed Cleanroom projects, compared against the SEL baseline.

The Cleanroom experiments required significant changes to the standard SEL development methodology and thus extensive training, preparation, and careful study execution. As in all such experiments, we generated detailed experimentation plans that described the goals, the questions that had to be addressed, and the metrics that had to be collected to answer the questions. Because Cleanroom consists of many specific methods — such as box-structure design, statistical testing, and rigorous inspections — each particular method had to be analyzed, along with the Cleanroom methodology itself. As a result of these projects, a slightly modified Cleanroom approach was deemed beneficial for smaller SEL projects. Anecdotal evidence from the recently completed fourth Cleanroom project confirms the effectiveness of Cleanroom. The revised Cleanroom-process model was captured in a process handbook for future applications to SEL projects. We have analyzed and applied many other methodologies in this way.

Packaging. Once we have identified beneficial methods and technologies, we provide feedback for future projects by capturing the process in standards, tools, and training. The SEL has produced a set of standards for its own use that reflect the results of its studies. Such standards must continually evolve to capture modified characteristics of the process (the SEL typically updates its basic standard every five years.) Standards we have pro-

TABLE 1
EARLY SEL BASELINE

Project (number & name)	Reuse (percent)	Mission Cost* (staff months)	Reliability (error/KSLOC)
1. GROAGSS	14	381	4.42
2. COBEAGSS	12	348	5.22
3. GOESAGSS	12	261	5.18
4. UARSAGSS	10	675	2.81
5. GROSIM	18	79	8.91
6. COBSIM	11	39	4.45
7. GOESIM	29	96	1.72
8. UARSTELS	35	80	2.96

** Mission cost = cost of telemetry simulator + cost of AGSS (GRO = projects 1 + 5, COBE = 2 + 6, GOES = 3 + 7, UARS = 4 + 8).*

TABLE 2
CURRENT SEL BASELINE

Project (number & name)	Reuse (percent)	Cost* (staff months)	Reliability (error/KSLOC)
1. EUVEAGSS	18	155	1.22
2. SAMPEX	83	77	.76
3. WINDPOLR	18	476	n/a†
4. EUVETELS	96	36	.41
5. SAMPEXTS	95	21	.48
6. POWITS	69	77	2.39
7. TOMSTELS	97	n/a‡	.23
8. FASTELS	92	n/a‡	.69

** Mission cost = cost of telemetry simulator + cost of AGSS (GRO = projects 1 + 5, COBE = 2+6, GOES = 3+7, UARS = 4 + 8).*
† Excluded because it used the Cleanroom development methodology, which counts errors differently.
‡ Total mission cost for TOMS and FAST cannot be calculated because AGSSs are incomplete (they are not included in the cost baseline).

duced include:

- ♦ *Manager's Handbook for Software Development,*[1]
- ♦ *Recommended Approach to Software Development,*[2] and
- ♦ *The SEL Relations and Models.*[3]

In addition to the evolving development standards, policies, and training material, successful packaging includes generating experiment results in the form of post-development analysis, formal papers, and guidebooks for applying specific software techniques.

IMPACT OF SEL

Our studies have involved many technologies, ranging from development and management practices to automation aids and technologies that affect the full life cycle. We have collected and archived detailed information so we can assess the impact of technologies on both the software process and product.

Product impact. To determine the effect of sustained SEL efforts as measured against our major goals, we routinely compare groups of projects developed at different times. Projects are grouped on the basis of size, mission complexity, mission characteristics, language, and platform. On these characteristic projects, we compared defect rates, cost, schedule, and levels of reuse. The reuse levels were studied carefully with the full expectation that there would be a correlation between higher reuse and lower cost and defect rates. These characteristic projects become our "baselines." Table 1 shows an early baseline — eight projects completed between 1985 and 1989. These projects were all ground-based attitude-determination and -simulation systems ranging in size from 50,000 to 150,000 lines of code that were developed on large IBM mainframes. Each was also a success, meeting mission dates and requirements within acceptable cost. Table 2 shows the current SEL baseline, which comprises seven similar projects completed between 1990 and 1994.

As the tables show, the early baseline projects had a reliability rate that ranged from 1.7 to 8.9 errors per 1,000 lines of code, with an average rate of 4.5 errors. The current baseline projects had a reliability rate ranging from 0.2 to 2.4 errors per 1,000 lines of code, with an average rate of 1 error. This is about a 75-percent reduction over the eight-year period.

The dramatic increase in our reuse levels — aided by experimentation with techniques such as object-oriented development and domain-engineering concepts — have been a major contributor to improved project cost and quality. Reuse, along with increased productivity, also contributed to a significant decrease in project cost. We examined selected missions from the two baselines and found that, although the total lines of code per mission remained relatively equal, the total mission cost decreased significantly. The average mission cost in the early baseline ranged from 357 to 755 staff-months, with an average of 490. The current baseline projects had costs ranging from 98 to 277 staff-months with an average of 210. This is a decrease in average cost per mission of more than 50 percent over the eight-year period. This reduction occurred despite the increased mission complexity, shown in Table 3.

Process impact. The most significant changes in the SEL environment are illustrated by the standards, training programs, and development approaches incorporated into the FDD

TABLE 3
COMPARING INCREASE IN BASELINE COMPLEXITY

Attribute	Early SEL baseline	Current SEL baseline
Control	Spin stabilized	Three-axis stabilized
Sensors	1	8 to 11
Torques	1	2 to 3
Onboard computer	Analog simple control	Digital control
Telemetry	5	12 to 15
Data rates	2.2 kbs	32 kbs
Accuracy	1 degree	0.02 degree

process. Although specific techniques and methods have had a measurable impact on a class of projects, significant improvement to the software-development process — and an overall change in the environment — has occurred because we have continuously incorporated detailed techniques into higher level organizational processes.

The most significant process attributes that distinguish our current production environment from that of a decade earlier include:

♦ Process change and improvement has been infused as a standard business practice. All standards and training material now contain elements of our continuous-improvement approach to experimentation.

♦ Measurement is now our way of doing business rather than an add-on to development. Measurement is as much a part of our software standards as documentation. It is expected, applied, and effective.

♦ Change is driven by process *and* product. As the process-improvement program matured over the years, our concern for product attributes grew to equal our concern for process attributes. Product goals are always defined before process change is infused. Measures of product are thus as important as those of process (if not more so).

♦ Change is bottom-up. Although process-improvement analysts originally assumed they could work independently of developers, we have realized over the years that change must be guided by development-project experience. Direct input from developers as well as measures extracted from development activities are key factors in change.

♦ "People-oriented" technologies are emphasized, rather than automation. The most effective process changes are those that leverage the thinking of developers. These include reviews, inspections, Cleanroom techniques, management practices, and independent-testing techniques — all of which are driven by disciplined programmers and managers. Automation techniques have sometimes provided improvement, but people-driven approaches have had farther reaching impacts.

The SEL has invested approximately 11 percent of its total software budget into process-improvement. This expense includes project overhead, as well as overhead for data archiving and processing and process and product analysis. We have maintained detailed records so we can accurately record and report process-improvement costs.

Our investment in process-improvement has brought many benefits. The cost, defect rates, and cycle time of flight-dynamics software have decreased significantly since we started the program. Today, our software developers are building better software more efficiently — using many techniques and methods considered experimental only a few years ago. Their progress has been facilitated throughout by the SEL focus on defining organizational goals, expanding domain understanding, and judiciously applying new technology, allowing the FDD to maximize the lessons from local experience. ♦

REFERENCES

1. L. Landis et al., "Manager's Handbook for Software Development," Revision 1, Tech. Report SEL-84-101, Software Eng. Laboratory, Greenbelt, Md., 1989.
2. L. Landis et al., "Recommended Approach to Software Development," Revision 3, Tech. Report SEL-81-305, Software Eng. Laboratory, Greenbelt, Md. 1992.
3. W. Decker, R. Hendrick, and J. Valett, "Relationships, Models and Measurement Rules," Tech. Report SEL-91-001, Software Eng. Laboratory, Greenbelt, Md., 1991.
4. V.R. Basili and D.M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Trans. Software Eng.*, Nov. 1984, pp. 728-738.
5. V.R. Basili, "Software Development: A Paradigm for the Future," *Proc. Compsac*, IEEE CS Press, Los Alamitos, Calif., 1989, pp. 471-485.
6. V.R. Basili and S. Green, "Software Process Evolution at the SEL," *IEEE Software*, July 1994, pp. 58-66.

Victor Basili is a professor in the Institute for Advanced Computer Studies and the Computer Science Department at the University of Maryland. He is co-editor-in-chief of the *International Journal on Empirical Software Engineering.*

Marvin Zelkowitz is a professor in the Institute for Advanced Computer Studies and the Computer Science Department at the University of Maryland and has been involved with the SEL since its inception in 1976. His research interests include language design, environments, and formal methods.

Frank McGarry is a senior member of the executive staff at Computer Sciences Corporation. Previously at NASA, he was a founding director of the SEL in 1976.

Jerry Page is the vice president of the System Science Division at Computer Sciences Corporation. Until last year, he managed SEL activities within CSC.

Sharon Waligora has worked for the Computer Science Corporation since 1974 and directs the CSC branch of the SEL, leading efforts in software process improvement, process definition, and measurement activities.

Rose Pajerski has worked for the Goddard Space Flight Center for more than 20 years and directs the GSFC branch of the SEL. Her research interests include testing processes, systems management through measurement, and tailoring approaches for process-improvement programs.

Address questions about this article to Basili at the Department of Computer Science, University of Maryland, 4121 A.V. Williams, College Park, MD 20742; basili@cs.umd.edu.

Page intentionally left blank

THE EXPERIENCE FACTORY
STRATEGY AND PRACTICE

Victor R. Basili
basili@cs.umd.edu

Gianluigi Caldiera
gcaldiera@cs.umd.edu

Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, Maryland 20742

ABSTRACT

The quality movement, that has had in recent years a dramatic impact on all industrial sectors, has recently reached the systems and software industry. Although some concepts of quality management, originally developed for other product types, can be applied to software, its specificity as a product which is developed and not produced requires a special approach. This paper introduces a quality paradigm specifically tailored on the problems of the systems and software industry.

Reuse of products, processes and experience originating from the system life cycle is seen today as a feasible solution to the problem of developing higher quality systems at a lower cost. In fact, quality improvement is very often achieved by defining and developing an appropriate set of strategic capabilities and core competencies to support them. A strategic capability is, in this context, a corporate goal defined by the business position of the organization and implemented by key business processes. Strategic capabilities are supported by core competencies, which are aggregate technologies tailored to the specific needs of the organization in performing the needed business processes. Core competencies are non-transitional, have a consistent evolution, and are typically fueled by multiple technologies. Their selection and development requires commitment, investment and leadership.

The paradigm introduced in this paper for developing core competencies is the Quality Improvement Paradigm which consists of six steps:

1. *Characterize the environment*
2. *Set the goals*
3. *Choose the process*
4. *Execute the process*
5. *Analyze the process data*
6. *Package experience*

The process must be supported by a goal-oriented approach to measurement and control, and an organizational infrastructure, called Experience Factory. The Experience Factory is a logical and physical organization distinct from the project organizations it supports. Its goal is development and support of core competencies through capitalization and reuse of life cycle experience and products.

The paper introduces the major concepts of the proposed approach, discusses their relationship with other approaches used in the industry, and presents a case in which those concepts have been successfully applied.

This work was supported by NASA Grant NSG-5123 and by Hughes Applied Information Systems, Inc.

1. INTRODUCTION

The presence of software in almost every activity and institution is a characteristic of our society. Our dependence on software becomes evident when software problems and related events make the headlines of newspapers. However, this dependency on software, although highly visible, is not yet well understood by the business community. Software is still too often perceived as the easiest part of a system, the part that can be easily modified and adapted to fit to the main business of the organization.

This idea that "software is easy" or, ultimately, "cheap" is hard to eradicate, even when there is substantial evidence that it is not true anymore. In particular, there is a certain difficulty in dealing with software quality, both it terms of definition (What is quality software?) and implementation of quality programs (How can we produce quality software?).

The starting point of every discussion on software quality is the recognition that software is an industrial product whose quality can be managed in a similar way to the quality of other products or services. A software system is the result of the concurrent effort of teams of people working according to a traditional engineering paradigm (a conception phase followed by an implementation phase, very often with several iterations). In fact, we call "software engineering" the systematic approach to the development, operation and maintenance of software systems (and associated documentation and data).

As with every industrial product, the quality of software is defined as "fitness for use" over its lifetime. Therefore, the goal of a quality management program is to incorporate quality into a software system in the most economically convenient way, i.e., by designing a high quality system. The challenge of software quality is to implement techniques and programs in order to fill the existing gap between demand and our ability to produce high-quality software in a cost-effective way.

The software product, however, presents the following critical combination of characteristics:

- *Software is a logical aggregate of invisible parts*: The quality of such aggregate depends on the appropriateness of the logical structuring of the parts and on a precise and easy-to-understand documentation of this structure;

- *Software is designed for user applications which are expected to evolve continuously*: The quality of application software depends on the precise conceptual understanding of user needs, and on the adaptability of design to a changing environment; good communication between designers and users, and user perception are essential components of good software design;

- *Software is developed and not produced*: Each software product is like a prototype, therefore many statistical concepts that help us in measuring and controlling quality in industrial products do not apply completely to software products;

- *Software is a human based technology*: The quality of the software product is dependent on the individuals involved, therefore appropriate use of individual skills, individual satisfaction and motivation are key issues in achieving substantial improvements in quality and productivity.

We believe that the quality of a software system should and can be managed in two ways. First, the effectiveness of the software development process should be improved by reducing the amount of rework and reusing software artifacts across segments of a project or different projects. Second, plans for controlled, sustained, and continuous improvement should be developed and implemented based on facts and data.

But software engineering does not make extensive use of quantitative data. Therefore software quality management is based on a very immature and unstable paradigm. A major problem is that many data regarding the quality of a system can only be observed, and measured when the system is implemented. Unfortunately, at that stage the correction of a design defect requires the redesign of some, sometimes large and complex, components and is very expensive. In order to prevent the occurrence of expensive defects in the final product, quality management must focus on the early stages of the engineering process, in particular on the requirements analysis and design phases, and use quantitative data in order to record and support inspection and decision making. Those early stages are, however, the ones in which the process is less defined and controllable with quantitative data. Therefore, software engineering projects do not regularly collect data and build models based upon them.

There are many software project that can be considered successful from a quality point of view; generally this means that the techniques and procedures applied in the project have been effective, in particular those aimed at assuring quality. The goal of quality management is to make this success repeatable in other projects, by transferring the knowledge and the experience that are at the roots of that success to the rest of the organization. Therefore, a software organization that manages quality should have, besides the quality assurance infrastructure associated with each project, a corporate infrastructure that links together and transcends the single projects by capitalizing on successes and learning from failures.

Quality management and infrastructure, however, do not just happen; they must be planned and implemented by the organization through specific programs and investments. This paper is about the need for a strategic approach to software quality management, as a part of a corporate strategy for software, aimed at pursuing and improving quality as an organization and not as a group of individual projects.

We will motivate the need for such an approach, discuss it in the context of some of the most relevant concepts developed by the management disciplines, and provide a framework for a solution, which has been applied in practice with convincing results.

We believe there is no solution that can be mechanically transferred and applied to every organization (the famous "silver bullet"), and this applies also to the concepts presented in this paper. The proposed approach, however, can be used by every organization, after appropriate customization, in order to improve software quality in a controllable way.

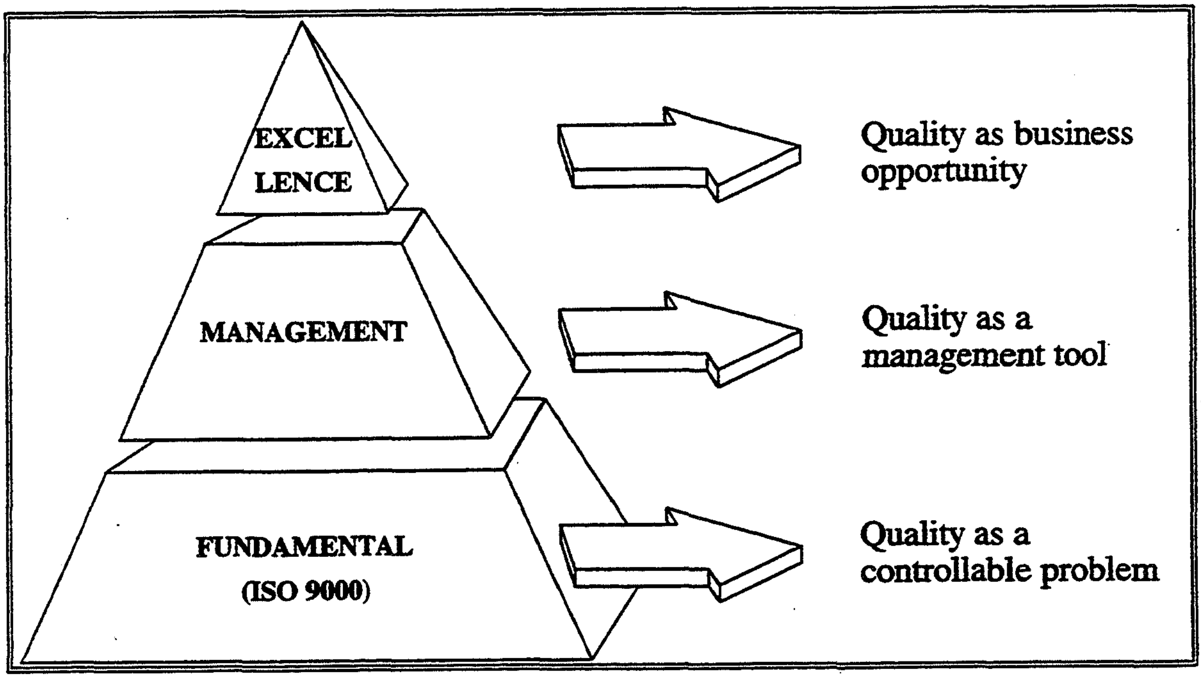
2. THE PROBLEM OF SOFTWARE QUALITY

Quality is the totality of characteristics of a product or service "that bear on its ability to satisfy stated or implied needs" [ISO1]. It is a multidimensional concept that includes the entity of interest (the product or service), the viewpoint on that entity (the user, the producer, a regulatory agency, etc.) and the quality attributes of that entity (the characteristics that make it fit for use). A recent international standards [ISO3] identifies the following characteristics:

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

In some cases, such as regulated environments in which some safety critical factors must be determined (aeronautics, nuclear power, etc.), these attributes are specified by a standard or a contract; but in the majority of cases they are identified and defined during the design process, and modified throughout the life cycle of the system. The ability of an organization to identify and define the quality attributes that are closer to the "stated or implied needs" of a user is the critical success factor in the market of the 90's.

Figure 1


EXCEL
LENCE
MANAGEMENT
FUNDAMENTAL
(ISO 9000)
Quality as business opportunity
Quality as a management tool
Quality as a controllable problem

Today the success of a software organization is measured by its cost/performance attributes: it delivers (or updates) the needed systems generally on time and without budget overruns. In the longer run, though, if we take into account today's market, characterized by shrinking budgets and increased global competition, we can expect, for the second half of the '90s, that the most successful organizations will probably be the ones that have been able to converge to better levels of productivity and quality. The influence of international standards such as the ISO 9000 Series [ISO2] is already evident. Many organizations are now seeking registration and the ability to develop quality systems in compliance with the requirements of the standard. Registration, however, is a means and not an end: spending resources on developing a quality system without a quality improvement program that uses it to gain a competitive advantage would be a waste of money. This is why, along with ISO 9000 registration programs, we see quality improvement programs being started. We can expect that in a few years all this movement will lead to a higher quality baseline for all the software that is being purchased and developed around the world. On top of this baseline the organizations will be able to build their own quality management programs and their continuous improvement strategies. In this way quality will complete its transformation from problem (search for defects) to tool (defined processes) to business opportunity used to distinguish an organization from its competitors(Figure 1).

At that point, the real advantage will come from the ability of the software organization to deliver solutions that not only satisfy, but also anticipate the needs of the system users, enhancing their business and adding a substantial amount of value to their products and services [Hamel and Prahalad, 1991]. Competition in the '90s is a more complex and dynamic playing field, in which the basic factors for success are the understanding of trends and the response to changing needs. The traditional rigidity of software organizations must to be adapted to the new ground rules. New professional skills, beyond the traditional programmer/analyst/manager triangle, are necessary in order to capitalize on the experience of the organization and work on specific lines of business instead of developing isolated products.

If we survey the approaches to software quality available to the industry, we see a variety of paradigms, mostly coming from the manufacturing industry.

Some organizations apply to their software processes an improvement process based on the Shewart-Deming Cycle [Deming, 1986]. This approach provides a methodology for managing change throughout the steps of a production process by analyzing the impact of those changes on the data derived from the process. The methodology is articulated in four phases:

- Plan: Define quality improvement goals and targets and determine methods for reaching those goals; prepare an implementation plan.
- Do: Execute the implementation plan and collect data.
- Check: Verify the improved performance using the data collected from the process and take corrective actions when needed.
- Act: Standardize the improvements and install them into the process.

Some organizations use the Total Quality Management (TQM) approach, which is a derivative of the PDCA method applied to all business processes in the organization [Feigenbaum, 1991]. Actually, more than a specific method TQM is a family of management philosophies based on the fact that quality is measured by the user of a product, and that everyone in the organization has specific responsibilities for the quality of the final outcome. Therefore, in TQM programs, quality improvements, identified during a preliminary characterization effort, are usually experimented by pilot groups and then institutionalized across the whole organization. The TQM approach usually results in the establishment of cross-functional quality improvement teams chartered to addressing specific quality improvements within a strategic quality plan developed by the top management.

A different approach is adopted by organizations that model their improvement on an external scale that is meant to represent the best practices in quality. The goals of the improvement program are, in this case, not internally generated but suggested by those best practices. A model of this kind, which is today very popular in both the USA and Europe, is the SEI Capability Maturity Model [SEI; Bootstrap] which measures the maturity of a software organization on the basis of its dependence on individual skills and on the presence of certain technologies. In a low maturity organization, the success of a task depends on the efforts of people involved in it, professionals and managers. Their ability to control risk, to solve or even prevent problems is the major asset of the organization. In a more mature organization, the success is based on the use of sound managerial and engineering techniques coordinated by a pervasive, well-defined set of processes for the execution of the needed tasks. At the highest level of maturity, the organization effectively capitalizes on its experiences and improves its processes. The improvement is achieved by bringing the organization through these levels of maturity.

All these approaches, and variations on them, have been used by the software industry, with mixed outcomes. Some outstanding successes have been reported, such as the one shown in Figure 2 [Dion, 1993], by combining those approaches. The major problem with all these approaches is that they either do not deal specifically with the nature of the software product (Deming Cycle, TQM) or, if they do, they assume that there is a consistent picture of what a good software product or process is (SEI model).

We argue that this is not enough for two reasons: the first one is that in order to be really effective a software quality program should deal with the nature of the software business itself; the second is that there is really no such thing as an explicit consistent picture of a good software product.

Figure 2

Raytheon Experience

Costs

- $ 1 million: Investment on the improvement program for each year (1987-1992)

Benefits

- $ 15.8 million: Rework costs eliminated

Return on investment

- 7.7 : 1 in 1990

Changes in % project time by cost type from 1988 to 1990

- Performance: Cost of building it right the first time, from 34% to 55%;
- Non conformance: Cost of rework, from 44% to 18%;
- Appraisal: Cost of testing, from 15% to 15%;
- Prevention: Cost of preventing non-conformance, from 7% to 12%.

On one hand, if we look at processes and technologies in isolation, like in the Plan/Do/Check/Act and TQM approaches, we have very little chance to get to the right level of abstraction that provides reusable units across different processes. Those approaches do not really build "model abstractions" because they manipulate the process explicitly. For instance: if we apply TQM to the order entry process, we have well defined elementary actions performed to enter an order. We can describe them with a flow chart and analyze the process, apply changes and assess their impact. We will have very soon many instances of that process to build a control chart and bring it under control. Unfortunately, the same approach cannot be used on a software process (e.g., structured design), which cannot be reduced to elementary units and is not replicated many times in a short period.

On the other hand, if we base our judgment upon an external model, like in the SEI and similar approaches, we might loose characteristics that make an organization's environment "special." Those characteristics are, in many cases, at the roots of the competitive advantage of that organization, therefore their loss is very damaging for the improvement program.

The approach that will be presented in the next sections of this paper is an attempt to learn from the successes obtained through the different paradigms sketched in this section, and to avoid the problems encountered in their application to software environments. It rests on the *lean enterprise concept* [Womack, 1989] by concentrating production and resources on value-added activities that represent the critical business processes of the organization. Such processes, after having been recognized, are conceptually redesigned in a modular way and associated with models, data, techniques and tools, in order to reuse them according to the needs and characteristics of specific projects. Total quality management [Feigenbaum, 1991] and Concurrent engineering [Dewan and Riedl, 1993] can be used in order to keep the structure efficient, responsive to the needs of any external entity (customer or supplier), and to make it rest upon partnership and participation, with many feedbacks and measures of the effectiveness of communication.

3. TOWARDS A MATURE SOFTWARE ORGANIZATION

If we analyze carefully some of the most successful and trend-setting business stories of the last 10 years [Stalk, Evans and Shulman, 1992], we can ascribe the reported successes to the application of four basic principles:

1. Business processes are the building blocks of the corporate strategy.

2. Competitive success depends on understanding and transforming the key business processes into strategic capabilities.

3. Strategic capabilities are created by sustained long-term investments in a support infrastructure that links together and transcends the business units.

4. A capability-based strategy must be sponsored by the top management of the corporation.

It is important to understand these four principles in the context of on a software organization.

The first principle sets the focus on business processes: this is consistent with the current tendency to emphasize the role of software processes in a successful project. Software is a logical aggregation and an intellectual product, which is, therefore, strongly dependent on the processes executed for developing or maintaining it. The analysis of those processes and the ability to reuse them in the appropriate context are a key competitive factor for every software organization. The corporate strategy must focus on identification and characterization of the key business processes used in developing and maintaining software, so that the business units, relieved from process related concerns, can focus more on the individual systems and services that are developed and delivered to individual clients.

The second principle is about "strategic understanding" of business processes. This means that the organization must understand its key business processes sufficiently to transform them into reusable units available to all its business units where needed. Not every process used in the organization has the characteristics of criticality that make it worthy of being transformed into a strategic capability: it is only from the analysis of the relationship between software processes and the mission of the organization that we can obtain a strategic level of understanding and a consolidated hypothesis of what should become a strategic capability. A system developer or integrator, for instance, produces software in order to deliver services to a particular group of users (e.g., electronic messaging). In this case a good cost/benefit ratio for the system or service is probably the most crucial issue. Therefore, the process of making acceptable estimates and to develop a plan based on them has a criticality definitely higher than the process of assuring the highest possible reliability. On the other hand, for a manufacturer of systems dependent on software (e.g., cellular phones) the cost/benefit ratio for software is distributed over a large number of products and therefore not extremely crucial for the single software package. Therefore, the process of assuring reliability has a higher criticality in comparison with the ability of making acceptable estimates of software costs.

The third and the fourth principles call for long-term investments and top management sponsorship, which translates into a permanent structure that develops and supports the reuse of the strategic capabilities. This is particularly new for the software industry, which is, in its large majority, driven by its business units and, therefore, has little ability to capitalize on experiences and capabilities. The required permanent structure is designed to provide a double support cycle:

- Control cycle: Support is provided to the everyday operation of software projects by comparing their current performance with the normal performance of similar projects;

- Capitalization cycle: Support is provided to future projects by continually learning from past experience and packaging this experience in a reusable way.

The development of strategic capabilities and competencies to support them, which is the key to all four of the presented principles, has, in the case of software, some basic requirements:

1. The organization must understand the software process and product.

2. The organization must define its business needs and its concept of process and product quality.

3. The organization must evaluate every aspect of the business process, including previous successes and failures.

4. The organization must collect and use information for project control.

5. Each project should provide information that allows the organization to have a formal quality improvement program in place, i.e. the organization should be able to control its processes, to tailor them to individual project needs and learn from its own experiences.

6. Competencies must be built in critical areas of the business by packaging and reusing clusters of experience relevant to the organization's business.

Part of the problem with the software business is the lack of understanding of the nature of software and software development. To some extent, software is different from most products. First of all, software is developed in the creative, intellectual sense, rather than produced in the manufacturing sense, i.e., each software system is developed rather than manufactured. Second, there is a non-visible nature to software. Unlike an automobile or a television set, it is hard to see the structure or the function of software, or to reason about it in a straightforward way. Therefore, the development of strategic capabilities in software requires understanding, model building and continuous feedback from the process.

This means that we must rethink the software business and expand our focus to a new set of problems and the techniques needed to solve them. Unfortunately, the traditional orientation of a software project is based on a case-by-case problem solving attitude; the development of strategic capabilities is based, instead, on an experience reuse and organizational sharing attitude. Figure 3 outlines the traditional focus of software development and problem solving, along with the expanded focus, proposed here for experience reuse.

The obvious question to be asked now is: are there any practical models that can be used in order to develop a strategy with the new focus? Such practical models can be software organizations that have tried to implement a capability-based strategy (or at least parts of it) and have carefully collected lessons learned and data, empirical studies in-the-large based on the scientific method (observe, formulate a hypothesis, measure and analyze, validate/refute the hypothesis) that have published their findings in a workable form, controlled experiments in-the-small.

Figure 3

Traditional Focus	New Extended Focus
• Delivering specific products and services	• Developing capabilities
• Decomposing a complex problem into simpler ones	• Unifying different solutions into more general ones
• Design/implementation process	• Analysis/Synthesis process
• Instantiation	• Generalization and formalization
• Validation and verification	• Experimentation

In Section 5 we will illustrate an experience that we, together with large part of the software engineering community, consider a practical model. The reason for choosing this one, besides the personal involvement of the authors of this paper with it, which provides us with considerable insight, is its almost unique blend of an organizational strategy aimed at continuous improvement, of a data-based approach to decision making, of an experimental paradigm, along with many years of continuous operation and data collection.

4 A STRATEGY FOR IMPROVEMENT

This section will present a strategy for improvement based on the development of strategic capabilities.

The main concept of this strategy is the central role played by a methodological framework addressing the development and improvement of strategic capabilities in form of reusable experience. This framework will be presented and discussed in the form of a process called "Quality Improvement Paradigm" [Basili, 1985]. In order to manage this conceptual framework we will need two tools

- A control tool: The goal-oriented approach to measurement addressing the issue of supporting the improvement process with quantitative information [Basili and Weiss, 1984];

- An organizational tool: An infrastructure aimed at capitalization and reuse of software experience and products [Basili, 1989].

In the next section we will see the methodological framework and the associated tools at work in a specific and practical example.

4.1 THE QUALITY IMPROVEMENT PARADIGM

A *strategic capability* is for us a corporate goal defined by the business position of the organization and implemented by key business processes. Strategic capabilities of software organizations are identified by the analysis of the categories of products/services that the organization intends to deliver in the future, of the level of project control needed in order to deliver those products/services at the appropriate level of quality, and of the strengths and weaknesses of the organization. Examples of strategic capabilities are

- Certify the reliability of the system that is being released for acceptance by the customer;
- Have a design-to-cost process, i.e., tailor the design of a software system to the amount of available resources (money, people, computers, etc.);
- Use flexible standards, i.e. standards that can, case by case, be tailored to the needs and the characteristics of each project;
- Have a short cycle-time, i.e., reduce the elapsed time from the identification of a solution to its deployment.

Strategic capabilities are always supported by *core competencies,* which are aggregate technologies tailored to the specific needs of the organization in performing the needed business processes. For instance: in order to certify the reliability of a system, an organization needs to master the quality assurance process owning competencies such as statistical testing and reliability modeling; in order to design to cost the organization must use flexible processes owning competencies such as process modeling and control, and concurrent engineering.

Core competencies have characteristics that distinguish them from simple technologies or clusters of technologies:

- They are non-transitional: although sometimes they appear to be fashionable concepts, they don't come and go;
- They have a consistent evolution: a paradigm for their interpretation and application is built over time and some consensus is generated throughout the user community;
- They require commitment, investment and leadership;
- They are typically fueled by and work with multiple technologies;
- They generally support multiple product/service lines.

The acquisition of core competencies that support the strategic capabilities is the goal of the process we will present in this section. If a competency is a key factor in a strategic capability, the organization must be sure to own, control and properly maintain this competency at state-of-the-art level, and know how to tailor it to the characteristics of specific projects and business units.

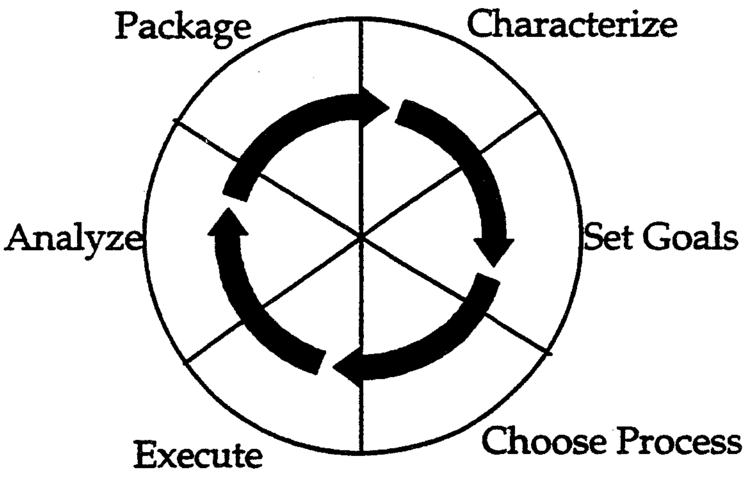
Strategic capabilities come into the improvement process as constituents of characteristics and goals. On the basis of the characteristics of the environment and of the transformation of those capabilities into specific goals for the software organization, the improvement paradigm provides a disciplined way to build the competencies necessary to support those capabilities.

The improvement process is articulated into the following six steps (Figure 4):

1. *Characterize*: Understand the environment based upon available models, data, intuition, etc. Establish baselines with the existing business processes in the organization and characterize their criticality.

2. *Set Goals*: On the basis of the initial characterization and of the capabilities that have a strategic relevance to the organization, set quantifiable goals for successful project and organization performance and improvement. The reasonable expectations are defined based upon the baseline provided by the characterization step.

Figure 4



3. *Choose Process*: On the basis of the characterization of the environment and of the goals that have been set, choose the appropriate processes for improvement, and supporting methods and tools, making sure that they are consistent with the goals that have been set.

4. *Execute*: Perform the processes constructing the products and providing project feedback based upon the data on goal achievement that are being collected. The processes will be

executed according to the needs dictated by the problem and to the process chosen in the previous phase.

5. *Analyze*: At the end of the execution, analyze the data and the information gathered to evaluate the current practices, determine problems, record findings, and make recommendations for future project improvements.

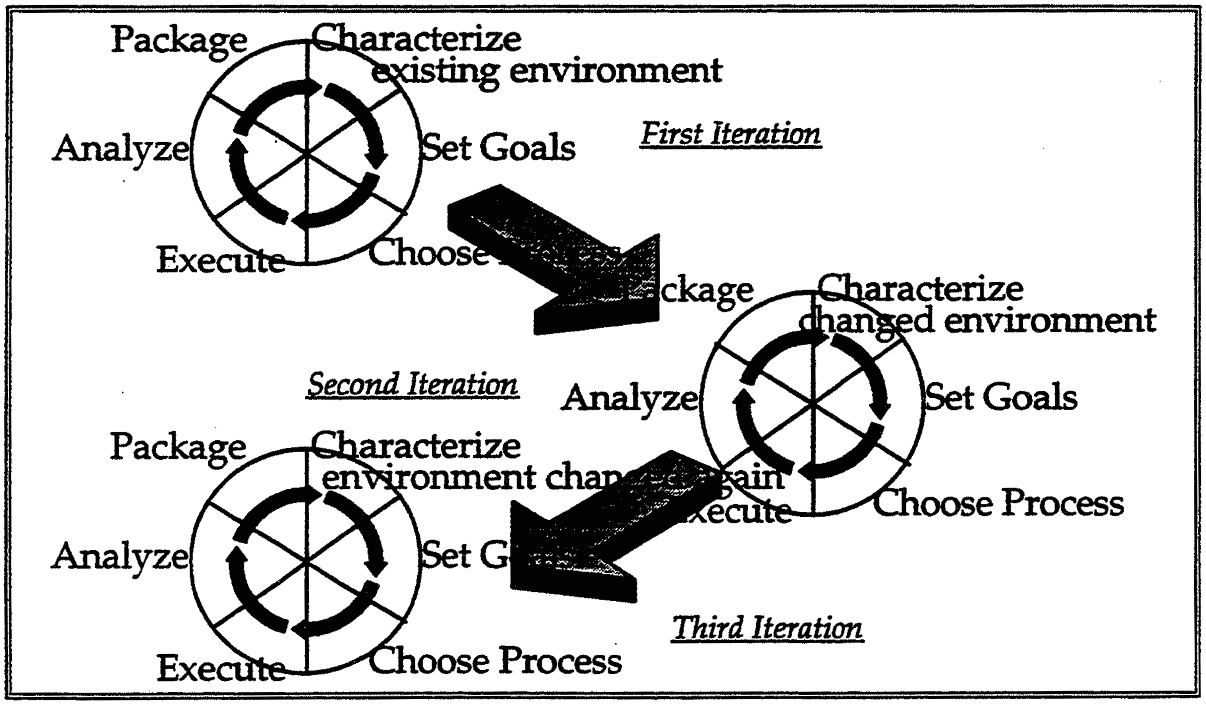
6. *Package*: Consolidate the experience gained in the form of new, or updated and refined, models and other forms of structured knowledge gained from this and prior projects, and store it in an experience base so it is available for future projects.

The Quality Improvement Paradigm implements the two major cycles, control and capitalization, introduced in section 3:

- The project feedback cycle (control cycle) is the feedback that is provided to the project during the execution phase: whatever the goals of the organization, the project should use its resources in the best possible way; therefore quantitative indicators at project and task level are useful in order to prevent and solve problems, monitor and support the project, realign the process with the goals;
- The corporate feedback cycle (capitalization cycle) is the feedback that is provided to the organization and has the purpose of
    - Providing analytical information about project performance at project completion time by comparing the project data with the nominal range in the organization and analyzing concordance and discrepancy;
    - Understanding what happened, capturing experience and devising ways to transfer that experience across domains;
    - Accumulating reusable experience in the form of software artifacts that are applicable to other projects and are, in general, improved based on the performed analysis.

The execution of the quality improvement paradigm by an organization is structured as an iterative process that repeatedly characterizes the environment, sets appropriate goals and chooses the process in order to achieve those goals, then proceeds with the execution and the analytical phases. At each iteration characteristics and goals are redefined and improved (Figure 5).

Figure 5



The reader has probably realized at this point that there is a deep similarity between the QIP and the Total Quality Management (TQM) philosophy. Figure 6 outlines some other correspondences between the two models.

The relationship between the QIP and the Plan/Do/Check/Act cycle is even closer. Both approaches are an offspring of the modern scientific method: first an hypothesis is generated, then an experiment is planned in order to validate the hypothesis, data are collected and analyzed, and the hypothesis is evaluated. The concept of feedback is also critical to both approaches: during the execution of the processes that have been planned and at the end of the execution data are analyzed in order to understand the impact of the changes introduced into the process. The real major difference between the two approaches appears at the end of the cycle: the PDCA approach incorporates the changes into the normal operation of the process, while the QIP develops a series of models that reflect the changes. This is due, as we said before, to the relatively smaller number of process instances that we have in the case of a software process, when compared with a manufacturing process.

Figure 6

TQM Total Quality Management	QIP Quality Improvement Paradigm
• Implements a corporation-wide quality improvement program	• Implements a program for reuse and improvement of software experience, artifacts, and processes
• Focuses on customer satisfaction and partnership for quality	• Focuses on customer satisfaction and partnership for quality
• Customers are both external and internal to the organization	• Capitalizes on project achievements
	• Customers are both external and internal to the organization
• Develops a flexible corporate culture	• Incorporates flexibility into the software process and product
• Bases decision making on facts	• Bases decision making on facts and data collected across different projects

4.2 THE GOAL-ORIENTED MEASUREMENT

The Goal/Question/Metric Approach [Basili and Weiss, 1984; Basili and Rombach, 1988] provides a method to identify and control key business processes in a measurable way. It is used to define metrics over the software project, process and product in such a way that the resulting metrics are tailored to the organization and to its goals, and reflect the quality values of the different viewpoints (developers, users, operators, etc.).

The result of the application of the Goal/Question/Metric Approach is the specification of a measurement system targeting a particular set of issues and a set of rules for the interpretation of the measurement data. The resulting measurement model has three levels:

1. Conceptual level (GOAL): A goal is defined for an object, for a variety of reasons, with respect to various models of quality, from

various points of view, relative to a particular environment. Objects of measurement include

- Products: Artifacts, deliverables and documents that are produced during the system life cycle; E.g., specifications, designs, programs, test suites.

- Processes: Software related activities normally associated with time; E.g., specifying, designing, testing, interviewing.

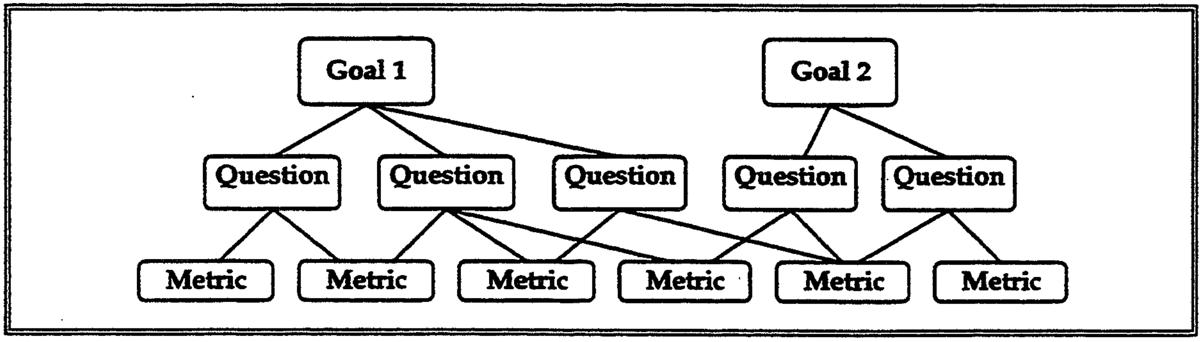
- Resources: Items used by processes in order to produce their outputs; E.g., personnel, hardware, software, office space.

- Knowledge objects: Models of the behavior of other items derived from past observations; E.g., resource models, reliability models.

2. Operational level (QUESTION): A set of questions is used to define in a quantitative way the goal and to characterize the way the specific goal is going to be interpreted based on some characterizing model. Questions try to characterize the object of measurement (product, process, resource, knowledge object) with respect to a selected quality issue and to determine its quality from the selected viewpoint.

3. Quantitative level (METRIC): A set of data is associated with every question in order to answer it in a quantitative way.

Figure 7



A GQM model is a hierarchical structure (Figure 7) starting with a goal (specifying purpose of measurement, object to be measured, issue to be measured, and viewpoint from which the measure is taken). In order to give an example of application of the Goal/Question/Metric approach, let's suppose we want to improve the timeliness of change request processing during the maintenance phase of the life-cycle of a system. The resulting goal will specify a purpose (improve), a process (change request processing), a viewpoint (project manager), and a quality issue (timeliness) (Figure 8). The goal is refined into several questions that usually break down the issue into its major components. The goal of the example can be refined to a series of questions, about, for instance, turn-around time and resources used. Each question is then refined into metrics. The questions of our example can, for instance, be answered by metrics comparing specific turn-around times with the average ones. The same metric can be used to answer different questions under the same goal. Several GQM models can also have questions and metrics in common, making sure that, when the measure is actually taken, the different viewpoints are taken into account correctly (i.e., the metric might have different values when taken from different viewpoints). The Goal/Question/Metric Model of our example is shown in Figure 8.

Figure 8

Goal	Purpose	Improve
	Issue	the timeliness of
	Object (process)	change request processing
	Viewpoint	from the project manager's viewpoint
Question		Is the performance of the process improving?
Metrics		$\frac{\text{Current average turnaround time}}{\text{Baseline average turnaround time}}$ Subjective rating of manager's satisfaction
Question		Is the distribution of resources changing?
Metrics		Percent effort spent on problem analysis Percent effort spent on solution identification Percent effort spent on solution implementation Percent effort spent on solution testing

In conclusion, we can also use the Goal/Question/Metric Approach for long range corporate goal setting and evaluation. The evaluation of a project can be enhanced by analyzing it in the context of several other projects. We can expand our level of feedback and understanding by defining the appropriate synthesis procedure for transforming specific, valuable information into more general packages of experience. As a part of the Quality Improvement Paradigm, we can learn more about the definition and application of the Goal/Question/Metric Approach in a formal way, just as we would learn about any other experiences.

4.3 EXPERIENCE FACTORY: THE CAPABILITY-BASED ORGANIZATION

The concept of the Experience Factory [Basili, 1989] has been introduced in order to institutionalize the collective learning of the organization that is at the root of continuous improvement and competitive advantage.

Reuse of experience and collective learning cannot be left to the imagination of single, very talented, managers: in a capability-based organization they become a corporate concern like the portfolio of businesses or the company assets. *The experience factory is the organization that supports reuse of experience and collective learning by developing, updating and providing upon request clusters of competencies to the project organizations* . We call these clusters of competencies, experience packages. The project organizations supply the experience factory with their products, the plans, processes and models used in their development, and the data gathered during development and operation; the experience factory transforms them into reusable units and supplies them to the project organizations, together with specific support made of monitoring and consulting.

The experience factory organization can be a logical and/or physical organization, but it is important that its activities are clearly identified and made independent from those of the project organization.

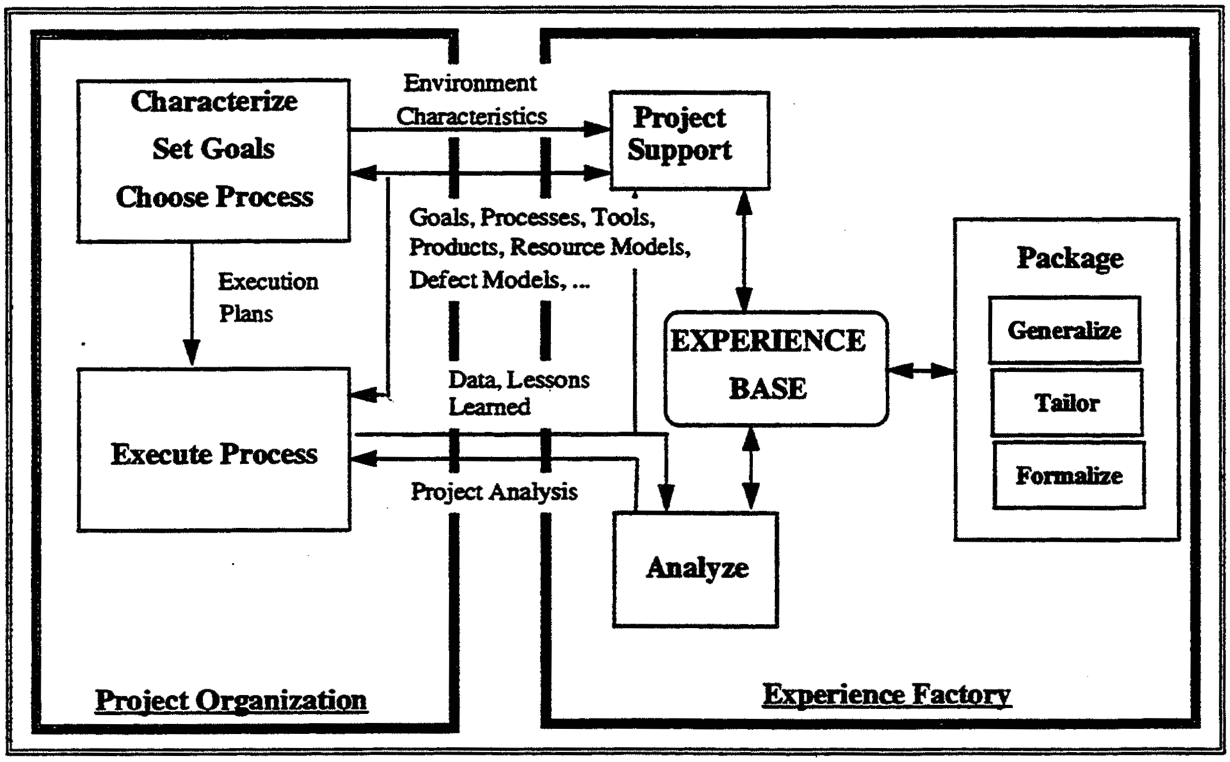
As we have seen at the beginning of this paper, the packaging of experience is based on tenets and techniques that are different from the problem solving activity used in project development. Therefore the projects and the factory will have different process models: each project will choose its process model based upon the characteristics of the software product that will be delivered, while the experience factory will define (and change) its process model based upon the nature of the work, and organizational and performance issues.

Figure 9 provides a high-level picture of the experience factory organization and highlights activities and information flows among the component sub-organizations.

The project organization, whose goal is to produce and maintain software, provides the experience factory with project and environment characteristics, development data, resource usage information, quality records, and process information. This provides feedback on the actual performance of the models processed by the experience factory and utilized by the project.

The experience factory provides direct feedback to each project, together with goals and models tailored from similar projects. It also produces and provides upon request baselines, tools, lessons learned, and data, parametrized in some form in order to be adapted to the specific characteristics of a project. The support personnel sustain and facilitate the interaction between developers and analysts, by saving and maintaining the information, making it efficiently retrievable, and controlling and monitoring the access to it.

Figure 9



The main product of the experience factory is a set of core competencies packaged as aggregates of technologies. Figure 10 shows some examples of core competencies and the corresponding aggregation of technologies:

Core competencies can be implemented in a variety of formats. We call these formats "experience packages". Their content and structure vary based upon the kind of experience clustered in it. There is, generally, a central element that determines what the package is: a software life cycle product or process, a mathematical relationship, an empirical or theoretical model, a data base, etc. We can use this central element as identifier of the experience package and produce a taxonomy of experience packages based upon the characteristics of this central element; e.g.:

- Product packages: Programs, Architectures, Designs;

Figure 10

Core Competencies	Aggregate Technologies
• Use of an integrated software engineering environment tailored to one or more specific application domains	⇦ Tool integration ⇦ Domain analysis and architectures ⇦ Data sharing and communication in heterogeneous environments
• Availability of reusable components (modules, algorithms, architectures) and tools portable across different platforms	⇦ Reuse libraries, mechanisms and methods ⇦ Domain analysis and architectures ⇦ Object-oriented techniques
• Availability and use of a software management environment based on "local" data for estimate, control and prediction of projects	⇦ Measurement and data collection and analysis ⇦ Data and process modeling ⇦ Defect counting, categorization and analysis

- Tool packages: Constructive and Analytic Tools;

- Process packages: Process Models, Methods;

- Relationship packages: Cost and Defect Models, Resource Models, etc.;

- Management packages: Guidelines, Decision Support Models;

- Data packages: Defined and validated data, Standardized data, etc.

The operation of the two components is based on the Quality Improvement Paradigm introduced in the previous section. Each component performs activities in all six steps, but for each step one component has a leadership role.

In the first three phases (Characterize, Set Goals, and Choose Process) the focus of the operation is on planning, therefore the project organization has a leading role and is supported by the analysts of the experience factory. The outcome of these three phases is, on the project organization side, a project plan associated with a management control framework, and on the experience factory side a support plan also associated with a management control framework. The project plan describes the phases and the activities of the project, with their products, mutual dependencies, milestones and resources. As far as the experience factory side is concerned, the plan describes the support that the experience factory will provide for each phase and activity, also with products, mutual dependencies, milestones and resources. The two parts of the plan are obviously integrated although executed by different components. The management control frameworks are composed of data (metrics) and models for monitoring the execution of the plan.

In the fourth phase (Execute) the focus of the operation is on delivering the product or service assigned to the project organization, therefore the project organization has again a leading role, and is supported by the experience factory. The outcome of this phase is the product or service, which represent a set of potentially reusable products, processes, and experiences.

In the fifth and the sixth phases (Analyze and Package) the focus of the operation is on capturing project experience and making it available to future similar projects, therefore the experience factory has a leading role and is supported by the project organization that is the repository of that experience. The outcomes of these phases are lessons learned with recommendations for future improvements, and new or updated experience packages incorporating the experience gained during the project execution.

Structuring a software development organization as an experience factory offers the ability to learn from every project, constantly increase the maturity of the organization and incorporate new technologies into the life cycle. In the long term, it supports the overall evolution of the organization from a project-based one, where all activities are aimed at the successful execution of current project tasks, to a capability-based one, which executes those tasks and capitalizes on their execution.

Some important benefits that an organization derives from structuring itself as an experience factory are

- To establish an improvement process for software substantiated and controlled by quantitative data;
- To produce a repository of software data and models which are empirically based on the everyday practice of the organization;
- To develop an internal support organization that represents a limited overhead and provides substantial cost and quality performance benefits;
- To provide a mechanism for identifying, assessing and incorporating into the process, new technologies that have proven to be valuable in similar contexts;
- To incorporate reuse into the software development process and support it;
- To approach in a more software specific way a Total Quality Management program.

The concept of experience factory is an extension and a redefinition of the concept of software factory, as it has evolved from the original meaning of integrated environment to the one of flexible software manufacturing environment [Cusumano, 1991]. The major difference is that, while the software factory is thought of as an independent unit producing code by using an integrated development environment, the experience factory handles all kind of software-related experience. The software factory can be seen as a part of the experience factory, recognizing in this way that its potential benefits can be fully exploited only within this framework.

5. IMPROVEMENT IN PRACTICE: THE NASA SOFTWARE ENGINEERING LABORATORY

In this section we will present and discuss a practical example of experience factory organization. We will show how its operation is based on the Quality Improvement Paradigm and we will use the case of a specific technology in order to illustrate the execution of the steps of the paradigm.

The organization that provides the example is the Software Engineering Laboratory (SEL) at NASA Goddard Space Flight Center. The laboratory was established in 1976 as a cooperative effort among the Department of Computer Science of the University of Maryland, The National Aeronautic and Space Administration Goddard Space Flight Center (NASA/GSFC), and the Computer Sciences Corporation (CSC). The goal of the SEL was to understand and improve key software development processes and products within a specific organization, the Flight Dynamics Division.

In general, the goals, the structure and the operation of the SEL have evolved from an initial stage, a laboratory dedicated to experimentation and measurement, to a full scale organization aimed at reusing experience and developing strategic capabilities. At the same time, the awareness of the quality improvement process used in the laboratory has generated the operational paradigm described in this paper as Quality Improvement Paradigm. Today the SEL represents a practical and operational example of experience factory [Basili et al., 1992].

The current structure of the SEL is based on three components:

- *Developers*, who provide products, plans used in development, and data gathered during development and operation (the Project Organization);

- *Analysts*, who transform these objects provided by the developers into reusable units and supply them back to the developers; they provide specific support to the projects on the use of the analyzed and synthesized information, tailoring it to a format which is usable by and useful to a current software effort (the Experience Factory proper);

- *Support infrastructure,* which provides services to the developers, on one hand, by supporting data collection and retrieval, and to the analysts, on the other hand, by managing the library of stored information and its catalogs (the Experience Base Support).

The activities of these three sub-organizations, although not separated and independent from each other, have their own goal and process models and plans. Figure 11 outlines the difference in focus among the three sub-organizations.

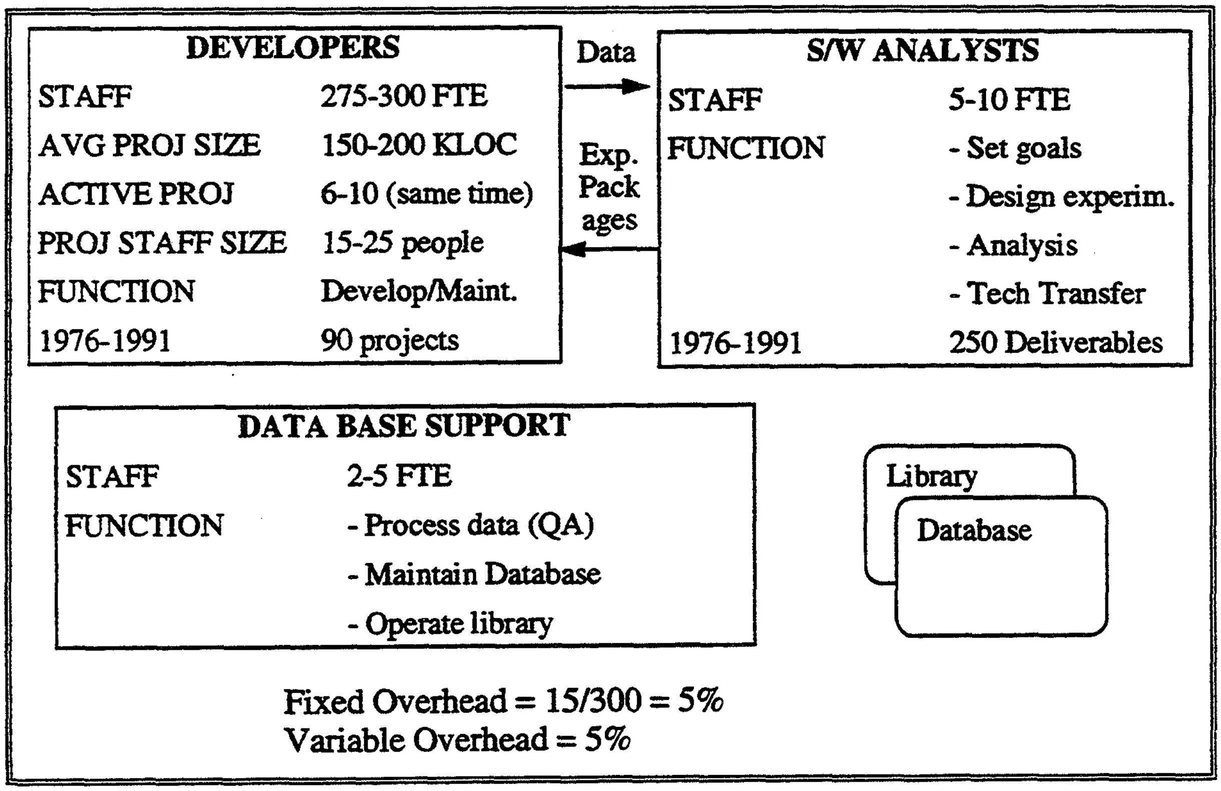
Figure 11

DEVELOPERS FOCUS	ANALYSTS FOCUS	SUPPORT INFRASTRUCTURE FOCUS
Software development	Experience packaging	Support developers and analysts
Single application	Application domain	Organization
Decompose a problem into simpler ones	Generalize and formalize solutions and products	Categorize and organize
Tailor and apply the process	Analyze and synthesize the process	Store and retrieve the process information
Validation and verification	Experimentation	Efficient retrieval

Figure 12 gives an idea of the overall size of the organization and of it components.

We will now show the operation of the SEL following the development of a particular core competence through the six steps of the improvement paradigm.

Figure 12



DEVELOPERS	
STAFF	275-300 FTE
AVG PROJ SIZE	150-200 KLOC
ACTIVE PROJ	6-10 (same time)
PROJ STAFF SIZE	15-25 people
FUNCTION	Develop/Maint.
1976-1991	90 projects

S/W ANALYSTS	
STAFF	5-10 FTE
FUNCTION	- Set goals - Design experim. - Analysis - Tech Transfer
1976-1991	250 Deliverables

DATA BASE SUPPORT	
STAFF	2-5 FTE
FUNCTION	- Process data (QA) - Maintain Database - Operate library

Fixed Overhead = 15/300 = 5%
Variable Overhead = 5%

In the late 80's the software engineering community, within and outside NASA, was discussing, among other technologies, the Ada programming language environment and technology [Ada, 1983]: the language had been developed under a major effort of the US Department of Defense and its application was being considered also in areas outside DoD. NASA was, at that time, considering the use of the Ada technology in some major projects such as the Space Station. More and more systems would have used Ada as development environment, and many organizations would have to be involved with it. In consideration of this fact Ada had to be transformed from simple technology to core competence for the software development organizations within NASA.

Associated with Ada there was the issue of object-oriented technologies. It is not very important for our discussion that our reader knows what is an object-oriented design technique. Anyway, Figure 13 provides some basic characteristic elements [Sommerville, 1992] of the object-oriented approach.

Figure 13

Characteristics of the Object-Oriented Approach

- A system is seen as a set of objects having at each time a specific state and behavior
- Objects interact with each other by exchanging messages
- Objects are organized into classes based on common characteristics and behaviors
- All information about the state or the implementation of an object is held within the object itself and cannot be deliberately or accidentally used by other objects

The Ada language environment implements several of those features and can be, to a certain extent, considered object-oriented. The design of systems to be implemented in Ada definitely takes advantage of the concepts of object-oriented design. Therefore, from the beginning, there was the impression in the SEL that the two technologies should be packaged together into a core competence supporting the strategic capability of delivering systems with better quality and lower delivery cost. After recognizing that this capability had a strategic value for the organization, the SEL selected Ada and the object-oriented design technology for supporting it, measured its benefits, and provided supporting data to the decision of using the technology.

The process followed is illustrated in the following steps according to the QIP:

1. Characterize: In 1985, the SEL had achieved a good understanding of how software was developed in the Flight Dynamics Division. The development processes had been defined and models had been built in order to improve the manageability of the process. The standard development methodology, based on the traditional design and build approach, had been integrated with concepts aimed at continuously evolving systems by successive enhancements.

2. Set Goals: Realizing that object-oriented techniques, implemented in the design and programming environments that support new languages, like C++ and Ada, offered potential for major improvements in the areas of productivity, quality and reusability of software products and processes, the SEL decided to develop a core competence around object-oriented

design and the use of the programming language Ada. The first step was to set up expectations and goals against which results would be measured. The SEL well-established baseline and set of measures provided an excellent basis for comparison. Expectations included

- A change in the effort distribution of development activities: an increase of the effort on early phases, e.g., design, and a decrease of the effort on late phases, e.g., testing;
- Increased reuse of software modules, both verbatim and with modification;
- Decreased maintenance costs due to the better quality of reusable components;
- Increased reliability as a result of lower global error rates, fewer high-impact interface errors, and fewer design errors.

3. Choose process: The SEL decided to approach the development of the desired core competence by experimenting with Ada and object-oriented design in a "real" project. Two version of the same system would be developed

System A: To be developed using FORTRAN and following the standard methodology based on functional decomposition. This system will become operational and its development will follow the ordinary schedule constraints.

System B: To be developed using Ada and following an object-oriented methodology called OOD. This system will not become operational.

The data derived from the development of System B would be compared with those derived from the development of System A. Particular attention would be dedicated to quality and productivity data. The data collection and comparison would be based on the Goal Question Metric Model shown in Figure 14.

Figure 14

Goal	Purpose	Evaluate the impact of
	Object	the object-oriented approach and Ada
	Issue	on the quality and productivity
	Viewpoint	within the Flight Dynamics Division
Question	1	What is the impact on the cost to develop software?
Metrics	1.1	Number of hours per statement developed for System A
	1.2	Number of hours per statement developed for System B
Question	2	What is the impact on the cost to deliver software?
Metrics	2.1	Number of hours per statement included in System A
	2.2	Number of hours per statement included in System B
Question	3	What is the impact on the quality of the delivered software?
Metrics	3.1	Number of defects per 1000 lines of code in System A
	3.2	Number of defects per 1000 lines of code in System B
Question	4	What was the amount of reuse that occurred?
Metrics	4.1	Percentage of reused code

4. Execute: System A and B were implemented and the desired metrics were collected. During the development changes had to be applied to the approach that was used for using Ada and also adaptations had to be made in order to use OOD. For instance: some review procedures that were particularly suited for a design based on functional decomposition did not fit the approach used for System B. Therefore new review procedures were drafted for that development.

5. Analyze: The data collected based on the previous GQM model showed an increase of the cost to develop (Metrics 1.1 and 1.2) that was interpreted as due on one hand to the inexperience of the organization with the new technology and on the other hand to the intrinsic characteristics of the technology itself. The data also showed an increase in the cost to deliver (Metrics 2.1 and 2.2) interpreted as due to the same

causes. The overall quality of System B showed an improvement over System A (Metrics 3.2 and 3.1) in terms of a substantially lower error density. Reuse data across systems (Metric 4.1) were obviously not available for System B because of the new implementation technology. The comparative data are shown in Figure 15.

Figure 15

Measure	System A	System B
Cost to develop (Hrs per Stm)	0.70	1.00
Cost to deliver (Hrs per Stm)	0.65	1.00
Defect density (Def. per 1000 lines of code)	3.90	1.80
Reuse (%)	30%	N/A

6. Package: The laboratory tailored and packaged an internal version of the methodology which adjusted and extended OOD for use in a specific environment and on a specific application domain. Commercial training courses, supplemented with limited project-specific training, constituted the early training in the techniques. The laboratory also produced experience reports containing the lessons learned using the new technology and recommending refinements to the methodology and the standards.

The data collected from the first execution of the process were encouraging, especially on the quality issue, but not conclusive. Therefore new executions were decided and carried over in the following years. In conjunction with the development methodology, a programming language style guide was developed, that provided coding standards for the local Ada environment. At least 10 projects have been completed by the SEL using an object-oriented technology derived from the one used for System B, but constantly modified and improved. The size of single projects, measured in thousand lines of source code (KSLOC), ranges from small (38 KSLOC) to large (185 KSLOC). Some characteristics of an object-oriented development, using Ada, emerged early and have remained rather constant: no significant change has been observed, for instance, in the effort distribution or in the error classification. Other characteristics emerged later and took time to stabilize: reuse has increased dramatically after the first projects, going from a traditionally constant figure of 30% reuse across different projects, to a current 96% (89% verbatim reuse).

Over the years the use of the object-oriented approach and the expertise with Ada have matured. Source code analysis of the systems developed with the new technology has revealed a maturing use of key features of Ada that have no equivalent in the programming environments traditionally used at NASA. Such features were not only used more often in more recent systems, but they were also used in more sophisticated ways, as revealed by specific metrics used to this purpose. Moreover, the use of object-oriented design and Ada features has stabilized over the last 3 years, creating an SEL baseline for object-oriented developments.

The charts shown in Figure 16 represent the trend of some significant indicators.

The cost to develop code in the new environment has remained higher than the cost to develop code in the old one. However, because of the high reuse rates obtained through the object-oriented paradigm, the cost to deliver a system in the new environment has significantly decreased and lies now well below the old cost to deliver.

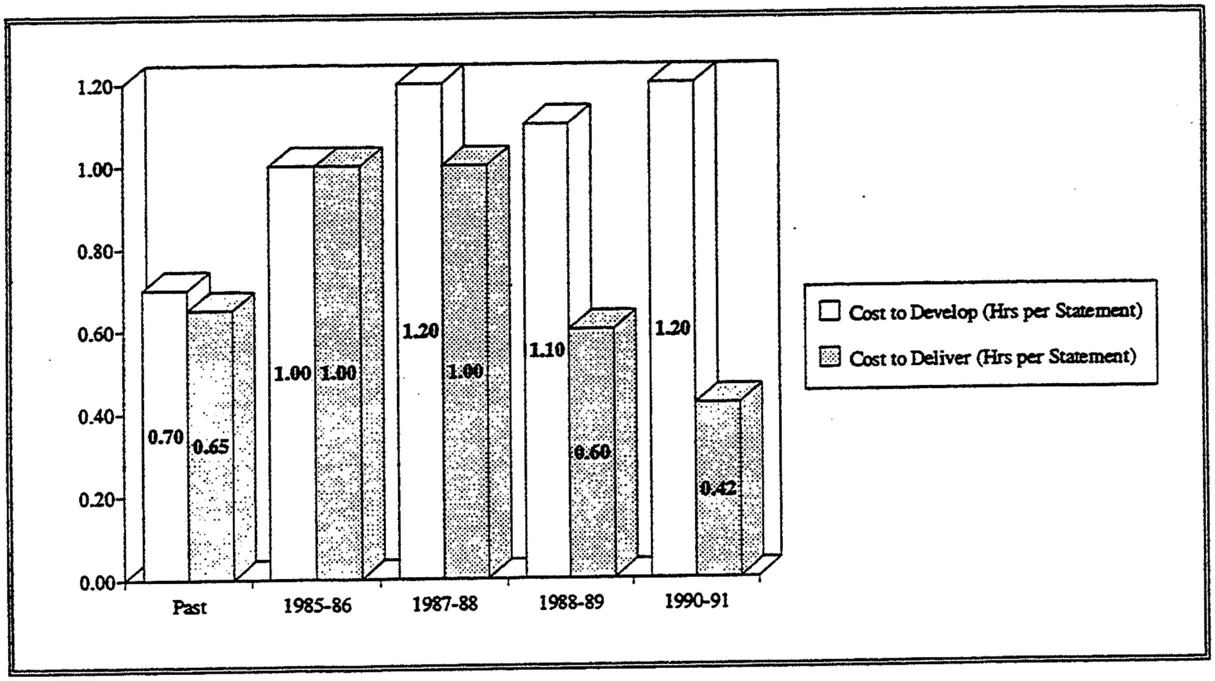
The reliability of the systems developed in the new environment has improved over the years with the maturing of the technology. Although the error rates were significantly lower than the traditional ones, they have continued to decrease even further: again, the high level of reuse in the later systems is a major contributor to this greatly improved reliability.

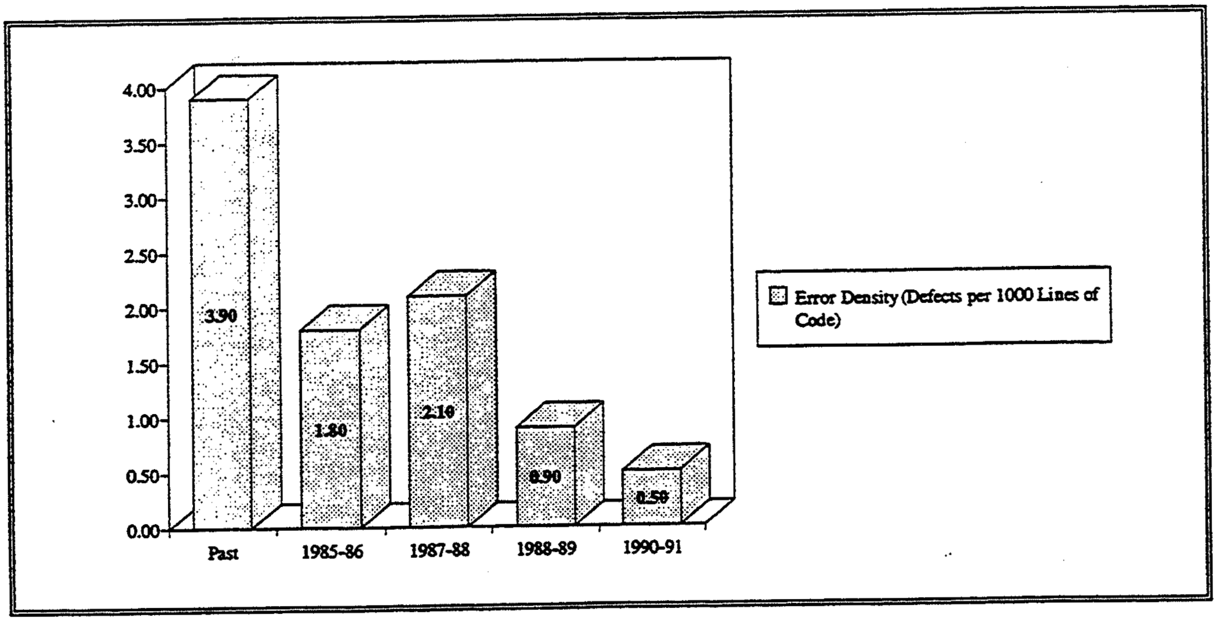
Because of the stabilization of the technology and apparent benefit to the organization, the object-oriented development methodology has been packaged and incorporated into the current technology baseline and is a core competence of the organization. And this is where things stand today.

Although the technology of object-oriented design will continue to be refined within the SEL, it has now progressed through all stages, moving from a candidate trial methodology to a fully integrated and packaged part of the standard methodology, ready for further incremental improvement.

The example we have just shown illustrates also the relationship between a competence (object-oriented technology) and a target capability (deliver high quality at low cost), and shows how innovative technologies can enter the production cycle of mature organizations in a systematic way. Although the topic of technology transfer is not within the scope of this paper, it is clear from the SEL example that the model we derive from it outlines a solution to some major technology transfer issues.

Figure 16


1.20
1.00
0.80
0.60
0.40
0.20
0.00
0.70
0.65
1.00
1.00
1.20
1.00
1.10
0.60
1.20
0.42
Past
1985-86
1987-88
1988-89
1990-91
□ Cost to Develop (Hrs per Statement)
▨ Cost to Deliver (Hrs per Statement)


4.00
3.50
3.00
2.50
2.00
1.50
1.00
0.50
0.00
3.90
1.80
2.10
0.90
0.50
Past
1985-86
1987-88
1988-89
1990-91
▨ Error Density (Defects per 1000 Lines of Code)

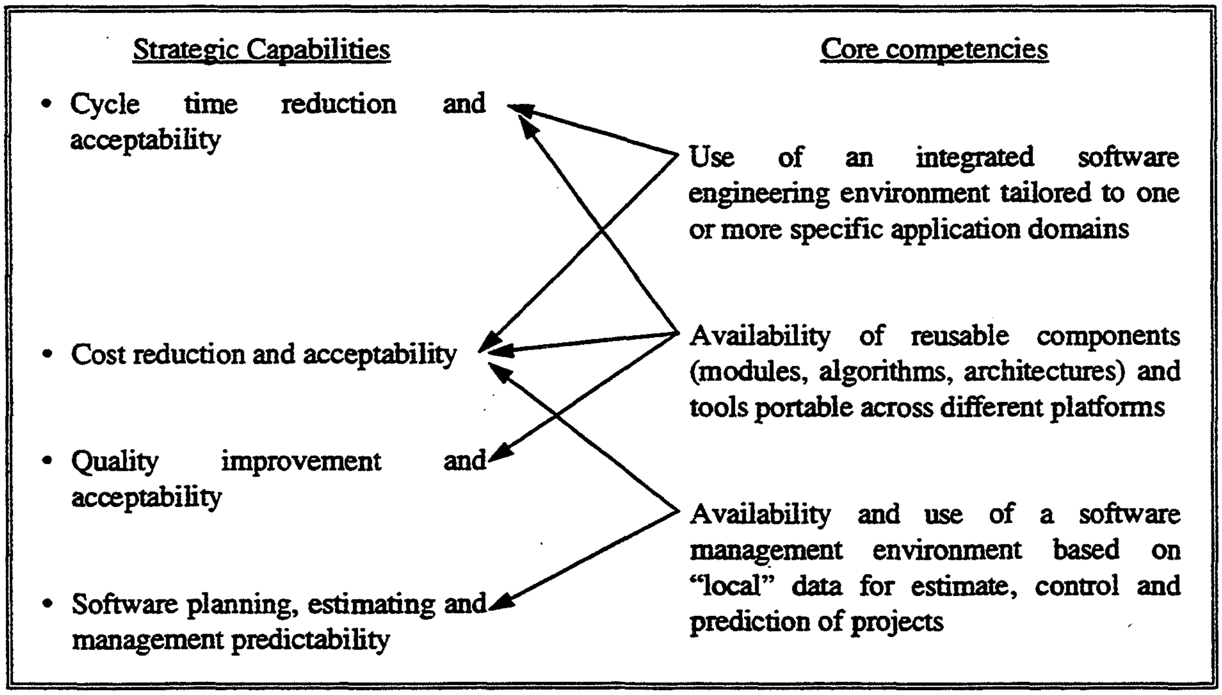
The purpose of an experience factory organization is larger than technology transfer: it is capability transfer and reuse. If these capabilities are already consolidated into a technology, available within the organization or outside it, then the process is a process of technology transfer. If the capabilities are present in the organization as informal experience, products prepared for other purposes, and lessons learned, then the process is different from technology transfer.

6. CONCLUSIONS

Clearly the nineties will be the quality era for software and there is a growing need to develop or adapt quality improvement approaches to the software business. Our approach to software quality improvement, as it has been presented in this paper, is based on the exploitation and reuse of the critical capabilities of an organization across different projects based on business needs.

The relationship between core competencies and strategic capabilities is established by the kind of products and services the organization wants to deliver and is specified by the strategic planning process. A possible mapping is shown as an example in Figure 17, in the case of an organization whose main business is development of systems and software for user applications.

Figure 17



Strategic Capabilities

- Cycle time reduction and acceptability
- Cost reduction and acceptability
- Quality improvement and acceptability
- Software planning, estimating and management predictability

Core competencies

Use of an integrated software engineering environment tailored to one or more specific application domains

Availability of reusable components (modules, algorithms, architectures) and tools portable across different platforms

Availability and use of a software management environment based on "local" data for estimate, control and prediction of projects

In this paper we have shown, through the NASA example, that all these ideas are practically feasible and have been successfully applied in a production environment in order to create a continuously improving organization.

But what does "continuously improving organization" really mean? It is an organization that can manipulate its processes to achieve various product characteristics. This requires that the organization has a process and an organizational structure to

- Understand its processes and products;
- Measure and model its business processes;
- Define process and product quality explicitly, and tailor the definitions to the environment;
- Understand the relationship between process and product quality;
- Control project performance with respect to quality;
- Evaluate project success and failure with respect to quality;
- Learn from experience by repeating successes and avoiding failures.

Using the Quality Improvement Paradigm/Experience Factory Organization approach the organization has a good chance to achieve all these capabilities, and to move up in the quality excellence scale faster, because it focuses on its strategic capabilities and value added activities. The Experience Factory Organization is the lean enterprise model for the system and software business.

ACKNOWLEDGMENTS

The authors would like to acknowledge the contributions given to this paper by all those who participated in the experiences and discussions that originated the concepts presented in this paper. The list of their names would be too long. A particular acknowledgment goes to the personnel of the Software Engineering Laboratory at NASA Goddard Space Flight Center and Frank McGarry who made it possible. Other acknowledgments go to Jerry Page (CSC), Tony Jordano (SAIC), Bob Yacobellis (Motorola), Paolo Sigillo (Italsiel).

REFERENCES

[Ada, 1983]
ANSI/MIL-STD-1815A 1983: *Reference Manual for the Ada Programming Language.*

[Basili and Weiss, 1984]
V. R. Basili, D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data", *IEEE Transactions on Software Engineering,* November 1984, pp. 728-738.

[Basili, 1985]
V. R. Basili, "Quantitative Evaluation of a Software Engineering Methodology", *Proceedings of the First Pan Pacific Computer Conference,* Melbourne, Australia, September 1985.

[Basili and Rombach, 1988]
V. R. Basili, H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments", *IEEE Transactions on Software Engineering,* June 1988, pp. 758-773.

[Basili, 1989]
V. R. Basili, "Software Development: A Paradigm for the Future (Keynote Address)", *Proceedings COMPSAC '89,* Orlando, FL, September 1989, pp. 471-485.

[Basili, Caldiera, and Cantone, 1992]
V. R. Basili, G. Caldiera and G. Cantone, "A Reference Architecture for the Component Factory", *ACM Transactions on Software Engineering and Methodology,* Vol. 1, No. 1, January 1992, pp. 53-80.

[Basili, Caldiera, McGarry, Pajerski, Page, and Waligora, 1992]
V. R. Basili, G. Caldiera, F. McGarry, R. Pajerski, J. Page, and S. Waligora, "The Software Engineering Laboratory - An Operational Software Experience Factory", *Proceedings of the Fourteenth International Conference on Software Engineering,* Melbourne, Australia, May 1992.

[Bootstrap]
2I Industrial Informatics, *BOOTSTRAP Project Proposal and Mission Statement,* 2I GmbH, Haierweg 20e, D7800 Freiburg, Germany, 1990, 1991

[Cusumano, 1991]

M.A. Cusumano, *Japan's Software Factories*, Oxford University Press, New York, 1991.

[Deming, 1986]
W. Edwards Deming, *Out of the Crisis*, MIT Center for Advanced Engineering Study, MIT Press, Cambridge, MA, 1986.

[Dewan and Riedl, 1993]
P. Dewan and J.Riedl, "Toward Computer-Supported Concurrent Software Engineering", *IEEE Computer, Special issue on Computer Support for Concurrent Engineering*, January 1993, pp. 17-27.

[Dion, 1993]
R. Dion, "Process Improvement and the Corporate Balance Sheet", *IEEE Software*, July 1993, pp. 28-35.

[Feigenbaum, 1991]
A. V.. Feigenbaum, *Total Quality Control*, Fortieth Anniversary Edition, Mc Graw Hill, New York, NY, 1991.

[Hamel and Prahalad, 1990]
G. Hamel, C. K. Prahalad, The Core Competence of the Corporation, *Harvard Business Review*, Vol. ?, No. ?, July-August 1991, pp. 79-??.

[Hamel and Prahalad, 1991]
G. Hamel, C. K. Prahalad, Corporate Imagination and Expeditionary Marketing, *Harvard Business Review*, Vol. 69, No. 4, July-August 1991, pp. 81-92.

[ISO1]
ISO 8402: 1986 *Quality - Vocabulary*

[ISO2]

ISO 9000: 1987 *Quality Management and Quality Assurance Standards - Guidelines for Selection and Use*

ISO 9001: 1987 *Quality Systems - Model for Quality Assurance in Design/Development, Production, Installation and Servicing*

ISO 9001-3: 1991 *Quality Management and Quality Assurance Standards - Guidelines for the Application of ISO 9001 to the Development, Supply and Maintenance of Software*

[ISO3]
ISO 9126: 1991 *Software Product Evaluation - Quality Characteristics and Guidelines for their Use*

[SEI]
W. S. Humphrey, W. L. Sweet, *A Method for Assessing the Software Engineering Capability of Contractors*, Software Engineering Institute, Technical Report, CMU/SEI-87-TR-23, September 1987.

M. C. Paulk, B. Curtis, M. B. Chrissis, *Capability Maturity Model for Software*, Software Engineering Institute, Technical Report, CMU/SEI-91-TR-24, August 1991.

[Stalk, Evans, and Shulman, 1992]
G. Stalk, P. Evans, and L. E. Shulman, "Competing on Capabilities: The New Rules of Corporate Strategy", *Harvard Business Review*, Vol. 70, No. 2, March-April 1992, pp. 57-69.

[Sommerville, 1992]
Ian Sommerville, *Software Engineering*, Fourth Edition, Addison-Wesley, Wokingham, England, 1992.

[Womack, 1989]
J. P. Womack, D. T. Jones, D. Roos, D. S. Carpenter, *The Machine that Changed the World*, Rawson Associates (MIT Study on Lean Production), New York, NY, 1989.

Page intentionally left blank

The Experience Factory and Its Relationship to Other Quality Approaches

VICTOR R. BASILI

Institute for Advanced Computer Studies
and
Department of Computer Science
University of Maryland
College Park, Maryland

Abstract

This chapter describes the principles behind a specific set of integrated software quality improvement approaches which include the Quality Improvement Paradigm, an evolutionary and experimental improvement framework based on the scientific method and tailored for the software business, the Goal/Question/Metric Paradigm, a paradigm for establishing project and corporate goals and a mechanism for measuring against those goals, and the Experience Factory Organization, an organizational approach for building software competencies and supplying them to projects on demand. It then compares these approaches to a set of approaches used in other businesses, such as the Plan–Do–Check–Act, Total Quality Management, Lean Enterprise Systems, and the Capability Maturity Model.

1. Introduction

The concepts of quality improvement have permeated many businesses. It is clear that the nineties will be the quality era for software and there is a growing need to develop or adapt quality improvement approaches to the software business. Thus we must understand software as an artifact and software development as a business.

Any successful business requires a combination of technical and managerial solutions. It requires that we understand the processes and products of the business, i.e., that we know the business. It requires that we define our business needs and the means to achieve them, i.e., we must define our process and product qualities. We need to define closed loop processes so that we can feed back information for project control. We need to evaluate every aspect of the business, so we must analyze our successes and failures. We must learn from our experiences, i.e., each project should provide information that allows us to do business better the next time. We must build competencies in our areas of business by packaging our successful experiences for reuse and then we must reuse our successful experiences or our competencies as the way we do business.

Since the business we are dealing with is software, we must understand the nature of software and software development. Some of the most basic premises assumed in this work are that:

The *software discipline is evolutionary and experimental;* it is a laboratory science. Thus we must experiment with techniques to see how and when they really work, to understand their limits, and to understand how to improve them.

Software is development not production. We do not produce the same things over and over but rather each product is different from the last. Thus, unlike in production environments, we do not have lots of data points to provide us with reasonably accurate models for statistical quality control.

The technologies of the discipline are human based. It does not matter how high we raise the level of discourse or the virtual machine, the development of solutions is still based on individual creativity and human ability will always create variations in the studies.

There is a lack of models that allow us to reason about the process and the product. This is an artifact of several of the above observations. Since we have been unable to build reliable, mathematically tractable models, we have tended not to build any. And those that we have, we do not always understand in context.

All software is not the same; process is a variable, goals are variable, content varies, etc. We have often made the simplifying assumption that software is software is software. But this is no more true that hardware is hardware is hardware. Building a satellite and a toaster are not the same thing, any more than building a microcode for a toaster and the flight dynamic software for the satellite are the same thing.

Packaged, reusable, experiences require additional resources in the form of organization, processes, people, etc. The requirement that we build packages of reusable experiences implies that we must learn by analyzing and synthesizing our experiences. These activities are not a byproduct of software development, they require their own set of processes and resources.

2. Experience Factory/Quality Improvement Paradigm

The *Experience Factory/Quality Improvement Paradigm* (EF/QIP) (Basili, 1985, 1989; Basili and Rombach, 1987, 1988) aims at addressing the issues of quality improvement in the software business by providing a mechanism for continuous improvement through the experimentation, packaging, and reuse of experiences based on a business's needs. The approach has been evolving since 1976 based on lessons learned in the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) Software Engineering Laboratory (SEL) (Basili *et al.,* 1992).

The basis for the approach is the QIP, which consists of six fundamental steps:

Characterize the current project and its environment with respect to models and metrics.

Set the quantifiable *goals* for successful project performance and improvement.

Choose the appropriate *process* model and supporting methods and tools for this project.

Execute the processes, construct the products, collect and validate the prescribed data, and analyze it to provide real-time feedback for corrective action.

Analyze the data to evaluate the current practices, determine problems, record findings, and make recommendations for future project improvements.

Package the experience in the form of updated and refined models and other forms of structured knowledge gained from this and prior projects and save it in an experience base to be reused on future projects.

Although it is difficult to describe the QIP in great detail here, we will provide a little more insight into the preceding six steps here.

Characterizing the Project and Environment. Based on a set of models of what we know about our business we need to classify the current project with respect to a variety of characteristics, distinguish the relevant project environment for the current project, and find the class of projects with similar characteristics and goals. This provides a context for goal definition, reusable experiences and objects, process selection, evaluation and comparison, and prediction. There are a large variety of project characteristics and environmental factors that need to be modeled and baselined. They include various people factors, such as the number of people, level of expertise, group organization, problem experience, process experience; problem factors, such as the application domain, newness to state of the art, susceptibility to change, problem constraints, etc.;

process factors, such as the life cycle model, methods, techniques, tools, programming language, and other notations; product factors, such as deliverables, system size, required qualities, e.g., reliability, portability, etc.; and resource factors, such as target and development machines, calendar time, budget, existing software, etc.

Goal Setting and Measurement. We need to establish goals for the processes and products. These goals should be measurable, driven by models of the business. There are a variety of mechanisms for defining measurable goals: Quality Function Deployment Approach (QFD) (Kogure and Akao, 1983), the Goal/Question/Metric Paradigm (GQM) (Weiss and Basili, 1985), and Software Quality Metrics Approach (SQM) (McCall *et al.,* 1977).

We have used the GQM as the mechanism for defining, tracking, and evaluating the set of operational goals, using measurement. These goals may be defined for any object, for a variety of reasons, with respect to various models of quality, from various points of view, relative to a particular environment. For example, goals should be defined from a variety of points of view: user, customer, project manager, corporation, etc.

A goal is defined by filling in a set of values for the various parameters in the template. Template parameters included purpose (what object and why), perspective (what aspect and who), and the environmental characteristics (where).

Purpose:

Analyze some
  (objects: process, products, other experience models)
for the purpose of
  (why: characterization, evaluation, prediction, motivation, improvement)

Perspective:

With respect to
  (focus: cost, correctness, defect removal, changes, reliability, user friendliness, . . .)
from the point of view of
  (who: user, customer, manager, developer, corporation, . . .)

Environment:

In the following context
  (problem factors, people factors, resource factors, process factors, . . .)

Example:

Analyze the (system testing method) for the purpose of (evaluation) with respect to a model of (defect removal effectiveness) from the point of view of the (developer) in the following context: the standard NASA/GSFC environment, i.e., process model (SEL version of the waterfall model, . . .), application (ground support software for satellites), machine (running on a DEC 780 under VMS), etc.

The goals are defined in an operational, tractable way by refining them into a set of quantifiable questions that are used to extract the appropriate information from the models of the object of interest and the focus. The questions and models define the metrics and the metrics, in turn, specify the data that needs to be collected. The models provide a framework for interpretation.

Thus, the GQM is used to (1) specify the goals for the organization and the projects, (2) trace those goals to the data that are intended to define these goals operationally, and (3) provide a framework for interpreting the data to understand and evaluate the achievement of the goals, (4) and support the development of data models based on experience.

Choosing the Execution Model. We need to be able to choose a generic process model appropriate to the specific context, environment, project characteristics, and goals established for the project at hand, as well as any goals established for the organization, e.g., experimentation with various processes or other experience objects. This implies we need to understand under what conditions various processes are effective. All processes must be defined to be measurable and defined in terms of the goals they must satisfy. The concept of defining goals for processes will be made clearer in later chapters.

Once we have chosen a particular process model, we must tailor it to the project and choose the specific integrated set of sub-processes, such as methods and techniques, appropriate for the project. In practice, the selection of processes is iterative with the redefinition of goals and even some environmental and project characteristics. It is important that the execution model resulting from these first three steps be integrated in terms of its context, goals, and processes. The real goal is to have a set of processes that will help the developer satisfy the goals set for the project in the given environment. This may sometimes require that we manipulate all three sets of variables to ensure this consistency.

Executing the Processes. The development process must support the access and reuse packaged experience of all kinds. On the other hand, it needs to be supported by various types of analyses, some done in close to real time for feedback for corrective action. To support this analysis, data needs to be collected from the project. But this data collection must be integrated into the processes—it must not be an add on, e.g., defect classification forms part of configuration control mechanism. Processes must be defined to be measurable to begin with, e.g., design inspections can be defined so that we keep track of the various activities, the effort expended in those activities, such as peer reading, and the effects of those activities, such as the number and types of defects found. This allows us to measure such things as domain understanding (how well the process performer understands the object of study and the application domain) and assures that the processes are well defined and can evolve.

Support activities, such as data validation, education and training in the models, and metrics and data forms are also important. Automated support necessary to support mechanical tasks and deal with the large amounts of data and information needed for analysis. It should be noted, however, that most of the data cannot be automatically collected. This is because the more interesting and insightful data tends to require human response.

The kinds of data collected include: resource data such as, effort by activity, phase, type of personnel, computer time, and calendar time; change and defect data, such as changes and defects by various classification schemes, process data such as process definition, process conformance, and domain understanding; product data such as product characteristics, both logical, e.g., application domain, function, and physical, e.g., size, structure, and use and context information, e.g., who will be using the product and how will they be using it so we can build operational profiles.

Analyzing the Data. Based on the goals, we interpret the data that has been collected. We can use this data to characterize and understand, so we can answer questions like ''What project characteristics effect the choice of processes, methods and techniques?'' and ''Which phase is typically the greatest source of errors?'' We can use the data to evaluate and analyze to answer questions like ''What is the statement coverage of the acceptance test plan?'' and ''Does the Cleanroom Process reduce the rework effort?'' We can use the data to predict and control to answer questions like ''Given a set of project characteristics, what is the expected cost and reliability, based upon our history?'' and ''Given the specific characteristics of all the modules in the system, which modules are most likely to have defects so I can concentrate the reading or testing effort on them?'' We can use the data to motivate and improve so we can answer questions such as ''For what classes of errors is a particular technique most effective?'' and ''What are the best combination of approaches to use for a project with a continually evolving set of requirements based on our organization's experience?''

Packaging the Models. We need to define and refine models of all forms of experiences, e.g., resource models and baselines, change and defect baselines and models, product models and baselines, process definitions and models, method and technique evaluations, products and product parts, quality models, and lessons learned. These can appear in a variety of forms, e.g., we can have mathematical models, informal relationships, histograms, algorithms, and procedures, based on our experience with their application in similar projects, so they may be reused in future projects. Packaging also includes training, deployment, and institutionalization.

The six steps of the QIP can be combined in various ways to provide different views into the activities. First note that there are two feedback loops, a project feedback loop that takes place in the execution phase and an organizational feedback loop that takes place after a project is completed. The organizational learning loop changes the organization's understanding of the world by the packaging of what was learned from the last project and as part of the characterization and baselining of the environment for the new project. It should be noted that there are numerous other loops visible at lower levels of instantiation, but these high-level loops are the most important from an organizational structure point of view.

One high-level organizational view of the paradigm is that we must *understand* (characterize), *assess* (set goals, choose processes, execute processes, analyze data), and *package* (package experience). Another view is to *plan* for a project (characterize, set goals, choose processes), *develop* it (execute processes), and then *learn* from the experience (execute processes, analyze data).

2.1 The Experience Factory Organization

To support the Improvement Paradigm, an organizational structure called the Experience Factory Organization (EFO) was developed. It recognizes the fact that improving the software process and product requires the continual accumulation of evaluated experiences (*learning*), in a form that can be effectively understood and modified (*experience models*), stored in a repository of integrated experience models (*experience base*), that can be accessed or modified to meet the needs of the current project (*reuse*).

Systematic *learning* requires support for recording, *off-line* generalizing, tailoring, formalizing, and synthesizing of experience. The off-line requirement is based on the fact that reuse requires separate resources to create reusable objects. Packaging and *modeling* useful experience requires a variety of models and formal notations that are tailorable, extendible, understandable, flexible, and accessible.

An effective *experience base* must contain accessible and integrated set of models that capture the *local* experiences. Systematic *reuse* requires support for using existing experience and on-line generalizing or tailoring or candidate experience.

This combination of ingredients requires an organizational structure that supports: a software evolution model that supports reuse, processes for learning, packaging, and storing experience, and the integration of these two functions. It requires separate logical or physical organizations with different focuses and priorities, process models, expertise requirements.

We divide the functions into a *Project Organization* whose focus/priority is product delivery, supported by packaged reusable experiences, and an *Experience Factory* whose focus is to support project developments by analyzing and synthesizing all kinds of experience, acting as a repository for such experience, and supplying that experience to various projects on demand.

The *Experience Factory* packages experience by building informal, formal or schematized, and productized models and measures of various software processes, products, and other forms of knowledge via people, documents, and automated support.

The Experience Factory deals with reuse of all kinds of knowledge and experience. But what makes us think we can be successful with reuse this time, when we have not been so successful in the past. Part of the reason is that we are not talking about reuse of only code in isolation but about reuse of all kinds of experience and of the context for that experience. The Experience Factory recognizes and provides support for the fact that experience requires the appropriate context definition for to be reusable and it needs to be identified and analyzed for its reuse potential. It recognizes that experience cannot always be reused as is, that it needs to be tailored and packaged to make it easy to reuse. In the past, reuse of experience has been too informal, and has not been supported by the organization. It has to be fully incorporated into the development or maintenance process models. Another major issue is that a project's focus is delivery, not reuse, i.e., reuse cannot be a by-product of software development. It requires a separate organization to support the packaging and reuse of local experience.

The Experience Factory really represents a paradigm shift from current software development thinking. It separates the types of activities that need to be performed by assigning them to different organizations, recognizing that they truly represent different processes and focuses. Project personnel are primarily responsible for the planning and development activities—the *Project Organization* (Fig. 1) and a separate organization, the *Experience Factory* (Fig. 2) is primarily responsible for the learning and technology transfer activities. In the *Project Organization*, we are problem solving. The processes we perform to solve a problem consist

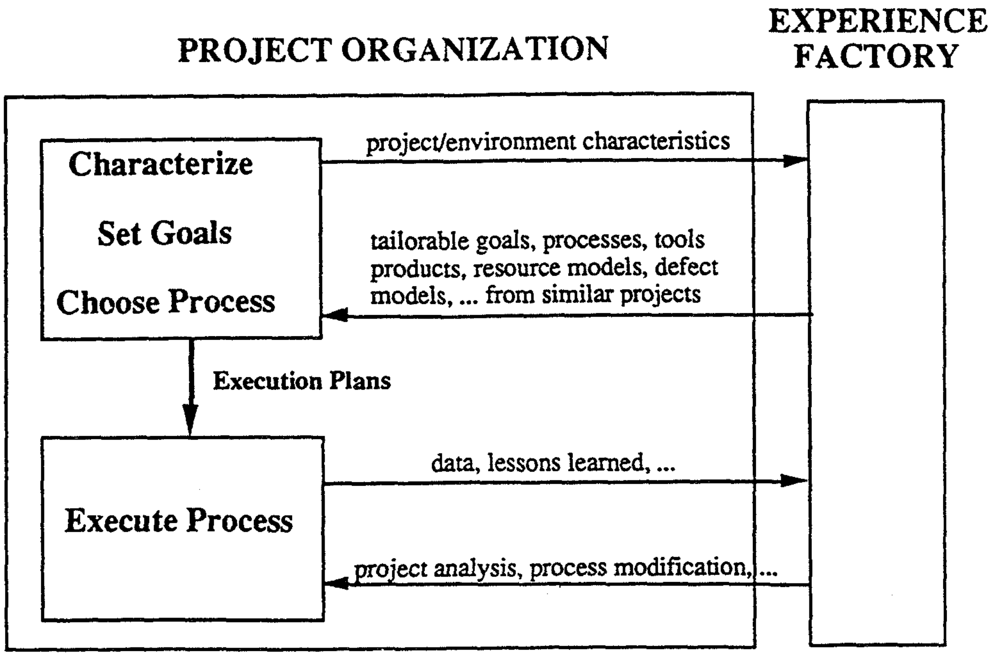

PROJECT ORGANIZATION
EXPERIENCE FACTORY
Characterize
Set Goals
Choose Process
project/environment characteristics
tailorable goals, processes, tools products, resource models, defect models, ... from similar projects
Execution Plans
Execute Process
data, lessons learned, ...
project analysis, process modification, ...

FIG. 1. The Project Organization.

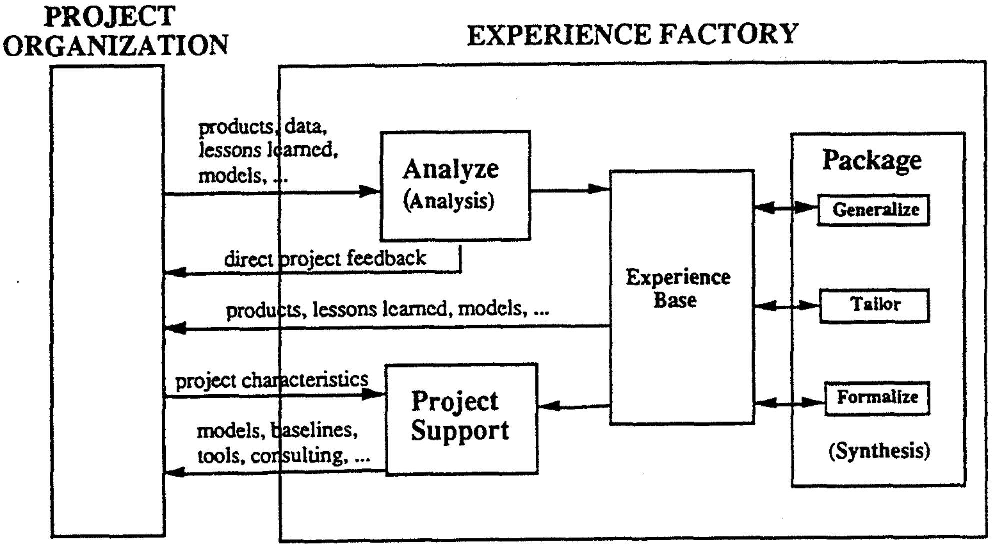


FIG. 2. The Experience Factory.

of the decomposition of a problem into simpler ones, instantiation of higher-level solutions into lower-level detail, the design and implementation of various solution processes, and activities such as validation and verification. In the *Experience Factory,* we are understanding solutions and packaging experience for reuse. The processes we perform are the unification of different solutions and redefinition of the problem, generalization and formalization of solutions in order to abstract them and make them easy to access and modify, an analysis synthesis process enabling us to understand and abstract, and various experimentation activities so we can learn. These sets of activities are totally different.

2.2 Examples of Packaged Experience in the SEL

The SEL has been in existence since 1976 and is a consortium of three organizations: NASA/GSFC, the University of Maryland, and Computer Sciences Corporation (McGarry, 1985; Basili *et al.,* 1992). Its goals have been to (1) understand the software process in a particular environment, (2) determine the impact of available technologies, and (3) infuse identified/refined methods back into the development process. The approach has been to identify technologies with potential, apply and extract detailed data in a production environment (experiments), and measure the impact (cost, reliability, quality, etc.).

Over the years we have learned a great deal and have packaged all kinds of experience. We have built resource models and baselines, e.g., local cost models, resource allocation models; change and defect models and baselines, e.g., defect prediction models; types of defects expected for the application, product models, and baselines, e.g., actual vs. expected product size, library access; over time, process definitions and models, e.g., process models for Cleanroom, Ada waterfall model; method and technique models and evaluations, e.g., best method for finding interface faults; products and product models, e.g., Ada generics for simulation of satellite orbits; a variety of quality models, e.g., reliability models, defect slippage models, ease of change models; and a library of lessons learned, e.g., risks associated with an Ada development (Basili *et al.,* 1992; Basili and Green, 1994).

We have used a variety of forms for packaged experience. There are equations defining the relationship between variables, e.g., effort = $1.48*\text{KSLOC}^{.98}$, number of runs = 108 + 150*KSLOC†; histograms or pie charts of raw or analyzed data, e.g., classes of faults: 30% data, 24% interface, 16% control, 15% initialization, 15% computation; graphs defining ranges of ''normal,'' e.g., graphs of size growth over time with confidence levels; specific lessons learned associated with project types, phases, activities, e.g., reading by stepwise abstraction is most effective for finding interface faults; or in the form of risks or recommendations, e.g., definition of a unit for unit test in Ada needs to be carefully defined; and models or algorithms specifying the processes, methods, or techniques, e.g., an SADT diagram defining design inspections with the reading technique being a variable on the focus and reader perspective.

Note that these packaged experiences are representative of software development in the Flight Dynamics Division at NASA/GSFC. They take into account the local characteristics and are tailored to that environment. Another organization might have different models or even different variables for their models and therefore could not simply use these models. This inability to just use someone else's models is a result of all software not being the same.

These models are used on new projects to help management control development (Valett, 1987) and provide the organization with a basis for improvement based on experimentation with new methods. It is an example of the EF/QIP in practice.

2.3 In Summary

How does the EF/QIP approach work in practice? You begin by getting a commitment. You then define the organizational structure and the associated processes. This means collecting data to establish baselines, e.g., defects and resources, that are process and product independent, and then measuring your strengths and weaknesses to provide a business focus and goals for improvement, and establishing product quality baselines. Using this information about your business, you select and experiment with methods and techniques to improve your processes based on your product quality needs and you then evaluate your improvement based on existing resource and defect baselines. You can define and tailor better and more measurable processes, based on the experience and knowledge gained within your own environment. You must measure for process conformance and domain understanding to make sure that your results are valid.

† KSLOC is thousands of source lines of code.

In this way, you begin to understand the relationship between some process characteristics and product qualities and are able to manipulate some processes to achieve those product characteristics. As you change your processes you will establish new baselines and learn where the next place for improvement might be.

The SEL experience is that the cost of the Experience Factory activities amounts to about 11% of the total software expenditures. The majority of this cost (approximately 7%) has gone into analysis rather than data collection and archiving. However, the overall benefits have been measurable. Defect rates have decreased from an average of about 4.5 per KLOC to about 1 per KLOC. Cost per system has shrunk from an average of about 490 staff months to about 210 staff months and the amount of reuse has jumped from an average of about 20% to about 79%. Thus, the cost of running an Experience Factory has more than paid for itself in the lowering of the cost to develop new systems, meanwhile achieving an improvement in the quality of those systems.

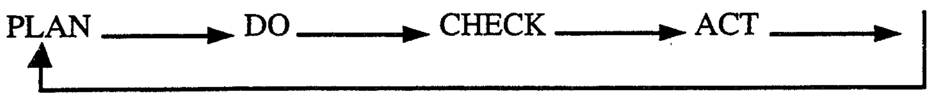
3. A Comparison with Other Improvement Paradigms

Aside from the *Experience Factory/Quality Improvement Paradigm,* there have been a variety of organizational frameworks proposed to improve quality for various businesses. The ones discussed here include:

Plan–Do–Check–Act is a QIP based on a feedback cycle for optimizing a single process model or production line. *Total Quality Management* represents a management approach to long-term success through customer satisfaction based on the participation of all members of an organization. The *SEI Capability Maturity Model* is a staged process improvement based on assessment with regard to a set of key process areas until you reach level 5 which represents continuous process improvement. *Lean (software) Development* represents a principle supporting the concentration of the production on "value-added" activities and the elimination or reduction of "not-value-added" activities. In what follows, we will try to define these concepts in a little more detail to distinguish and compare them. We will focus only on the major drivers of each approach.

3.1 Plan–Do–Check–Act Cycle (PDCA)

The approach is based on work by Shewart (1931) and was made popular by Deming (1986). The goal of this approach is to optimize and improve a single process model/production line. It uses such techniques as feedback loops and statistical quality control to experiment with methods for improvement and build predictive models of the product.



If a family of processes (P) produces a family of products (X) then the approach yields a series of versions of product X (each meant to be an improvement of X), produced by a series of modifications (improvements) to the processes P,

$$P_0, P_1, P_2, \ldots, P_n \rightarrow X_0, X_1, X_2, \ldots, X_n$$

where P_i, represents an improvement over P_{i-1} and X_i has better quality than X_{i-1}. The basic procedure involves four basic steps:

Plan: Develop a plan for effective improvement, e.g., quality measurement criteria are set up as targets and methods for achieving the quality criteria are established.

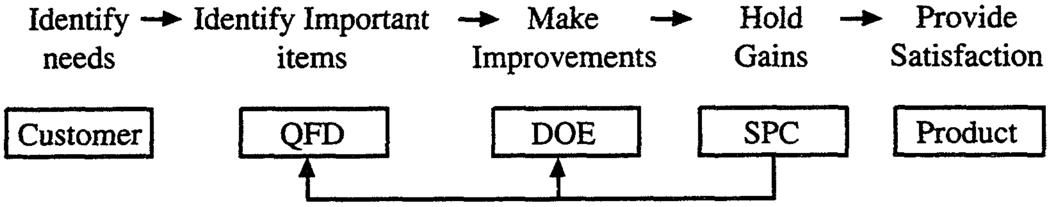
Do: The plan is carried out, preferably on a small scale, i.e., the product is produced by complying with development standards and quality guidelines.

Check: The effects of the plan are observed; at each stage of development, the product is checked against the individual quality criteria set up in the Plan phase.

Act: The results are studied to determine what was learned and what can be predicted, e.g., corrective action is taken based upon problem reports.

3.2 Total Quality Management (TQM)

The term Total Quality Management (TQM) was coined by the Naval Air Systems Command in 1985 to describe its Japanese-style management approach to quality improvement (Feigenbaum, 1991). The goal of TQM is to generate institutional commitment to success through customer satisfaction. The approaches to achieving TQM vary greatly in practice so to provide some basis for comparison, we offer the approach being applied at Hughes. Hughes uses such techniques as QFD, design of experiments (DOE), and statistical process control (SPC), to improve the product through the process.



The approach has similar characteristics to the PDCA approach. If Process (P) → Product (X) then the approach yields

$$P_0, P_1, P_2, \ldots, P_n \rightarrow X_0, X_1, X_2, \ldots, X_n$$

where P_i, represents an improvement over P_{i-1} and X_i provides better customer satisfaction than X_{i-1}.

In this approach, after identifying the needs of the customer, you use QFD to identify important items in the development of the system. DOE is employed to make improvements and SPC is used to control the process and hold whatever gains have been made. This should then provide the specified satisfaction in the product based upon the customer needs.

3.3 SEI Capability Maturity Model (CMM)

The approach is based upon organizational and quality management maturity models developed by Likert (1967) and Crosby (1980), respectively. A software maturity model was developed by Radice *et al.* (1985) while he was at IBM. It was made popular by Humphrey (1989) at the SEI. The goal of the approach is to achieve a level 5 maturity rating, i.e., continuous process improvement via defect prevention, technology innovation, and process change management.

As part of the approach, a five-level process maturity model is defined (Fig. 3). A maturity level is defined based on repeated assessment of an organization's capability in key process areas (KPA). KPAs include such processes as Requirements Management, Software Project Planning, Project Tracking and Oversight, Configuration Management, Quality Assurance, and Subcontractor Management. Improvement is achieved by action plans for processes that had a poor assessment result.

Thus, if a Process (*P*) is level i then modify the process based upon the key processes of the model until the process model is at level i + 1. Different KPSAs play a role at different levels.

The SEI has developed a Process Improvement Cycle to support the movement through process levels. Basically it consists of the following activities:

- Initialize
  - Establish sponsorship
  - Create vision and strategy
  - Establish improvement structure
- For each Maturity level:
  - Characterize current practice in terms of KPAs
  - Assessment recommendations

Level	Focus
5 Optimizing	Continuous Process Improvement
4 Managed	Product & Process Quality
3 Defined	Engineering Process
2 Repeatable	Project Management
1 Initial	Heros

FIG. 3. CMM maturity levels.

Revise strategy (generate action plans and prioritize KPAs)
For each KPA:
Establish process action teams
Implement tactical plan, define processes, plan and execute pilot(s), plan and execute
Institutionalize
Document and analyze lessons
Revise organizational approach

3.4 Lean Enterprise Management

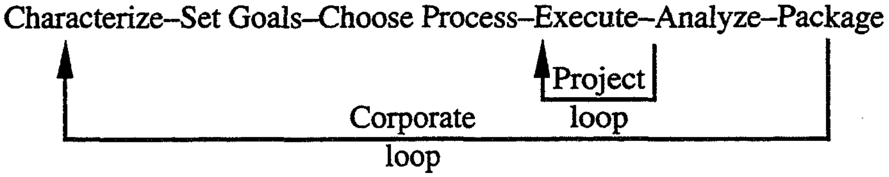
The approach is based on a philosophy that has been used to improve factory output. Womack *et al.* (1990), have written a book on the application of lean enterprises in the automotive industry. The goal is to build software using the minimal set of activities needed, eliminating nonessential steps, i.e., tailoring the process to the product needs. The approach uses such concepts as technology management, human-centered management, decentralized organization, quality management, supplier and customer integration, and internationalization/regionalization.

Given the characteristics for product *V*, select the appropriate mix of subprocesses *pi, qj, rk* . . . to satisfy the goals for *V*, yielding a minimal tailored process *PV* which is composed of *pi, qj, rk* . . .

Process (*PV*) → Product (*V*)

3.5 Comparing the Approaches

As stated above, the Quality Improvement Paradigm has evolved over 17 years based on lessons learned in the SEL (Basili, 1985, 1989; Basili and Rombach, 1987, 1988; Basili *et al.*, 1992). Its goal is to build a continually improving organization based upon its evolving goals and an assessment of its status relative to those goals. The approach uses internal assessment against the organizations own goals and status (rather than process areas) and such techniques as GQM, model building, and qualitative/quantitative analysis to improve the product through the process.


Characterize–Set Goals–Choose Process–Execute–Analyze–Package

If Processes (P_X, Q_Y, R_Z, . . .) → *Products* (*X, Y, Z*, . . .) and we want to build *V*, then based on an understanding of the relationship between P_X, Q_Y, R_Z,

. . . and *X, Y, Z,* . . . and goals for *V* we select the appropriate mix of processes *pi, qj, rk* . . . to satisfy the goals for *V,* yielding a tailored

$$\text{Process } (PV) \rightarrow \text{Product } (V)$$

The EF/QIP is similar to the PDCA in that they are both based on the scientific method. They are both evolutionary paradigms, based on feedback loops from product to process. The process is improved via experiments; process modifications are tried and evaluated and that is how learning takes place.

The major differences are due to the fact that the PDCA paradigm is based on production, i.e., it attempts to optimize a single process model/production line, whereas the QIP is aimed at development. In development, we rarely replicate the same thing twice. In production, we can collect a sufficient set of data based upon continual repetition of the same process to develop quantitative models of the process that will allow us to evaluate and predict quite accurately the effects of the single process model. We can use statistical quality control approaches with small tolerances. This is difficult for development, i.e., we must learn form one process about another, so our models are less rigorous and more abstract. Development processes are also more human based. This again effects the building, use, and accuracy of the types of models we can build. So although development models may be based on experimentation, the building of baselines and statistical sampling, the error estimates are typically high.

The EF/QIP approach is compatible with TQM in that it can cover goals that are customer satisfaction driven and it is based on the philosophy that quality is everyone's job. That is, everyone is part of the technology infusion process. Someone can be on the project team on one project and on the experimenting team on another. All the project personnel play the major role in the feedback mechanism. If they are not using the technology right it can be because they don't understand it, e.g., it wasn't taught right, it doesn't fit or interface with other project activities, it needs to be tailored, or it simply doesn't work. You need the user to tell you how to change it. The EF/QIP philosophy is that no method is ''packaged'' that hasn't been tried (applied, analyzed, tailored). The fact that it is based upon evolution, measurement, and experimentation is consistent with TQM.

The differences between EF/QIP and TQM are based on the fact that the QIP offers specific steps and model types and is defined specifically for the software domain.

The EF/QIP approach is most similar to the concepts of Lean Enterprise Management in that they are both based upon the scientific method/PDCA philosophy. They both use feedback loops from product to process and learn from experiments. More specifically, they are both based upon the ideas of tailoring a set of processes to meet particular problem/product under development. The goal is to generate an optimum set of processes, based upon models of the business and our experience about the relationship between process characteristics and product characteristics.

The major differences are once again based upon the fact that LEM was developed for production rather than development and so model building is based on continual repetition of the same process. Thus, one can gather sufficient data to develop accurate models for statistical quality control. Since the EF/QIP is based on development and the processes are human based, we must learn from the application of one set of processes in a particular environment about another set of processes in different environment. So the model building is more difficult, the models are less accurate, and we have to be cautious in the application of the models. This learning across projects or products also requires two major feedback loops, rather than one. In production, one is sufficient because the process being changed on the product line is the same one that is being packaged for all other products. In the EF/QIP, the project feedback loop is used to help fix the process for the particular project under development and it is with the corporate feedback loop that we must learn by analysis and syntheses across different product developments.

The EF/QIP organization is different from the SEI CMM approach, in that the latter is really more an assessment approach rather than an improvement approach.

In the EF/QIP approach, you pull yourself up from the top rather than pushing up from the bottom. At step 1 you start with a level 5 style organization even though you do not yet have level 5 process capabilities. That is, you are driven by an understanding of your business, your product and process problems, your business goals, your experience with methods, etc. You learn from your business, not from an external model of process. You make process improvements based upon an understanding of the relationship between process and product in your organization. Technology infusion is motivated by the local problems, so people are more willing to try something new.

But what does a level 5 organization really mean? It is an organization that can manipulate process to achieve various product characteristics. This requires that we have a process and an organizational structure to help us: understand our processes and products, measure and model the project and the organization, define and tailor process and product qualities explicitly, understand the relationship between process and product qualities, feed back information for project control, experiment with methods and techniques, evaluate our successes and failures, learn from our experiences, package successful experiences, and reuse successful experiences. This is compatible with the EF/QIP organization.

QIP is not incompatible with the SEI CMM model in that you can still use key process assessments to evaluate where you stand (along with your internal goals, needs, etc.). However, using the EF/QIP, the chances are that you will move up the maturity scale faster. You will have more experience early on operating within an improvement organization structure, and you can demonstrate product improvement benefits early.

4. Conclusion

Important characteristics of the EF/QIP process indicate the fact that it is iterative; you should converge over time so don't be overly concerned with perfecting any step on the first pass. However, the better your initial guess at the baselines the quicker you will converge.

No method is ''packaged'' that hasn't been tried (applied, analyzed, tailored). Everyone is part of the technology infusion process. Someone can be on the project team on one project and on the experimenting team on another. Project personnel play the major role in the feedback mechanism. We need to learn from them about the effective use of technology. If they are not using the technology right it can be because they don't understand it or it wasn't taught right, it doesn't fit/interface with other project activities, it needs to be tailored, or it doesn't work and you need the user to tell you how to change it. Technology infusion is motivated by the local problems, so people are more willing to try something new. In addition, it is important to evaluate process conformance and domain understanding or you have very little basis for understanding and assessment.

The integration of the Improvement Paradigm, the Goal/Question/Metric Paradigm, and the EFO provides a framework for software engineering development, maintenance, and research. It takes advantage of the experimental nature of software engineering. Based upon our experience in the SEL and other organizations, it helps us understand how software is built and where the problems are, define and formalize effective models of process and product, evaluate the process and the product in the right context, predict and control process and product qualities, package and reuse successful experiences, and feed back experience to current and future projects. It can be applied today and evolve with technology.

The approach provides a framework for defining quality operationally relative to the project and the organization, justification for selecting and tailoring the appropriate methods and tools for the project and the organization, a mechanism for evaluating the quality of the process and the product relative to the specific project goals, and a mechanism for improving the organization's ability to develop quality systems productively. The approach is being adopted by several organizations to varying degrees, such as Motorola and HP, but it is not a simple solution and it requires long-term commitment by top-level management.

In summary, the QIP approach provides for a separation of concerns and focus in differentiating between problem solving and experience modeling and packaging. It offers a support for learning and reuse and a means of formalizing and integrating management and development technologies. It allows for the generation of a tangible corporate asset: an experience base of software competencies. It offers a Lean Enterprise Management approach compatible with TQM while providing a level 5 CMM organizational structure. It links focused research with development. Best of all you can start small, evolve and expand, e.g., focus on a homogeneous set of projects or a particular set of packages and build from there. So any company can begin new and evolve.

References

Basili, V. R. (1985). Quantitative evaluation of software engineering methodology. *In* "Proceedings of the 1st Pan Pacific Computer Conference, Melbourne, Australia" (also available as Technical Report TR-1519, Department of Computer Science, University of Maryland, College Park, 1985).

Basili, V. R. (1989). Software development: A paradigm for the future. *In* "Proceedings of the 13th Annual International Computer Software and Applications Conference (COMPSAC), Keynote Address, Orlando, FL."

Basili, V. R., and Green, S. (1994). Software process evolution at the SEL. *IEEE Software Mag.*, July, pp. 58–66.

Basili, V. R., and Rombach, H. D. (1987). Tailoring the software process to project goals and environments. *In* "Proceedings of the 9th International Conference on Software Engineering, Monterey, CA," pp. 345–357.

Basili, V. R., and Rombach, H. D. (1988). The TAME Project: Towards improvement-oriented software environments. *IEEE Trans. Software Eng.* **SE-14**(6), 758–773.

Basili, V. R., Caldiera, G., McGarry, F., Pajerski, R., Page, G., and Waligora, S. (1992). The software engineering laboratory—an operational software experience factory. *In* "Proceedings of the International Conference on Software Engineering," pp. 370–381.

Crosby, P. B. (1980). Quality is Free: The art of making quality certain. New American Library, New York.

Deming, W. E. (1986). "Out of the Crisis." MIT Center for Advanced Engineering Study, MIT Press, Cambridge MA.

Feigenbaum, A. V. (1991). "Total Quality Control," 40th Anniv. Ed. McGraw-Hill, New York.

Humphrey, W. S. (1989). "Managing the Software Process," SEI Ser. Software Eng. Addison-Wesley, Reading, MA.

Kogure M., and Akao, Y. (1983). Quality function deployment and CWQC in Japan. *Qual. Prog.*, October, pp. 25–29.

Likert, R. (1967). "The Human Organization: Its Management and Value." McGraw-Hill, New York.

McCall, J. A., Richards, P. K., and Walters, G. F. (1977). "Factors in Software Quality," RADC TR-77-369.

McGarry, F. E. (1985). "Recent SEL Studies," Proc. 10th Annu. Software Eng. Workshop. NASA Goddard Space Flight Center, Greenbelt, MD.

Paulk, M. C., Curtis, B., Chrissis, M. B., and Weber, C. V. Capability Maturity Model for Software, Version 1.1, Technical Report SEI-93-TR-024.

Radice, R., Harding, A. J., Munnis, P. E., and Phillips, R. W. (1985). A programming process study. *IBM Syst. J.* **24**(2).

Shewhart, W. A. (1931). "Economic Control of Quality of Manufactured Product." Van Nostrand, New York.

Software Engineering Institute (1933). "Capability Maturity Model," CMU/SEI-93-TR-25 Version 1.1. Carnegie-Mellon University, Pittsburgh, PA.

Valett, J. D. (1987). The dynamic management information tool (DYNAMITE): Analysis of the prototype, requirements and operational scenarios. M.Sc. Thesis, University of Maryland, College Park.

Weiss, D. M., and Basili, V. R. (1985). Evaluating software development by analysis of changes: Some data from the software engineering laboratory. *IEEE Trans. Software Eng.* **SE-11**(2), 157–168.

Womack, J. P., Jones, D. T., and Roos, D. (1990). "The Machine that Changed the World: Based on the Massachusetts Institue of Technology 5-Million Dollar 5-Year Study on the Future of the Automobile." Rawson Associates, New York.

omit
THIS
PAGE

SECTION 3—SOFTWARE MODELS

The technical paper included in this section was originally prepared as indicated below.

- "Characterizing and Assessing a Large-Scale Software Maintenance Organization," L. Briand, W. Melo, C. Seaman, and V. Basili, *Proceedings of the 17th International Conference on Software Engineering*, Seattle, Washington, U.S.A., April 23–30, 1995

Page intentionally left blank

Characterizing and Assessing a Large-Scale Software Maintenance Organization*

Lionel Briand
CRIM
1801 McGill College Av.
Montréal (Quebec)
H3A 2N4, Canada
Lionel.Briand@crim.ca

Walcélio Melo, Carolyn Seaman and Victor Basili
Computer Science Department
Institute for Advanced Computer Studies
University of Maryland
College Park, MD, 20742
{melo | carolyns | basili}@cs.umd.edu

Abstract

One important component of a software process is the organizational context in which the process is enacted. This component is often missing or incomplete in current process modeling approaches. One technique for modeling this perspective is the Actor-Dependency (AD) Model. This paper reports on a case study which used this approach to analyze and assess a large software maintenance organization. Our goal was to identify the approach's strengths and weaknesses while providing practical recommendations for improvement and research directions. The AD model was found to be very useful in capturing the important properties of the organizational context of the maintenance process, and aided in the understanding of the flaws found in this process. However, a number of opportunities for extending and improving the AD model were identified. Among others, there is a need to incorporate quantitative information to complement the qualitative model.

1. Introduction

It has now been recognized that, in order to improve the quality of software products, it is necessary to enhance the quality of the software processes used to develop and maintain them. This requires the understanding and modeling of these processes in order to be able to analyze and assess them. Following the example of other engineering disciplines, where empirical approaches to management have been successfully applied, several methodologies have been defined for allowing the characterization and incremental improvement of software processes [Basili, 1989; Lanphar, 1990]. The modeling of software development and maintenance processes is a necessary component of these approaches. A number of modeling techniques have been developed, e.g., [Lott and Rombach, 1993; Finkelstein et al, 1994; Melo and Belkhatir, 1994].

What has received less attention in the literature is the need to model the organizational context in which a development process executes. It is not possible to fully understand and analyze such process issues as information flow, division of work, and coordination without including the organizational context in the analysis. Organizational context refers to characteristics of relationships between process participants. Such relationships include, among others, the management hierarchy, the structure of ad hoc working groups, and seating arrangements. Some process modeling approaches attempt to include mechanisms in their formalisms to deal with organizational structure [Curtis et. al., 1992], but not to any great level of detail. Some formalisms have specifically focused on organizational modeling [Rein, 1992; Benus, 1994], but these lack the mechanisms and flexibility necessary for quantitative analysis (discussed later).

One approach to organization and process modeling appears particularly promising [Yu and Myopoulos, 1994]. This approach is very new (it was presented at last year's ICSE) and thus lacks significant validation through use. One goal of this paper is to report an early experience with this promising new approach.

Consistent with the philosophy presented above, [Briand et. al., 1994] have developed an auditing process specifically aimed at software maintenance processes and organizations.

*During this work the authors were, in part, supported by the following grants: Briand by NSF grant 01-5-24845 and CRIM, Melo by UMIACS, Seaman by IBM-Toronto grant 91-1-20011, and Basili by NASA grant NSG-5123.

ICSE '95, Seattle, Washington USA
© 1995 ACM 0-89791-708-1/95/0004...$3.50

Such an approach requires, to a certain level of detail, the modeling of processes and organizations. In this context, the Actor-Dependency modeling technique [Yu and Mylopoulos, 1994] mentioned above appeared to be suitable because of its capability to capture numerous kinds of constraints and dependencies frequently encountered in complex organizations. In addition, this technique proved in practice to be intuitive and to facilitate communication with the maintenance staff of the studied organization. This paper reports one experience of using the Actor-Dependency technique to help analyze a large software maintenance organization. We evaluate the approach's strengths and weaknesses while providing practical recommendations for improvement. In Section 2, we briefly describe the Actor-Dependency modeling approach and one of its extensions that we have used in our study. Section 3 presents the case study we conducted. In Section 4 we evaluate the advantages and weaknesses of the AD approach. Finally, in Section 5, we present a number of suggestions for future work, both with the AD model and software organization modeling in general.

2. The Actor-Dependency Modeling Approach

The most important characteristic of the modeling approach presented by Yu et al, for our purposes, is its capability to fully represent the organizational context in which a development process is performed. This language provides a basic organizational model with several enhancements, only one of which we will describe here. The basic Actor-Dependency model represents an organizational structure as a network of dependencies among organizational entities, or actors. The enhancement which we have used, called the Agent-Role-Position (ARP) model, provides a useful decomposition of the actors themselves. These two representations are described briefly in the following sections. For a more detailed description, see [Yu and Mylopoulos, 1993].

2.1. The basic Actor-Dependency (AD) model

In this model, an organization is described as a network of interdependencies among active organizational entities, i.e., actors. A node in such a network represents an organizational actor, and a link indicates a dependency between two actors. Examples of actors are: someone who inspects units, a project manager, or the person who gives authorization for final shipment. Documents to be produced, goals to be achieved, and tasks to be performed are examples of dependencies between actors. When an actor, A1, depends on A2, through a dependency D1, it means that A1 cannot achieve, or cannot efficiently achieve, its goals if A2 is not able or willing to fulfill its commitment to D1. The AD model provides four types of dependencies between actors:

- In a *goal dependency*, an actor (the depender) depends on another actor (the dependee) to achieve a certain goal or state, or fulfill a certain condition (the dependum). The depender does not specify how the dependee should do this. A fully built configuration, a completed quality assessment, or 90% test coverage of a software component might be examples of goal dependencies if no specific procedures are provided to the dependee(s).
- In a *task dependency*, the depender relies on the dependee to perform some task. This is very similar to a goal dependency, except that the depender specifies how the task is to be performed by the dependee, without making the goal to be achieved by the task explicit. Unit inspections are examples of task dependencies if specific standard procedures are to be followed.
- In a *resource dependency*, the depender relies on the dependee for the availability of an entity (physical or informational). Software artifacts (e.g. designs, source code, binary code), software tools, and any kind of computational resources are examples of resource dependencies.
- A *soft-goal dependency* is similar to a goal dependency, except that the goal to be achieved is not sharply defined, but requires clarification between depender and dependee. The criteria used to judge whether or not the goal has been achieved is uncertain. Soft-goals are used to capture informal concepts which cannot be expressed as precisely defined conditions, as are goal dependencies. High product quality, user-friendliness, and user satisfaction are common examples of soft-goals because in most environments, they are not precisely defined.

Three different categories of dependencies can be established based on degree of criticality:

- *Open dependency*: the depender's goals should not be significantly affected if the dependee does not fulfill his or her commitment.
- *Committed dependency*: some planned course of action, related to some goal(s) of the depender, will fail if the dependee fails to provide what he or she has committed to.
- *Critical dependency*: failure of the dependee to fulfill his or her commitment would result in the failure of all known courses of action towards the achievement of some goal(s) of the depender.

The concepts of open, committed, and critical dependencies can be used to help understand actors' vulnerabilities and associated risks. In addition, we can identify ways in which actors alleviate this risk. A commitment is said to be:

- *enforceable* if the depender can cause some goal of the dependee to fail.
- *assured* if there is evidence that the dependee has an interest in delivering the dependum.
- *insured* if the depender can find alternative ways to have his or her dependum delivered.

In summary, a dependency is characterized by three attributes: type, level of criticality, and its associated risk-management mechanisms. The type (resource, soft-goal, goal, and task) represents the issue captured by the dependency, while the level of criticality indicates how important the dependency is to the depender. Risk-management mechanisms allow the depender to reduce the vulnerability associated with a dependency.

Figure 1 shows a simple example of an AD model. A Manager oversees a Tester and a Developer. The Manager depends on the Tester to test. This is a task dependency because there is a defined set of procedures that the Tester must follow. In contrast, the Manager also depends on the Developer to develop, but the Developer has complete freedom to follow whatever process he or she wishes, so this is expressed as a goal dependency. Both the Tester and the Developer depend on the Manager for positive evaluations, where there are specific criteria to define "positive", thus these are goal dependencies. The Tester depends on the Developer to provide the code to be tested (a resource), while the Developer depends on the Tester to test the code well (good coverage). Assuming that there are no defined criteria for "good" coverage, this is a soft-goal dependency.

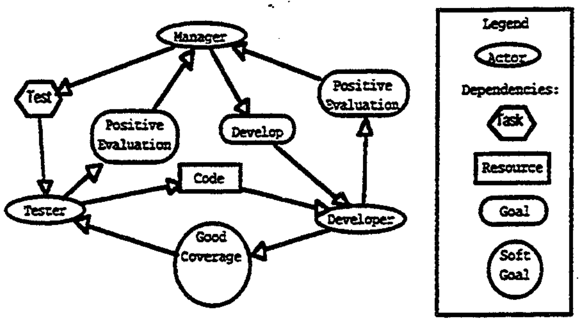


Figure 1: A simple example of an AD model

2.2. The Agent-Role-Position (ARP) decomposition

In the previous section, what we referred to as an actor is in fact a composite notion that can be refined in several ways to provide different views of the organization. *Agents*, *roles*, and *positions* are three possible specializations of the notion of actor which are related as follows:

- an agent occupies one or more positions
- an agent plays one or more roles.
- a position can cover different roles in different contexts

Figure 2 shows an example of an actor decomposition. These three types of specialization are useful in several ways. They can be used to represent the organization at different levels of detail. At a very high level, one might use only unspecialized actors. Positions provide more detail, but still provide a high-level view. Roles provide yet more detail, and the use of agents allows the modeler to specify even specific individuals. The ARP decomposition could be especially useful when extending the use of AD models to quantitative analysis. This is described in more detail in Section 5.3.3. A Case Study using the AD Model

3.1. Background

Before describing our results, some background is necessary in order for the reader to understand the analysis which follows. We will first describe the development environment which serves as our context. Second, the auditing process used in the case study will be described.

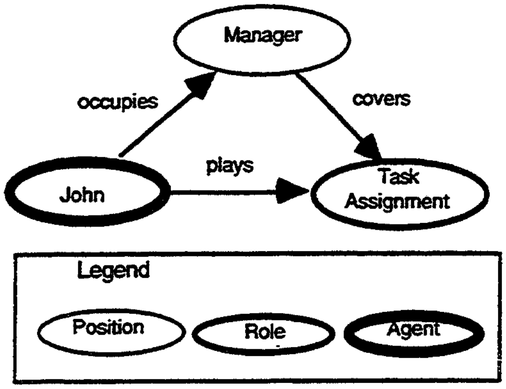


Figure 2. Associated Agent, Position, and Role

3.1.1. The Studied Maintenance Organization

This study took place in the Flight Dynamics Division (FDD) of the NASA Goddard Space Flight Center (GSFC). Over one hundred software systems for the control and prediction of satellite orbits, trajectories and attitude, totalling about 3.5 million lines of code, are maintained. Many of these systems are maintained over a very long period of time, and regularly produce new releases. About 80 people are involved in the maintenance of these systems. This study focused on three systems in particular, which ranged from 156 to 260 thousand lines of FORTRAN code, and from 7 to 26 years of age.

Numerous communication, schedule, budget and technical problems arise with each release. This results in somewhat unstable change requirements all along the release process, a high turnover in some projects and difficulties in meeting deadlines. There was a need to study these phenomena.

More precisely, our framework for this study is the Software Engineering Laboratory (SEL). The SEL is a joint venture between the University of Maryland, CSC and NASA. The SEL is an organization aimed at improving NASA-FDD software development processes based on measurement and empirical analysis. Recently, responding to the growing cost of software maintenance at NASA-FDD, the SEL has initiated a program aimed at characterizing, evaluating and improving its maintenance processes. The first step in this direction was a set of studies conducted using the auditing technique described below.

3.1.2. The Maintenance Process Auditing Methodology

In [Briand et. al., 1994], a qualitative and inductive methodology has been proposed in order to characterize and audit software maintenance processes and organizations and thereby identify their specific problems and needs. This methodology encompasses a set of procedures which attempts to determine causal links between maintenance problems and flaws in the maintenance organization and process. This allows for a set of concrete steps to be taken for maintenance quality and productivity improvement, based on a tangible understanding of the relevant maintenance issues in a particular maintenance environment. The steps of this methodology can be summarized as follows:

Step 1: Identify the organizational entities with which the maintenance team interacts and the organizational structure in which maintainers operate. In this step the distinct teams and their roles in a change process are identified. Information flows between actors should also be determined.

Step 2: Identify the phases involved in the creation of a new system release. Software artifacts produced and consumed by each phase must be identified. Actors responsible for producing and validating the output artifacts of each phase have to be identified and located in the organizational structure defined in Step 1.

Step 3: Identify the generic activities involved in each phase, i.e. decompose life-cycle phases to a lower level of granularity. Identify, for each low-level activity, its inputs and outputs and the actors responsible for them.

Step 4: Select one or several past releases for analysis in order to better understand process and organization flaws.

Step 5: Analyze the problems that occurred while performing the software changes in the selected releases in order to produced a causal analysis document. The knowledge and understanding acquired through steps 1-3 are necessary in order to understand, interpret and formalize the information described in the causal analysis document.

Step 6: Establish the frequency and consequences of problems due to flaws in the organizational structure and the maintenance process by analyzing the information gathered in Step 5.

Modeling the organizational context of the maintenance process was a very important step in the above analysis process. A model of the organization was necessary for communication with maintenance process participants. Gathering organizational information and building the model was critical to our understanding of the work environment and differences across projects. The model was also useful in checking the consistency and completeness of the maintenance process model. For example, the organizational model allowed us to determine whether or not all organizational actors had defined roles in the process model. During this preliminary study, the following requirements were identified for an optimal organizational modeling technique:

R1: The modeling methodology had to facilitate the detection of conflicts between organizational structures and goals. For example, inconsistencies between the expectations and intentions of interfacing actors seemed to be a promising area of investigation.

R2: We needed to capture many different types of relationships between actors. These included relationships that contributed to information flow, work flow, and fulfillment of goals. The explicit and comprehensive modeling of all types of relationships was necessary in this context.

R3: Different types of organizational entities had to be captured: individuals, their official position in the organizational structure, and their roles and activities in the maintenance process. This was important not only to be able to model at different levels of detail, but also to provide different views of the organization, each relaying different information.

R4: Links between the organization and the maintenance process model had to be represented explicitly.

R5: The notation had to aid in communication through intuitive concepts and graphical representation.

As a starting point, we decided to use the Actor-Dependency model introduced by Yu et al in order to reach these objectives. The AD model, as we shall see, meets many of our requirements.

In the next section, we provide the extended AD model of our maintenance organization where, for the sake of simplification, we use only positions (one possible specialization of actors) as vertices of the graph.

3.2. AD Organizational Model

The organizational model in Figure 3 is very complex despite important simplifications (e.g., agents and roles are not represented). This shows how intricate the network of dependencies in a large software maintenance organization can be. The lessons learned with respect to the maintenance organization are presented below and the approach's advantages and drawbacks are the focus of the next section.

The organizational model presented in Figure 3 was built using information from a variety of sources: we read maintenance standards release documents, interviewed people involved in the change and configuration management process, analyzed release management reports, and studied the official organization charts.

The model is by necessity incomplete. We have focused on those positions and activities which contribute to the maintenance process only. So there are many other actors in the NASA-FDD organization which do not appear in the AD graph. As well, we have aggregated some of the positions where appropriate. For example, Maintenance Management includes a large number of separate actors, but for the purposes of our analysis, they can be treated as an aggregate. Because only the primary dependencies are shown at this level of detail, nearly all of them are shown as critical. This issue will be discussed in more detail later. Below are listed the positions shown in the figure, and a short explanation of their specific roles:

Testers present acceptance test plans, perform acceptance test and provide change requests to the maintainers when necessary.

Users suggest, control and approve performed changes.

QA Engineer controls maintainers' work (e.g., conformance to standards), attends release meetings, and audits delivery packages.

Configuration Manager integrates updates into the system, coordinates the production and release of versions of the system, and provides tracking of change requests.

Maintenance management grants preliminary approvals of maintenance change requests and release definitions.

Maintainers: analyze changes, make recommendations, perform changes, perform unit and change validation testing after linking the modified units to the existing system, perform validation and regression testing after the system is recompiled by the *Configuration Manager.*

Process Analyst collects and analyzes data from all projects and packages data to be reused.

NASA Management is officially responsible for selecting software changes, gives official authorizations, and provides the budget.

The resulting organizational model was validated through use, within the context of the auditing methodology presented above. The modeling of the maintenance process, the release documents, and the causal analysis of maintenance problems allowed us to check the model for consistency and completeness.

3.3 Lessons Learned

Below are the main flaws that were found in the maintenance process and which we reported to the maintenance organization. In all cases, the flaws were uncovered, or at least better understood, by studying the AD model.

Task Leader

From our analysis, it appears that the Task Leader is a very central position. This is clearly illustrated in Figure 3. The centrality of the Task Leader gives rise to two possible problems: overloading of the person filling this position, and over-dependence of the project on this one position. Analysis of the Task Leader's role decomposition, especially in conjunction with quantitative analysis, would be helpful in determining the extent of these problems, and possible solutions.

Quality Assurance

Standards conformance and quality inspections were not perceived by the task leaders and maintainers as critical. They considered these processes mainly bureaucratic. This is reflected in the (non-)criticality symbols on the corresponding dependencies in Figure 3. This pointed out a weakness in the process and organization that could be remedied through more suitable inspection procedures and better definition and communication of quality needs.

Requirements

In Figure 3, Unambiguous requirements (a dependency between the Task Leader and the User) is not an enforceable soft-goal dependency since the users and maintainers (including the Task Leader) belong to two different management hierarchies. In other words, the Task Leader and User are so far removed from each other in the network of management dependencies that the Task Leader has no practical recourse for ensuring that the User provides unambiguous requirements. Note that the management dependencies are included in the AD model, but have been omitted from Figure 3 to simplify the diagram. Moreover, the fact that this dependency is a soft-goal and not a goal raises another issue: standards for defining unambiguous requirements should be defined and applied. The lack of such standards indicates that the organization is still immature in this area.

Data Collection

Process analysts attempt to collect data in order to evaluate and better predict the maintenance process. However, such a procedure is inherently difficult to enforce when maintainers do not clearly understand the benefits of such data collection, in terms of useful feedback. In terms of the AD model, the Process Analyst's dependency on the Maintainer is a vulnerability, with no reciprocal dependency to serve as a risk management mechanism available to reduce that vulnerability.

4. Evaluation of the AD Model

4.1. Advantages

The notions of enforcement and assurance, as well as the modeling of goal and soft-goal dependencies, helped us to detect potential problem areas, such as critical dependencies that are not enforceable and for which there were no clear assurances of commitment. The Task Leader's need for unambiguous requirements is an example of such an inconsistency. This seemed to fulfill, at least partially, requirement R1.

The AD model captured all the information, work, and resource flows through resource and task dependencies. This allowed us to identify inconsistencies between what some agents needed and the support that they were actually getting. The problem of the Process Analyst's need for development data from maintainers is an example of this. We also found that the soft-goal dependency in particular was useful in highlighting areas in which the environment was immature. The unambiguous requirements dependency exemplifies this situation. The various types of dependencies in the AD model therefore fulfilled requirement R2.

The actor decomposition extension to the AD model makes a clear distinction between various organizational entities by defining and differentiating roles, positions and agents as different specializations of actors (requirement R3). This allowed us to extract different information from different versions of the AD model, using different specializations of the actors. For example, we found that the model remained fairly stable from project to project when nodes represented positions (as in Figure 3). However, when we used the role specialization, significant differences appeared between projects. For example, roles of managers often varied significantly, depending on their technical background. This served to show that process participants found the freedom to tailor their work to the situation, while the official organizational structure could remain stable. Roles also provide a way to create explicit links between the organizational model and any process model composed of consistently defined activities (requirement R4).

Many interactions with various members of the maintenance organization were necessary in order to clarify inconsistencies and insure completeness. The AD model played an important role in this communication, because it facilitated the exchange and comparison of perceptions about the organizational structure. It served as a good communication tool (requirement R5).

4.2. Issues

Despite the numerous advantages of the AD model mentioned in the previous section, some problems have been identified and should be the subject of further research.

Classification of dependencies

Once a dependency has been identified, it is not always straightforward to classify it according to the defined taxonomy (requirement R2). One example is the difficulty in distinguishing between a task dependency and a goal dependency. A task may be partially defined (e.g., through standards) but some significant degree of freedom can exist for the dependee whose understanding of the task objectives may or may not be complete. It is for this reason that we have included no task dependencies in our AD model (see Figure 3). Also, the borderline between soft-goals and (hard-)goals is not always clear. When is a goal sufficiently defined to be classified as a (hard-)goal? More precise guidelines are needed in order to classify dependencies in an appropriate fashion.

Another inadequacy of the classification scheme is in the case of information dependencies. As defined, information dependencies are one type of resource dependency. However, a need for information is different in nature from a need for time, money, or personnel resources. From a data analysis point of view, information dependencies are described by different attributes than those that would be used to describe other resource dependencies. For this reason, any kind of information flow analysis necessitates the treatment of information as a separate type of dependency.

Criticality of dependencies

No precise and unambiguous definition exists to classify a dependency as critical, committed or open, which impedes fulfillment of requirement R1. Because of this, most of the dependencies in our context appeared critical since they were certainly important from the dependee's perspective. It was, from a practical perspective, difficult to determine if they were really indispensable.

Another difficulty with identifying committed and open dependencies is that practitioners often do not mention them in interviews and they are usually not included explicitly in process documents. We have found that direct observation is the only effective way to capture such secondary dependencies. This is time- and effort-consuming. Furthermore, when modeling at the level of detail of our model, it is sufficient to include only the primary dependencies, which are usually critical .

Interactions between dependencies

The notions of enforcement, assurance and insurance are extremely useful but they are difficult to represent explicitly in the AD model representation (requirement R5). These notions need to be captured explicitly by the organizational model. In the next section, we suggest a way to do this by treating these three mechanisms as interactions between dependencies.

5. Suggestions

Based on this case study and our evaluation of the AD model, we provide some suggestions which may be useful to those wishing to extend the AD concepts, and to those who are engaged in organization modeling.

5.1. An Entity-Relationship Model

We believe two of the most important problems that arose in our work with Actor-Dependency models have a common solution. The first issue is the need to clearly define the information that needs to be collected, particularly in a quantitative analysis effort. The second issue is that of separating organizational from process concerns since they require different types of analyses and solutions. The Entity-Relationship Model, shown in Figure 4 and discussed in the next two sections, addresses both of these issues.

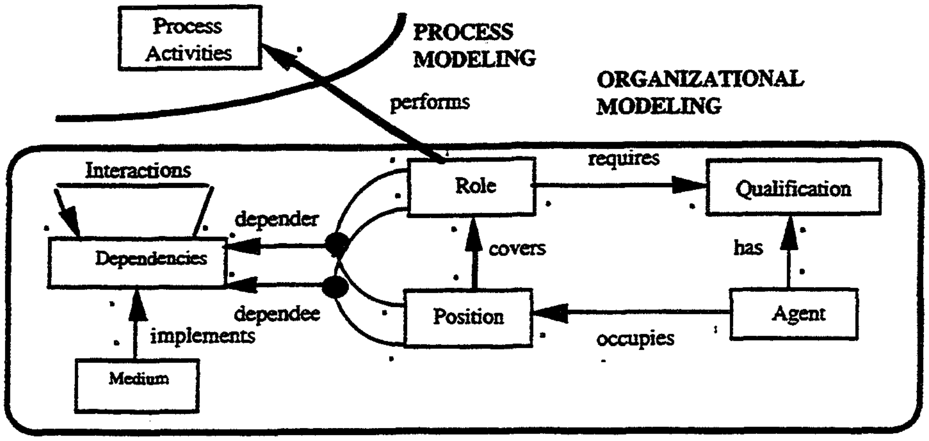


Figure 4: Modified ER model for AD graphs

5.1.1. ER Model

Defining precisely the entities and attributes of interest is not only necessary for data analysis, but also helps clarify the modeling approach itself. One entity that we have added in Figure 4 is the Qualification entity. An agent "has" one or more qualifications, e.g., maintaining ground satellite software systems. Moreover, based on experience, it may be determined that some role "requires" specific qualifications, e.g., experience with Ada. Comparison of the required qualifications and the actual organizational set-up appears useful for identifying high-risk organizational patterns.

We have retained the agent/role/position decomposition of an actor defined by Yu et al, which we found very useful. The ER model also shows "depender" and "dependee" as ternary relationships. This reflects the fact that a depender or dependee of a dependency can be either a role or a position. A role may be functionally dependent on another role in order to perform a given process activity. Positions are usually interdependent because of the need for authorization or authority. However, we believe that dependencies are not inherent to agents themselves, at least not in our context.

We have also added a new entity, Medium, which is the communication medium used to implement a particular dependency (especially information dependencies). This entity is used in some types of quantitative analysis, which is described in a later section. Finally, dependencies are related to each other and this is captured through the interaction relationship, also described in a later section. However, this ER model requires further definition (e.g., attributes should be specified), validation, and refinement.

5.1.2. Linking an organization model with a process model

The ER model also makes explicit the relationship, and the separation, between process and organization. Analysis of an organization is aided by the isolation of organizational issues (e.g., information flow, distribution of work) from purely process concerns (e.g., task scheduling, concurrency). This is especially true when dealing with quantitative data analysis. Process entities and organization entities are described by different quantitative attributes. Separation of these attributes clarifies the analysis. Although organization and process raise separate issues, their effects are related. Understanding the relationship between organization and process is crucial to making improvements to either aspect of the environment (requirement R4). For example, the "performs" relationship can link a role to a set of activities, which may be seen as lower-level roles. The entity Process Activity is itself related to other entities in the process model not specified here.

5.2. Dependency Interactions

Interactions between dependencies need to be modeled. There are several different types of these interactions which may be seen as relationships from a source to a target dependence:

1) Being committed to the source dependency makes the commitment to the target dependency more difficult. This represents a *negative assurance*.
2) The source dependency is an additional motivation to the target dependency. This represents a *positive assurance*.

3) The source dependency's failure can provoke the failure of the target dependency. One dependency's depender is the other dependency's dependee and vice-versa. This represents a dependency *enforcement*.
4) Failure of one dependency is mitigated by the other dependency. Both dependencies have the same depender but different dependees. This is a dependency *insurance*.

If a depender can count on many dependees to deliver a dependum, we can say that the dependency is *insured*. In this case, different dependees can be committed to the same dependum. This can be graphically represented in a AD graph in a fashion similar to OR branches in AND-OR trees. For example, Figure 5 shows a case where a Task Leader (depender) can count on a Maintainer or a Tester (two different dependees) for delivering the "Test Plan & Results" (a particular dependum).

We can also provide a representation for expressing assurance interactions between dependencies, shown in Figure 6. All nodes in the diagram are dependencies, and the arrows between them represent either negative or positive assurances.[1] In Figure 6, all the soft-goals contribute positively (are positive assurances) to the goal "High-quality release", but all but one contribute negatively to "Release on time". All of these dependencies have to be previously defined in the AD model.

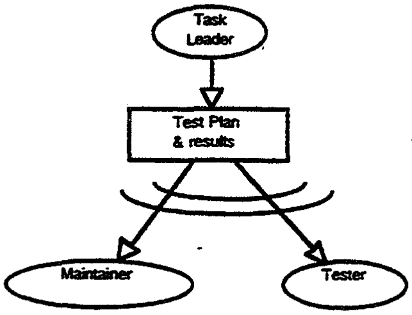


Figure 5: Insurance Representation.

Our suggestion for representing dependency enforcements is a variation of the above. A dependency which enforces another dependency can be seen as one which completely assures it. So our representation uses the same arrows between dependencies shown in Figure 6, with the infinity symbol ("∞") in place of the plus ("+") or minus ("-").

5.3. Use of quantitative data

The use of quantitative data is critical to the useful analysis of development processes and organizations. Without quantitative information, the analysis results are not sufficient to effectively compare alternatives and to make decisions. Qualitative analysis, while important for intuitive understanding and insight, must be taken further to provide a basis for action. For example, [Perry et. al., 1994] have recently attempted to characterize and quantify the workload of software developers across software development process activities.

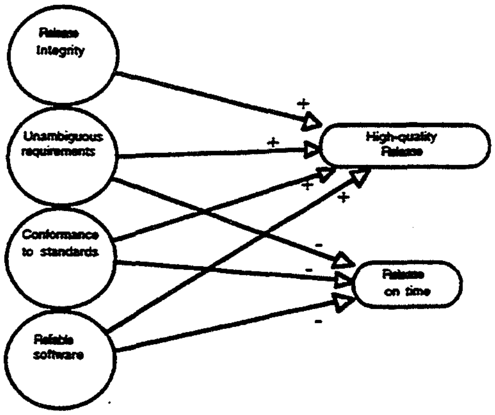


Figure 6: Representing Assurances

In fact, the AD model is particularly well suited to incorporating data, although there is not an explicit facility for this in the modeling methodology. One way to perform such analysis is to associate attributes with the various AD entities (positions, roles, dependencies, etc.). The attributes could be used to hold the quantitative information. Then analysis tools can be used to analyze the AD graph, by making calculations, based on the data, according to the structure represented in the graph.

One type of quantitative analysis, which has already been alluded to, is information flow analysis. Information dependencies (one type of resource dependency) can have attached to them attributes such as frequency and amount of information. Each information dependency is also related to the different communication media (the entity Medium in Figure 4) that it uses to pass information, e.g. phone, email, formal and informal documents, formal and informal meetings. The many-to-many relationships between dependencies and their media also have attributes (e.g., effort). Such attributes are captured by defining metrics and collecting the appropriate data. An example of such an attribute is the computation, for each information dependency, of the product of the dependency frequency, the amount of information, and the effort associated with the medium related to the dependency. This product gives a quantitative assessment of the effort expended to satisfy the information dependency. Summing these values for each pair of actors in the AD graph shows how much effort the

[1] Readers familiar with the work of Yu et al will find this notation similar to their Issue Argumentation model, which we did not make use of in our work. However, our notation which we present here has different semantics than the IA model, and the two should not be confused.

pair expends in passing information to each other. This information can be used to support such management decisions as how to fill different positions, how to locate these people, and what communication media to make available. Without quantitative analysis, these decisions are subject to guesswork, trial and error, and the personal expertise of the manager. For more on metrics for organizational information flow, see [Seaman, 1994].

There are several possible applications of quantitative analysis in relation to the actor/position/role decomposition. For example, during the course of our study, we noticed that many differences between projects were reflected in variations in the breakdown of positions into roles. In other words, the people filling the same positions in different projects divided their effort differently among their various roles. These variations were usually symptomatic of differences in management strategy and leadership style. Data needs to be collected to capture the important variations in effort breakdown across organizations and projects. This data must then be attached to entities in the AD model so that it can be used to analyze variations in job structure. For example, suppose that we wanted to find out which projects require a manager with technical expertise. If we have quantitative data available on the effort breakdown of the different managers, then we can easily see which managers spend a high proportion of their time on technical activities. This information can be used in choosing people to fill different management positions. Variations in effort breakdown can also be represented in an AD graph by varying the thickness of the lines which join a position with its various roles, as shown in Figure 7.

Effort breakdown is only one example of the many possibilities for analysis of the role/position/agent structure of actors. Qualification analysis, which would involve the Qualifications entity in Figure 4, is another example. Understanding the sharing of tasks and responsibilities is another area in which quantitative analysis could be useful. All of these involve the evaluation of quantitative attributes attached to roles, positions, agents, and the links (occupies, contains, performs) between them.

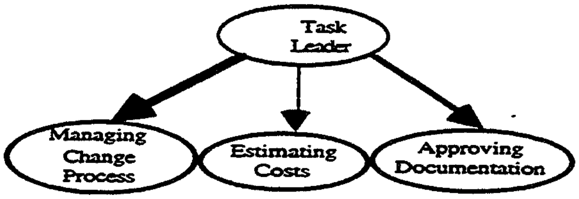


Figure 7. Representing effort breakdown per role

5.4. Acquisition process

Any modeling effort requires that a great deal of information be collected from the environment being modeled. Building an AD model requires collecting information about all the people in the environment, the details of their jobs and assignments, whom they depend on to complete their tasks and reach their goals, etc. Our experience has shown that it is useful to follow a defined process for gathering this information, which we will call an *acquisition process*. The acquisition process which we followed, with modifications motivated by our experience, is briefly presented in this section. The steps are as follows:

Step 1: First, we determined the official, (usually) hierarchical structure of the organization. Normally this information can be found in official organization charts. This gives us the set of positions and the basic reporting hierarchy.

Step 2: We determine the roles covered by the positions by interviewing the people in each position, and then, to check for consistency, their supervisors and subordinates. Process descriptions, if available, often contain some of this information. However, when using process descriptions, the modeler must check carefully for process conformance.

Step 3: In this step, we focus on the goal, resource, and task dependencies that exist along the vertical links in the reporting hierarchy. To do this, we interview members of different departments or teams, as well as the supervisors of those teams. Also, direct observation of supervisors, called "shadowing", can be useful in determining exactly what is requested of, and provided by supervisors for their subordinates.

Step 4: Here we focus on resource (usually informational) and goal dependencies between members of the same team. Direct observation (through shadowing or observation of meetings) is also useful here. Interviews and process documents can also be used to identify dependencies.

Step 5: Finally, we determine the informational and goal dependencies between different teams. These are often harder to identify, as they are not always explicit. Direct observation is especially important here, as often actors do not recognize their own subtle dependencies on other teams. It is also very important in this step to carefully check for enforcement, assurance, and insurance mechanisms, since dependers and dependees work in different parts of the management hierarchy.

6. Conclusions

This paper presents the experience of using the Actor-Dependency modeling approach to model and analyze a large scale maintenance organization. The AD model was found to be very useful at capturing the important properties of the organizational context of the maintenance process, and aided in the understanding of the flaws found in this process. There were, however, some inadequacies of the approach, which we have addressed through a set of proposed suggestions. However, those must be seen as research directions and need to be further investigated.

One major potential extension of the approach is to use quantitative data and analysis methods, within the framework of an AD model. Qualitative methods are not sufficient to differentiate organizations and especially variations across projects. Measurement is therefore necessary for studying organizations.

The AD model also needs automated support for real-scale organizations. This is required to allow the user to analyze a real-time organization and define complementary views of the studied organizations, at different levels of refinement, at different levels of completeness. Automated support is especially crucial for the use of quantitative analysis.We need also to better define the relationship between the organization and the development process. Separating organizational concerns from process concerns, but considering them in conjunction with each other, is a crucial element in the comprehensive study of development environments (see [Seaman, 1994]). Finally, collecting information about an organization for building an accurate AD model is a complex task. Therefore, based on experience, we need to define an optimal data acquisition process that can be tailored to various maintenance environments.

Acknowledgements

We want to thank the referees and Khaled El Eman for their suggestions that helped improve both the content and the form of this paper. We would also like to thank the people at NASA and CSC who participated in this work.

References

Basili, V.R. (1989). "Software Development: A Paradigm for the Future". In *Proceedings, COMPSAC '89*, Orlando, FL, September.

Benus, B. (1994). "Detailed Design Document; Organisation Model Tool". Technical Report KADS-II/M6/UvA/057/1.0, University of Amsterdam, May.

Briand, L. C.; Basili, V. R.; Kim, Y.M.; Squier, D. R. (1994). "A Change Analysis Process to Characterize Software Maintenance Projects". In *Proc. of the IEEE Int'l Conf. on Software Maintenance*. Victoria, Canada, September.

Curtis, B.; Kellner, M.I.; Over, J. (1992). "Process Modeling". *Communications of the ACM*, 35(9):75-90, September.

Finkelstein, A.; Kramer, J.; Nuseibeh, B., eds.(1994). *Software Process Modeling and Technology*. Research Studies Press (distributed by Wiley & Sons).

Lanphar, R. (1990). "Quantitative Process Management in Software Engineering, a Reconciliation Between Process and Product Views". *Journal of Systems and Software*, 12:243-248.

Lott, C.M.; Rombach, H.D. (1993). "Measurement-based Guidance of Software Projects Using Explicit Project Plans". *Information and Software Technology*, 35(6/7):407-19, June/July.

Melo, W. and Belkhatir, N. (1994). "Collaborating Software Engineering Processes in Tempo". *Journal of the Brazilian Computer Society*, 1(1):24-35.

Perry, D.; Staudenmayer, N.; Votta, L. (1994), "People, Organizations, and Process Improvement". *IEEE Software*, 11(4):36-45.

Rein, G.L. (1992), "Organization Design Viewed as a Group Process Using Coordination Technology". MCC Technical Report No. CT-039-92, February.

Seaman, C.B. (1994), "Using the OPT Improvement Approach in the SQL/DS Development Environment". In *Proceedings of CASCON'94*, (CD-ROM version), Toronto, Canada, October.

Yu, E.; Mylopoulos, J. (1993). "An Actor Dependency Model of Organizational Work - with Application to Business Process Reengineering". In *Proc. Conference on Organizational Computing Systems (COOCS 93)*, Milpitas, CA, November.

Yu, E. S.; Mylopoulos, J. (1994). "Understanding 'Why' in Software Process Modeling, Analysis, and Design". In *Proc. of the 16th IEEE Int'l Conf. on Software Engineering*. Sorrento, Italy. pp. 159-168.

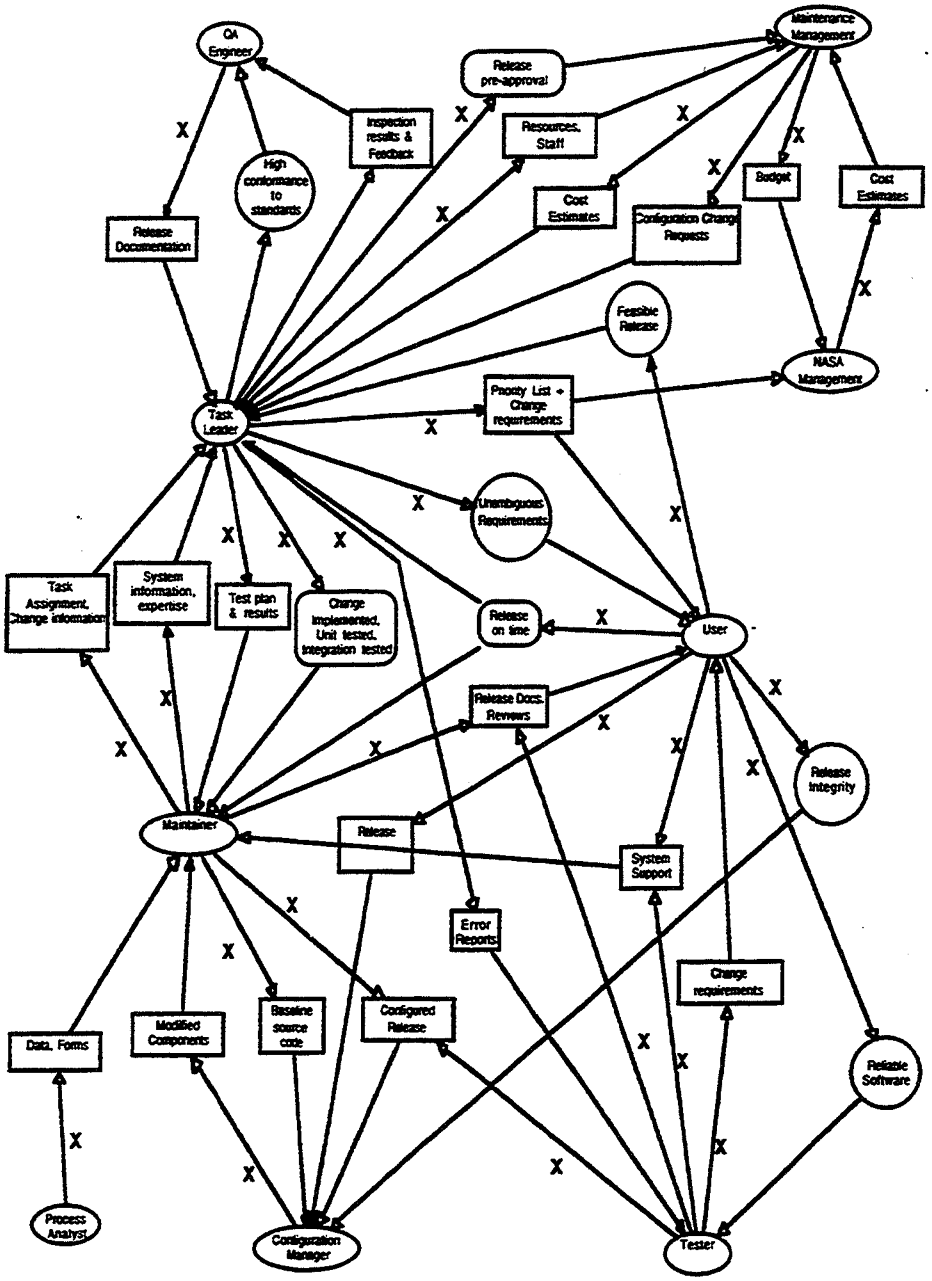


Figure 3: AD Model of a Maintenance Organization.

OMIT
THIS
PAGE

SECTION 4—SOFTWARE MEASUREMENT

The technical papers included in this section were originally prepared as indicated below.

- *Goal-Driven Definition of Product Metrics Based on Properties*, L. Briand, S. Morasca, and V. R. Basili, University of Maryland, Computer Science Technical Report, CS-TR-3346, UMIACS-TR-94-106, December 1994
- *Property-based Software Engineering Measurement*, L. Briand, S. Morasca, and V. R. Basili., University of Maryland, Computer Science Technical Report, CS-TR-3368, UMIACS-TR-94-119, November 1994
- *An Analysis of Errors in a Reuse-Oriented Development Environment*, W. M. Thomas, A. Delis, and V. R. Basili, University of Maryland, Computer Science Technical Report, CS-TR-3424, UMIACS-TR-95-24, February 1995
- *A Validation of Object-Oriented Design Metrics*, V. R. Basili, L. Briand, and W. L. Melo, University of Maryland, Computer Science Technical Report, CS-TR-3443, UMIACS-TR-95-40, April 1995

Page intentionally left blank

Goal-Driven Definition of Product Metrics Based on Properties

S5-61
415780
361 098
24P

Lionel Briand*, Sandro Morasca**, Victor R. Basili*

* Computer Science Department
University of Maryland, College Park, MD, 20742
{lionel, basili}@cs.umd.edu

** Dipartimento di Elettronica e Informazione
Politecnico di Milano
Piazza Leonardo da Vinci 32, I-20133 Milano, Italy
morasca@elet.polimi.it

Abstract

Defining product metrics requires a rigorous and disciplined approach, because useful metrics depend, to a very large extent, on one's goals and assumptions about the studied software process. Unlike in more mature scientific fields, it appears difficult to devise a "universal" set of metrics in software engineering, that can be used across application environments.

We propose an approach for the definition of product metrics which is driven by the experimental goals of measurement, expressed via the GQM paradigm, and is based on the mathematical properties of the metrics. This approach integrates several research contributions from the literature into a consistent , practical and rigorous approach.

The approach we outline should not be considered as a complete and definitive solution, but as a starting point for discussion about a product metric definition approach widely accepted in the software engineering community. At this point, we intend to provide an intellectual process that we think is necessary to define sound software product metrics. A precise and complete documentation of such an approach will provide the information needed to make the assessment and reuse of a new metric possible. Thus, product metrics are supported by a solid theory which facilitates their review and refinement. Moreover, their definition is made less exploratory and, as a consequence, one is less likely to identify spurious correlations between process and product metrics.

1. Introduction

Metrics can help address some of the most critical issues in software development and provide support for planning, monitoring, controlling and evaluating the software process. However, past approaches for designing new software metrics very seldom addressed a specific objective explicitly,

This work was supported in part by NASA grant NSG–5123, UMIACS, and NSF grant 01-5-24845. Sandro Morasca was also supported by grants from MURST and CNR.

and were usually not based upon assumptions/information about the characteristics of the development environment under study. These include descriptions of organizational structure and work procedures, guidelines, standards, etc. This frequently led to some degree of fuzziness in the metric definitions, properties, and underlying concepts, making the use of the metrics difficult, their interpretation hazardous, and the results of the various validation studies somewhat contradictory [IS88, K88].

As a consequence, the number of available metrics in the literature is quite large, but the number of used and useful metrics in industry is small. It is our position that, in order to make software measurement a viable part of the solutions to software engineering issues, metrics must be defined according to clear assumptions about the process under study and an explicit definition of the specific goal(s) to be addressed. Based on these goals and assumptions, desirable metric properties may be identified and used to direct and constrain the search for metrics. *Such an approach appears particularly necessary for product metrics since these metrics are often more complex than process metrics and address phenomena that are poorly understood.*

The goal of this paper is to specify (based on our experience [BMB93, BBH93, BMB94(a)]) a practical metric definition approach, specifically aimed at product metrics, and usable as a practical *guideline* to design technically sound and useful metrics. The focus will be the construction of prediction systems, which is a crucial application of measurement. Not all activities in this approach can, at this point, be fully formalized, nor do we believe that they will be completely formalized in the future. We think that formal techniques can be very effective in providing support for better understanding and analyzing software processes and products—indeed, we advocate the need for a formal definition of metrics' mathematical properties. However, the definition of a metric is a very human-intensive activity, which cannot be described and analyzed in a fully formal way. We believe that our metric definition approach may be better detailed, refined, and tailored to fit the needs of different application contexts. This will be made possible through the experience gained by using this metric definition approach across several environments. Thus, this work should be considered as a contribution towards a satisfactory solution. We point out what information ought to be provided when one proposes a new metric in order to make its review and refinement possible. Furthermore, we determine what intellectual process one should go through to ensure the technical soundness and practical usefulness of the defined metrics. A purely exploratory approach to metric definition would have for a consequence the experimental evaluation of a large number of relationships between product metrics (possibly not supported by any theory) and development process characteristics (e.g., effort). A simple probability calculation [F91] shows that this kind of approach is likely to lead to the identification of spurious statistical relationships, e.g., correlations uniquely due to coincidence.

Several important research issues involved in the definition of such an approach have already been investigated. Basili et al. [B92] [BR88] have provided templates to define operational experimental goals for software measurement. Melton et al. have studied product abstraction properties [MGB90]. Weyuker [W88] and Tian and Zelkowitz [TZ92] have studied desirable properties for complexity metrics. In addition, the latter authors provided a property-based classification scheme for such metrics. Fenton and Melton [FM90], and Zuse [Z90] have investigated the use of measurement theory to determine measurement scales. Finally, Schneidewind has proposed a validation framework for metrics [S92]. All this research needs to be integrated into a consistent and practical metric definition approach.

The paper is organized as follows. In the next section, we provide an overview of a practical metric design approach in part inspired by the work referenced above. and augmented with some new ideas. Then, in the subsequent sections, we separately show each step of our metric design approach in detail (Sections 3-8). Section 9 outlines the directions for future work.

2. Overview of Our Metric Definition Approach

We provide here an overview of the steps composing this approach, as illustrated in Figure 1 by a Data Flow Diagram. The remaining sections will go in detail through all the issues involved in each of the steps and will provide examples.

Step 1: Define Experimental Goal(s)

Define the experimental goal(s) of the data collection, based on the general corporate objectives (e.g. reduce cycle time) and the available information about the studied development environment (e.g., weaknesses, problems). This step requires goal definition techniques. The Goal/Question/Metric paradigm (GQM) [B92] [BR88] is one of the approaches that can be used to this end. It provides a set of templates to define experimental goals and refines them into concrete and realistic questions, which subsequently lead to the definition of metrics. For instance, a GQM goal is:

Analyze *software components* for the purpose of *prediction* with respect to *the number of faults* from the viewpoint of *the project manager*.

(We will use this very simple example to illustrate the steps of our approach during this concise overview.) A GQM goal specifies the object(s) of study (*software components*), the purpose of measurement (*prediction*), the quality focus of interest *(the number of faults)*, and viewpoint (*project manager)* from which measurement is performed. The goal strongly impacts all other steps of the metric definition approach and the information they need. For instance, the object of study and the viewpoint are used to determine the product artifacts and information to be taken into account. The GQM paradigm uses descriptive models (e.g., definition of complexity metrics) and predictive models (e.g., cost models) in order to achieve the experimental goals it specifies. However, the GQM paradigm does not specify how to generate these models. In this paper, we expand the GQM paradigm to address this issue with respect to product descriptive models. As we will see in Section 6, questions about product characteristics are no longer necessary in our approach. However, GQM questions on the confidence with which assumptions are stated and on the quality (e.g., accuracy of collection procedures, granularity) of data to be collected [B92, BR88] still need to be asked. We will not address this issue, which is beyond the scope of this paper.

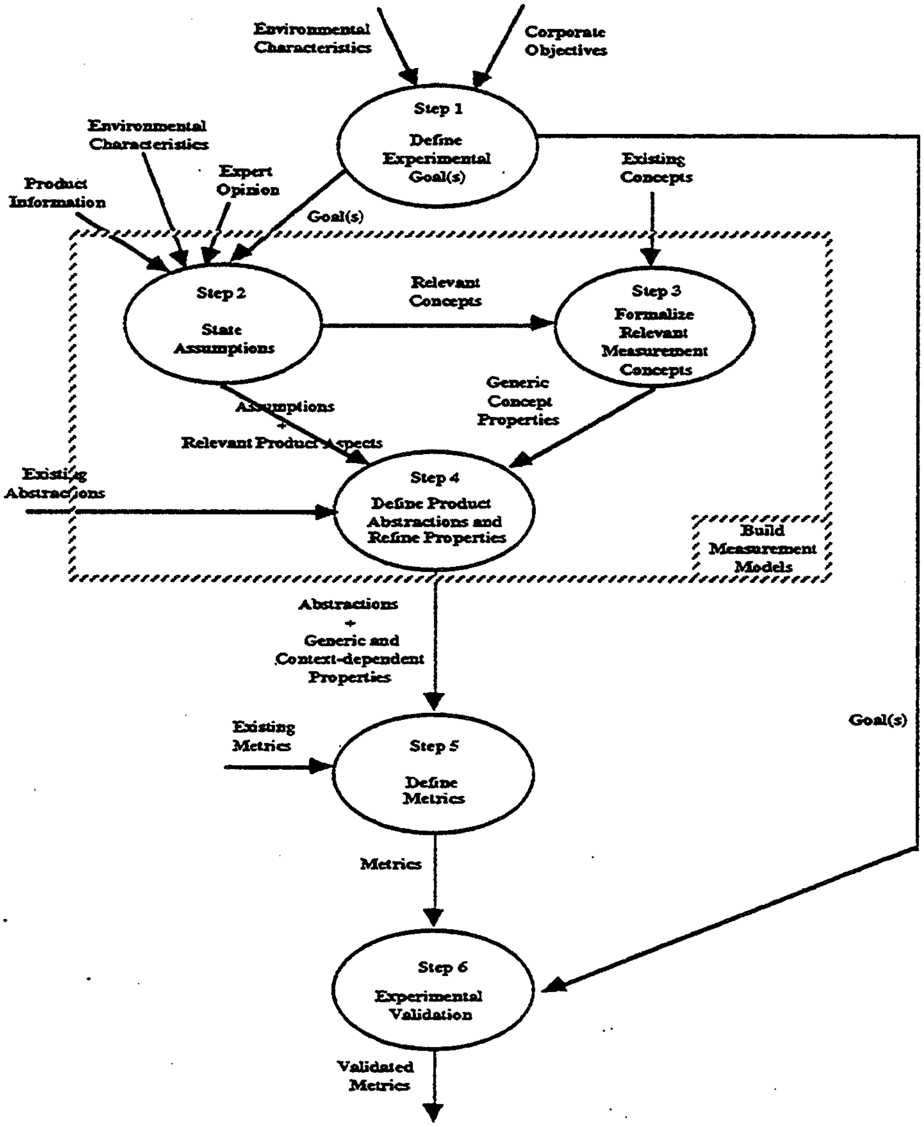

Environmental Characteristics
Corporate Objectives
Step 1
Define Experimental Goal(s)
Environmental Characteristics
Expert Opinion
Product Information
Goal(s)
Existing Concepts
Step 2
State Assumptions
Relevant Concepts
Step 3
Formalize Relevant Measurement Concepts
Assumptions + Relevant Product Aspects
Generic Concept Properties
Existing Abstractions
Step 4
Define Product Abstractions and Refine Properties
Build Measurement Models
Abstractions + Generic and Context-dependent Properties
Existing Metrics
Step 5
Define Metrics
Goal(s)
Metrics
Step 6
Experimental Validation
Validated Metrics

Figure 1. Goal-Driven and Property-Based Definition Approach for Product Metrics

Step 2: State Assumptions

Based on the object of study and the quality focus (as defined by the experimental goals, Step 1), a set of relevant assumptions must be stated to embody our intuitive knowledge about the development environment and object(s) of study. Assumptions implicitely define an order on the set of objects of study with respect to the quality focus [MGB90]. For instance, components are ordered with respect to their error-proneness. Furthermore, while stating these assumptions, relevant measurement concepts are identified, e.g., size. For instance, based on developers' interviews and a careful study of the development environment, we might assume that the larger the number of sequential blocks of statements and conditional statements in a program, the higher the number of faults. From this assumption, size appears to be a possibly relevant *measurement concept*. As an input for this step, we need information on the development environment (e.g., descriptive process model), product information and expert opinion as an intuitive basis for the assumptions. Besides assumptions, the outputs of this step also include

- a set of relevant measurement concepts (e.g., size)
- a better definition of the relevant aspects of the object of study (e.g., statement blocks and control flow)

Step 3: Formalize Relevant Measurement Concepts

Relevant *measurement concepts* of interest are formally defined (e.g., size, complexity, coupling, cohesion) through their mathematical properties. Thus, they are clearly characterized and the search for metrics is guided and constrained by these generic properties. This makes the search for metrics less exploratory and provides precise mathematical criteria for assessing the soundness of the metrics to be defined. The mathematical properties characterizing the concepts are identified independently from the concept instantiation into a metric [TZ92] [Z90] [W88] and are therefore referred to as *generic concept properties*. With reference to our simple example, we can say that a property of size is that it is non-negative. As opposed to other papers on the subject, we believe that these properties are subjective even though some of them might be widely accepted. However, it appears that, for a matter of convenience, a universal set of properties should be defined for the most important concepts used by the software engineering community, as is the case for more mature engineering disciplines. It is important, when defining metrics, that one precisely determines the meaning of concepts like size or complexity. Existing definitions may, however, be reused when available and, conversely, the newly created concepts may be stored so that they may be eventually reused.

Step 4: Define Product Abstractions and Refine Properties

One needs to define abstractions of the object of study that capture all the information (i.e., objects, attributes, relationships) needed to express the assumptions and the relevant product aspects they refer to. Some examples of product abstractions are data flow graphs, data dependency graphs, and control flow graphs. These abstractions will be representations of the object of study that will help us express useful properties and define metrics. For our example, we may assume that control flow graphs are suitable abstractions with respect to the set of assumptions and the concepts defined.

Once useful abstractions are defined, a set of new properties is added to the generic concept properties. The objective is to formalize the assumptions stated in Step 3: The intuitive ordering of the objects of study (e.g.,components) with respect to the quality focus (e.g., components' error-proneness) must be preserved by the ordering of abstractions (e.g.,components' control flow graphs) with respect to each measurement concept (e.g., components' size) [MGB90]. For instance, under the assumption stated in Step 3, and given two control flow graphs G1 and G2, we can preserve the intuitive ordering captured by the assumption if we define the following size property: the size of G1 is greater than the size of G2 if G1 has more nodes than G2. These additional properties allow us to tailor the generic concepts to any particular quality focus and set of assumptions. It should be noted that the added properties must be consistent with the generic properties defined in Step 2. These added properties are specific to a given context of measurement (i.e., goal, concept, assumptions, abstractions) and are referred to as *context-dependent properties*. At this point, if the defined abstractions are not fully adequate to define the context-dependent properties, this step can be reiterated.

Steps 2, 3, and 4, taken as a whole, can be seen as a macro-step in which *measurement models* [F91] (i.e., abstractions and generic/context-dependent properties, main outputs of Step 4) are defined based on the experimental goals, environmental characteristics, and product information (inputs of Step 2).

Step 5: Define Metrics

Metrics are defined based upon the defined product abstraction(s), concepts and their associated properties. Existing metrics can also be reused if they satisfy the defined properties. With respect to our example, size can be simply measured as the number of nodes in a control flow graph. We are not able, at this point, to select optimal metrics from those metrics satisfying the generic and context-dependent properties. Experimental validation (Step 6) will help us do so.

Step 6: Experimental Validation

After defining metrics in Step 5, the data collected on the actual products must be used to validate the assumptions upon which the metrics are built. The procedure to follow for experimental validation varies significantly depending on the purpose of measurement. With respect to prediction, which is our main focus here, one needs to validate the product metrics with respect to their statistical relationship to the quality focus of interest. For example, we might find a very strong correlation between the defined size metric and a simple descriptive model of error-proneness, e.g., the number of faults. If the assumptions are not supported by the experimental results, we need to repeat from Step 2, re-consider the assumptions and properties, then re-define new metrics. The definition and validation of metrics are performed iteratively until the metric validation yields satisfactory results [S92].

It is important to mention that most of the outputs (e.g., product abstractions, assumptions) of the steps defined above are reusable. They should be packaged and stored so that they can be efficiently and effectively reused [BR88]. In a mature development environment, inputs for most of those steps should come from reused knowledge.

Moreover, many refinement loops are not represented in Figure 1. For example, as we said in the description of Step 6, poor experimental results may trigger the need for refining assumptions. This is an important issue that needs further investigation.

In the remainder of this paper, we will use this definition approach to define data flow size and complexity metrics as simple examples. Each step will be discussed in detail in a different section. Each section contains three subsections:

- Definition of the step
- Examples
- Discussion of related issues.

3. Define Experimental Goal(s) (Step 1)

Definition

In this section, we apply the first step of the Goal/Question/Metric paradigm [B92, BR88] to set the measurement goals. Here is a summary of templates that can be used to define goals:

Object of study: products, processes, resources
Purpose: characterization, evaluation, prediction, improvement, ...
Quality focus: cost, correctness, defect removal, changes, reliability, ...
Viewpoint: user, customer, manager, developer, corporation, ...

A detailed description of the GQM paradigm is beyond the scope of the paper. A comprehensive description of the GQM paradigm can be found in [B92, BR88].

It is important to note that the four goal dimensions mentioned above have a direct impact on the remaining steps of the metric definition approach and, from a more general perspective, the whole data collection program. This can be summarized as follows:

The *object of study* helps determine the

- software artifacts that are to be modeled by mathematical abstractions in order to be analyzable (Step 4).

- assumptions (Step 2) that may be relevant because related to the object of study.

The *purpose* points out what is the intended use of the metrics to be defined and therefore the

- type of data to be collected, e.g., process improvement requires additional data over process prediction (e.g., with respect to development effort), in order to allow for the determination of optimal techniques and methods. For example, performance data are needed in sufficient amount to ensure a minimal level of confidence in the improvement decisions.
- amount of data to be collected, e.g., if prediction usually requires more data than characterization so that the identified relationships are statistically significant. Characterization only requires the data to be representative of what is to be characterized.

The *quality focus* helps determine the

- dependent variable against which the defined product metrics are going to be experimentally validated (Step 6) [S92]. This dependent variable will in fact be a descriptive model of the quality focus. For instance, number of requirement changes per month per thousand of lines of code is a descriptive model of requirement instability. Since there may be alternative models, validation may require the use of several dependent variables. In this case, if inconsistent experimental results are obtained, the dependent variables are very likely to actually capture different quality focuses.
- assumptions (Step 2) linking the object of study characteristics to the quality focus of interest.

The *viewpoint* helps determine

- the point in time at which characterizations, predictions, or evaluations should be carried out and therefore what product information will be available to define product abstractions and metrics (Steps 4, 5).
- what information is costly or difficult to acquire and consequently, what information should be left out of the model if it does not show a sufficiently strong impact on the quality focus (Steps 5, 6).
- the definition of descriptive models of the quality focus. For example, from the user's point of view, error-proneness may be defined as the mean time to failure, whereas, from the tester point of view, it may be defined as the number of errors occuring during the test phase.

In this framework, we will not derive questions from goals as suggested by the GQM paradigm. A justification will be provided in Section 6.

Example of a goal

Let us assume that one of the corporate objectives is to reduce development time, and more particularly the time spent on testing activities. Assuming that previous studies have shown that errors are usually concentrated in a small number of "difficult" components (example of information about the development environment), the following experimental goal seems pertinent. By identifying error-prone components, we may concentrate verification activities where needed and, thereby, reduce effort.

Goal G

Object of study: component
Purpose: prediction
Quality focus: error-proneness
Viewpoint: tester

Let us take an example to illustrate the impact of the defined experimental goal on our metric definition approach. We know from the *object of study* that we have to define relevant component mathematical abstractions so we can derive component metrics. We know from the *purpose* of measurement that we need to collect enough data about the quality focus to allow a statistically significant validation of the relationships between the component metrics to be defined and the quality focus. This requires that we better define our quality focus: error-proneness. Very likely, we need to determine precisely how to count defects, e.g., what testing and inspection phases should be taken into account?, are all errors equal or should they be weighted according to a predefined error taxonomy? Such questions are also dependent on the particular *viewpoint*. In our example, testers want to find out where errors are and more particularly critical errors (according to their own definition of criticality). Therefore, errors will be weighted according to the level of criticality of their consequences. Similarly, errors could be weighted according to the correction effort they require. The determination of suitable error counting procedures will depend on the particular application of the predictive model to be built and therefore on the viewpoint of our experimental goal.

In the next sections, we will discuss more precisely about the impact of experimental goals on the definition of software product metrics.

Discussion

The definition of the goals is a fundamental phase, since all other steps in our approach are affected by the experimental goals. Therefore, extra care must be used when setting the goals. Specific descriptive process models and knowledge acquisition techniques can be used to better understand the issues that are most relevant to software development in a software organization. Careful application of the GQM paradigm provides two important results:

- Data collection is ensured to respond to the specific needs of the software organization;
- The derivation of metrics from explicit goals and the definition of explicit measurement models (output of Step 4 of our approach) allow the analyst to specify *a priori* the interpretation mechanisms associated with the collected data. This prevents *a posteriori* search for patterns which are not based on precise assumptions.

4. State Assumptions (Step 2)

Definition

We have to state assumptions (see examples below) about some aspects of the software process under study that are relevant to the experimental goals. These assumptions capture our intuitive understanding of the studied phenomena and need to be explicit so they can be discussed, questioned and refined. Various sources of information can be used to devise pertinent assumptions. A thorough understanding of the working procedures, methodologies and techniques used in the studied development environment, combined with the interview of domain experts, is usually very helpful [BBK94]. The set of assumptions defines an ordering on the set of products [MGB90] with respect to the quality focus. This ordering will be used to evaluate the adequacy of the metrics defined in the remainder of this approach.

An *assumption* is a statement believed to be true about the relationship between the *quality focus* and the characteristics of the *object of study*.

Stating assumptions helps identify the measurement concepts (e.g., size, complexity) that are characteristics of the object of study relevant to the goal. In addition, assumptions allow us to identify artifacts, or parts of artifacts (e.g., definitions, condition expressions), that must be taken into account for the definition of suitable product abstractions.

Examples of assumptions

In order to capture our intuitive understanding about data flow size and complexity, we define the following assumptions.

Assumption 1:
The larger the number of definitions and condition expressions, the larger the likelihood of error.

Assumption 2:
The larger the number of definitions and condition expressions "depending" on a definition D, the larger the probability of ripple effects if D is to be created or modified.

Assumption 3:
The larger the number of definitions on which a definition or a condition expression D "depends", the more difficult it is to create and understand D.

Assumption 4:
The larger the "distance" between two definitions or condition expression D1 and D2, where D2 depends on D1, the more difficult the control of ripple effects on D2 if D1 is to be created or modified.

The concepts between quotes are not defined: they make sense on an intuitive level. They will be formally defined later, either via the definition of product abstractions (as is the case of "dependency"), or additional concept properties in Step 4 (as is the case of "distance").

Discussion

At this point, several sets of consistent assumptions could be defined. This would lead to multiple categories of metrics, reflecting the inherent uncertainty associated with the assumptions. In Step 6, experimental results will eventually help us select the best category of metrics for each concept. For example, we could assume that when a condition expression CE (as opposed to a definition) depends on a definition D, this increases the probability of misunderstanding and ripple effect between D and CE. This stems from the fact that condition expressions also have an implicit effect on the definitions in the block they control. This additional assumption (referred to as Assumption 5) affects the metric definition approach, as we show in the following steps.

5. Formalize Relevant Measurement Concepts (Step 3)

Definition

The relevant measurement concepts are defined by specifying the mathematical properties that are believed to characterize them. In our framework, these properties should be used to constrain and guide the search for new metrics. In addition, as shown in [BMB94(b)], intuition may lead to properties showing awkward mathematical properties[1]. One should always make sure that a metric exhibits properties that are essential for its technical soundness. These properties are independent from both any specific product abstraction and any future instantiation of the concept into any specific metric. Therefore, they are called generic.

[1] The authors of this paper were several times misled in the definition of software metrics that were intuitively appealing, but, after a more thorough analysis, showed inconvenient and unsubstantiated properties.

A *measurement concept* is a class of metrics characterized by a set of mathematical properties (i.e., *generic concept properties*) and associated with an intuitive software product characteristic, e.g., size.

The generic properties associated with a measurement concept should not be contradictory—there must be at least one metric that satisfy them. Moreover, these properties should hold for the admissible transformations [Z90] of the scale of measurement (i.e., nominal, ordinal, interval, ratio, absolute) on which it is intended to define metrics. In other words, there should not be any contradiction between the scale of measurement which is assumed while using and interpreting a defined metric and its generic properties.

Examples of concepts and their generic properties

In this example, we provide properties that are, in our opinion, generic for metrics related to size and complexity. These concepts are believed to be relevant with respect to many experimental goals and applications, and in particular with respect to the goal defined above. As for complexity, the properties we define are related to the properties several authors have already provided in the literature (see [LJS92, TZ92, W88]). However, since we may want to use these properties on artifacts other than software code and on abstractions other than control-flow graphs, we formalized them in a more general manner. A thorough discussion of these properties—which is beyond the scope of this paper—can be found in [BMB94(b)]. These properties are provided as an example. Nevertheless, in the metric definition approach we outline in this paper, other sets of properties [TZ92] [W88] may be used, since the selection of properties is, to some extent, subjective.

Size and complexity are concepts related to systems, in general, i.e., one can speak about the size of a system and the complexity of a system. In our general framework—recall that we want these properties to be as independent as possible from any specific product abstraction—, a system is characterized by its elements and the relationships between them.

Definition 1: Representation of Systems and Modules

A *system* S will be represented as a pair <E,R>, where E represents the set of elements of S, and R is a binary relation on E ($R \subseteq E \times E$) representing the relationships between S's elements.

Given a system S = <E,R>, a system m = $<E_m,R_m>$ is a *module* of S if and only if $E_m \subseteq E$, $R_m \subseteq E \times E$, and $R_m \subseteq R$. This will be denoted by $m \subseteq S$.

◊

As an example, E can be defined as the set of code statements and R as the set of control flows from one statement to another. A module m may be a code fragment or a subprogram.

Concept: Size

Intuitively, size is recognized as being an important measurement concept. According to our framework, size cannot be negative (property Size.1), and we expect it to be null when a system does not contain any elements (property Size.2). When modules do not have elements in common, we expect size to be additive (property Size.3).

Definition 2: Size
The size of a system S is a function Size(S) that is characterized by the following properties Size.1 - Size.3.

◊

Property *Size.1*: Non-negativity
The size of a system S = <E,R> is non-negative

$$\text{Size}(S) \geq 0 \qquad \text{(Size.I)}$$

◊

Property *Size.2*: Null Value
The size of a system S = <E,R> is null if E is empty

$$E = \varnothing \Rightarrow \text{Size}(S) = 0 \qquad \text{(Size.II)}$$

◊

Property *Size.3*: Module Additivity
The size of a system S = <E,R> is equal to the sum of the sizes of two of its modules $m_1 = <E_{m1},R_{m1}>$ and $m_2 = <E_{m2},R_{m2}>$ such that any element of S is an element of either m_1 or m_2

$$(m_1 \subseteq S \textbf{ and } m_2 \subseteq S \textbf{ and } E = E_{m1} \cup E_{m2} \textbf{ and } E_{m1} \cap E_{m2} = \varnothing) \Rightarrow \text{Size}(S) = \text{Size}(m_1) + \text{Size}(m_2) \qquad \text{(Size.III)}$$

◊

The last property Size.3 provides the means to compute the size of a system S = <E,R> from the knowledge of the size of its—disjoint—modules $m_e = <\{e\},R_e>$ whose set of elements is composed of a different element e of E[2].

$$\text{Size}(S) = \sum_{e \in E} \text{Size}(m_e) \qquad \text{(Size.IV)}$$

Therefore, adding elements to a system cannot decrease its size

[2] For each m_e, it is either $R_e = \varnothing$ or $R_e = \{<e,e>\}$.

$(S' = <E',R'> \textbf{ and } S'' = <E'',R''> \textbf{ and } E' \subseteq E'') \Rightarrow Size(S') \leq Size(S'')$ (Size.V)

From the above properties Size.1 - Size.3, it also follows that the size of a system $S = <E,R>$ is not greater than the sum of the sizes of any pair of its modules $m_1 = <E_{m1},R_{m1}>$ and $m_2 = <E_{m2},R_{m2}>$, such that any element of S is an element of m_1, or m_2, or both, i.e.,

$(m_1 \subseteq S \textbf{ and } m_2 \subseteq S \textbf{ and } E = E_{m1} \cup E_{m2}) \Rightarrow Size(S) \leq Size(m_1) + Size(m_2)$ (Size.VI)

The size of a system built by merging such modules cannot be greater than the sum of the sizes of the modules, due to the presence of common elements (e.g., lines of code, operators, class methods).

Properties Size.1-Size.3 hold when applying the admissible transformation of the ratio scale [F91]. Therefore, there is no contradiction between our concept of size and the definition of size metrics on a ratio scale.

Concept: Complexity

Intuitively, the complexity of a product is a measurement concept that is considered extremely relevant to system properties. It has been studied by several researchers [BMB94(b)]. In our framework, we expect product complexity to be non-negative (property Complexity.1) and to be null (property Complexity.2) when there are no relationships between the elements of a system. However, it could be argued that the complexity of a system whose elements are not connected to each other does not need to be necessarily null, because each element of E may have some complexity of its own. In our view, complexity is a system property that depends on the relationships between elements, and is not an isolated element's property [BMB94(b)].

Complexity should not be sensitive to representation conventions with respect to the direction of arcs representing system relationships (property Complexity.3). A relation can be represented in either an "active" (R) or "passive" (R^{-1}) form. The system and the relationships between its elements are not affected by these two equivalent representation conventions, so a complexity metric should be insensitive to this.

Also, the complexity of a system S should be at least as much as the sum of the complexities of any collections of its modules, such that no two modules share relationships, but may only share elements (property Complexity.4). *We believe that this property is the one that most strongly differentiates complexity from the other system concepts.* Intuitively, this property may be explained by two phenomena. First, the transitive closure of R is a larger graph than the graph obtained as the union of the transitive closures of R' and R" (where R' and R" are contained in R). As a consequence, if any kind of *indirect* (i.e., *transitive*) relationships between elements is considered in the computation of complexity, then the complexity of S may be larger than the sum of its modules' complexities, when the modules do not share any relationship. Otherwise, they are equal.

Second, merging modules may implicitely generate relationships (note $R' \cup R'' \subseteq R$ in formula Complexity.IV's premise) between the elements of each module (e.g., definition-use relationships may be created when blocks are merged into a common system). As a consequence of the above properties, system complexity should not decrease when the set of system relationships is increased (property Complexity.4).

Last, the complexity of a system made of disjoint modules is the sum of the complexities of the single modules (property Complexity.5). Consistent with property Complexity.4, this property is intuitively justified by the fact that the transitive closure of a graph composed of several disjoint subgraphs is equal to the union of the transitive closures of each subgraph taken in isolation. Furthermore, if two modules are put together in the same system, but they are not merged, i.e., they are still two disjoint module in this system, then no additional relationships are generated from the elements of one to the elements of the other.

Definition 3: Complexity

The complexity of a system S is a function Complexity(S) that is characterized by the following properties Complexity.1 - Complexity.5.

◊

Property *Complexity*.1: Non-negativity

The complexity of a system S = <E,R> is non-negative

$$\text{Complexity}(S) \geq 0 \qquad \text{(Complexity.I)}$$

◊

Property *Complexity*.2: Null Value

The complexity of a system S = <E,R> is null if R is empty

$$R = \emptyset \Rightarrow \text{Complexity}(S) = 0 \qquad \text{(Complexity.II)}$$

◊

Property *Complexity*.3: Symmetry

The complexity of a system S = <E,R> does not depend on the convention chosen to represent the relationships between its elements

$$(S = \langle E,R \rangle \textbf{ and } S^{-1} = \langle E,R^{-1} \rangle) \Rightarrow \text{Complexity}(S) = \text{Complexity}(S^{-1}) \qquad \text{(Complexity.III)}$$

◊

Property *Complexity*.4: Module Monotonicity

The complexity of a system S = <E,R> is no less than the sum of the complexities of any two of its modules with no relationships in common

$$(S = \langle E,R\rangle \textbf{ and } m_1 = \langle E_{m1},R_{m1}\rangle \textbf{ and } m_2 = \langle E_{m2},R_{m2}\rangle$$
$$\textbf{and } E_{m1} \cup E_{m2} \subseteq E \textbf{ and } R_{m1} \cup R_{m2} \subseteq R \textbf{ and } R_{m1} \cap R_{m2} = \emptyset)$$
$$\Rightarrow \text{Complexity}(S) \geq \text{Complexity}(m_1) + \text{Complexity}(m_2)$$
$$\text{(Complexity.IV)}$$

◊

Property *Complexity*.5: Disjoint Module Additivity
The complexity of a system $S = \langle E,R\rangle$ composed of two disjoint modules $m_1 = \langle E_{m1},R_{m1}\rangle$, $m_2 = \langle E_{m2},R_{m2}\rangle$ is equal to the sum of the complexities of the two modules

$$(S = \langle E_{m1} \cup E_{m2},R_{m1} \cup R_{m2}\rangle \textbf{ and } E_{m1} \cap E_{m2} = \emptyset \textbf{ and } R_{m1} \cap R_{m2} = \emptyset)$$
$$\Rightarrow \text{Complexity}(S) = \text{Complexity}(m_1) + \text{Complexity}(m_2)$$
$$\text{(Complexity.V)}$$

◊

As a consequence of the above properties Complexity.1 - Complexity.5, it can be shown that the complexity of a system is no less than the complexity of any of its modules, i.e., adding relationships between elements of a system does not decrease its complexity

$$(S' = \langle E,R'\rangle \textbf{ and } S'' = \langle E,R''\rangle \textbf{ and } R' \subseteq R'')$$
$$\Rightarrow \text{Complexity}(S') \leq \text{Complexity}(S'')$$
$$\text{(Complexity.VI)}$$

Properties Complexity.1 - Complexity.5 hold when applying the admissible transformations of the ratio scale. Therefore, there is no contradiction between our concept of Complexity and the definition of Complexity metrics on a ratio scale.

Discussion

The paragraphs above, stating the motivations and justifications for size and complexity concepts, illustrate the subjectivity of the metric definition approach. However, it is important that all concept properties be explicitly justified and motivated so that their limitations may be understood and the discussion on their validity may be facilitated.

6. Define Product Abstractions and Refine Concept Properties (Step 4)

Definition

We first need to define an abstraction that helps us precisely capture and define all the concepts involved in the stated assumptions. Abstractions are mathematical representations of the product(s) (usually graphs). Products have to be mapped into abstractions so they become analyzable and some of their attributes become quantifiable [MGBB90]. The choice should be entirely guided by the experimental goals (i.e., the object of study and the quality focus) and the set of assumptions, that is, the abstractions must capture all the concepts involved in the set of assumptions related to the object of study. The mapping from the product to the abstraction needs to be checked for completeness, i.e., Does the abstraction contain all the relationships between nodes that one wants to capture? Is the level of granularity of the abstraction nodes sufficient to represent accurately the product? One way of assessing the suitability of an abstraction is to study the effect of relevant modifications in the product and assess its impact on the abstraction, e.g., number of nodes and edges added or removed, change of topology in a graph. Several abstractions capturing control flow, data flow and data dependency information are available in the literature [M90, BBC88, O80]. However, an even larger variety of abstractions can be derived from software products.

The set of properties associated with each concept is expanded so as to formalize the order existing on the set of abstractions with respect to each concept as defined by the assumptions. Therefore, the order formalized by the newly introduced properties is intended to preserve the order defined by the assumptions so that concepts have a monotonic relationship with the quality focus of interest. For example, given that the quality focus is error-proneness and that a Definition-Use (D-U) graph DUG1 is defined as more complex than another graph DUG2 and assuming that there is a monotonic relationship between error-proneness and complexity, we expect the assumptions to state that the product corresponding to DUG1 is more error-prone than that of DUG2.

These properties are specific to a given context of measurement (i.e., goal, concept, assumptions, abstractions) and are referred to as *context-dependent properties*. They will, most of the time, capture effects on the ordering of abstractions when modifications are performed on these abstraction. These modifications will often be what is referenced as atomic modifications in [Z90], adding / removing / moving / substituting an edge/node. They will be useful in order to constrain and guide the search for metrics (Step 5).

Examples

In our example, D-U graphs are a suitable abstraction since they capture concepts such as definitions, condition expressions, uses. D-U graphs are directed graphs where nodes are statements or conditions and arcs are definition-use clear paths [RW82]. Moreover, concepts such as "dependencies" or "distance" can be derived from such graphs. A definition or a condition expression "depends" on a definition when the variable/constant defined in the latter is used in the former. A suitable definition of "distance" between two definitions will be provided in the next section.

Concept: Size

Property *CD1*: Count of definitions
If a graph DUG_1 has at least as many definitions and condition expressions as another graph DUG_2, then $Size(DUG_1) \geq Size(DUG_2)$.

◊

The above property CD1 is not implied by the generic properties Size.1-Size.3, since DUG_1 and DUG_2 have nothing to do with each other, i.e., they are not related by any inclusion relationship (DUG_2 is not necessarily included in DUG_1).

Concept: Complexity

Property *CD2*: Sum of distances
Let DUG_1 and DUG_2 be two Definition-Use graphs. If the sum of the distances between all pairs of nodes in DUG_1 is greater than the sum of distances between all pairs of nodes in DUG_2, then $Complexity(DUG_1) > Complexity(DUG_2)$.

◊

The distance between two nodes is the number of arcs in the longest path between the two nodes that contains no repetitions of elementary cycles (cycles that do not traverse the same arc twice). As an example, the distance between nodes b and c in the D-U graph of Figure 2 is 4, i.e., the number of arcs of the path {<b,c>,<c,e>,<e,b>,<b,c>}. In this path, the arc <b,c> is traversed twice, but it is only traversed once in the cycles {<b,c>,<c,e>,<e,b>} and {<c,e>,<e,b>,<b,c>} contained in the path. When several paths exist between two nodes, we select the longest one because the shortest or average path distance would not satisfy the monotonicity property (Complexity.4). For instance, adding an arc in a graph may decrease the length of the shortest path between two nodes. The distance between two unrelated nodes is zero because the absence of relation does not add any complexity, consistent with the generic property Complexity.2. This shows how generic properties constrain the definition of metrics and help make the right decisions. As an example of distance calculations, consider the D-U graph in Figure 2.

If Assumption 5 is considered, a different abstraction is necessary: Data-Dependency (D-D) graphs [BBC88]. This abstraction captures the links between condition expressions and the definitions they can affect. In this case, the following property holds:

Property *CD3*: Definitions versus condition expressions
Let DDG_1 and DDG_2 be two Data Dependency graphs. If DDG_2 is identical to DDG_1 except for the fact that one of the condition expressions of DDG_1 has been substituted with a definition to form DDG_2, then $Complexity(DDG_1) > Complexity(DDG_2)$. In other words, a condition expression is the source of more complexity than a definition.

◊

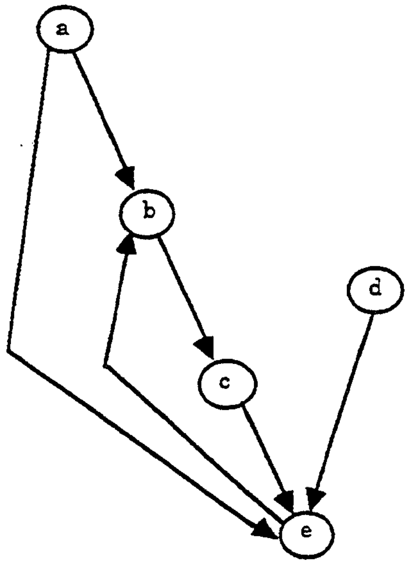


Figure 2. Example of D-U graph

The distances between the nodes in Figure 2 are computed in Table 1.

	a	b	c	d	e
a	0	5	6	0	6
b	0	3	4	0	5
c	0	5	3	0	4
d	0	5	6	0	4
e	0	4	5	0	3

Table 1. Distances between the nodes of the D-U graph in Figure 2

Discussion

According to the GQM paradigm, questions must be derived from goals. In our particular framework, questions about product characteristics (e.g., what is the complexity of a component?) are not necessary and the outputs of Steps 2, 3, and 4 may be seen as a more rigorous substitute to questions. Thus, metrics are not intended to answer questions but to validate assumptions. However, as we have shown, there may be aspects of the relevant environmental characteristics that cannot be explicitly modeled, e.g., the quality of the data and the validity of the assumptions, so questions may still be necessary to support the full interpretation of the metrics.

As pointed out in [FM90, F94], not all abstractions may be comparable with respect to a particular measurement concept. In such cases, it appears difficult to define a total order on the set of abstractions and only a partial order can be obtained [MGB90]. Ultimately, statistical analysis can only be conducted independently on comparable subsets of abstractions.

One of the main difficulties of this step is to ensure that the set of context-dependent properties is complete. Completeness is reached when the properties can fully describe the ordering of abstractions, i.e., when any pair of comparable abstractions can be ordered by using the stated properties or their combination.

It is also necessary to verify that the newly introduced context-dependent properties define metrics whose scales are consistent with those defined by the generic properties, i.e., ratio, interval, ordinal, nominal.

7. Define Metrics (Step 5)

Definition

For each concept, metrics are defined by using the abstractions' elements and relationships and are checked against the concepts' generic and context-dependent properties. Management and resource constraints are taken into account at this point for defining convenient metrics. This step may require approximations which must be performed explicitly, based on a solid theory, and in a controlled manner. At this stage, we are not able to select the best among alternative metrics satisfying generic and context-dependent properties. Experimental validation (Step 6) will help us perform such a selection. As a necessary precondition to carrying out a meaningful experimental validation, the measurement scale (i.e., nominal, ordinal, interval, ratio, absolute [FM90], [Z90]) of the metrics must be clearly identified. This prevents metrics from being misused (e.g., taking the average value of an ordinal metric, which is meaningless).

Examples

Concept: Size

A simple size metric is given by the number of definitions and condition expressions, i.e., the number of nodes in the Definition-Use graph. Other size metrics can be devised, by associating a weight with each node. However, this would require that additional assumptions be made.

Concept: Complexity

The most straightforward metric that comes to mind is the number of arcs in the graph. However, this does not take into account Assumption 4 since distances between pairs of nodes may not have an impact on the metric. In this context, a complexity metric that seems relevant and that satisfies the generic and context-dependent properties is the sum of distances between every pair of nodes in the DUG graph.

$$\mathrm{Cplx(DUG)}=\sum_{i=1}^{|E|}\sum_{j=1}^{|E|}\mathrm{Distance}(\mathrm{Node}_i,\mathrm{Node}_j) \qquad (1)$$

where Node_i, $\mathrm{Node}_j \in E$.

If Assumption 5 and Property CD3 are taken into account, then another complexity metric can be defined as follows

$$\mathrm{Cplx(DDG)}=\sum_{i=1}^{|E|}\sum_{j=1}^{|E|}\mathrm{Distance}(\mathrm{Node}_i,\mathrm{Node}_j) \qquad (2)$$

Note that the formula is identical but the abstraction used is different, i.e., Data-Dependency Graphs (DDG). This metric is therefore different from the one in (1). The weight of condition expressions in formula (2) has increased since path distances are made longer by the link between condition expressions and the definitions that belong to the block they control.

Discussion

Once metrics have been defined, it must be proven that they are consistent with the generic and context-dependent properties. With reference to our examples, it can be easily shown that the metrics we define for size and complexity satisfy their respective sets of generic and context-dependent properties. Thus, they can be shown to preserve the intuitive order defined on the abstractions with respect to the quality focus.

8. Experimental Validation of the Metrics (Step 6)

After defining metrics in Step 5, the data collected on actual software products and processes must be used to validate the metrics experimentally. This is done differently according to the purpose of measurement. With respect to prediction, it is required to validate the assumptions on which the product metrics are based. In other words, significant statistical relationships must be identified between the product metrics and the quality focus (or rather a particular descriptive model of the quality focus) and, furthermore, these relationships must be consistent with what is specified by the assumptions. Validation procedures for other measurement purposes (e.g., characterization) will not be discussed here.

With respect to prediction, experimental validation may be seen as a search for statistical relationships between metrics of the *object of study* and a descriptive models of the *quality focus* (e.g., # error for Error-proneness).

Numerous analysis techniques, both univariate and multivariate [S92, BBH93, DG84], exist in the statistical and machine learning literature. If such assumptions and properties are not validated, we need to repeat from Step 2, re-consider the assumptions and properties, then re-define new metrics. This metric definition/validation cycle is iterated until the metric validation yields satisfactory results. Since extensive material is available on the subject, we will not describe this step any further.

9. Conclusions and Future Work

Product metrics need to be defined in a rigourous and disciplined manner based on a precisely stated experimental goal, assumptions, properties, and a thorough experimental validation. In order to do so, we propose a definition approach that is intended to help analysts develop product metrics. This approach integrates many contributions from the literature and is intended to be the starting point for a practical product metric definition approach to be discussed by the software engineering community, on both the academic and industrial sides. This approach is the result of our past experience [BMB93, BBH93, BMB94(a)] and is validated through realistic examples.

Our future work encompasses a more detailed study and validation of each of the steps involved in the metric definition approach. In this framework, we proposed definitions for the measurement concepts usually encountered in software engineering, such as complexity, size, coupling, cohesion, etc [BMB94(b)]. Such a work aims at building a formal, unambiguous, and comprehensive theory. Also, we need to better understand how experimental results can be used to guide the refinement of metric. The refinement process of metrics needs to be better understood and defined so that metrics can evolve with the increase in understanding and refinement of the studied development processes. Last, we need to better identify what can be reused across environments and projects, e.g., metrics, assumptions, measurement concepts, product abstractions.

Acknowledgments

We would like to thank William Agresti, Dieter Rombach, Yong-Mi Kim, Bryan Hsueh, Wolfgang Heuser, Oliver Laitenberger, and Manoel Mendonça for their help in reviewing the early drafts of this paper.

References

[B92] V. Basili, "Software Modeling and Measurement: The Goal/Question/Metric Paradigm" University of Maryland, Department of Computer Science, Tech. Rep. CS-TR-2956, 1992.

[BBC88] J. Bieman et al, "A Standard Representation of Imperative Language Programs for Data Collection and Software Measures Specification", *J. Syst. Software*, vol. 8, pp. 13-37, 1988.

[BBH93] L. Briand, V. Basili and C. Hetmanski, "Developing Interpretable Models with Optimized Set Reduction for Identifying High Risk Software Components," *IEEE Trans. Software Eng.*, 19 (11), November, 1993.

[BBK94] L. Briand, V. Basili, Y. M. Kim and D. Squier, "A Change Analysis Process to Characterize Software Maintenance Projects," IEEE Conference on Software Maintenance, September 1994, Victoria, British Columbia, Canada.

[BMB93] L. Briand, S. Morasca, V. Basili, "Assessing Software Maintainability at the End of High-Level Design", IEEE Conference on Software Maintenance, September 1993, Montreal, Quebec, Canada.

[BMB94(a)] L. Briand, S. Morasca, V. Basili, "Defining and Validating High-Level Design Metrics", CS-TR 3301, UMIACS-TR 94-75, University of Maryland, College Park

[BMB94(b)] L. Briand, S. Morasca, V. Basili, "Property-based Software Engineering Measurement," CS-TR 3368, UMIACS-TR 94-119, University of Maryland, College Park

[BR88] V. Basili and D. Rombach, "The Tame Project: Towards Improvement-Oriented Software Environments," *IEEE Trans. Software Eng.*, vol. 14, no. 6, pp. 758-773, June 1988.

[DG84] W. Dillon and M. Goldstein, *Multivariate Analysis: Methods and Applications*, Wiley and Sons, 1984.

[F91] N. Fenton, "Software Metrics, A Rigorous Approach," Chapman&Hall, 1991.

[F94] N. Fenton, "Software Measurement: A Necessary Scientific Basis", *IEEE Trans. Software Eng.*, vol. 20, no. 3, pp. 199-206, March 1994.

[FM90] N. Fenton and A. Melton, "Deriving Structurally Based Software Measures", *J. Syst. Software*, vol. 12, pp. 177-187, 1990.

[IS88] D. Ince, M. Shepperd, "System Design Metrics: a Review and Perspective," Proc. Software Engineering 88, pages 23-27, 1988

[K88] B. Kitchenham, "An Evaluation of Software Structure Metrics," Proc. COMPSAC 88, 1988

[LJS91] K. B. Lakshmanan, S. Jayaprakash, and P. K. Sinha, "Properties of Control-Flow Complexity Measures," *IEEE Trans. Software Eng.*, vol. 17, no. 12, pp. 1289-1295, Dec. 1991.

[M90] L. Moser, "Data Dependency Graphs for Ada Programs", *IEEE Trans. Software Eng.*, vol. 16, no. 5, pp. 498-509, May 1990.

[MGB90] A. C. Melton, D.A. Gustafson, J. M. Bieman, and A. A. Baker, "Mathematical Perspective of Software Measures Research," *IEE Software Eng. J.*, vol. 5, no. 5, pp. 246-254, 1990.

[O80] E. I. Oviedo, "Control Flow, Data Flow and Program Complexity," *Proc. COMPSAC*, Nov. 1980, pp. 146-152.

[RW82] S. Rapps and E. Weyuker, "Data flow analysis test techniques for program test data selection", in *Proc. 6th Int. Conf. on Software Engineering*, Sept. 1982, pp. 272-278

[S92] N. F. Schneidewind, "Methodology for Validating Software Metrics," *IEEE Trans. Software Eng.*, vol. 18, no. 5, pp. 410-422, May 1992.

[TZ92] J. Tian and M. V. Zelkowitz, "A Formal Program Complexity Model and Its Application," *J. Syst. Software*, vol. 17, pp. 253-266, 1992.

[W88] E. J. Weyuker, "Evaluating Software Complexity Measures," *IEEE Trans. Software Eng.*, vol. 14, no. 9, pp. 1357-1365, Sept. 1988.

[Z90] H. Zuse, *Software Complexity: Measures and Methods*. Amsterdam: de Gruyter, 1990.

Property-based Software Engineering Measurement

Lionel Briand
CRIM
1801 McGill College Avenue
Montréal (Quebec), H3A 2N4
Canada
Lionel.Briand@crim.ca

Sandro Morasca
Dip. di Elettronica e Informazione
Politecnico di Milano
Piazza Leonardo da Vinci 32
I-20133 Milano, Italy
morasca@elet.polimi.it

Victor R. Basili
Computer Science Department
University of Maryland
College Park, MD 20742
basili@cs.umd.edu

Abstract

Little theory exists in the field of software system measurement. Concepts such as complexity, coupling, cohesion or even size are very often subject to interpretation and appear to have inconsistent definitions in the literature. As a consequence, there is little guidance provided to the analyst attempting to define proper measures for specific problems. Many controversies in the literature are simply misunderstandings and stem from the fact that some people talk about different measurement concepts under the same label (complexity is the most common case).

There is a need to define unambiguously the most important measurement concepts used in the measurement of software products. One way of doing so is to define precisely what mathematical properties characterize these concepts, regardless of the specific software artifacts to which these concepts are applied. Such a mathematical framework could generate a consensus in the software engineering community and provide a means for better communication among researchers, better guidelines for analysts, and better evaluation methods for commercial static analyzers for practitioners.

In this paper, we propose a mathematical framework which is generic, because it is not specific to any particular software artifact, and rigorous, because it is based on precise mathematical concepts. This framework defines several important measurement concepts (size, length, complexity, cohesion, coupling). It does not intend to be complete or fully objective; other frameworks could have been proposed and different choices could have been made. However, we believe that the formalisms and properties we introduce are convenient and intuitive. In addition, we have reviewed the literature on this subject and compared it with our work. This framework contributes constructively to a firmer theoretical ground of software measurement.

1. Introduction

Many concepts have been introduced through the years to define the characteristics of the artifacts produced during the software process. For instance, one speaks of size and complexity of software specification, design, and code, or cohesion and coupling of a software design or code. Several techniques have been introduced, with the goal of producing software which is better with respect to these concepts. As an example, Parnas [P72] design principles attempt to decrease coupling between modules, and increase cohesion within modules. These concepts are used as a guide to choose among alternative techniques or artifacts. For instance, a technique may be preferred over another because it yields artifacts that are less complex; an artifact may be preferred over another because it is less complex. In turn, lower complexity is believed to provide advantages such as lower maintenance time and cost. This shows the importance of a clear and unambiguous understanding of what these concepts actually mean, to make choices on more objective bases. The

This work was supported in part by NASA grant NSG-5123, UMIACS, NSF grant 01-5-24845, MURST, and CNR. This Tecnical Report is also available as Internal Report 94.078, Politecnico di Milano, Dipartimento di Elettronica e Informazione.

definition of relevant concepts (i.e., classes of software characterization measures) is the first step towards quantitative assessment of software artifacts and techniques, which is needed to assess risk and find optimal trade-offs between software quality, schedule and cost of development.

To capture these concepts in a quantitative fashion, hundreds of software measures have been defined in the literature. However, the vast majority of these measures did not survive the proposal phase, and did not manage to get accepted in the academic or industrial worlds. One reason for this is the fact that they have not been built using a clearly defined process for defining software measures. As we propose in [BMB94(b)], such a process should be driven by clearly identified measurement goals and knowledge of the software process. One of its crucial activities is the precise definition of relevant concepts, necessary to lay down a rigorous framework for software engineering measures and to define meaningful and well-founded software measures. The theoretical soundness of a measure, i.e., the fact that it really measures the software characteristic it is supposed to measure, is an obvious prerequisite for its acceptability and use. The exploratory process of looking for correlations is not an acceptable scientific validation process in itself if it is not accompanied by a solid theory to support it. Unfortunately, new software measures are very often defined to capture elusive concepts such as complexity, cohesion, coupling, connectivity, etc. (Only size can be thought to be reasonably well understood.) Thus, it is impossible to assess the theoretical soundness of newly proposed measures, and the acceptance of a new measure is mostly a matter of belief.

To this end, several proposals have appeared in the literature [LJS91, TZ92, W88] in recent years to provide desirable properties for software measures. These works (especially [W88]) have been used to "validate" existing and newly proposed software measures. Surprisingly, whenever a new measure which was proposed as a software complexity measure did not satisfy the set of properties against which it was checked, several authors failed to conclude that their measure was not a software complexity measure, e.g., [CK94, H92]. Instead, they concluded that their measure was a complexity measure that does not satisfy that set of properties for complexity measures. What they actually did was provide an absolute definition of a software complexity measure and check whether the properties were consistent with respect to the measure, i.e., check the properties against their own measure.

This situation would be unacceptable in other engineering or mathematical fields. For instance, suppose that one defines a new measure, claiming it is a distance measure. Suppose also that that measure fails to satisfy the triangle inequality, which is the characterizing property of distance measures. The natural conclusion would be to realize that that is not a distance measure, rather than to say that it is a distance measure that does not satisfy the conditions for a distance measure. However, it is true that none of the sets of properties proposed so far has reached so wide an acceptance to be considered "the" right set of necessary properties for complexity. It is our position that this odd situation is due to the fact that there are several different concepts that are still covered by the same word: complexity.

Within the set of commonly mentioned software characteristics, size and complexity are the ones that have received the widest attention. However, the majority of authors have been inclined to believe that a measure captures either size or complexity, as if, besides size, all other concepts related to software characteristics could be grouped under the unique name of complexity. Sometimes, even size has been considered as a particular kind of complexity measure.

Actually, these concepts capture different software characteristics, and, until they are clearly separated and their similarities and differences clearly studied, it will be impossible to reach any kind of consensus on the properties that characterize each concept relevant to the definition of software measures. The goal of this paper is to lay down the basis for a discussion on this subject, by providing properties for a—partial—set of measurement concepts that are relevant for the definition of software measures. Many of the measure properties proposed in the literature are generic in the sense that they do not characterize specific measurement concepts but are relevant to all syntactically-based measures (see [S92, TZ92, W88]). In this paper, we want to focus on properties that differentiate measurement concepts such as size, complexity, coupling, etc. Thus, we want to identify and clarify the essential properties behind these concepts that are commonplace in software engineering and form important classes of measures. Thus, researchers will be able to validate their new measures by checking properties specifically relevant to the class (or concept) they belong to (e.g., size should be additive). *By no means should these properties be regarded as* *the unique set of properties that can be possibly defined for a given concept.* Rather, we want to provide a theoretically sound and convenient solution for differentiating a set of well known concepts and check their analogies and conflicts. Possible applications of such a framework are to guide researchers in their search for new measures and help practitioners evaluate the adequacy of measures provided by commercial tools.

We also believe that the investigation of measures should also address artifacts produced in the software process other than code. It is commonly believed that the early software process phases are the most important ones, since the rest of the development depends on the artifacts they produce. Oftentimes, the concepts (e.g., size, complexity, cohesion, coupling) which are believed relevant with respect to code are also relevant for other artifacts. To this end, the properties we propose will be general enough to be applicable to a wide set of artifacts.

The paper is organized as follows. In Section 2, we introduce the basic definitions of our framework. Section 3 provides a set of properties that characterize and formalize intuitively relevant measurement concepts: size, length, complexity, cohesion, coupling. We also discuss the relationships and differences between the different concepts. Some of the best-known measures are used as examples to illustrate our points. Section 4 contains comparisons and discussions regarding the set of properties for complexity measures defined in the paper and in the literature. The conclusions and directions for future work come in Section 5.

2. Basic Definitions

Before introducing the necessary properties for the set of concepts we intend to study, we provide basic definitions related to the objects of study (to which these concepts can be applied), e.g., size and complexity of *what*?

Systems and modules

Two of the concepts we will investigate, namely, size (Section 3.1) and complexity (Section 3.3) are related to systems, in general, i.e., one can speak about the size of a system and the complexity of a system. We also introduce a new concept, length (Section 3.2), which is related to systems. In our general framework—recall that we want these properties to be as independent as possible of any product abstraction—, a system is characterized by its elements and the relationships between them. Thus, we do not reduce the number of possible system representations, as elements and relationships can be defined according to needs.

Definition 1: Representation of Systems and Modules
A *system* S will be represented as a pair <E,R>, where E represents the set of elements of S, and R is a binary relation on E ($R \subseteq E \times E$) representing the relationships between S's elements.

Given a system S = <E,R>, a system m = $<E_m,R_m>$ is a *module* of S if and only if $E_m \subseteq E$, $R_m \subseteq E \times E$, and $R_m \subseteq R$. As an example, E can be defined as the set of code statements and R as the set of control flows from one statement to another. A module m may be a code segment or a subprogram.

The elements of a module are connected to the elements of the rest of the system by incoming and outgoing relationships. The set InputR(m) of relationships from elements outside module m = $<E_m,R_m>$ to those of module m is defined as

$$\text{InputR}(m) = \{<e_1,e_2> \in R \mid e_2 \in E_m \textbf{ and } e_1 \in E - E_m\}$$

The set OutputR(m) of relationships from the elements of a module m = $<E_m,R_m>$ to those of the rest of the system is defined as

$$\text{OutputR}(m) = \{<e_1,e_2> \in R \mid e_1 \in E_m \textbf{ and } e_2 \in E - E_m\}$$

◊

We now introduce inclusion, union, intersection operations for modules and the definitions of empty and disjoint modules, which will be used often in the remainder of the paper. For notational convenience, they will be denoted by extending the usual set-theoretic notation. We will illustrate these operations by means of the system $S = \langle E,R \rangle$ represented in Figure 1, where $E = \{a,b,c,d,e,f,g,h,i,j,k,l,m\}$ and $R = \{\langle b,a \rangle,\langle b,f \rangle,\langle c,b \rangle,\langle c,d \rangle,\langle c,g \rangle,\langle d,f \rangle,\langle e,g \rangle,\langle f,i \rangle, \langle f,k \rangle,\langle g,m \rangle,\langle h,a \rangle,\langle h,i \rangle,\langle i,j \rangle,\langle k,j \rangle,\langle k,l \rangle\}$. We will consider the following modules

- $m_1 = \langle E_{m1},R_{m1} \rangle = \langle \{a,b,f,i,j,k\},\{\langle b,a \rangle,\langle b,f \rangle,\langle f,i \rangle,\langle f,k \rangle,\langle i,j \rangle,\langle k,j \rangle\}$ (area filled with ▧)
- $m_2 = \langle E_{m2},R_{m2} \rangle = \langle \{f,j,k\},\{\langle f,k \rangle,\langle k,j \rangle\}$ (area filled with ▩)
- $m_3 = \langle E_{m3},R_{m3} \rangle = \langle \{c,d,e,f,g,j,k,m\},\{\langle c,d \rangle,\langle c,g \rangle,\langle d,f \rangle,\langle e,g \rangle,\langle f,k \rangle,\langle g,m \rangle, \langle k,j \rangle\} \rangle$ (area filled with ▨)
- $m_4 = \langle E_{m4},R_{m4} \rangle = \langle \{d,e,g\},\{\langle e,g \rangle\} \rangle$ (area filled with ▦)

Inclusion. Module $m_1 = \langle E_{m1},R_{m1} \rangle$ is said to be included in module $m_2 = \langle E_{m2},R_{m2} \rangle$ (notation: $m_1 \subseteq m_2$) if $E_{m1} \subseteq E_{m2}$ and $R_{m1} \subseteq R_{m2}$. In Figure 1, $m_4 \subseteq m_3$.

Union. The union of modules $m_1 = \langle E_{m1},R_{m1} \rangle$ and $m_2 = \langle E_{m2},R_{m2} \rangle$ (notation: $m_1 \cup m_2$) is the module $\langle E_{m1} \cup E_{m2},R_{m1} \cup R_{m2} \rangle$. In Figure 1, the union of modules m_1 and m_3 is module $m_{13} = \langle \{a,b,c,d,e,f,g,i,j,k,m\}, \{\langle b,a \rangle,\langle b,f \rangle,\langle c,d \rangle,\langle c,g \rangle,\langle d,f \rangle,\langle e,g \rangle,\langle f,i \rangle, \langle f,k \rangle,\langle g,m \rangle,\langle i,j \rangle,\langle k,j \rangle\}$ (area filled with ▧ or ▩ or ▨).

Intersection. The intersection of modules $m_1 = \langle E_{m1},R_{m1} \rangle$ and $m_2 = \langle E_{m2}, R_{m2} \rangle$ (notation: $m_1 \cap m_2$) is the module $\langle E_{m1} \cap E_{m2},R_{m1} \cap R_{m2} \rangle$. In Figure 1, $m_2 = m_1 \cap m_3$.

Empty module. Module $\langle \varnothing,\varnothing \rangle$ (denoted by $\varnothing$) is the empty module.

Disjoint modules. Modules m_1 and m_2 are said to be disjoint if $m_1 \cap m_2 = \varnothing$. In Figure 1, $m_1 \cap m_4 = \varnothing$.

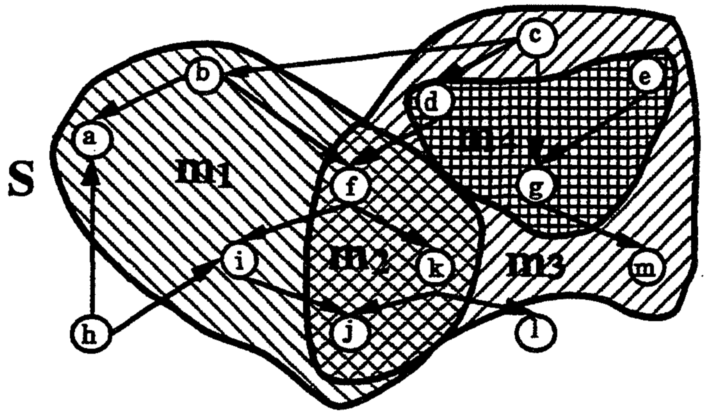


Figure 1. Operations on modules.

Since in this framework modules are just subsystems, all systems can theoretically be decomposed into modules. The definition of a module for a particular measure in a specific context is just a matter of convenience and programming environment (e.g., language) constraints.

Modular systems

The other two concepts we will investigate, cohesion (Section 3.4) and coupling (Section 3.5), are meaningful only with reference to systems that are provided with a modular decomposition, i.e., one can speak about cohesion and coupling of a whole system only if it is structured into modules. One can also speak about cohesion and coupling of a single module within a whole system.

Definition 2: Representation of Modular Systems
The 3-tuple MS = <E,R,M> represents a modular system if S = <E,R> is a system according to Definition 1, and M is a collection of modules of S such that

$\forall e \in E\ (\exists m \in M\ (m = <E_m,R_m>$ **and** $e \in E_m))$ **and**

$\forall m_1,m_2 \in M\ (m_1 = <E_{m1},R_{m1}>$ **and** $m_2 = <E_{m2},R_{m2}>$ **and** $E_{m1} \cap E_{m2} = \emptyset)$

i.e, the set of elements E of MS is partitioned into the sets of elements of the modules.

We denote the union of all the R_m's as IR. It is the set of *intra-module relationships*. Since the modules are disjoint, the union of all OutputR(m)'s is equal to the union of all InputR(m)'s, which is equal to R-IR. It is the set of *inter-module relationships*.

◊

As an example, E can be the set of all declarations of a set of Ada modules, R the set of dependencies between them, and M the set of Ada modules.

Figure 2 shows a modular system MS = <E,R,M>, obtained by partitioning the set of elements of the system in Figure 1 in a different way. In this modular system, E and R are the same as in system S in Figure 1, and M = $\{m_1,m_2,m_3\}$. Besides, IR = {<b,a>,<c,d>,<c,g>, <e,g>,<f,i>,<f,k>,<g,m>,<h,a>,<i,j>,<k,j>,<k,l>}.

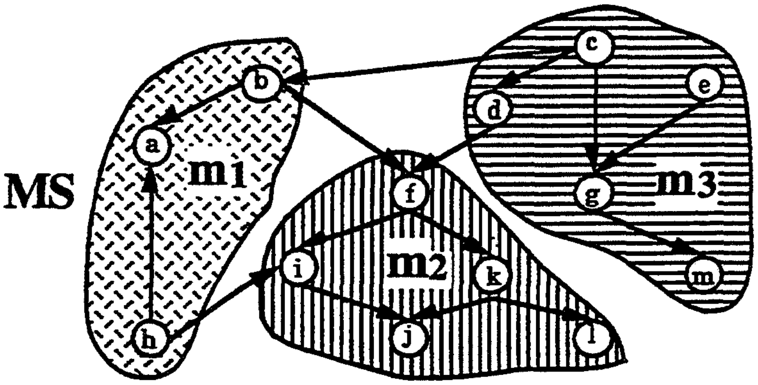


Figure 2. A modular system.

It should be noted that some measures do not take into account the modular structure of a system. As already mentioned, our concepts of size and complexity (defined in Sections 3.1 and 3.3) are such examples, i.e., in a modular system MS = <E,R,M>, one computes size and complexity of the system S = <E,R>, and M is not considered.

We have defined concept properties using a graph-theoretic approach to allow us to be general and precise. It is general because our properties are defined so that no restriction applies to the definition of vertices and arcs. Many well known product abstractions fit this framework, e.g., data dependency graphs, definition-use graphs, control flow graphs, USES graphs, Is_Component_of graphs, etc. It is precise because, based on a well defined formalism, all the concepts used can be mathematically defined, e.g., system, module, modular system, and so can the properties presented in the next section.

3. Concepts of Measurement and Properties

It should be noted that the concepts defined below are to some extent subjective. However, we wish to assign them intuitive and convenient properties. We consider these properties necessary but not sufficient because they do not guarantee that the measures for which they hold are useful or even make sense. On the other hand, these properties will constrain the search for measures and therefore make the measure definition process more rigorous and less exploratory [BMB94(b)]. Several relevant concepts are studied: size, length, complexity, cohesion, and coupling. They do not represent an exhaustive list but a starting point for discussion that should eventually lead to a standard definition set in the software engineering community.

3.1. Size

Motivation

Intuitively, size is recognized as being an important measurement concept. According to our framework, size cannot be negative (property Size.1), and we expect it to be null when a system does not contain any elements (property Size.2). When modules do not have elements in common, we expect size to be additive (property Size.3).

Definition 3: Size
The size of a system S is a function Size(S) that is characterized by the following properties Size.1 - Size.3.

◊

Property *Size.1*: Non-negativity
The size of a system S = <E,R> is non-negative

$$\text{Size}(S) \geq 0 \qquad \text{(Size.I)}$$

◊

Property *Size.2*: Null Value
The size of a system S = <E,R> is null if E is empty

$$E = \varnothing \Rightarrow \text{Size}(S) = 0 \qquad \text{(Size.II)}$$

◊

Property *Size.3*: Module Additivity
The size of a system S = <E,R> is equal to the sum of the sizes of two of its modules $m_1 = <E_{m1},R_{m1}>$ and $m_2 = <E_{m2},R_{m2}>$ such that any element of S is an element of either m_1 or m_2

$$(m_1 \subseteq S \textbf{ and } m_2 \subseteq S \textbf{ and } E = E_{m1} \cup E_{m2} \textbf{ and } E_{m1} \cap E_{m2} = \varnothing) \Rightarrow \text{Size}(S) = \text{Size}(m_1) + \text{Size}(m_2) \qquad \text{(Size.III)}$$

◊

For instance, the size of the system in Figure 2 is the sum of the sizes of its three modules m_1,m_2,m_3.

The last property Size.3 provides the means to compute the size of a system S = <E,R> from the knowledge of the size of its—disjoint—modules $m_e = <\{e\},R_e>$ whose set of elements is composed of a different element e of E[1].

[1] For each m_e, it is either $R_e = \varnothing$ or $R_e = \{<e,e>\}$.

$$\mathrm{Size}(S) = \sum_{e \in E} \mathrm{Size}(m_e) \qquad \text{(Size.IV)}$$

Therefore, adding elements to a system cannot decrease its size (*size monotonicity property*)

$$(S' = \langle E',R' \rangle \textbf{ and } S'' = \langle E'',R'' \rangle \textbf{ and } E' \subseteq E'') \Rightarrow \mathrm{Size}(S') \le \mathrm{Size}(S'') \qquad \text{(Size.V)}$$

From the above properties Size.1 - Size.3, it follows that the size of a system $S = \langle E,R \rangle$ is not greater than the sum of the sizes of any pair of its modules $m_1 = \langle E_{m1},R_{m1} \rangle$ and $m_2 = \langle E_{m2},R_{m2} \rangle$, such that any element of S is an element of m_1, or m_2, or both, i.e.,

$$(m_1 \subseteq S \textbf{ and } m_2 \subseteq S \textbf{ and } E = E_{m1} \cup E_{m2}) \Rightarrow \mathrm{Size}(S) \le \mathrm{Size}(m_1) + \mathrm{Size}(m_2) \qquad \text{(Size.VI)}$$

The size of a system built by merging such modules cannot be greater than the sum of the sizes of the modules, due to the presence of common elements (e.g., lines of code, operators, class methods).

Properties Size.1 - Size.3 hold when applying the admissible transformation of the ratio scale [F91]. Therefore, there is no contradiction between our concept of size and the definition of size measures on a ratio scale.

Examples and counterexamples of size measures

Several measures introduced in the literature can be classified as size measures, according to our properties Size.1 - Size.3. With reference to code measures, we have: LOC, #Statements, #Modules, #Procedures, Halstead's Length [H77], #Occurrences of Operators, #Occurrences of Operands, #Unique Operators, #Unique Operands. In each of the above cases, the representation of a program as a system is quite straightforward. Each counted entity is an element, and the relationship between elements is just the sequential relationship.

Some other measures that have been introduced as size measures do not satisfy the above properties. Instances are the Estimator of length, and Volume [H77], which are not additive when software modules are disjoint (property Size.3). Indeed, for both measures, the value obtained when two disjoint software modules are concatenated may be less than the sum of the values obtained for each module, since they may contain common operators or operands. Note that, in this context, the graph is just the sequence of operand and operator occurrences. Disjoint code segments are disjoint subgraphs.

On the other hand, other measures, that are meant to capture other concepts, are indeed size measures. For instance, in the object-oriented suite of measures defined in [CK94], Weighted Methods per Class (WMC) is defined as the sum of the complexities of methods in a class. Implicitly, the program is seen as a directed acyclic graph (a hierarchy) whose terminal nodes are methods, and whose nonterminal nodes are classes. When two classes without methods in common are merged, the resulting class's WMC is equal to the sum of the two WMC's of the original classes (property Size.3 is satisfied). When two classes with methods in common are merged, then the WMC of the resulting class may be lower than the sum of the WMC's of the two original classes (formula Size.VI). Therefore, since all size properties hold (it is straightforward to show that properties Size.1 and Size.2 are true for WMC), this is a class size measure. However, WMC does not satisfy our properties for complexity measures (see Section 3.3). Likewise, NOC (Number Of Children of a class) and Response For a Class (RFC) [CK94] are other size measures, according to our properties.

3.2. Length

Motivation

Properties Size.1 - Size.3 characterize the concept of size as is commonly intended in software engineering. Actually, the concept of size may have different interpretations in everyday life, depending on the measurement goal. For instance, suppose we want to park a car in a parallel parking space. Then, the "size" we are interested in is the maximum distance between two points of the car linked by a segment parallel to the car's motion direction. The above properties Size.1 - Size.3 do not aim at defining such a measure of size. With respect to physical objects, volume and weight satisfy the above properties. In the particular case that the objects are unidimensional (or that we are interested in carrying out measurements with respect to only one dimension), then these concepts coincide.

In order to differentiate this measurement concept from size, we call it *length*. Length is non-negative (property Length.1), and equal to 0 when there are no elements in the system (property Length.2). In extreme situations where systems are composed of unrelated elements this property allows length to be non-null. If a new relationship is introduced between two elements belonging to the same connected component[2] of the graph representing a system, the length of the new system is not greater than the length of the original system (property Length.3). The idea is that, in this case, a new relationship may make the elements it connects "closer" than they were. This new relationship may reduce the maximum distance between elements in the connected component of the graph, but it may never increase it. On the other hand, if a new relationship is introduced between two elements belonging to two different connected components, the length of the new system is not smaller than the length of the original system. This stems from the fact that the new relationship creates a new connected component, where the maximum distance between two elements cannot be less than the maximum distance between any two elements of either original connected component (property Length.4). Length is not additive for disjoint modules. The length of a system containing several disjoint modules is the maximum length among them (property Length.5).

Definition 4: Length
The length of a system S is a function Length(S) characterized by the following properties Length.1 - Length.4.

◊

Property *Length.1*: Non-negativity
The length of a system S = <E,R> is non-negative

Length(S) $\geq$ 0 (Length.I)

◊

Property *Length.2*: Null Value
The length of a system S = <E,R> is null if E is empty

$(E = \varnothing) \Rightarrow (\text{Length}(S) = 0)$ (Length.II)

◊

Property *Length.3*: Non-increasing Monotonicity for Connected Components
Let S be a system and m be a module of S such that m is represented by a connected component of the graph representing S. Adding relationships between elements of m does not increase the length of S

[2] Here, two elements of a system S are said to belong to the same connected component if there is a path from one to the other in the non-directed graph obtained from the graph representing S by removing directions in the arcs.

$$\begin{array}{l}(S = \langle E,R \rangle \textbf{ and } m = \langle E_m,R_m \rangle \textbf{ and } m \subseteq S \\ \quad \textbf{and } m \text{ "is a connected component of S" } \textbf{and} \\ \qquad\qquad S' = \langle E,R' \rangle \textbf{ and } R' = R \cup \{\langle e_1,e_2 \rangle\} \textbf{ and } \langle e_1,e_2 \rangle \notin R \\ \qquad\qquad\quad \textbf{and } e_1 \in E_{m1} \textbf{ and } e_2 \in E_{m1}) \Rightarrow \text{Length}(S) \geq \text{Length}(S') \quad \text{(Length.III)}\end{array}$$

◊

Property *Length.4*: Non-decreasing Monotonicity for Non-connected Components
Let S be a system and m_1 and m_2 be two modules of S such that m_1 and m_2 are represented by two separate connected components of the graph representing S. Adding relationships from elements of m_1 to elements of m_2 does not decrease the length of S

$$\begin{array}{l}(S = \langle E,R \rangle \textbf{ and } m_1 = \langle E_{m1},R_{m1} \rangle \textbf{ and } m_2 = \langle E_{m2},R_{m2} \rangle \\ \quad \textbf{and } m_1 \subseteq S \textbf{ and } m_2 \subseteq S \text{ "are separate connected components of S" } \textbf{and} \\ \qquad\qquad S' = \langle E,R' \rangle \textbf{ and } R' = R \cup \{\langle e_1,e_2 \rangle\} \textbf{ and } \langle e_1,e_2 \rangle \notin R \\ \qquad\qquad\quad \textbf{and } e_1 \in E_{m1} \textbf{ and } e_2 \in E_{m2}) \Rightarrow \text{Length}(S') \geq \text{Length}(S) \quad \text{(Length.IV)}\end{array}$$

◊

Property *Length.5*: Disjoint Modules
The length of a system $S = \langle E,R \rangle$ made of two disjoint modules m_1, m_2 is equal to the maximum of the lengths of m_1 and m_2

$$\begin{array}{l}(S = m_1 \cup m_2 \textbf{ and } m_1 \cap m_2 = \emptyset \textbf{ and } E = E_{m1} \cup E_{m2}) \Rightarrow \\ \qquad\qquad \text{Length}(S) = \max\{\text{Length}(m_1),\text{Length}(m_2)\} \qquad \text{(Length.V)}\end{array}$$

◊

Let us illustrate the last three properties with systems S, S', S", represented in Figure 3. We will assume that $m_1 = m'_1 = m''_1$, $m_2 = m'_2 = m''_2$, and $m_3 = m'_3 = m''_3$. The length of system S, composed of the three connected components m_1, m_2, and m_3, is the maximum value among the lengths of m_1, m_2, and m_3 (property Length.V). System S' differs from system S only because of the added relationship <c,m> (represented by the thick dashed arrow), which connects two elements already belonging to a connected component of S, m_3. The length of system S' is not greater than the length of S (property Length.III). System S" differs from system S only because of the added relationship <b,f> (represented by the thick solid arrow), which connects two elements belonging to two different connected components of S, m_1 and m_2. The length of system S" is not less than the length of S (property Length.IV).

Properties Length.1 - Length.5 hold when applying the admissible transformation of the ratio scale. Therefore, there is no contradiction between our concept of length and the definition of length measures on a ratio scale.

Examples of length measures

Several measures can be defined at the system or module level based on the length concept. A typical example is the depth of a hierarchy. Therefore, the nesting depth in a program [F91] and DIT (Depth of Inheritance Tree—which is actually a hierarchy, in the general case) defined in [CK94] are length measures.

3.3. *Complexity*

Motivation

Intuitively, complexity is a measurement concept that is considered extremely relevant to system properties. It has been studied by several researchers (see Section 4 for a comparison between our framework and the literature). In our framework, we expect complexity to be non-negative (property Complexity.1) and to be null (property Complexity.2) when there are no relationships between the elements of a system. However, it could be argued that the complexity of a system whose elements are not connected to each other does not need to be necessarily null, because each element of E may have some complexity of its own. In our view, complexity is a system property that depends on the relationships between elements, and is not an isolated element's property. The complexity that an element taken in isolation may—intuitively—bring can only originate from the relationships between its "subelements." For instance, in a modular system, each module can be viewed as a "high-level element" encapsulating "subelements." However, if we want to consider the system as composed of such "high-level elements" (E), we should not "unpack" them, but only consider them and their relationships, without considering their "subelements" (E'). Otherwise, if we want to consider the contribution of the relationships between "subelements" (R'), we actually have to represent the system as S = <E', R∪R'>.

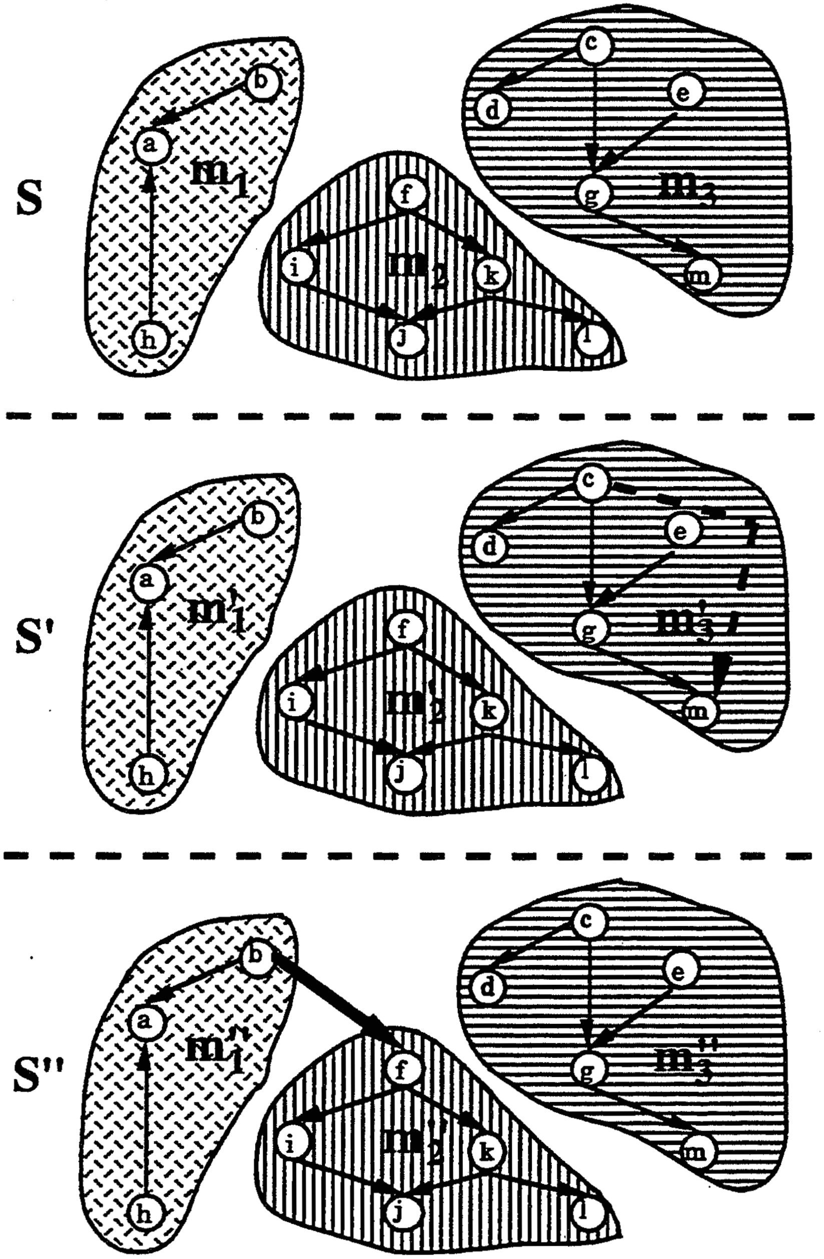


Figure 3. Properties of length.

Complexity should not be sensitive to representation conventions with respect to the direction of arcs representing system relationships (property Complexity.3). A relation can be represented in either an "active" (R) or "passive" (R^{-1}) form. The system and the relationships between its elements are not affected by these two equivalent representation conventions, so a complexity measure should be insensitive to this.

Also, the complexity of a system S should be at least as much as the sum of the complexities of any collections of its modules, such that no two modules share relationships, but may only share elements (property Complexity.4). *We believe that this property is the one that most strongly differentiates complexity from the other system concepts.* Intuitively, this property may be explained by two phenomena. First, the transitive closure of R is a larger graph than the graph obtained as the union of the transitive closures of R' and R" (where R' and R" are contained in R). As a consequence, if any kind of *indirect* (i.e., *transitive*) relationships between elements is considered in the computation of complexity, then the complexity of S may be larger than the sum of its modules' complexities, when the modules do not share any relationship. Otherwise, they are equal. Second, merging modules may implicitely generate between the elements of each modules. (e.g., definition-use relationships may be created when blocks are merged into a common system). As a consequence of the above properties, system complexity should not decrease when the set of system relationships is increased (property Complexity.4).

However, it has been argued that it is not always the case that the more relationships between the elements of a system, the more complex the system. For instance, it has been argued that adding a relationship between two elements may make the understanding of the system easier, since it clarifies the relationship between the two. This is certainly true, but we want to point out that this assertion is related to understandability, rather than complexity, and that complexity is only *one* of the factors that contribute to understandability. There are other factors that have a strong influence on understandability, such as the amount of available context information and knowledge about a system. In the literature [MGB90], it has been argued that the inner loop of the ShellSort algorithm, taken in isolation, is less understandable than the whole algorithm, since the role of the inner loop in the algorithm cannot be fully understood without the rest of the algorithm. This shows that understandability improves because a larger amount of context information is available, rather than because the complexity of the ShellSort algorithm is less than that of its inner loop. As another example, a relationship between two elements of a system may be added to *explicitly* state a relationship between them that was implicit or uncertain. This adds to our knowledge of the system, while, at the same time, increases complexity (according to our properties). In some cases (see above examples), the gain in context information/knowledge may overcome the increase in complexity and, as a result, may improve understandability. This stems from the fact that several phenomena concurrently affect understandability and does not mean in any way that an increase in complexity increases understandability.

Last, the complexity of a system made of disjoint modules is the sum of the complexities of the single modules (property Complexity.5). Consistent with property Complexity.4, this property is intuitively justified by the fact that the transitive closure of a graph composed of several disjoint subgraphs is equal to the union of the transitive closures of each subgraph taken in isolation. Furthermore, if two modules are put together in the same system, but they are not merged, i.e., they are still two disjoint module in this system, then no additional relationships are generated from the elements of one to the elements of the other.

The properties we define for complexity are, to a limited extent, a generalization of the properties several authors have already provided in the literature (see [LJS91, TZ92, W88]) for software code complexity, usually for control flow graphs. We generalize them because we may want to use them on artifacts other than software code and on abstractions other than control flow graphs.

Definition 5: Complexity

The complexity of a system S is a function Complexity(S) that is characterized by the following properties Complexity.1 - Complexity.5.

◊

Property *Complexity*.1: Non-negativity
The complexity of a system S = <E,R> is non-negative

Complexity(S) ≥ 0 (Complexity.I)

◊

Property *Complexity*.2: Null Value
The complexity of a system S = <E,R> is null if R is empty

$R = \varnothing \Rightarrow \text{Complexity}(S) = 0$ (Complexity.II)

◊

Property *Complexity*.3: Symmetry
The complexity of a system S = <E,R> does not depend on the convention chosen to represent the relationships between its elements

$(S=\langle E,R\rangle$ **and** $S^{-1}=\langle E,R^{-1}\rangle) \Rightarrow \text{Complexity}(S) = \text{Complexity}(S^{-1})$ (Complexity.III)

◊

Property *Complexity*.4: Module Monotonicity
The complexity of a system S = <E,R> is no less than the sum of the complexities of any two of its modules with no relationships in common

$(S = \langle E,R\rangle$ **and** $m_1 = \langle E_{m1},R_{m1}\rangle$ **and** $m_2 = \langle E_{m2},R_{m2}\rangle$
and $m_1 \cup m_2 \subseteq S$ **and** $R_{m1} \cap R_{m2} = \varnothing)$
$\Rightarrow \text{Complexity}(S) \geq \text{Complexity}(m_1)+\text{Complexity}(m_2)$ (Complexity.IV)

◊

For instance, the complexity of the system shown in Figure 4 is not smaller than the sum of the complexities of m_1 and m_2.

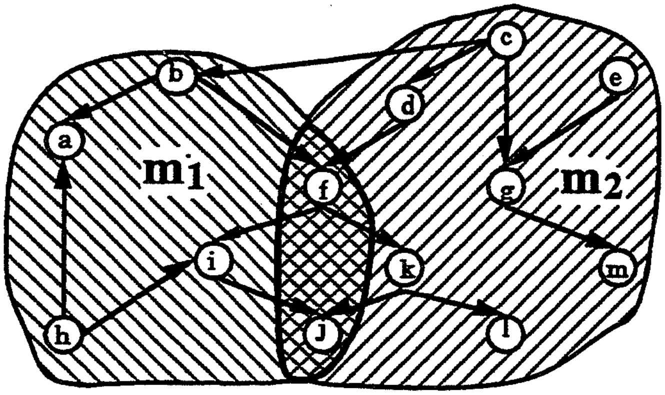


Figure 4. Module monotonicity of complexity.

Property *Complexity*.5: Disjoint Module Additivity
The complexity of a system S = <E,R> composed of two disjoint modules m_1, m_2 is equal to the sum of the complexities of the two modules

$(S = \langle E,R\rangle$ **and** $S = m_1 \cup m_2$ **and** $m_1 \cap m_2 = \varnothing)$
$\Rightarrow \text{Complexity}(S) = \text{Complexity}(m_1) + \text{Complexity}(m_2)$ (Complexity.V)

◊

For instance, the complexity of system S in Figure 2 is the sum of the complexities of its modules m_1, m_2, and m_3.

As a consequence of the above properties Complexity.1 - Complexity.5, it can be shown that adding relationships between elements of a system does not decrease its complexity

$$(S' = \langle E,R' \rangle \textbf{ and } S'' = \langle E,R'' \rangle \textbf{ and } R' \subseteq R'') \Rightarrow \text{Complexity}(S') \leq \text{Complexity}(S'') \qquad \text{(Complexity.VI)}$$

Properties Complexity.1 - Complexity.5 hold when applying the admissible transformation of the ratio scale. Therefore, there is no contradiction between our concept of complexity and the definition of complexity measures on a ratio scale.

Comprehensive comparisons and discussions of previous work in the area of complexity properties are provided in Section 4.

Examples and counterexamples of complexity measures

In [O80], Oviedo proposed a data flow complexity measure (DF). In this case, systems are programs, modules are program blocks, elements are variable definitions or uses, and relationships are defined between the definition of a given variable and its uses. The measure in [O80] is simply defined as the number of definition-use pairs in a block or a program. Property Complexity.4 holds. Given two modules (i.e., program blocks) which may only have common elements (i.e., no definition-use relationship is contained in both), the whole system (i.e., program) has a number of relationships (i.e., definition-use relationships) which is at least equal to the sum of the numbers of definition-use relationships of each module. Property Complexity.5 holds as well. The number of definition-use relationships of a system composed of two disjoint modules (i.e., blocks between which no definition-use relationship exists), is equal to the sum of the numbers of definition-use relationships of each module. As a conclusion, DF is a complexity measure according to our definition.

In [McC76], McCabe proposed a control flow complexity measure. Given a control flow graph $G = \langle E,R \rangle$ (which corresponds—unchanged—to a system for our framework), Cyclomatic Complexity is defined as

$$v(G) = |R| - |E| + 2p$$

where p is the number of connected components of G. Let us now check whether v(G) is a complexity measure according to our definition. It is straightforward to show that, except Complexity.4, the other properties hold. In order to check property Complexity.4, let $G = \langle E,R \rangle$ be a control flow graph and $G_1 = \langle E_1,R_1 \rangle$ and $G_2 = \langle E_2,R_2 \rangle$ two non-disjoint control flow subgraphs of G such that they have nodes in common but no relationships. We have to require that G_1 and G_2 be control flow subgraphs, because cyclomatic complexity is defined only for control flow graphs, i.e., graphs composed of connected components, each of which has a start node—a node with no incoming arcs—and an end node—a node with no outgoing arcs. Property Complexity.4 requires that the following inequality be true for all such G_1 and G_2

$$|R| - |E| + 2p \geq |R_1| - |E_1| + 2p_1 + |R_2| - |E_2| + 2p_2$$

i.e., $2(p_1 + p_2 - p) \leq |E_1| + |E_2| - |E|$, where p_1 and p_2 are the number of connected components in G_1 and G_2, respectively. This is not always true. For instance, consider Figure 5. G has 3 elements and 1 connected component; G_1 and G_2 have 2 nodes and 1 connected component apiece. Therefore, the above inequality is not true in this case, and the cyclomatic number is not a complexity measure according to our definition. However, it can be shown that v(G)-p satisfies all the above complexity properties. From a practical perspective, especially in large systems, this correction does not have a significant impact on the value of the measure.

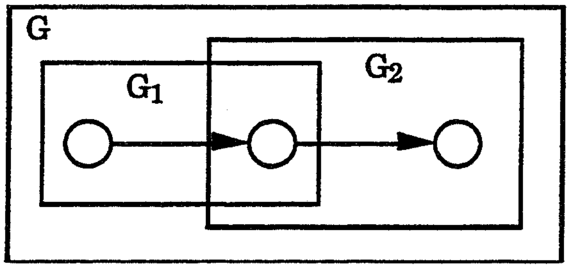


Figure 5. Control flow graph.

Henry and Kafura [HK81] proposed an information flow complexity measure. In this context, elements are subprogram variables or parameters, modules are subprograms, relationships are either fan-in's or fan-out's. For a subprogram SP, the complexity is expressed as *length.(fan-in.fan-out)*^{2}, where fan-in and fan-out are, respectively, the local (as defined in [HK81]) information flows from other subprograms to SP, and from SP to other subprograms. Such local information flows can be represented as relationships between parameters/variables of SP and parameters/variables of the other subprograms. Subprograms' parameters/variables are the system elements and the subprograms' fan-in and fan-out links are the relationships. Any size measure can be used for *length* (in [HK81] LOC was used). The justification for multiplying *length* and *(fan-in.fan-out)*^{2} was that "The complexity of a procedure depends on two factors: the complexity of the procedure code and the complexity of the procedure's connections to its environment." The complexity of the procedure code is taken into account by *length*; the complexity of the subprogram's connections to its environment is taken into account by *(fan-in.fan-out)*^{2}. The complexity of a system is defined as the sum of the complexities of the individual subprograms. For the measure defined above, properties Complexity.1 - Complexity.4 hold. However, property Complexity.5 does not hold since, given two disjoint modules S_1 and S_2 with a measured information flow of, respectively, *length*$_1$*.(fan-in*$_1$*.fan-out*$_1$*)*^{2} and *length*$_2$*.(fan-in*$_2$*.fan-out*$_2$*)*^{2}, the following statement is true:

$$\text{length.(fan-in.fan-out)}^2 \geq \text{length}_1.(\text{fan-in}_1.\text{fan-out}_1)^2 + \text{length}_2.(\text{fan-in}_2.\text{fan-out}_2)^2$$

where length = length$_1$ + length$_2$, fan-in = fan-in$_1$ + fan-in$_2$, and fan-out = fan-out$_1$ + fan-out$_2$.

However, equality does not hold because of the exponent 2, which is not fully justified, and multiplication of fan-in and fan-out. Therefore, Henry and Kafura [HK81] information flow measure is not a complexity measure according to our definition. However, fan-in and fan-out taken as separate measures, without exponent 2, are complexity measures according to our definition since all the required properties hold.

Similar measures have been used in [C90] and referred to as *structural complexity* (SC) and defined as:

$$SC = \frac{\sum_{i \in [1..n]} \text{fan-out}^2(\text{subroutine}_i)}{n}$$

Once again, property Complexity.5 does not hold because fan-out is squared in the formula.

A metric suite for object-oriented design is proposed in [CK94]. A system is an object oriented design, modules are classes, elements are either methods or instance variables (depending on the measure considered) and relationships are calls to methods or uses of instance variables by other methods. These measures are validated against Weyuker's properties for complexity measures, thereby implicitely implying that they were complexity measures. However, none of the measures defined by [CK94] is a complexity measure according to our properties:

- Weighted Methods per Class (WMC) and Number Of Children of a class (NOC) are size measures (see Section 3.1);
- Depth of Inheritance Tree (DIT) is a length measure (see Section 3.2);
- Coupling Between Object classes (CBO) is a coupling measure (see Section 3.4);
- Response For a Class (RFC) is a size and coupling measure (see Sections 3.1 and 3.5);
- Lack of COhesion in Methods (LCOM) cannot be classified in our framework. This is consistent with what was said in the introduction: our framework does not cover all possible measurement concepts.

This is not surprising. In [CK94], it is shown that all of the above measures do not satisfy Weyuker's property 9, which is a weaker form of property Complexity.4 (see Section 4).

3.4. Cohesion

Motivation

The concept of cohesion has been used with reference to modules or modular systems. It assesses the tightness with which "related" program features are "grouped together" in systems or modules. It is assumed that the better the programmer is able to encapsulate related program features together, the more reliable and maintainable the system [F91]. This assumption seems to be supported by experimental results [BMB94(a)]. Intuitively, we expect cohesion to be non-negative and, more importantly, to be normalized (property Cohesion.1) so that the measure is independent of the size of the modular system or module. Moreover, if there are no internal relationships in a module or in all the modules in a system, we expect cohesion to be null (property Cohesion.2) for that module or for the system, since, as far as we know, there is no relationship between the elements and there is no evidence they should be encapsulated together. Additional internal relationships in modules cannot decrease cohesion since they are supposed to be additional evidence to encapsulate system elements together (property Cohesion.3). When two (or more) modules showing no relationships between them are merged, cohesion cannot increase because seemingly unrelated elements are encapsulated together (property Cohesion.4).

Since the cohesion (and, as we will see in Section 3.5, the coupling) of modules and entire modular systems have similar sets of properties, both will be described at the same time by using brackets and the alternation symbol '|'. For instance, the notation [A|B], where A and B are phrases, will denote the fact that phrase A applies to module cohesion, and phrase B applies to entire system cohesion.

Definition 6: Cohesion of a [Module | Modular System]
The cohesion of a [module $m = <E_m,R_m>$ of a modular system MS | modular system MS] is a function [Cohesion(m)|Cohesion(MS)] characterized by the following properties Cohesion.1-Cohesion.4.

◊

Property *Cohesion.1*: Non-negativity and Normalization
The cohesion of a [module $m = <E_m,R_m>$ of a modular system MS = <E,R,M> | modular system MS = <E,R,M>] belongs to a specified interval

$$[\ \text{Cohesion}(m) \in [0,\text{Max}] \quad | \quad \text{Cohesion}(MS) \in [0,\text{Max}]\ ] \qquad \text{(Cohesion.I)}$$

◊

Normalization allows meaningful comparisons between the cohesions of different [modules|modular systems], since they all belong to the same interval.

Property *Cohesion.2:* Null Value.
The cohesion of a [module $m = <E_m,R_m>$ of a modular system $MS = <E,R,M>$ | modular system $MS = <E,R,M>$] is null if [R_m|IR] is empty

$[\ R_m = \emptyset \Rightarrow \text{Cohesion}(m) = 0 \mid IR = \emptyset \Rightarrow \text{Cohesion}(MS) = 0\ ]$ (Cohesion.II)

(Recall that IR is the set of intra-module relationships, defined in Definition 2.)

◊

If there is no intra-module relationship among the elements of a (all) module(s), then the module (system) cohesion is null.

Property *Cohesion.3:* Monotonicity.
Let $MS' = <E,R',M'>$ and $MS" = <E,R",M">$ be two modular systems (with the same set of elements E) such that there exist two modules $m' = <E_m,R_{m'}>$ and $m" = <E_m,R_{m"}>$ (with the same set of elements E_m) belonging to M' and M" respectively, such that $R' - R_{m'} = R" - R_{m"}$, and $R_{m'} \subseteq R_{m"}$ (which implies $IR' \subseteq IR"$). Then,

$[\ \text{Cohesion}(m') \leq \text{Cohesion}(m") \mid \text{Cohesion}(MS') \leq \text{Cohesion}(MS")\ ]$ (Cohesion.III)

◊

Adding intra-module relationships does not decrease [module|modular system] cohesion. For instance, suppose that systems S, S', and S" in Figure 3 are viewed as modular systems $MS = <E,R,M>$, $MS' = <E',R',M'>$, and $MS" = <E",R",M">$ (with $M = \{m_1,m_2,m_3\}$, $M' = \{m'_1,m'_2,m'_3\}$, and $M" = \{m"_1,m"_2,m"_3\}$). We have [$\text{Cohesion}(m'_3) \geq \text{Cohesion}(m_3)$ | $\text{Cohesion}(MS') \geq \text{Cohesion}(MS)$].

Property* Cohesion.4: *Cohesive Modules.
Let $MS' = <E,R,M'>$ and $MS" = <E,R,M">$ be two modular systems (with the same underlying system $<E,R>$) such that $M" = M' - \{m'_1,m'_2\} \cup \{m"\}$, with $m'_1 \in M'$, $m'_2 \in M'$, $m" \notin M'$, and $m" = m'_1 \cup m'_2$. (The two modules m'_1 and m'_2 are replaced by the module m", union of m'_1 and m'_2.) If no relationships exist between the elements belonging to m'_1 and m'_2, i.e., $\text{InputR}(m'_1) \cap \text{OutputR}(m'_2) = \emptyset$ and $\text{InputR}(m'_2) \cap \text{OutputR}(m'_1) = \emptyset$, then

$[\ \max\{\text{Cohesion}(m'_1),\text{Cohesion}(m'_2)\} \geq \text{Cohesion}(m") \mid$
$\text{Cohesion}(MS') \geq \text{Cohesion}(MS")\ ]$ (Cohesion.IV)

◊

The cohesion of a [module|modular system] obtained by putting together two unrelated modules is not greater than the [maximum cohesion of the two original modules|the cohesion of the original modular system].
Properties Cohesion.1 - Cohesion.4 hold when applying the admissible transformation of the ratio scale. Therefore, there is no contradiction between our concept of cohesion and the definition of cohesion measures on a ratio scale.

Examples of cohesion measures

In [BMB94(a)], cohesion measures for high-level design are defined and validated, at both the abstract data type (module) and system (program) levels. For brevity's sake, the term software part here denotes either a module or a program. A high-level design is seen as a collection of modules, each of which exports and imports constants, types, variables, and procedures/functions. A widely accepted software engineering principle prescribes that each module be highly cohesive, i.e., its elements be tightly related to each other. [BMB94(a)] focuses on investigating whether high cohesion values are related to lower error-proneness, due to the fact that the changes required by a change in a module are confined in a well-encapsulated part of the overall program. To this end, the exported feature A is said to interact with feature B if the change of one of A's definitions or uses may require a change in one of B's definitions or uses.

In the approach of the present paper, each feature exported by a module is an element of the system, and the interactions between them are the relationships between elements. A module according to [BMB94(a)] is represented by a module according to the definition of the present paper. At high-level design time, not all interactions between the features of a module are known, since the features may interact in the body of a module, and not in its visible part. Given a software part sp, three cohesion measures NRCI(sp), PRCI(sp), and ORCI(sp) (respectively, Neutral, Pessimistic, and Optimistic Ratio of Cohesive Interactions) are defined for software as follows

$$\mathrm{NRCI(sp)} = \frac{\mathrm{\#KnownInteractions(sp)}}{\mathrm{\#MaxInteractions(sp)\text{-}\#UnknownInteractions(sp)}}$$

$$\mathrm{PRCI(sp)} = \frac{\mathrm{\#KnownInteractions(sp)}}{\mathrm{\#MaxInteractions(sp)}}$$

$$\mathrm{ORCI(sp)} = \frac{\mathrm{\#KnownInteractions(sp)+\#UnknownInteractions(sp)}}{\mathrm{\#MaxInteractions(sp)}}$$

where #MaxInteractions(sp) is the maximum number of possible intra-module interactions between the features exported by each module of the software part sp. (Inter-module interactions are not considered cohesive; they may contribute to coupling, instead.) All three measures satisfy the above properties Cohesion.1 - Cohesion.4.

Other examples of cohesion measures can be found in [BO94], where new functional cohesion measures are introduced. Given a procedure, function, or main program, only *data tokens* (i.e., the occurrence of a definition or use of a variable or a constant) are taken into account. The *data slice* for a data token is the sequence of all those data tokens in the program that can influence the statement in which the data token appears, or can be influenced by that statement. Being a sequence, a data slice is ordered: it lists its data tokens in order of appearance in the procedure, function or main program. If more than one data slice exists, some data tokens may belong to more than one data slice: these are called *glue tokens*. A subset of the glue tokens may belong to all data slices: these are called *super-glue tokens*. Functional cohesion measures are defined based on data tokens, glue tokens, and super-glue tokens. This approach can be represented in our framework as follows. A data token is an element of the system, and a data slice is represented as a sequence of nodes and arcs. The resulting graph is a Directed Acyclic Graph, which represents a module. ([BO94] introduces functional cohesion measures for single procedures, functions, or main programs.) Given a procedure, function, or main program p, the following measures SFC(p) (Strong Functional Cohesion), WFC(p) (Weak Functional Cohesion), and A(p) (adhesiveness) are introduced

$$\mathrm{SFC(p)} = \frac{\mathrm{\#SuperGlueTokens}}{\mathrm{\#AllTokens}}$$

$$\mathrm{WFC(p)} = \frac{\mathrm{\#GlueTokens}}{\mathrm{\#AllTokens}}$$

$$\mathrm{A(p)} = \frac{\sum_{\mathrm{GT} \in \mathrm{GlueTokens}} \mathrm{\#SlicesContainingGlueTokenGT}}{\mathrm{\#AllTokens.\#DataSlices}}$$

It can be shown that these measures satisfy the above properties Cohesion.1 - Cohesion.4.

3.5. Coupling

Motivation

The concept of coupling has been used with reference to modules or modular systems. Intuitively, it captures the amount of relationship between the elements belonging to different modules of a system. Given a module m, two kinds of coupling can be defined: inbound coupling and outbound coupling. The former captures the amount of relationships from elements outside m to elements inside m; the latter the amount of relationships from elements inside m to elements outside m.

We expect coupling to be non-negative (property Coupling.1), and null when there are no relationships among modules (property Coupling.2). When additional relationships are created across modules, we expect coupling not to decrease since these modules become more interdependent (property Coupling.3). Merging modules can only decrease coupling since there may exist relationships among them and therefore, inter-module relationships may have disappeared (property Coupling.4, property Coupling.5).

In what follows, when referring to module coupling, we will use the word coupling to denote either inbound or outbound coupling, and OuterR(m) to denote either InputR(m) or OutputR(m).

Definition 7: Coupling of a [Module | Modular System]
The coupling of a [module m = $<E_m,R_m>$ of a modular system MS|modular system MS] is a function [Coupling(m)|Coupling(MS)] characterized by the following properties Coupling.1 - Coupling.5.

◊

Property *Coupling.1*: Non-negativity
The coupling of a [module m = $<E_m,R_m>$ of a modular system|modular system MS] is non-negative

$[\ \text{Coupling}(m) \geq 0 \quad |\ \text{Coupling}(MS) \geq 0\ ]$ (Coupling.I)

◊

Property *Coupling.2*: Null Value
The coupling of a [module m = $<E_m,R_m>$ of a modular system|modular system MS = <E,R,M>] is null if [OuterR(m)|R-IR] is empty

$[\ \text{OuterR}(m)=\varnothing \Rightarrow \text{Coupling}(m)=0\ |\ R\text{-}IR=\varnothing \Rightarrow \text{Coupling}(MS)=0\ ]$ (Coupling.II)

◊

Property *Coupling.3*: Monotonicity
Let MS' = <E,R',M'> and MS" = <E,R",M"> be two modular systems (with the same set of elements E) such that there exist two modules m' ∈ M', m" ∈ M" such that R' - OuterR(m') = R" - OuterR(m"), and OuterR(m') ⊆ OuterR(m"). Then,

$[\ \text{Coupling}(m')\leq\text{Coupling}(m")\ |\ \text{Coupling}(MS')\leq\text{Coupling}(MS")\ ]$ (Coupling.III)

◊

Adding inter-module relationships does not decrease coupling. For instance, if systems S, and S" in Figure 3 are viewed as modular systems (see Section 3.4), we have [Coupling($m"_1$) ≥ Coupling(m_1) | Cohesion(MS") ≥ Cohesion(MS)].

Property *Coupling.4*: Merging of Modules
Let MS' = <E',R',M'> and MS" = <E",R",M"> be two modular systems such that E' = E", R' = R", and M" = M' - {m'_1,m'_2} ∪ {m"}, where $m'_1 = <E_{m'1},R_{m'1}>$, $m'_2 = <E_{m'2},R_{m'2}>$, and m" = $<E_{m"},R_{m"}>$, with m'_1 ∈ M', m'_2 ∈ M', m" ∉ M', and $E_{m"} = E_{m'1} \cup E_{m'2}$ and $R_{m"} = R_{m'1} \cup$

$R_{m'_2}$. (The two modules m'_1 and m'_2 are replaced by the module m'', whose elements and relationships are the union of those of m'_1 and m'_2.) Then

$$[\ \text{Coupling}(m'_1) + \text{Coupling}(m'_2) \geq \text{Coupling}(m'')\ |\ \text{Coupling}(MS') \geq \text{Coupling}(MS'')\ ] \qquad (\text{Coupling.IV})$$

◊

The coupling of a [module|modular system] obtained by merging two modules is not greater than the [sum of the couplings of the two original modules|coupling of the original modular system], since the two modules may have common inter-module relationships. For instance, suppose that the modular system MS_{12} in Figure 6 is obtained from the modular system MS in Figure 2 by merging modules m_1 and m_2 into module m_{12}. Then, we have [Coupling(m_1) + Coupling(m_2) ≥ Coupling(m_{12}) | Coupling(MS) ≥ Coupling(MS_{12})].

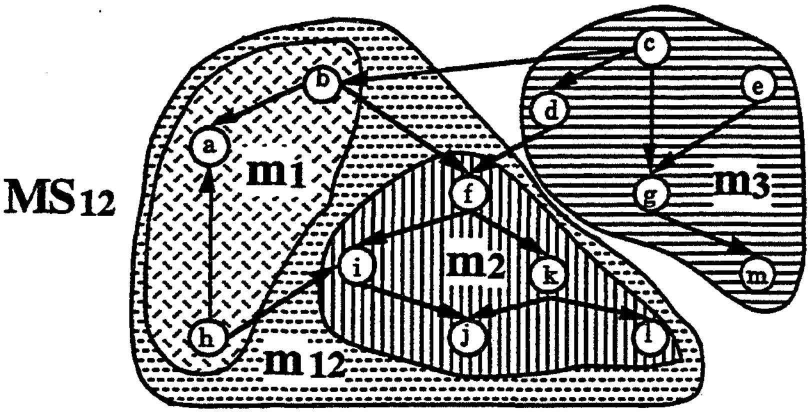


Figure 6. The effect of merging modules on coupling.

Property *Coupling.5*: Disjoint Module Additivity
Let MS' = <E,R,M'> and MS" = <E,R,M"> be two modular systems (with the same underlying system <E,R>) such that $M'' = M' - \{m'_1, m'_2\} \cup \{m''\}$, with $m'_1 \in M'$, $m'_2 \in M'$, $m'' \notin M'$, and $m'' = m'_1 \cup m'_2$. (The two modules m'_1 and m'_2 are replaced by the module m'', union of m'_1 and m'_2.) If no relationships exist between the elements belonging to m'_1 and m'_2, i.e., $\text{InputR}(m'_1) \cap \text{OutputR}(m'_2) = \varnothing$ and $\text{InputR}(m'_2) \cap \text{OutputR}(m'_1) = \varnothing$, then

$$[\ \text{Coupling}(m'_1) + \text{Coupling}(m'_2) = \text{Coupling}(m'')\ |\ \text{Coupling}(MS') = \text{Coupling}(MS'')\ ] \qquad (\text{Coupling.V})$$

◊

The coupling of a [module|modular system] obtained by merging two unrelated modules is equal to the [sum of the couplings of the two original modules|coupling of the original modular system].

Properties Coupling.1 - Coupling.5 hold when applying the admissible transformations of the ratio scale. Therefore, there is no contradiction between our concept of coupling and the definition of coupling measures on a ratio scale.

Examples and counterexamples of coupling measures

Fenton has defined an ordinal coupling measure between pairs of subroutines [F91] as follows:

$$C(S, S') = i + \frac{n}{n+1}$$

where i is the number corresponding to the worst coupling type (according to Myers' ordinal scale [F91]) and n the number of interconnections between S and S', i.e., global variables and formal parameters. In this case, systems are programs, modules are subroutines, elements are formal parameters and global variables. If coupling for the whole system is defined as the sum of coupling values between all subroutine pairs, properties Coupling.1 - Coupling.5 hold for this measures and we label it as a coupling measure. However, Fenton proposes to calculate the median of all the pair values as a system coupling measure. In this case, property Coupling.3 does not hold since the median may decrease when inter-module relationships are added. Similarly for Coupling.4, when subroutines are merged and inter-module relationships are lost, the median may increase. Therefore, the system coupling measure proposed by Fenton is not a coupling measure according to our definitions.

In [BMB94(a)], coupling measures for high-level design are defined and validated, at both the module (abstract data type) and system (program) levels. They are based on the notion of interaction introduced in the examples of Section 3.4. Import Coupling of a module m is defined as the extent to which m depends on imported external data declarations. Similarly, export coupling of m is defined as the extent to which m's data declarations affect the other data declarations in the system. At the system level, coupling is the extent to which the modules are related to each other. Given a module m, Import Coupling of m (denoted by IC(m)) is the number of interactions between data declarations external to m and the data declarations within m. Given a module m, Export Coupling of m (denoted by EC(m)) is the number of interactions between the data declarations within m and the data declarations external to m. As shown in [BMB94(a)], our coupling properties hold for these measures.

Coupling Between Object classes (CBO) of a class is defined in [CK94] as the number of other classes to which it is coupled. It is a coupling measure. Properties Coupling.1 and Coupling.2 are obviously satisfied. Property Coupling.3 is satisfied, since CBO cannot decrease by adding one more relationship between features belonging to different classes (i.e., one class uses one more method or instance variable belonging to another class). Property Coupling.4 is satisfied: CBO can only remain constant or decrease when two classes are grouped into one. Property Coupling.4 is also satisfied.

Response For a Class (RFC) [CK94] is a size and a coupling measure at the same time (see Section 3.1). Methods are elements, calls are relationships, classes are modules. Coupling.3 holds, since adding outside method calls to a class can only increase RFC and Coupling.4 holds because merging classes does not change RFC's value since RFC does not distinguish between inside and outside method calls. Similarly, when there are no calls between the classes' methods, Coupling.5 holds. This result is to be expected since RFC is the result of the addition of two terms: the number of methods in the class, a size measure, and the number of methods called, a coupling measure.

3.6. Comparison of Concept Properties

We want to summarize the important differences and similarities between the system concepts introduced in this paper. Table 1 uses only criteria that can be compared across the concepts of size, length, complexity, cohesion, and coupling. First, it is important to recall that coupling and cohesion are only defined in the context of modular systems, whereas size, length and complexity are defined for all systems.

Second, the concepts appear to have the null value (second column) and monotonicity (third column) properties based on different sets. The behavior of a measure with respect to variations in such sets characterizes the nature of the measure itself, i.e., the concept(s) it captures. As RFC, defined in [CK94], shows (see Sections 3.1 and 3.5), the same measure may satisfy the sets of properties associated with different concepts. As a matter of fact, similar sets of properties associated with different concepts are not contradictory.

Third, when systems are made of disjoint modules, size, complexity and coupling are additive (properties Size.3, Complexity.5, and Coupling.5). Cohesion and length are not additive.

Concepts\Properties	Null Value	Monotonicity	Additivity
Size	$E = \emptyset$	E	Yes
Length	$E=\emptyset$	R	No
Complexity	$R = \emptyset$	R	Yes
System Cohesion	$IR=\emptyset$	IR	No
System Coupling	$R\text{-}IR=\emptyset$	R-IR	Yes

Table 1: Comparison of concept properties

This summary shows that these concepts are really different with respect to basic properties. Therefore, it appears that desirable properties are likely to vary from one measurement concept to another.

4. Comparison with Related Work

We mainly compare our approach with the other approaches for defining sets of properties for software complexity measures, because they have been studied more extensively and thoroughly than other kinds of measures. Besides, we compare our approach with the axioms introduced by Fenton and Melton [FM90] for software coupling measures. As already mentioned, our approach generalizes previous work on properties for defining complexity measures. Unlike previous approaches, it is not constrained to deal with software code only, but, because of its generality, can be applied to other artifacts produced during the software lifecycle, namely, software specifications and designs. Moreover, it is not defined based on some control flow operations, like sequencing or nesting, but on a general representation, i.e., a graph.

Weyuker[3]

Weyuker's work [W88] is one of the first attempts to formalize the fuzzy concept of program complexity. This work has been discussed by many authors [CK94, F91, LJS91, TZ92, Z91] and is still a point of reference and comparison for anyone investigating the topic of software complexity.

To make Weyuker's properties comparable with ours, we will assume that a program according to Weyuker is a system according to our definition; a program body is a module of a system. A whole program is built by combining program bodies, by means of sequential, conditional, and iterative constructs (plus the program and output statements, which can be seen as "special" program bodies), and, correspondingly, a system can be built from its constituent modules. Since some of Weyuker's properties are based on the sequencing between pairs of program bodies P and Q, we provide more details about the representation of sequencing in our framework. Sequencing of program bodies P and Q is obtained via the composition operation (P;Q). Correspondingly, if $S_P = <E_P,R_P>$ and $S_Q = <E_Q,R_Q>$ are the modules representing the two program bodies P and Q[4], then, we will denote the representation of P;Q as $S_{P;Q} = <E_{P;Q},R_{P;Q}>$. In what follows, we will assume that $E_{P;Q} = E_P \cup E_Q$ and $R_{P;Q}\quad R_P \cup R_Q$, i.e., the representation of the composition of two program bodies contains the elements of the representation of each program body, and at least contains all the relationships belonging to each of the representations of program bodies. In other words, S_P and S_Q are modules of $S_{P;Q}$.

[3] We will list properties/axioms by the initial of the proponents. So, Weyuker's properties will be referred to as W1, W2, ..., W9, Tian and Zelkowitz's as TZ1 to TZ5, and Lakshmanian et alii's as L1 to L9.

[4] In what follows, we will use the notation $S_P = <E_P,R_P>$ to denote the representation of program body P.

W1: *A complexity measure must not be "too coarse" (1).*
$\exists$ S_P, S_Q Complexity(S_P) $\neq$ Complexity(S_Q)

W2: *A complexity measure must not be "too coarse" (2).* Given the nonnegative number c, there are only finitely many systems of complexity c.

W3: *A complexity measure must not be "too fine."* There are distinct systems S_P and S_Q such that Complexity(S_P) = Complexity(S_Q).

W4: *Functionality.* There is no one-to-one correspondence between functionality and complexity
$\exists$ S_P,S_Q P and Q are functionally equivalent **and** Complexity(S_P) $\neq$ Complexity(S_Q)

W5: *Monotonicity with respect to composition.*
$\forall$ S_P,S_Q
Complexity(S_P) $\leq$ Complexity($S_{P;Q}$) **and** Complexity(S_Q) $\leq$ Complexity($S_{P;Q}$)

W6: *The contribution of a module in terms of the overall system complexity may depend on the rest of the system.*
(a) $\exists$ S_P, S_Q, S_T Complexity(S_P) = Complexity(S_Q) **and** Complexity($S_{P;T}$) $\neq$ Complexity($S_{Q;T}$)

(b) $\exists$ S_P, S_Q, S_T Complexity(S_P) = Complexity(S_Q) **and** Complexity($S_{T;P}$) $\neq$ Complexity($S_{T;Q}$)

W7: *A complexity measure is sensitive to the permutation of statements.*
$\exists$ S_P, S_Q Q is formed by permuting the order of statements of P **and** Complexity(S_P) $\neq$ Complexity(S_Q)

W8: *Renaming.* If P is a renaming of Q, then Complexity(S_P)=Complexity(S_Q).

W9: *Module monotonicity.*
$\exists$ S_P, S_Q Complexity(S_P) + Complexity(S_Q) $\leq$ Complexity($S_{P;Q}$)

Analysis of Weyuker's properties

W1, W2, W3, W4, W8: These are not implied by our properties, but they do not contradict any of them, so they can be added to our set, if desired. However, we think that these properties are general to all syntactically-based product measures and do not appear useful in our framework to differentiate concepts.

W5: This is implied by our properties, as shown by inequality (Complexity.VI), since S_P and S_Q are modules of $S_{P;Q}$.

W6, W7: These properties are not implied by the above properties Complexity.1 - Complexity.5. However, they show a very important and delicate point in the context of complexity measure definition.

By assuming properties W6(a) and W6(b) to be false, one forces all complexity measures to be strongly related to control flow, since this would exclude that the composition of two program bodies may yield additional relationships between elements (e.g., data declarations) of the two program bodies. If properties W6(a) and W6(b) are assumed true, one forces all complexity measures to be sensitive to at least one other kind of additional relationship.

Similarly, W7 states that the order of the statements, and therefore the control flow, should have an impact on all complexity measures. By assuming property W7 to be false, one forces all complexity measures to be insensitive to the ordering of statements. If property W7 is assumed true, one forces all complexity measures to be somehow sensitive to the ordering of statements, which may not always be useful.

W8: We analyze this property again, to better explain the relationship between complexity and understandability. According to this property, renaming does not affect complexity. However, it is a fact that renaming program variables by absurd or misleading names greatly impairs understandability. This shows that other factors, besides complexity, affect understandability and the other external qualities of software that are affected by complexity.

As for properties W1-W8, our approach is somewhat more liberal than Weyuker's. For instance, the constant null function is an acceptable complexity measure according to our properties, while it is not acceptable according to Weyuker's properties. It is evident that the usefulness of such a complexity measure is questionable. We think that properties should be used to check whether a measure actually addresses a given concept (e.g., complexity). However, given any set of properties, it is almost always possible to build a measure that satisfies them, but is of no practical interest (see [CS91]). At any rate, this is not a sensible reason to reject a set of properties associated with a concept (how many sensless measures could be defined that satisfy the three properties that characterize distance!). Rather, measures that satisfy a set of properties must be later assessed with regard to their usefulness.

W9: This is probably the most controversial property. The above properties Complexity.1 - Complexity.5 imply it. Actually, our properties imply the stronger form of W9, the unnumbered property following W9 in Weyuker's paper [W88] (see also [P84])

$\forall\ S_P, S_Q\ \text{Complexity}(S_P) + \text{Complexity}(S_Q) \leq \text{Complexity}(S_{P;Q})$

Weyuker rejects it on the basis that it might lead to contradictions: she argues that the effort needed to implement or understand the composition of a program body P with itself, is probably not twice as much as the effort needed for P alone. Our point is that complexity is not the only factor to be taken into account when evaluating the effort needed to implement or understand a program, nor is it proven that this effort is in any way "proportional" to product complexity.

Fenton

In addition to Weyuker's work, Fenton [F94] shows that, based on measurement-theoretic mathematical grounds, there is no chance that a general measure for software complexity will ever be found, nor even for control flow complexity, i.e., a more specific kind of complexity. We totally agree with that. By no means do we aim at defining a single complexity measure, which captures all kinds of complexity in a software artifact. Instead, our set of properties define constraints for any specific complexity measure, whatever facet of complexity it addresses.

Fenton and Melton [FM90] introduced two axioms that they believe should hold for coupling measures. Both axioms assume that coupling is a measure of connectivity of a system represented by its module design chart (or structure chart). The first axiom is similar to our monotonicity property (Coupling.3). It states that if the only difference between two module design charts D and D' is an extra interconnection in D', then the coupling of D' is higher than the coupling of D. The second axiom basically states that system coupling should be independent from the number of modules in the system. If a module is added and shows the same level of pairwise coupling as the already existing modules, then the coupling of the system remains constant. According to our properties, coupling is seen as a measure which is to a certain extent dependent on the number of modules in the system and we therefore do not have any equivalent axiom. This shows that the sets of properties that can be defined above are, to some extent, subjective.

Zuse

In his article in the *Encyclopaedia of Software Engineering* [ESE94 pp. 131-165], Zuse applies a measurement-theoretic approach to complexity measures. The focus is on the conditions that should be satisfied by empirical relational systems in order to provide them with additive ratio scale measures. This class of measures is a subset of ratio scale measures, characterized by the additivity property (Theorems 2 and 3 of [ESE94]). Given the set P of flowgraphs and a binary operation * between flowgraphs (e.g., concatenation), additive ratio scale complexity measures are such that, for each pair of flowgraphs P1, P2,

Complexity(P1*P2) = Complexity(P1) + Complexity(P2)

This property shows that a different concept of complexity is defined by Zuse, with respect to that defined by Weyuker's (W9) and our properties (Complexity.4). It is our belief that, by requiring that complexity measures be additive, important aspects of complexity may not be fully captured, and complexity measures actually become quite similar to size measures. Considering complexity as additive means that, when two modules are put together to form a new system, no additional dependencies between the elements of the modules should be taken into account in the computation of the system complexity. We believe this is a very questionable assumption for product complexity.

Tian and Zelkowitz

Tian and Zelkowitz [TZ92] have provided axioms (necessary properties) for complexity measures and a classification scheme based on additional program characteristics that identify important measure categories. In the approach, programs are represented by means of their abstract syntax trees (e.g., parse trees). To translate this representation into our framework, we will assume that the whole program, represented by the entire tree, is a system, and that any part of a program represented by a subtree is a module.

TZ1: Systems with identical functionality are comparable, i.e., there is an order relation between them with respect to complexity.
TZ2: A system is comparable with its module(s).
TZ3: Given a system S_Q and any module S_P whose root, in the abstract tree representation, is "far enough" from the root of S_Q, then S_P is not more complex than S_Q. In other words, "small" modules of a system are no more complex than the system.
TZ4: If an intuitive complexity order relation exists between two systems, it must be preserved by the complexity measure (it is a weakened form of the representation condition of Measurement Theory [F91]).
TZ5: Measures must not be too coarse and must show sufficient variability.

TZ1, TZ2, TZ5 do not differentiate software characteristics (concepts) and can be used for all syntactic product measures. **TZ3** can be derived from our set of properties. **TZ4** captures the basic purpose behind the definition of all measures: preserving an intuitive order on a set of software artifacts [MGB90].

The additional set of properties which is presented in [TZ92] is used to define a measure classification system. It determines whether or not a measure is based exclusively on the abstract syntax tree of the program, whether it is sensitive to renaming, whether it is sensitive to the context of definition or use of the measured program, whether it is determined entirely by the performed program operations regardless of their order and organization, and whether concatenation of programs always contribute positively toward the composite program complexity (i.e., system monotonicity).

Some of these properties are related to the properties defined in this paper and we believe they are characteristic properties of distinct system concepts (e.g., system monotonicity). Others do not differentiate the various concepts associated with syntactically-based measures (e.g., renaming).

Lakshmanian et al.

Lakshmanian et al. [LJS91] have attempted to define necessary properties for software complexity measures based on control flow graphs. In order to make these properties comparable to ours, we will use a notation similar to the one used to introduce Weyuker's properties. A program according to Lakshmanian et al. (represented by a control flow graph) is a system according to our definition, and a program segment is a module. In addition to sequencing, these properties use the nesting program construct denoted as @. "A program segment Z is said to be obtained by nesting [program segment] Y at the control location i in [program segment] X (denoted by $Y@X_i$) if the program segment X has at least one conditional branch, and if Y is embedded at location i in X in such a way that there exists at least one control flow path in the combined code Z that completely skips Y." "The notation Y@X refers to any nesting of Y in X if the specific location in X at which Y is embedded is immaterial."

In what follows, X, Y, Z will denote programs or program segments; S_X, S_Y, S_Z will denote the corresponding systems or modules according to our definition. Lakshmanian et al. [LJS91] introduce nine properties. However, only five out of them can be considered basic, since the remaining four can be derived from them. Therefore, below we will only discuss the compatibility of the basic properties with respect to our properties.

L1: Non-negativity.

L1(a): Null value.
If the program only contains sequential code (referred to as a basic block B) then

$\text{Complexity}(S_B) = 0$

L1(b): Positivity.
If the program X is not a basic block, then

$\text{Complexity}(S_X) > 0$

◊

Property L1 does not contradict any of our properties (in particular, Complexity 1 and Complexity 2).

L5: Additivity under sequencing.
$\text{Complexity}(S_{X;Y}) = \text{Complexity}(S_Y) + \text{Complexity}(S_X)$

◊

This property does not contradict properties Complexity.4 and Complexity.5, where the equality sign is allowed. By requiring that complexity be additive under sequencing, Lakshmanian et al take a viewpoint which is very similar to that of Zuse.

L6: Functional independence under nesting.
Adding a basic block B to a system X through nesting does not increase its complexity

$\text{Complexity}(S_{B@X}) = \text{Complexity}(S_X)$

◊

L7: Monotonicity under nesting.
$\text{Complexity}(S_{Y@X_i}) < \text{Complexity}(S_{Z@X_i})$ if $\text{Complexity}(S_Y) < \text{Complexity}(S_Z)$

◊

These properties are compatible with our properties.

L9: Sensitivity to nesting.
$\text{Complexity}(S_{X;Y}) < \text{Complexity}(S_{Y@X})$ if $\text{Complexity}(S_Y) > 0$

◊

This property does not contradict our properties.

In conclusion, none of the above properties contradicts our properties. However, the scope of these properties is limited to the sequencing and nesting of control flow graphs, and therefore to the study of control flow complexity.

As for the other properties, we now show how they can be derived from L1, L5, L6, L7, and L9.

L2: Functional independence under sequencing.
$\text{Complexity}(S_{X;B}) = \text{Complexity}(S_X)$

This property follows from L5 (first equality below) and L1 (second equality below):

$\text{Complexity}(S_{X;B}) = \text{Complexity}(S_X) + \text{Complexity}(S_B) = \text{Complexity}(S_X)$

◊

L3: Symmetry under sequencing.
$\text{Complexity}(S_{X;Y}) = \text{Complexity}(S_{Y;X})$

This property follows from L5 (both equalities)
$\text{Complexity}(S_{X;Y}) = \text{Complexity}(S_X) + \text{Complexity}(S_Y) = \text{Complexity}(S_{Y;X})$

◊

L4: Monotonicity under sequencing.
$\text{Complexity}(S_{X;Y}) < \text{Complexity}(S_{X;Z})$ if $\text{Complexity}(S_Y) < \text{Complexity}(S_Z)$
$\text{Complexity}(S_{X;Y}) = \text{Complexity}(S_{X;Z})$ if $\text{Complexity}(S_Y) = \text{Complexity}(S_Z)$

This property follows from L5:

if $\text{Complexity}(S_Y) < \text{Complexity}(S_Z)$, then
$\text{Complexity}(S_{X;Y}) = \text{Complexity}(S_X) + \text{Complexity}(S_Y)$
$< \text{Complexity}(S_X) + \text{Complexity}(S_Z) = \text{Complexity}(S_{X;Z})$
if $\text{Complexity}(S_Y) = \text{Complexity}(S_Z)$, then
$\text{Complexity}(S_{X;Y}) = \text{Complexity}(S_X) + \text{Complexity}(S_Y)$
$= \text{Complexity}(S_X) + \text{Complexity}(S_Z) = \text{Complexity}(S_{X;Z})$

◊

L8: Monotonicity under nesting.
$\text{Complexity}(S_Y) < \text{Complexity}(S_{Y@X})$

This property follows from L1 (first inequality below, since $\text{Complexity}(S_X)>0$—X cannot be a basic block), L5 (equality below) and L9 (second inequality below)

$\text{Complexity}(S_Y) < \text{Complexity}(S_X) + \text{Complexity}(S_Y)$
$= \text{Complexity}(S_{X;Y}) < \text{Complexity}(S_{Y@X})$

◊

5. Conclusion and Directions for Future Work

In order to provide some guidelines for the analyst in charge of defining product measures, we propose a framework for software measurement where various software measurement concepts are distinguished and their specific properties defined in a generic manner. Such a framework is, by its very nature, somewhat subjective and there are possible alternatives to it. However, it is a practical framework since the properties we capture are, we believe, interesting and all the concepts can be distinguished by different sets of properties.

For example, these properties can be used to guide the search for new product measures as shown in [BMB94(b)]. Moreover, we hope this framework will help avoid future confusion, often encountered in the literature, about what properties product measures should or should not have. Studying measure properties is important in order to provide discipline and rigor to the search for new product measures. However, the relevancy of a property to a given measure must be assessed in the context of a well defined measurement concept, e.g., one should not attempt to verify if a length measure is additive.

This framework does not prevent useless measures from being defined. The usefulness of a measure can only be assessed in a given context (i.e., with respect to a given experimental goal and environment) and after a thorough experimental validation [BMB94(b)]. This framework is not a global answer to the problems of software engineering measurement; it is just of the necessary components of a measure validation process as presented in [BMB94(b)].

Future research will include the definition of more specific measurement frameworks for particular product abstractions, e.g., control flow graphs, data dependency graphs. Also, new concepts could be defined, such as information content (in the information theory sense).

Acknowledgments

We would like to thank Wolfgang Heuser, Yong-Mi Kim, Bryan Hsueh, Oliver Laitenberger, Carolyn Seaman, and Marvin Zelkowitz for reviewing the drafts of this paper.

References

[BMB94(a)] L. Briand, S. Morasca, V. Basili, "Defining and Validating High-Level Design Metrics," CS-TR 3301, University of Maryland, College Park

[BMB94(b)] L. Briand, S. Morasca, and V. R. Basili, "A Goal-Driven Definition Process for Product Metrics Based on Properties," University of Maryland, Department of Computer Science, Tech. Rep. CS-TR-3346, UMIACS-TR-94-106, 1994. Submitted for publication.

[BO94] J. Bieman and L. M. Ott, "Measuring Functional Cohesion," *IEEE Trans. Software Eng.*, vol. 20, no. 8, pp. 644-657, August 1994.

[C90] D. Card, "Measuring Software Design Quality," Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1990.

[CK94] S. R. Chidamber and C. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Trans. Software Eng.*, vol. 20, no. 6, pp. 476-493, June 1994.

[CS91] J. C. Cherniavsky and C. H. Smith, "On Weyuker's Axioms for Software Complexity Measures," *IEEE Trans. Software Eng.*, vol. 17, no. 6, pp. 636-638, June 1991.

[ESE94] Encyclopaedia of Software Engineering, Wiley&Sons Inc., 1994

[F91] N. Fenton, "Software Metrics, A Rigorous Approach," Chapman&Hall, 1991.

[F94] N. Fenton, "Software Measurement: A Necessary Scientific Basis," *IEEE Trans. Software Eng.*, vol. 20, no. 3, pp. 199-206, March 1994.

[FM90] N. Fenton and A. Melton, "Deriving Structurally Based Software Measures," *J. Syst. Software*, vol. 12, pp. 177-187, 1990.

[H77] M. H. Halstead, "Elements of Software Science," Elsevier North-Holland, 1977.

[H92] W. Harrison, "An Entropy-Based Measure of Software Complexity," *IEEE Trans. Software Eng.*, vol. 18, no. 11, pp. 1025-1029, November 1992.

[HK81] S. Henry and D. Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Trans. Software Eng.*, vol. 7, no. 5, pp. 510-518, September 1981.

[LJS91] K. B. Lakshmanan, S. Jayaprakash, and P. K. Sinha, "Properties of Control-Flow Complexity Measures," *IEEE Trans. Software Eng.*, vol. 17, no. 12, pp. 1289-1295, Dec. 1991.

[McC76] T. J. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.*, vol. 2, no. 5, pp. 308-320, Apr. 1976.

[MGB90] A. C. Melton, D.A. Gustafson, J. M. Bieman, and A. A. Baker, "Mathematical Perspective of Software Measures Research," *IEE Software Eng. J.*, vol. 5, no. 5, pp. 246-254, 1990.

[O80] E. I. Oviedo, "Control Flow, Data Flow and Program Complexity," *Proc. IEEE COMPSAC*, Nov. 1980, pp. 146-152.

[P72] D. L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, pp. 1053-1058, May 1972.

[P84] R. E. Prather, "An Axiomatic Theory of Software Complexity Measure," *The Computer Journal*, vol 27, n. 4, pp. 340-346, 1984.

[S92] M. Shepperd, "Algebraic Models and Metric Validation," in Formal Aspects of Measurement (T. Denvir, R. Herman, and R. W. Whitty eds.), pp. 157-173, Lecture Notes in Computer Science, Springer Verlag, 1992.

[TZ92] J. Tian and M. V. Zelkowitz, "A Formal Program Complexity Model and Its Application," *J. Syst. Software*, vol. 17, pp. 253-266, 1992.

[W88] E. J. Weyuker, "Evaluating Software Complexity Measures," *IEEE Trans. Software Eng.*, vol. 14, no. 9, pp. 1357-1365, Sept. 1988.

[Z91] H. Zuse, *Software Complexity: Measures and Methods*. Amsterdam: de Gruyter, 1991.

AN ANALYSIS OF ERRORS IN A REUSE-ORIENTED DEVELOPMENT ENVIRONMENT*

William M. Thomas
Dept. of Computer Science
University of Maryland
College Park, MD 20742

Alex Delis
School of Information Systems
Queensland Univ. of Technology
Brisbane, QLD 4001, Australia

Victor R. Basili
Dept. of Computer Science
University of Maryland
College Park, MD 20742

Abstract

Component reuse is widely considered vital for obtaining significant improvement in development productivity. However, as an organization adopts a reuse-oriented development process,the nature of the problems in development is likely to change. In this paper, we use a measurement–based approach to better understand and evaluate an evolving reuse process. More specifically, we study the effects of reuse across seven projects in narrow domain from a single development organization. An analysis of the errors that occur in new and reused components across all phases of system development provides insight into the factors influencing the reuse process. We found significant differences between errors associated with new and various types of reused components in terms of the types of errors committed, when errors are introduced, and the effect that the errors have on the development process.

1 Introduction

Reuse has been advocated as a technique with great potential to increase software development productivity, reduce development cycle time, and improve product quality [AM87, Bro87, BP88]. However, reuse will not just happen–rather, components must be designed for reuse, and organizational elements must be in place to enable projects to take advantage of the reusable artifacts.

Basili and Rombach present a framework of comprehensive support for reuse, including organizational and methodological properties necessary to maximize the benefit of reuse [BR91]. For reuse to attain a significant role in an environment, organizational changes must be made to facilitate the change in development style. Maintaining a library of reusable parts may require resources including personnel, hardware, and software. While increasing

*This was supported in part by the National Aeronautics and Space Administration grant NSG–5123.

the amount of reuse in an environment may reduce certain development activities (e.g., code creation), it will also require additional effort in other activities (e.g., searching for components). With respect to product quality, it is also clear that "reused" does not imply "defect-free." An investigation into the benefits of reuse in the NASA Goddard Space Flight Center (NASA/GSFC) showed that even among components that were intended to be reused verbatim, while their error rate was an order of magnitude lower than newly created code, the error rate is still significant [TDB92]. By analyzing the nature of the defects in the reuse process, one can tailor the process appropriately to best achieve the organization's goals.

There have been several studies into techniques to stock an initial reuse library [CB91, DK93]. One factor to be considered is the structure of the candidate reusable component. Selby investigated various characteristics of new versus reused code in a large collection of FORTRAN projects [Sel88]. Basili and Perricone analyzed tradeoffs between creating a component from scratch versus modifying an existing component [BP84]. This work extends these studies by investigating the nature of errors occurring in a reuse oriented development environment, and drawing conclusions as to their impact in such an environment. In particular, we analyzed a collection of eight medium scale Ada projects developed over a five year period in the NASA/GSFC with respect to the defects found in newly developed and reused components. The goal of the study was to learn about the nature of problems associated with reuse-oriented software development, thereby allowing for improvement of the reuse process. We found significant differences between errors associated with new and with various types of reused components in terms of when errors are being introduced, the effect that they have on the development process, and the type of error being committed. We also found some similarites and some differences with the findings of other investigations into component reuse.

This paper is organized as follows. Section 2 provides a brief overview of reuse-oriented software development, while section 3 gives background about using error analysis for process improvement. Section 4 describes the goals of the study and the data analyzed. The findings from our analysis are presented in section 5, and section 6 summarizes and identifies the major conclusions.

2 Reuse-Oriented Software Development

Reuse has been cited as a technology with the potential to provide a significant increase in software development productivity and quality. For example, Jones estimates that only 15 percent of the developed software is unique to the applications for which it was developed [Jon84]. Reduced development cost is not the only benefit of reuse–in fact, the greatest benefit from reuse may be its impact on maintenance [LG84, Rom91]. The potential for substantial savings from reuse clearly exists. Unfortunately, achieving high levels of reuse still remains an difficult task. A number of issues must be addressed to effectively increase the level of reuse in an organization, including the forms of reuse, and language and organizational support to encourage reuse.

2.1 Types of Reuse

In this study we examined three modes of reuse:

- verbatim reuse, in which the component is unchanged,
- reuse with slight modification, in which the original component is slightly tailored for the new application,
- reuse with extensive modification, in which the original component is extensively altered for the new application.

While differentiating verbatim reuse and reuse via modification is trivial, distinguishing between slight modification and extensive modification is more difficult. Our intent is to distinguish between cases where a component is left essentially intact, but needs some small change for the new application, and cases where a component is significantly altered for its new use. The three types of reuse, and a their expected impact on development are described in the following paragraphs.

Intuitively, verbatim reuse appears to hold the greatest benefit to software development. Development effort is minimized and verification effort is reduced, since the component has previously been developed, tested, and used. There may be an increased cost in integration effort, as the reused component may not squarely fit in the new system, and the developers may not be as familiar with the reused component as they would be with a custom component.

Another means of reuse is achieved by slight modification of an existing component. Here a component remains for the most part unchanged, but is adapted slightly for the new application. For example, a sort routine may be modified to sort a different type of objects. An improvement in terms of reduced development effort and increased quality is expected, although perhaps not to the same degree as in the reused verbatim components. Again, the integration of modified components may be more difficult than that of newly created components; but, because the modified components may be adapted to better match the application, the integration is perhaps not as difficult as with the verbatim reused components. As with verbatim reuse, there may be new errors introduced in the component selection process. However, since the developer does have a greater understanding of the implementation of the modified component, one is more likely to detect that error earlier than if the component was reused verbatim.

Our third category of reuse occurs through extensive modification of an existing component. For example, one may want to change the underlying representation of a particular type while maintaining the operations on the type. If the component was not designed with the representation isolated in the implementation, this may require changes throughout the component. Reuse in this manner is likely to be beneficial only if the component is of a sufficient size and complexity to justify modification as opposed to simply creating a new component from scratch. Since much of the component is new, in many ways this type of reuse may appear similar to new development. However, there are some important distinctions. The number of coded lines is likely to be reduced relative to newly developed code, so one might expect a decrease in error density. However, the extensive modification activity may be more error prone than standard component creation, since the original abstraction is being significantly altered. This mode of component creation may result in more of a "hack" than a well-conceived component. New types of errors may arise, such as removing too much or not enough of the old component.

2.2 Language Issues in Software Reuse

The Ada programming language contains a number of constructs that encourage effective reuse, including packages and generics [Ich85, WCW85, GP87, EG90]. A package is used to group a collection of declarations, such as types, variables, procedures and functions. The package construct allows for the encapsulation of related entities, encouraging the creation of well-defined abstractions such as encapsulated data types. For example, a stack package of a particular type can be created, containing the element type and operations such as push and pop. Through a simple modification of the element type, the package can be adapted to support operation on a different type. This would enable one to move toward the second type of reuse, tailoring the component slightly to suit the new application.

Ada's generic construct provides more support for verbatim reuse, as it enables the creation of more abstract entities. A generic program unit is a template for a module. Instantiation of the generic program unit yields a module. The generic units may be parameterized, i.e., they may require the user to supply types or operations to create a module. This provides a great deal of flexibility in their use. For example, one may parameterize the stack package such that the user must supply the element type to create an instance of the stack. The generic stack can then be used without modification in support of a number of different types.

High levels of reuse may be achieved in languages without such features, however, the approach taken to achieve such reuse will be different. Such differences were reported in a study comparing FORTRAN and Ada reuse in the NASA/SEL [BWS93]. The Ada approach was to develop a set of generics that can be instantiated to support a variety of application types. In contrast, the FORTRAN approach was to develop a collection of libraries specific to each application type. On projects within a very narrow domain, both approaches achieved similar high levels of reuse. However, when there was a significant change in the domain, the Ada approach achieved a sizable amount of reuse (50 percent verbatim reuse), while the FORTRAN approach showed less than 10 percent verbatim reuse [BWS93]. Thus it would appear that the parameterized, generic approach is better suited to development in a dynamic, evolving domain.

While improved language features may help to enable reuse, they alone have not resulted in large-scale reuse in software development. There are other important factors involved–applications must be structured to allow and encourage reuse, and software organizations must be tailored to support a reuse-oriented development paradigm.

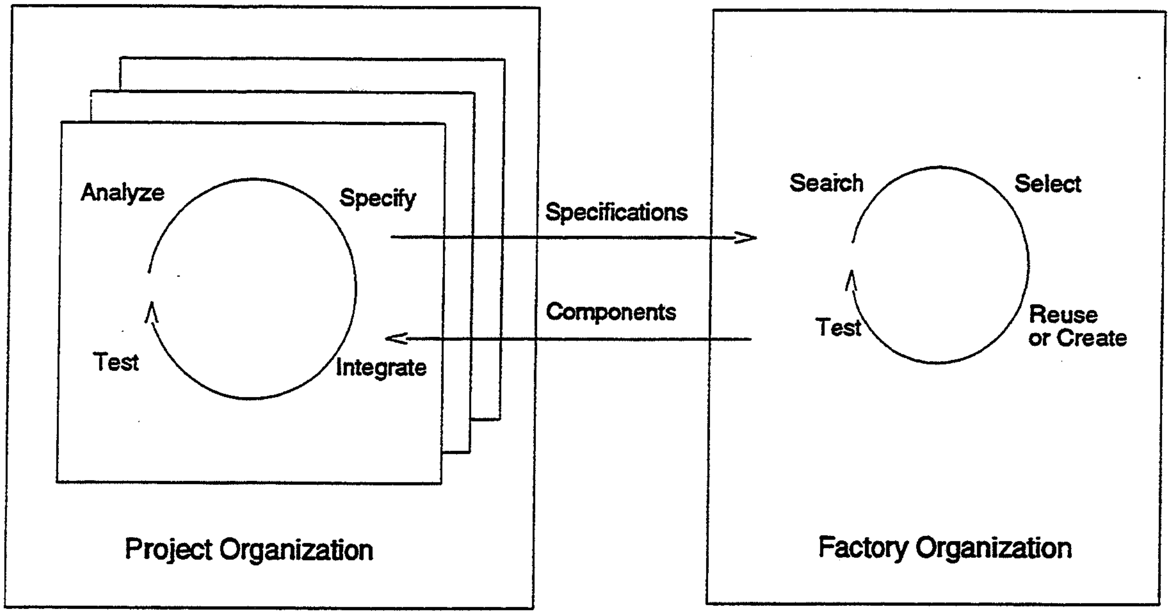


Figure 1: Interaction of a Project Organization with the Component Factory

2.3 Organizational Support for Reuse

One model that integrates reuse into a development is the "component factory" organization, which is a dual-organization structure consisting of two parts: a factory organization and a project organization. The factory organization provides software components in response to requests from the various projects being developed in the project organization [BCC92]. Figure 1 illustrates the component factory concept in support of a project organization. In this setting, the development organization makes requests to the component factory to provide components to be integrated into the desired product. If the component factory is effective, the activity of component creation can be significantly reduced, and the quality of the components that are delivered to the integration team can be increased, reducing the costs of development and of rework. The key features of the component factory are the repository of the components for future reuse, and the focus on flexibility and continuous improvement. Thus a measurement–oriented approach must be utilized, such as that proposed in the TAME project [BR88], which provides an experimental view of software development, allowing for analysis and learning about the effectiveness of the new technologies.

Reuse-oriented development will require some effort to be expended in activities that are not a part of traditional software development. For example, although the component factory will allow the effort spent in component creation to be reduced, it will also require additional activity in searching for and selecting the appropriate component for the particular application. These new activities may also be a potential source of errors in the system, and thus a source of rework effort. Introducing an activity of selecting a component from a repository may introduce new types of errors, for example, selecting a component that does not provide the intended function.

3 Using Error Analysis to Optimize the Development Process

The Quality Improvement Paradigm provides a framework to build a continually improving organization relative to its evolving set of goals [Bas85, BR88]. The QIP consists of six steps:

1. **Characterize** the current project and environment.
2. **Set Goals** for project performance and improvement.
3. **Choose processes**, as well as models and metrics, appropriate for the project.
4. **Execute** the processes, and collect the prescribed data, and provide real-time feedback for corrective action.
5. **Analyze** the data to evaluate current practices and make recommendations for future improvement.
6. **Package** the experience in a form suitable for reuse on future projects.

The first two steps deal with determining the nature of the project, including goals for performance and improvement. Based on the characterization and goals, the third step selects the most suitable processes for the project; establishes the measurement plan, including choosing appropriate models and metrics, and sets up the mechanism for real-time feedback as the project progresses. The fourth step starts the selected processes, collects and the data as prescribed by the measurement plan, and uses the selected models and metrics to provide feedback to the development organization. The fifth and sixth steps occur off-line, as the data is analyzed and packaged into the experience base for use in other projects.

Examining the various dimensions of errors in an organization can yield important lessons learned that may be used to improve software development. The goal of error analysis is to learn about the nature of errors in the current environment so that improvement can be made (e.g., process tailoring) in subsequent projects, and feedback can be provided to the current project. Thus error analysis can be associated with either of the two feedback loops in the model, the project loop, occurring in step 4, in which the results are in real-time provided back to the project, or the corporate loop, in steps 5 and 6, in which results are made available for subsequent projects in the organization. Our focus in this paper is on the corporate loop; i.e., the analysis and packaging steps for subsequent development, from the perspective of reuse-oriented software development.

A number of recent studies have shown that product metrics can be used to determine the areas in a program that are at a greater risk of containing a fault [AE92, SP88, BBH93, BTH93, MK92]. These studies indicate that models can be developed to isolate faulty components in a system based on characteristics of the components and their environment. Our goal is to develop an understanding of the differences between traditional development methods and reuse-oriented methods in terms of the characteristics of their errors. Increased knowledge about the types of errors in an environment can be used to optimize the process. for that environment.

Basili and Selby found that the effectiveness of error detection techniques varies with the type of fault encountered [BS87]. For example, code reading was found to be the most effective technique for isolating interface errors, while functional testing was found to be more effective at finding logic errors. As such, a-priori knowledge of the distribution of the type of errors allows one to select verification techniques most appropriate for the that distribution. Suppose two thirds of the errors are interface errors, and one third logic errors. In this case, we would want to be sure to use techniques that are effective in finding interface errors. Given a limited budget for verification and validation, we may choose to expend more resources in code reading and fewer in functional testing. On the other hand, if a different project is much more likely to have logic errors than interface errors, it may be more effective to focus the verification activities on structural testing.

Knowledge of when the errors are being introduced enables one to apply verification techniques at the most suitable time. If a large number of errors are being introduced in the design phase, adding design inspections to the development process may reduce the number of errors impacting later phases. On the other hand, if most errors are being introduced during coding, design inspections may not be as cost-effective. In this case, one may choose not to inspect design, but choose to have additional verification effort in the coding phase.

The QIP can be used to take advantage of such knowledge. To incorporate this reuse information into the development process, we can develop a mapping to the QIP. The first step of the QIP, characterize the project, can be tailored to include determining the amount and type of reuse expected on the project. The second step, select appropriate models, can include selecting models of expected error profiles based on the characterization of reuse. The third step is to select the appropriate processes. Here, one can choose the processes expected to be most effective for the expected error distribution. The fourth and fifth steps are to execute the processes, collect data, and feedback the results. This can be seen as measuring the actual reuse profile, and measuring the effectiveness of the error mitigation strategies, and making a determination of whether to modify the selected processes based on the new information. For example, if the actual reuse profile is very different from original expectations, one should attempt to understand the factors that led to the difference, and, if appropriate, develop a new projection of the expected error profile.

4 Description of the Analysis

Since its origin, The NASA/GSFC SEL has collected a wealth of data from their software development [SEL94]. Selby performed a study on the characteristics of reused components on a collection of FORTRAN projects from this environment [Sel88], in which the level of reuse averaged 32 percent. Because of the support for reuse provided by the Ada language, as discussed in section 2.2, we chose to analyze the Ada projects in this environment. A much higher level of reuse than what was reported in [Sel88] has been achieved more recently in this environment [Kes90]. The high levels of reuse have been attributed in part to the Ada language constructs and object-oriented methods [Kes90, Sta93, BWS93]. More recently,

Project ID	KSTMT	Pct. Total Reuse	Pct. Verbatim Reuse	Effort (SM)
A	27.1	31	4	175
B	14.4	31	13	85
C	13.7	38	19	72
D	24.8	85	27	117
E	13.8	97	88	30
F	12.8	78	44	73
G	13.7	100	89	16

Table 1: Overview of the Examined Projects

however, even the FORTRAN systems have been showing high levels of reuse, although the nature of the reuse is different than reuse in the Ada development environment.

We analyzed a collection of seven medium-scale Ada projects from a narrow domain, as all are simulators which were developed at the NASA/GSFC Flight Dynamics Division. An overview of the projects examined is provided in Table 1. The projects ranged in size from 61 to 184 thousand source lines, or 12.8 to 27.1 thousand Ada statements (KSTMT). They required development effort of 16 to 175 technical staff months. Reuse ranged from 4 to 89 percent (verbatim), and from 31 to 100 percent (verbatim and with modification).

While this environment is not organized along the lines of the Component Factory discussed in section 2, it does have some characteristics in common with that organization. In the SEL, generalized architectures were developed explicitly to facilitate large scale reuse from project to project [Sta93], so it is clear that significant effort has been applied towards the goal of reuse in the organization. As such, new systems have been developed in accordance with the packaged experience of reusable architectures, designs and code. One aspect of the Component Factory organization is the separate organization that produces or releases all reusable software products [BCC92]. While this feature is not present in the SEL, it is apparent that less effort is being spent on project-specific development activities. The percentage of effort spent in the Coding/Unit Test phase has dropped from 44 percent on an early simulator, to only 18 percent on one of the more recent simulators [Sta93]. This suggests that there is a significant leveraging of the stored experience, and as such, the observed effort on the SEL projects is becoming more in line with the profile one would expect in the Component Factory's project organization, i.e., dominated by design and testing activities.

We developed a set of questions with which to compare newly created, modified, and reused verbatim components:

1. What is the impact of reuse on error density?

2. Are errors in reused units easier to isolate or correct?

3. Are the errors typically being introduced at different phases?

4. Are errors associated with reused units detected earlier in the lifecycle?

Component Origin	No. Comp.	KSTMT	Pct. KSTMT
New	1095	44.2	36.5
Extensively Modified	152	8.8	7.2
Slightly Modified	517	21.6	17.8
Reused Verbatim	1495	46.6	38.5
All Components	3259	121.2	100.0

Table 2: Profile of each class of component origin

5. Are there different kinds of errors associated with reused units?
6. Are there structural differences between new and reused units?

Several types of data were used in our analyses. The first type of data has to do with the origin of a component—whether it was newly created or reused. At the time of component creation a form was filled out by the developer indicating the origin of the component–whether it was to be created new, reused from another component with extensive modification (more than 25 percent changed), reused with slight modification (less than 25 percent changed), or reused verbatim (without change). Table 2 provides a summary of the number of components and source statements in each category of component origin. A larger amount of source code was created in the new and reused verbatim categories than in either of the categories of reuse with modification.

The SEL uses "Change Report Forms" to collect data on changes to components for various reasons, such as error corrections, requirements changes, and planned enhancements. In this analysis, we examined the changes made to correct errors. For each reported error, the form identifies the modules that needed to be changed, the source of the error, (requirements, functional specification, design, code, or previous change), the type of the error (initialization, computational, data value, logic, internal interface, or external interface), and whether or not the error was one of omission (something was not done) or commission (something was done incorrectly).

Finally, we analyzed the systems with a source code static analysis tool, ASAP [Dou87], which provided us with a static profile of each compilation unit, including, for example, basic complexity measures such as McCabe's Cyclomatic Complexity and Halstead's Software Science, as well as counts of various types of declarations and statement usage. ASAP also identifies all **with** statements, so we were able to develop measures of the external declarations visible to each unit.

5 Results of the Analysis

This section presents the major findings from our analysis. We used non-parametric statistical methods to test the hypotheses there were significant differences among the classes

Component Origin	Ave. No. Statements	Ave. No. Parameters	Ave. No. Withs
New	45.8	2.1	3.5
Extensively Modified	59.9	2.1	7.5
Slightly Modified	41.6	1.9	4.0
Reused Verbatim	24.5	2.8	1.1
All Components	36.8	2.3	2.7

Table 3: Structural Characteristics of Subprogram Bodies

of component origin in terms of the the nature and impact of the errors in each class. Structural characteristics of the components are discussed in 5.1, and the remaining sections describe findings associated with with the various dimensions of errors.

5.1 Structural Characteristics

Table 3 shows a collection of measures that characterize the structure of compilation units by class of reuse. Only compilation units that are subprogram bodies were considered, so as not to bias the results with characteristics of instantiations or package specifications. The average number of Ada statements provides an indication of the typical size of a component. The number of parameters is a rough measure of the generality of a component. The number of context couples (i.e., the number of "with" statements) provides an indication of the external dependencies of a particular unit.

What we see is that the reused verbatim components are simpler in terms of their size and external dependencies, as evidenced by the number of source statements and with statements. The reused verbatim units average 24.5 statements and 1.1 withs per unit, while the new units average 45.8 statements and 3.4 withs per unit. The extensively modified units tend to be the most complex, as they average 59.9 statements and 7.5 withs per unit. The slightly modified units tend to be slightly smaller than the new units, but with roughly the same number of external dependencies. It is interesting to note that the extensively modified components are the most complex, both in terms of their size and external complexity. These results are similar to what was reported by Selby in his analysis of reuse in a collection of FORTRAN systems–the reused components tend to be simpler than newly created components in terms of size and interaction with other modules [Sel88]. This additional complexity may result in an increase in difficulty associated with these components in terms or their error density and error correction effort.

We did note one result that is in contrast to Selby's study. He reported that the verbatim reused modules tend to have a smaller interface than newly created units. We observed the opposite–that the verbatim reused modules tend to have more parameters than either the modified or new components. The verbatim reused components averaged 2.8 parameters per unit, versus 1.9 to 2.1 in the new and modified components. This difference is significant at the 0.01 level (i.e., there is less than a one percent chance that there actually is no difference

Project	Ave. No. Statements	Ave. No. Withs	Ave. No. Params.
A	15	0.3	1.9
B	14	0.2	1.8
C	14	0.2	1.8
D	18	0.9	2.7
E	31	1.1	3.0
F	26	1.2	2.1
G	26	1.5	3.1

Table 4: Structural Characteristics in Verbatim Reused Components as Reuse Increases

between the classes). Units that are more highly parameterized have an increased generality that may allow them to be more readily integrated into new applications. As such, we should expect to see a greater number of parameters in the unchanged modules. This difference may be indicative of the approach being taken to reuse in the environment. As previously noted, the Ada approach in this environment was based on the use of well-parameterized generics, while the FORTRAN approach was based on libraries of more specialized functions [BWS93]. As such, we might expect a lower level of parameterization in reused FORTRAN modules. Another reason for the difference from Selby's study may be that his measure of a module's interface is a sum of counts of the parameters and global references in the module. In the FORTRAN modules that he examined, this sum is likely to be dominated by the count of global references; as such, the variation in the count of subprogram parameters among the classes of reuse can not be observed.

Table 4 shows the profile of the reused components over time, as the projects are listed in chronological order of their development start date. We see an increasing complexity (expressed both in terms of module size and external dependencies) in the reused components. Also, we see a rise in the number of parameters per subprogram in the verbatim units, suggesting an increasing generality among them. Low level utility functions were the first to be reused, but as the organization gained reuse experience, more and more complex units were reused as well. Thus while utility functions may be among the best components to initially stock a repository, a reuse process is not limited to them. As an organization gains experience, more and more complex units, at higher levels of the application hierarchy may be reused.

5.2 Error Density

Table 5 shows the error and defect densities (errors/defect per thousand source statements) observed in each of the four classes of component origin. We use *error* to refer to a change report in which the reason for the change was attributed to an error correction. A change report can list several components as requiring correction due to a single error. We refer each instance of a component requiring modification due to an error as a *defect.*

Component Origin	No. Comp.	KSTMT	Defect Density	Error Density	S/A Err. Density
New	1095	44.2	24.8	13.0	8.4
Extensively Modified	152	8.8	19.5	14.0	8.9
Slightly Modified	517	21.6	10.5	7.4	2.5
Reused Verbatim	1495	46.6	2.1	1.2	0.7
All Components	3259	121.2	13.1	7.6	4.4

Table 5: Error densities in each class of component origin

As such, there can be several defects associated with a single error. Two measures of error density are shown–the first includes all errors from unit test through acceptance test, while the second only includes those detected in system and acceptance test. The first measure can provide an indication of the total amount of rework, while the second shows the amount that is occurring late in the development life-cycle. The measure of defect density shown in the table includes defects from unit through acceptance test.

We used a non-parametric test to obtain a statistical comparison of component error density by class of component origin. This comparison shows a significantly lower error density among the reused verbatim components compared to each of the other classes. Similarly, there is a significant difference between the slightly modified components, and the new and extensively modified components. No significant difference was observed between new and extensively modified components.

In terms of error density, reuse via extensive modification appears to yield no advantage over new code development. There is a benefit from reuse in terms of reduced error density when the reuse is verbatim or via slight modification. However, reuse through slight modification only shows about a 50 percent reduction in total error density, while verbatim reuse results in more than a 90 percent reduction. When we only look at the errors that are encountered during the system and acceptance test phases, we still see a greater than 90 percent reduction in defect density in the reused verbatim class (0.7 errors per KSLOC, compared to 8.4 errors per KSLOC in the new components). The slightly modified components, with 2.5 errors per KSLOC, show a reduction of nearly 70 percent compared to the new components, with 8.4 errors per KSLOC. Verbatim reuse clearly provides the most significant benefit to the development process in terms of reducing error density, but reuse via slight modification also provides a substantial improvement, one which is even more noticeable in the test phases.

A number of studies have found higher defect/error densities in smaller components than in larger components [BP84, SYTP85, LV89, MP93]. As shown in table 6, our data supports their findings. Small components (25 or less statements) have defect density more than twice that of the larger components (more than 25 statements), and this difference is highly significant. The only class of reuse where we saw no significant difference was the reused verbatim components, as they have the same defect density regardless of size. The defect density in the small components was more than twice that of the larger components in the new and extensively modified classes, and nearly four times greater in the slightly modified

Component Origin	Small		Large	
	No. Comp.	Def. Dens.	No. Comp.	Def. Dens.
New	638	49.8	457	19.8
Extensively Modified	67	35.7	85	17.7
Slightly Modified	283	26.5	234	7.4
Reused Verbatim	952	2.3	543	2.0
All Components	1940	22.6	1319	10.9

Table 6: Relationship of defect density and component size

class. One explanation for higher error density in the small components is that a system composed of small components will have more interfaces than a system composed of large components; and interfaces are frequently noted as a major source of error in development.

5.3 Error Isolation/Completion Difficulty

Basili and Perricone, in their study of a FORTRAN development project, reported that modified components typically required more correction effort than new components [BP84]. We see a similar result in the two classes of modified components, and also see the same pattern occurring in the reused verbatim components. Table 7 shows the percentage of errors in each class of reuse that were categorized as difficult to isolate or difficult to complete (defined as more than one day to isolate or complete, resp.), and the relative rework effort, a crude approximation of relative effort (staff-hours per KSTMT) in isolating and correcting these errors. In terms of effort to isolate, we see little difference among the classes of component origin. Newly created components had the smallest percentage of difficult-to-isolate errors, but it was not significantly different from any of the classes of reused components. This result is not surprising, as the isolation activity is associated more with understanding the intended functions rather than with their implementation. As such, the origin of the components may not have as great an impact on isolation effort as it will have on completion effort.

We do see an increase in the effort to complete an error in reused components relative to new components. The new components had the lowest percentage of errors requiring more than 1 day to complete a change and the reused verbatim components had the highest percentage, while the modified components fell in between. The difference between the new and the reused verbatim components is significant at the 0.05 level. One explanation for this effect is that the developers have a greater familiarity with the newly created components, so less time is needed to understand the components that must be changed. Another explanation is that the majority of the "easy" errors had previously been removed from the reused component, leaving only the more difficult ones.

To determine whether the increased error correction cost in the reused components outweighs benefit of their having fewer errors, we computed a rough measure of the amount of error rework expended in each class. Unfortunately, our data for effort spent in error

Component Origin	KSTMT	No. Errors.	Pct. Diff. Isolation	Pct. Diff. Completion	Rel. Rework Effort
New	44.2	574	12.4	10.1	118.3
Extensively Modified	8.8	124	14.5	17.7	157.4
Slightly Modified	21.6	160	13.8	13.1	76.8
Reused Verbatim	46.6	58	14.3	22.4	14.7
All Components	121.2	916	13.2	12.6	73.9

Table 7: Difficulty in error isolation/correction

correction and isolation is categorical, so we approximated the true effort simply by the midpoint of the category (μ). Rework was then computed as the sum of this approximation over all errors. Our relative rework measure (RR) was computed by dividing rework by the number of statements (S), i.e.:

$$RR = \frac{\sum_{i=1}^{n} \mu(e_i)}{S}.$$

Again, we used a non-parametric test to determine whether there is a significant difference in the relative rework effort among the four classes of component origin. The tests found a significant difference among the classes with one exception. When comparing the extensively modified components and the new components we found the level of significance to be only 0.18. There may be an increase in the rework cost of extensively modified components, however, our data does not confirm this. In any event, it is not clear whether such an increase in rework cost would be offset by the expected benefit of reduced component creation cost.

For all other pairs, the result was significant at the 0.01 level. Reuse via slight modification shows a 35 percent reduction in rework cost over newly created components, while verbatim reuse provides an 88 percent reduction. For these modes of reuse, the benefit of fewer errors clearly outweighs the cost of more difficult error correction. This measure of benefit is somewhat conservative, as it does not account for the expected reduction in component creation cost, or for the impact of errors as “obstacles” in the development process (e.g., the cost of delays due to effort spent correcting errors). As such, we expect these modes of reuse to yield an even greater improvement over new development. This shows that there is a shift in costs of reuse compared to traditional development, with the reuse-oriented development showing less development effort and fewer, but more costly, errors.

5.4 Source of Errors

Understanding the activity in which the error is introduced allows for corrective action to be applied at the appropriate time. Table 8 shows, for each class of component origin, the percentage of errors from each error source (when the error was introduced). Across all

Component Origin	Rqmts. or Fun. Spec.	Design	Code	Previous Change	Any Error
New	7.3	16.8	68.1	7.8	100
Extensively Modified	5.6	20.2	59.7	14.5	100
Slightly Modified	4.4	26.9	60.1	10.6	100
Reused Verbatim	3.4	3.4	74.1	19.0	100
All Components	5.7	18.2	66.1	10.0	100

Table 8: Percentage of errors in each class of error source by class of reuse

classes, coding errors are the most common error; however, errors associated with requirements, functional specification and design occur at a slightly higher rate in new components than in reused components. The Basili-Perricone study reported the opposite effect of reuse on the specification errors [BP84]. They found that modified modules had a higher proportion of specification errors than did the new modules, and explained the result by suggesting that the specification was not well-enough or appropriately defined to be used in different contexts. A similar result was reported by Endres [End75]. A difference from the environments examined in those studies is that reuse has been well planned for in this environment. The organization is not structured as a pure "component factory" as described in section 3, but it is moving in that direction. As such, the architecture, design and specifications have improved in this environment to better allow and encourage reuse. This result suggests that the reused functionality is more likely to be well specified. This is not surprising, since the reused components have been specified previously, with the expectation that they would be reused. As such, any specification errors are more likely to affect new components rather than reused components. The result also indicates that reuse, whether formal or informal, is occurring in this environment at a higher level than simply code.

A second item of interest is the increased percentage of design errors in the modified components. This suggests that there is increased difficulty in designing an adaptation of an existing component to a new role. This is more difficult because the reuser must be concerned with two pieces of information: the intended function and the existing function. In creating a new component, one only needs to be concerned with the intended function. A misunderstanding of the existing function can result in an error, and that error is likely to be attributed to the design.

5.5 Time of Error Detection

Errors detected late in the development life-cycle can have a much greater cost than those detected early. Table 9 shows, by class of component origin, the percentage of all errors and the more difficult errors that escape unit test. Across all errors, we see little difference between the classes of new, extensively modified, and reused verbatim components, as nearly two thirds of the errors in these classes escaped unit test. This is significantly higher than what we observed in the slightly modified components, where only 43 percent escaped unit

Component Origin	Pct. All Errors.	Pct. Diff. Isolation	Pct. Diff. Completion
New	69	86	80
Extensively Modified	66	81	87
Slightly Modified	43	74	58
Reused Verbatim	62	100	100
All Components	64	84	78

Table 9: Percentage of errors that escape unit test

Component Origin	Error of Omission	Both	Error of Comission	Any
New	35.4	28.6	36.0	100
Extensively Modified	40.3	29.4	30.3	100
Slightly Modified	39.6	20.8	39.6	100
Reused Verbatim	26.3	26.3	47.3	100
All Components	36.2	27.2	36.6	100

Table 10: Percentage of errors of omission and commission

test.

Of the difficult isolation errors (those taking more than one day to isolate), there is not much difference among the classes–a relative high percentage of these errors escape in all classes. However, again, the slightly modified components do show the lowest percentage. There is a significant reduction in the slightly modified class in the percentage of difficult-to-complete errors that escape unit test, as only 58 percent of these errors escape unit test, compared to 80 to 100 percent in the other classes. This suggests that the verification process is more effective in eliminating the difficult errors for the slightly modified components than for other modes of component creation.

5.6 Nature of the Errors

Table 10 shows the percentage of errors that were classified as one of omission, commission, or both. An error associated with a component that was reused verbatim is more likely to be error of commission, and less likely to be one of omission. This suggests that the reused component was typically complete, i.e., it contained the necessary functionality, but at times was in error.

Extensively modified components are more likely to have errors of omission than errors of commission. This may be an indication of the greater complexity of these components. Another possible explanation is that in the development of these components, the intended

Component Origin	Procedural	Interface	Data	All
New	41.2	14.1	44.6	100
Extensively Modified	47.6	17.7	34.7	100
Slightly Modified	31.8	31.2	36.9	100
Reused Verbatim	48.2	12.1	39.7	100
All Components	40.9	17.5	41.6	100

Table 11: Percent of errors of each type by class of component origin

function was not so clear, resulting in necessary parts being omitted. Additional review of the completeness of the design of these components may be a means for removing these errors at an earlier stage.

New and extensively modified components have a higher rate of errors that are classified as both omission and commission than do the slightly modified or reused verbatim components. This may be due to the nature of new development–it is more likely to result in a complex error.

5.7 Type of Errors

Table 11 shows the percentage of errors that were classified in each of the three classes: procedural, interface, and data. Procedural errors are those that were classified as either a computational or a logic error, interface errors are those that were classified as either an internal or external interface error, and data errors are those that were classified as either an initialization or a data value error.

We see a significant difference in the distribution of error types in the slightly modified components, as they have a much higher frequency of interface errors than any other class. This suggests that the nature of the modifications is likely to be associated with the interface. We also see that the new components are more likely to have data errors than the reused components. Basili and Perricone found the opposite effect, namely, that the modified components had a greater percentage of data errors than did the new components. These results suggest that a different approach has been taken toward reuse. In the FORTRAN project studied by Basili and Perricone, the approach may have been to tailor data values and initialization to adapt the component to the new application. The approach taken in the Ada environment is to create generalized modules that can be parameterized to create instances suitable for the new application. As such, one might expect fewer data errors in reused components in the Ada environment.

6 Conclusions

In this analysis we observed clear benefits from reuse–for example, reduced error density. We found that verbatim reuse provides a substantial improvement in error density (more than a 90 percent reduction) compared to new development. The other modes of reuse did not approach this level of improvement. Reuse via slight modification offered a 50 percent reduction in error density compared to new development, but the improvement with this mode of reuse was greater in errors detected late in development (a 70 percent reduction).

We observed a shift in costs of reuse-oriented development, with the reuse offering fewer, but more difficult errors. The effect of increased difficulty in error correction was apparent across the three modes of reuse, although it was less evident in the slightly modified components. In both the verbatim and slightly modified classes of reuse, the relative amount of rework was less than in new code. This suggests that while there is a cost of increased correction effort per error associated with such reuse, the cost is outweighed by the benefit of the reduced number of errors. Coupled with the reduction in development effort, these modes of reuse appear to offer a substantial benefit to development.

Reuse via extensive modification does not provide the reduction in error density that the other modes of reuse yield, and it also results in errors that typically were more difficult to isolate and correct than the errors in newly developed code. In terms of the rework due to the errors in these components, it appears that this mode of development is more costly than new development. However, extensive modification may offer savings in development effort that outweigh the increased cost of rework. This remains an issue for further study.

A different profile of errors was observed for different modes of reuse. For example, a greater percentage of design errors were observed in the modified components. The observed increase in design errors may be due to errors in the additional activities of understanding the function and implementation of the component to be modified, as well as due to the fact that less code was being written. Such information can be used to help in selecting appropriate verification methods for projects where there is significant reuse via modification. One may want to increase the effort in design reviews on such projects, while on projects dominated by new development, code reviews may receive more emphasis. This finding also suggests that one might want to investigate techniques to better describe the components stored in the experience base so that the likelihood of a misunderstanding of the function and implementation is lessened.

The experience with reuse in an organization and the approach taken toward reuse are likely to influence the nature of errors observed in the organization. In this study of an organization well experienced with reuse, we observe a number of effects that differed with findings from other studies of environments where reuse was not planned for to such an extent. The reused components appear to be simpler, have fewer dependencies, and be more parameterized than new components. However, as this organization gained reuse experience, the distinction became less apparent–more and more complex components, at higher levels in the application hierarchy were reused. As an organization moves toward a reuse-oriented development approach, it must evolve its practices to accommodate the new effects of reuse. In the context of the QIP, error analysis can be a useful mechanism to provide insight into the benefits and difficulties of reuse in software development.

References

[AE92] W. W. Agresti and W. M. Evanco. Projecting software defects from analyzing Ada designs. *IEEE Transactions on Software Engineering*, 18(11), November 1992.

[AM87] W. Agresti and F. McGarry. The Minnowbrook workshop on software reuse: A summary report. In W. Tracz, editor, *Software Reuse: Emerging Technology*. IEEE Computer Society Press, 1987.

[Bas85] V. R. Basili. Quantitative Evaluation of Software Methodology. In *Proceedings of the First Pan Pacific Computer Conference*, Australia, July 1985.

[BBH93] L. C. Briand, V.R. Basili, and C. J. Hetmanski. Developing interpretable models with optimized set reduction for identifying high-risk software components. *IEEE Transaction on Software Engineering*, 19(11), November 1993.

[BCC92] V. R. Basili, G. Caldiera, and G. Cantone. A Reference Architecture for the Component Factory. *ACM Transactions on Software Engineering and Methodology*, 1(1), January 1992.

[BP84] V. R. Basili and B. T. Perricone. Software errors and complexity: An empirical investigation. *Communications of the ACM*, 27(1), January 1984.

[BP88] B. W. Boehm and P. N. Papaccio. Understanding and Controlling Software Costs. *IEEE Transactions on Software Engineering*, 14(10), October 1988.

[BR88] V. Basili and D. Rombach. The TAME Project: Towards Improvement–Oriented Software Environments. *IEEE Transactions on Software Engineering*, 14(6), June 1988.

[BR91] V. R. Basili and H. D. Rombach. Support for Comprehensive Reuse. *Software Engineering Journal*, 6(5), September 1991.

[Bro87] F. P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4), April 1987.

[BS87] V. R. Basili and R. W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, 13(12), December 1987.

[BTH93] L. C. Briand, W. M. Thomas, and C. J. Hetmanski. Modeling and managing risk early in software development. In *Proceedings of the Fifteenth International Conference on Software Engineering*, May 1993.

[BWS93] J. Bailey, S. Waligora, and M. Stark. Impact of Ada in the flight dynamics division: Excitement and frustration. In *Proceedings of the 18th Annual Software Engineering Workshop*. NASA/GSFC, December 1993.

[CB91] G. Caldiera and V. R. Basili. Identifying and Qualifying Reusable Software Components. *IEEE Computer*, 24(2), February 1991.

[DK93] M. Dunn and J. Knight. Automating the detection of reusable parts in existing software. In *Proceedings of the 15th International Conference on Software Engineering*, Baltimore, Maryland, May 1993.

[Dou87] D. Doubleday. ASAP: Ada Static Analyzer Program. Technical Report CS–TR–1897, University of Maryland, May 1987.

[EG90] N. Ebel and C. Genillard. The reusability of Ada software components. In R. Gautier and P. Wallis, editors, *Software Reuse with Ada*. Peter Peregrinus Ltd., 1990.

[End75] A. Endres. An analysis of errors and their causes in system programs. In *Proceedings of the International Conference on Software Engineering*, April 1975.

[GP87] A. Gargaro and T. Pappas. Reusability issues and Ada. *IEEE Software*, July 1987.

[Ich85] J. Ichbiah. *The Rationale for the Ada Programming Language*. Cambridge University Press, 1985.

[Jon84] T. C. Jones. Reusability in programming: A survey of the state of the art. *IEEE Transactions on Software Engineering*, SE–10(5), September 1984.

[Kes90] R. Kester. SEL Ada Reuse Analysis and Representations. In *Proceedings of the 15th Annual GSFC Software Engineering Workshop*. NASA/GSFC, November 1990.

[LG84] R. Lanergan and C. Grasso. Software Engineering with Reusable Designs and Code. *IEEE Transactions on Software Engineering*, SE–10(5), September 1984.

[LV89] R. Lind and K. Vairavan. An experimental investigation of software metrics and their relationship to software development effort. *IEEE Transactions on Software Engineering*, 15(5), May 1989.

[MK92] J. Munson and T. Khoshgoftaar. The detection of fault-prone programs. *IEEE Transactions on Software Engineering*, 18(5), May 1992.

[MP93] K. Möller and D. Paulish. An empirical investigation of software fault distribution. In *Proceedings of the First International Software Metrics Symposium*, Baltimore, Maryland, May 1993.

[Rom91] H. D. Rombach. Software Reuse: A Key to the Maintenance Problem. *Information and Software Technology*, 33(1), January/February 1991.

[Sel88] R. Selby. Empirically analyzing software reuse in a production environment. In W. Tracz, editor, *Software Reuse: Emerging Technology*. IEEE Computer Society Press, 1988.

[SEL94] An Overview of the Software Enginnering Laboratory. Technical Report SEL-94-005, Software Engineering Laboratory, NASA Goddard Space Flight Center, December 1994.

[SP88] R.W. Selby and A.A. Porter. Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis. *IEEE Transactions on Software Engineering*, 14(11), November 1988.

[Sta93] M. Stark. Impacts of object-oriented technologies: Seven years of SEL studies. In *Proceedings of Eigth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, October 1993.

[SYTP85] V. Shen, T. Yu, S. Thebaut, and L. Paulsen. Identifying error-prone software–an empirical study. *IEEE Transactions on Software Engineering*, SE-11(4), April 1985.

[TDB92] W. M. Thomas, A. Delis, and V. R. Basili. An evaluation of Ada source code reuse. In J. van Katwijk, editor, *Ada: Moving Towards 2000 (Proceedings of the Ada–Europe International Conference)*, Zandvoort, The Netherlands, June 1992. Springer-Verlag.

[WCW85] A. Wolf, L. Clarke, and J. Wileden. Ada-based support for programmming in the large. *IEEE Software*, March 1985.

Page intentionally left blank

A VALIDATION OF OBJECT-ORIENTED DESIGN METRICS

Victor R. Basili*, Lionel Briand and Walcélio L. Melo***

***University of Maryland**
Dep. of Computer Sciences
Institute for Advanced Computer Studies
College Park, MD, 20770 USA
{basili|melo}@cs.umd.edu

****CRIM**
1801 McGill College Av.
Montréal (Quebec)
H3A 2N4, Canada
lbriand@crim.ca

Abstract

This paper presents the results of a study conducted at the University of Maryland in which we experimentally investigated the suite of Object-Oriented (OO) design metrics introduced by [Chidamber&Kemerer, 1994]. In order to do this, we assessed these metrics as predictors of fault-prone classes. This study is complementary to [Lie&Henry, 1993] where the same suite of metrics had been used to assess frequencies of maintenance changes to classes. To perform our validation accurately, we collected data on the development of eight medium-sized information management systems based on identical requirements. All eight projects were developed using a sequential life cycle model, a well-known OO analysis/design method and the C++ programming language. Based on experimental results, the advantages and drawbacks of these OO metrics are discussed and suggestions for improvement are provided. Several of Chidamber&Kemerer's OO metrics appear to be adequate to predict class fault-proneness during the early phases of the life-cycle. We also showed that they are, on our data set, better predictors than "traditional" code metrics, which can only be collected at a later phase of the software development processes.

Key-words: Object-Oriented Design Metrics; Error Prediction Model; Object-Oriented Software Development; C++ Programming Language.

During this work the authors were, in part, supported by the following grants: Basili by NASA grant NSG-5123, Briand by the IGLOO project and NFS grant 01-5-24845, and Melo by a Westinghouse grant and UMIACS.

Published as a Technical Report, Univ. of Maryland, Dep. of Computer Science, College Park, MD, 20742. April 1995. CS-TR-3443.

1. Introduction

1.1 Motivation

The development of a large software system is a time- and resource-consuming activity. Even with the increasing automation of software development activities, resources are still scarce. Therefore, we need to be able to provide accurate information and guidelines to managers to help them make decisions, plan and schedule activities, and allocate resources for the different software activities that take place during software evolution. Software metrics are thus necessary to identify where the resource issues are; they are a crucial source of information for decision-making [Harrison, 1994].

Testing of large systems is an example of a resource- and time-consuming activity. Applying equal testing and verification effort to all parts of a software system has become cost-prohibitive. Therefore, one needs to be able to identify fault-prone classes so that testing/verification effort can be concentrated on these classes [Harrison, 1988]. The availability of adequate product design metrics for characterizing error-prone classes is thus vital.

Dozens of product metrics have been proposed [Fenton, 1991], used, and, sometimes, experimentally validated in academia [Basili&Hutchens, 1982] and industry, e.g., number of lines of code, MacCabe complexity metric, etc. In fact, many companies have built their own cost, quality and resource prediction models based on product metrics. TRW [Boehm, 1981], the Software Engineering Laboratory (SEL) [McGarry *et. al.*, 1994] and Hewlett Packard [Grady, 1994] are examples of software organizations that have been using product metrics to build their cost, resource, defect, and productivity models.

1.2 Issues

In the last decade, many companies have started to introduce Object-Oriented (OO) technology into their software development environments. OO analysis/design methods, OO languages, and OO development environments are currently popular worldwide in both small and large software organizations. The insertion of OO technology in the software industry, however, has created new challenges for companies which use product metrics as a tool for monitoring, controlling and improving the way they develop and maintain software. Therefore, metrics which reflect the specificities of the OO paradigm must be defined and validated in order to be used in industry. Some studies have concluded that "traditional" product metrics are not sufficient for characterizing, assessing and predicting the quality of OO software systems. For example, based on a study at Texas Instruments, [Brooks, 1993] has reported that McCabe cyclomatic complexity appeared to be an inadequate metric for use in software development based on OO technology.

To address this issue, OO metrics have recently been proposed in the literature [Abreu&Carapuça, 1994; Chidamber&Kemerer, 1994]. However, most of them have not undergone a thorough and comprehensive experimental validation. [Briand *et.al.*, 1994] and [Lie&Henry, 1993] are rare exceptions in this respect. The work described in this paper is an additional step toward a thorough experimental validation of the OO metric suite defined in [Chidamber&Kemerer, 1994]. This paper presents the results of a study conducted at the University of Maryland in which we performed an experimental validation of that suite of OO metrics with regard to their ability to identify fault-prone classes. Data were collected during the development of eight medium-sized management information systems based on identical requirements. All eight projects were developed using a sequential life cycle model, a well-known Object-Oriented analysis/design method [Rumbaugh et al, 1991], and the C++ programming language [Stroustrup, 1991]. In fact, we used an experiment framework that should be representative of currently used technology in industrial settings. This study discusses the strengths and weaknesses of the validated OO metrics with respect to predicting faults across classes.

1.3. Outline

This paper is organized as follows. Section 2 presents the suite of OO metrics proposed by Chidamber&Kemerer (1994), and the methodology we used for experimental validation. Section 3 presents the data collected together with the statistical analysis of the data. Section 4 compares our study with other works on the subject. Finally, section 5 concludes the paper by presenting lessons learned and future work.

2. Description of the Study

2.1. Experiment goal

The goal of this study was to analyze experimentally the OO design metrics proposed in [Chidamber&Kemerer, 1994] for the purpose of evaluating whether or not these metrics are suitable for predicting the probability of detecting faulty classes. From [Chidamber&Kemerer, 1994], [Chidamber&Kemerer, 1995] and [Churcher&Shepperd, 1995], it is clear that the definitions of these metrics are not language independent. As a consequence, we had to slightly adjust some of Chidamber&Kemerer's metrics in order to reflect the specificities of C++. These metrics are as follows:

- Weighted Methods per Class (WMC). WMC measures the complexity of an individual class. Based on [Chidamber&Kemerer, 1994], if we consider all methods of a class to be equally complex, then WMC is simply the number of methods defined in each class. In this study, we adopted this approach for the sake of simplicity and because the choice of a complexity metric would be somewhat arbitrary since it is not fully specified in the metric suite. Thus, WMC is defined as being the number of all member functions and operators defined in each class. However, "friend" operators (C++ specific construct) are not counted. Member functions and operators inherited from the ancestors of a class are also not counted. This definition is identical the one described in [Chidamber&Kemerer, 1995]. The assumption behind this metric is that a class with significantly more member functions than its peers is more complex, and by consequence tends to be more fault-prone.

  Churcher&Shepperd (1995) have argued that WMC can be measured in different ways depending on how member functions and operations defined in a C++ class are counted. We believe that the different counting rules proposed by [Churcher&Shepperd, 1995] correspond to different metrics, similar to the WMC metric, and which must be experimentally validated as well. A validation of Churcher&Shepperd's WMC-like metrics is, however, beyond the scope of this paper.

- Depth of Inheritance Tree of a class (DIT) – DIT is defined as the maximum depth of the inheritance graph of each class. C++ allows multiple inheritance and therefore classes can be organized into a directed acyclic graph instead of trees. DIT, in our case, measures the number of ancestors of a class. The assumption behind this metric is that well-designed OO systems are those structured as forests of classes, rather than as one very large inheritance lattice. In other words, a class located deeper in a class inheritance lattice is supposed to be more fault-prone because the class inherits a large number of definitions from its ancestors.

- Number Of Children of a Class (NOC) – This is the number of direct descendants for each class. Classes with large number of children are difficult to modify and usually require more testing because the class potentially affects all of its children. Thus, a class with numerous children has to provide services in a larger number of contexts and must be more flexible. We expect this to introduce more complexity into the class design.

- Coupling Between Object classes (CBO) – A class is coupled to another one if it uses its member functions and/or instance variables. CBO provides the number of classes to which a given class is coupled. The assumption behind this metric is that highly coupled classes are more fault-prone than weakly coupled classes. So coupling between classes should be identified in order to concentrate testing and/or inspections on such classes.

- Response For a Class (RFC) – This is the number of methods that can potentially be executed in response to a message received by an object of that class. In our study, RFC is the number of functions directly invoked by member functions or operators of a class. The assumption here is that the larger the response set of a class, the higher the complexity of the class, and the more fault-prone and difficult to modify.

- Lack of Cohesion on Methods (LCOM) – This is the number of pairs of member functions without shared instance variables, minus the number of pairs of member functions with shared instance variables. However, the metric is set to 0 whenever the above subtraction is negative. A class with low cohesion among its methods suggests an inappropriate design, (i.e., the encapsulation of unrelated program objects and member functions that should not be together), which is likely to be fault-prone.

Readers acquainted with C++ can see that many particularities of C++ are not taken into account by Chidamber&Kemerer's metrics, e.g., C++ templates, friend classes, etc. In fact, additional work is necessary in order to extend the proposed OO metric set with metrics specifically tailored to C++.

2.2 Experimental framework

In order to experimentally validate the OO metrics proposed in [Chidamber&Kemerer, 1994] with regard to their capabilities to predict fault probability, we ran a controlled study over four months (from September to December, 1994). The population under study was a graduate level class offered by the Department of Computer Science at the University of Maryland. The students were not required to have previous experience or training in the application domain or OO methods. All students had some experience with C or C++ programming and relational databases and therefore had the basic skills necessary for such an experiment.

The students were randomly grouped into 8 teams. Each team developed a medium-sized management information system that supports the rental/return process of a hypothetical video rental business, and maintains customer and video databases.

The development process was performed according to a sequential software engineering life-cycle model derived from the Waterfall model. This model includes the following phases: Analysis, Design, Implementation, Testing, and Repair. At the end of each phase, a document was delivered: Analysis document, design document, code, error report, and finally, modified code, respectively.

Requirement specifications and design documents were checked in order to verify that they matched the system requirements. Errors found in these first two phases were reported to the students. This maximized the chances that the implementation began with a correct OO analysis/design. The testing phase was accomplished by an independent group composed of experienced software professionals. This group tested all systems according to similar test plans and using functional testing techniques. During the repair phase, the students were asked to correct their system based on the errors found by the independent test group.

OMT, an OO Analysis/Design method, was used during the analysis and design phases [Rumbaugh *et. al.*, 1991]. The C++ programming language, the GNU software development environment, and OSF/MOTIF were used during the implementation. Sparc Sun stations were used as the implementation platform. Therefore, the development environment and technology we used are representative of what is currently used in industry and academia.

The following libraries were provided to the students:

a) *MotifApp.* This public domain library provides a set of C++ classes on top of OSF/MOTIF for manipulation of windows, dialogs, menus, etc. [Young, 1992]. The MotifApp library provides a way to use the OSF/Motif widgets in an OO programming/design style.

b) *GNU library.* This public domain library is provided in the GNU C++ programming environment. It contains functions for manipulation of string, files, lists, etc.

c) *C++ database library.* This library provides a C++ implementation of multi-indexed B-Trees.

No special training was provided for the students in order to teach them how to use these libraries. However, a tutorial describing how to implement OSF/Motif applications was given to the students. In addition, a C++ programmer, familiar with OSF/Motif applications, was available to answer questions about the use of OSF/Motif widgets and the libraries. A hundred small programs exemplifying how to use OSF/Motif widgets were also provided. Finally, the code sources and the complete documentation of the libraries were made available. It is important to note that the students were not required to use the libraries and, depending on the particular design they adopted, different reuse choices were expected.

We also provided a specific domain application library in order to make our experiment more representative of the "real world". This library implemented the graphical user interface for insertion/removal of customers and was implemented in such a way that the main resources of the OSF/Motif widgets and MotifApp library were used. Therefore, this library contained a small part of the implementation required for the development of the rental system.

2.3. Data Collection

We collected: (1) the source code of the C++ programs delivered at the end of the implementation phase, (2) data about these programs, (3) data about errors found during the testing phase and fixes during the repair phase, and (4) the repaired source code of the C++ programs delivered at the end of the life cycle. GEN++ [Devanbu, 1992] was used to extract Chidamber&Kemerer's OO design metrics directly from the source code of the programs delivered at the end of the implementation phase. To collect items (2) and (3) , we used the following forms, which have been tailored from those used by the Software Engineering Laboratory [Heller et. al, 1992]:

- Defect Report Form.
- Component Origination Form.

In the following sections, we comment on the purpose of the Component Origination and Defect Report forms used in our experiment and the data they helped collect.

2.3.1 Defect Report Form

This form was used to gather data about (1) the defects found during the testing phase, (2) classes changed to correct such defects, and (3) the effort in correcting them. The latter includes:

- how long it took to determine precisely what change was needed. This includes the effort required for understanding the change or finding the cause of the error, locating where the change was to be made, and determining that all effects of the change were accounted for.
- how much time it took to implement the correction. This includes design changes, code modification, regression testing, and updates to documentation.

2.3.2 Component Origination Form

This form is used to record information that characterizes each class under development in the project at the time it goes into configuration management. Firstly, this form is used to capture whether the class has been developed from scratch or has been developed from a reused class. In the latter case, we collected the amount of modification (none, small or large) that was needed to meet the system requirements and design as well as the name of the reused class. By small/large, we mean that less/more than 25% of the original code had been modified, respectively. However, this kind of data was difficult to obtain because we do not have appropriate tools to collect this data automatically. As a simplification, we asked the developers to tell us if more or less than 25% of a class had been changed. In the former case, the class was labeled: *Extensively modified* and in the latter case: *Slightly modified.* Classes reused without modification were labeled: *verbatim reused.*

In addition, the name of the sub-system to which the class belonged was also collected. In our study, we had three types of sub-systems: graphical user interface (GUI), textual user interface (TUI), and database processing (DB).

3. Analysis of Experimental Results

In this section, we will attempt to assess experimentally whether the OO design metrics defined in [Chidamber&Kemerer, 1994] are suitable predictors of fault-prone classes. This will help us assess these metrics as quality indicators and how they compare to common code metrics. Thus, we intend to provide the type of empirical validation that we think is necessary before any attempt to use such metrics as objective and early indicators of quality. Section 3.1 shows the descriptive distributions of the OO metrics in the studied sample whereas Section 3.2 provides the results of univariate and multivariate analyses of the relationships between OO metrics and fault-proneness.

3.1. Analysis of Distributions

Figure 1 shows the distributions of the analyzed OO metrics based on 180 classes present in the studied systems. Table 1 provides common descriptive statistics of the metric distributions. These results indicate that inheritance hierarchies are somewhat flat (DIT) and that classes have, in general, few children (NOC). In addition, most classes show a lack of cohesion (LCOM) near 0. This latter metric does not seem to differentiate classes well and this stems from its definition which prevents any negative measure. This issue will be discussed further in Section 3.2.

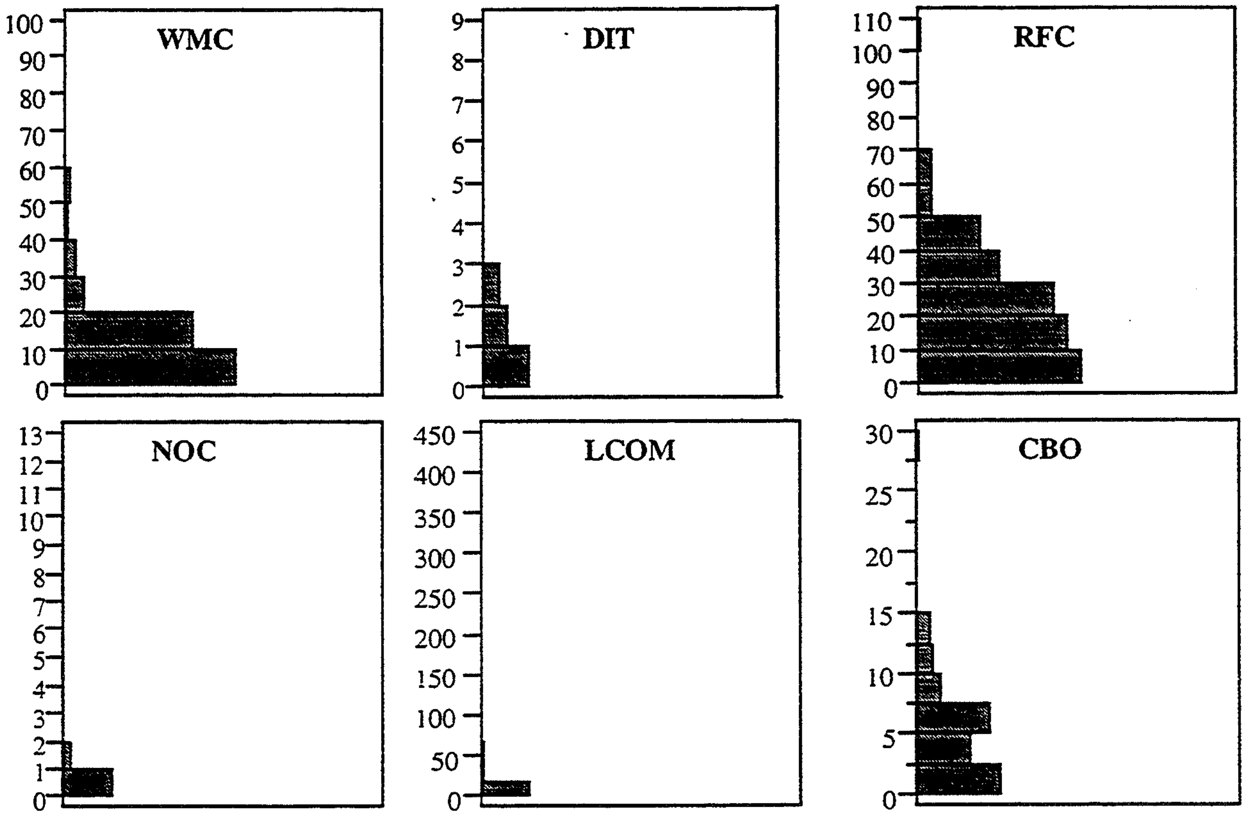


Figure 1: Distribution of the analyzed OO metrics

	WMC	DIT	RFC	NOC	LCOM	CBO
maximum	99.000	9.0000	105.00	13.000	426.00	30.00
minimum	1.0000	0.0000	0.0000	0.0000	0.0000	0.000
median	9.5000	0.0000	19.5000	0.0000	0.0000	5.000
Mean	13.3897	1.3179	33.9141	0.2308	9.7077	6.7962
Std Dev	14.9052	1.9896	33.3703	1.5377	63.7766	7.5614

Table 1: Descriptive statistics of the analyzed OO metrics.

Descriptive statistics will be useful to help us interpret the results of the analysis in the remainder of this section. In addition, they will facilitate comparisons of results from future similar studies.

3.2 The Relationships between Fault Probability and OO Metrics

3.2.1 Analysis Methodology

The response variable we use to validate the OO design metrics is binary, i.e., was a fault detected in a class during testing phases? We used logistic regression to analyze the relationship between metrics and the fault-proneness of classes. Logistic regression is a classification technique [Hosmer&Lemeshow, 1989] used in many experimental sciences based on maximum likelihood estimation. In this case, a careful outlier analysis must be performed in order to make sure that the observed trend is not the result of a few observations [Dillon&Goldstein, 1984], even though logistic regression is deemed to be more robust for outliers than least-square regression.

In particular, we first used univariate logistic regression, to evaluate the relationship of each of the metrics in isolation and fault-proneness. Then, we performed multivariate logistic regression, to evaluate the predictive capability of those metrics that had been assessed sufficiently significant in the univariate analysis (e.g., $\alpha < 0.10$ is a reasonable heuristic). This modeling process is further described in [Hosmer&Lemeshow, 1989].

A multivariate logistic regression model is based on the following relationship equation (the univariate logistic regression model is a special case of this, where only one variable appears):

$$\log(\frac{p}{1 - p}) = C_0 + C_1X_1 + C_2X_2 + \cdots + C_nX_n \qquad (1)$$

where p is the probability that a fault will be found in a class during the validation phase, and the X_i's are the OO metrics included as predictors in the model (called *covariates* of the logistic regression equation). In the two extreme cases, i.e., when a variable is either non-significant or entirely differentiates fault-prone classes, the curve (between p and any single X_i, i.e., assuming that all other X_j's are constant) approximates a horizontal line and a vertical line respectively. In between, the curve takes a flexible S shape. However, since p is unknown, the coefficients C_i will be estimated through a likelihood function optimization [Hosmer&Lemeshow, 1989]. This procedure assumes that all observations are statistically independent. When building the regression equations, each observation was weighted according to the number of faults detected in each class. The rationale is that each detection of a fault is considered as an independent event: Classes where no faults were detected were weighted 1.

Tables 2 and 3 contain the results we obtained through, respectively, univariate and multivariate logistic regression on all of the 180 classes. We report those related to the metrics that turned out to be the most significant across all eight development projects. For each metric, we provide the following statistics:

- Coefficient (appearing in Tables 2 and 3), the estimated regression coefficient. The larger the coefficient in absolute value, the stronger the impact of the explanatory variable on the probability p of a fault to be detected in a class.

- $\Delta\psi$ (appearing in Table 2 only), which is based on the notion of odd ratio [Hosmer&Lemeshow, 1989], and provides an evaluation of the impact of the metric on the response variable. More specifically, the odds ratio $\psi(X)$ represents the ratio between the probability of having a fault and the probability of not having a fault when the value of the metric is X. As an example, if, for a given value X, $\psi(X)$ is 2, then it is twice as likely that the class does contain a fault than that it does not contain a fault. The value of $\Delta\psi$ is computed by means of the following formula:

$$\Delta\psi = \frac{\psi(X+1)}{\psi(X)} \qquad (2)$$

Therefore, $\Delta\psi$ represents the reduction/increase in the odd ratio when the value X increases by 1 unit. This provides a more intuitive insight than regression coefficients into the impact of explanatory variables.

- The level of significance (α, appearing in Tables 2 and 3) provides an insight into the accuracy of the coefficient estimates. It tells the reader about the probability of the coefficient being different from zero by chance. Usually, a level of significance of $\alpha = 0.05$ (i.e., 5% probability) is used as a threshold to determine whether an explanatory variable is a significant predictor. However, the choice of a particular level of significance is ultimately a subjective decision and other levels such as $\alpha = 0.01$ or 0.1 are common. Also, the larger the level of significance, the larger the standard deviation of the estimated coefficients, and the less believable the calculated impact of the explanatory variables. The significance test is based on a likelihood ratio test [Hosmer&Lemeshow, 1989] commonly used in the framework of logistic regression.

Based on equation (1), the likelihood function of a data set of size *D* is:

$$L = \prod_{i=1}^{D} \pi(x_i) \qquad (3)$$

where:

$$\pi(x_i) = \frac{e^{(C_o + C_1 \bullet X_{i1} + \ldots + C_n \bullet X_{in}) \bullet Y_i}}{1 + e^{(C_o + C_1 \bullet X_{i1} + \ldots + C_n \bullet X_{in})}} \qquad (4)$$

where Y_i is assigned the value 1 if the class does not contain any fault, 0 otherwise. The n-dimensional vectors X_i contain the OO design metrics characterizing each of the *D* observations. Also, $\pi(X_i)$ represents the estimated probability for a class to contain (or not, depending on which is the case) a fault. The coefficients that will maximize the likelihood function will be the regression coefficient estimates. For mathematical convenience, $l = Ln[L]$, the *log-likelihood*, is usually maximized.

One of the global measure of goodness of fit we will use for logistic regression models is R^2, a statistic defined as:

$$R^2 = \frac{(l_0 - l_n)}{(l_0 - l_s)}$$

where

- l_0 is the log-likelihood function without using any covariate (just the intercept),
- l_n is the log-likelihood of the model including the *n* selected design metrics as covariates,
- l_s is the log-likelihood of the *saturated model*, i.e., where Y_i, (0 or 1) is substituted for each probability $\pi(X_i)$ in *l*. The log-likelihood l_s is the maximum value that can be assigned to *l*.

The higher the R^2, the more accurate the model. However, as opposed to the R^2 of least-square regression, high R^2's are rare for logistic regression because l_n rarely approaches the value of l_s since the computed $\pi(X_i)$'s in l_n rarely approach 1. The interested reader may refer to [Hosmer&Lemeshow, 1989] for a detailed introduction to logistic regression. Finally, R^2 may be described as a measure of the *proportion of total uncertainty* that is attributed to the model fit.

3.2.2 Univariate Analysis

In this section, we analyze the six OO metrics introduced in [Chidamber&Kemerer, 1994] (though slightly adapted to our context) with regard to the probability of fault detection in a class during test phases. In our case, it is equivalent for the logistic model to calculate the probability of a single fault to be detected in a class.

- Weighted Methods per Class (WMC) was shown to be somewhat significant ($\alpha = 0.06$) overall. For new and extensively modified classes and for UI (Graphical and Textual User Interface) classes, the results are much better: $\alpha = 0.0003$ and $\alpha = 0.001$, respectively. As expected, the larger the WMC, the larger the probability of fault detection. These results can be explained by the fact that the internal complexity does not have a strong impact if the class is reused verbatim or with very slight modifications. In that case, the class interface properties will have the most significant impact.

- Depth of Inheritance Tree of a class (DIT) was shown to be very significant ($\alpha = 0.0000$) overall. As expected, the larger the DIT, the larger the probability of defect detection. Again, results improve (Logistic R^2 goes from 0.06 to 0.13) when only new and extensively modified classes are considered.

- Response For a Class (RFC) was shown to be very significant overall ($\alpha = 0.0000$). Predictably, the larger the RFC, the larger the probability of defect detection. However, the logistic R^2 improved significantly for new and extensively modified classes and UI classes (from 0.06 to 0.24 and 0.36, respectively). Reasons are believed to be the same as for WMC for extensively modified classes. In addition, UI classes show a distribution which is significantly different from that of DB classes: the mean and median are significantly higher. This, as a result, may strengthen the impact of RFC when performing the analysis.

- Number Of Children of a Class (NOC) appeared to be very significant (except in the case of UI classes) but the observed trend is contrary to what was expected. The larger the NOC, the lower the probability of defect detection. This surprising trend can be explained by the combined facts that most classes do not have more than one child and that verbatim reused classes are somewhat associated with a large NOC. Since we have observed that reuse was a significant factor in fault density [Melo *et. al.*, 1995], this explains why large NOC classes are less fault-prone. Moreover, there is some instability across class subsets with respect to the impact of NOC on the probability of detecting a fault in a class (see $\Delta\psi$'s in Table 2). This may be explained in part by the lack of variability on this measurement scale (see distributions in Figure 1).

- Lack of Cohesion on Methods (LCOM) was shown to be insignificant in all cases (this is why the results are not shown in Table 2) and this should be expected since the distribution of LCOM shows a lack of variability and a few very large outliers. This stems in part from the definition of LCOM where the metric is set to 0 when the number of class pairs sharing variable instances is larger than that of the ones not sharing any instances. This definition is definitely not appropriate in our case since it sets cohesion to 0 for classes with very different cohesions and keeps us from analyzing the actual impact of cohesion based on our data sample.
- Coupling Between Object classes (CBO) is significant and more particularly so for UI classes ($\alpha = 0.0000$ and $R^2 = 0.17$). No satisfactory explanation could be found for differences in pattern between UI and DB classes.

It is important to remember, when looking at the results in Table 2, that the various metrics have different units. Some of these units represent "big steps" on each respective measurement scale while others represent "smaller steps". As a consequence, some coefficients show a very small impact (i.e., $\Delta\psi$'s) when compared to others. This is not, however, a valid criterion to evaluate the predictive usefulness of such metrics.

Most importantly, besides NOC, all metrics appear to have a very stable impact across various categories of classes (i.e., DB, UI, New-Ext, etc.). This is somewhat encouraging since it tells us that, in that respect, the various types of components are comparable. If we were considering different types of faults separately, results might be different. Such a refinement is, however, part of our future research plans.

Metrics	Coefficient	$\Delta\psi$	α	R^2	Classes
WMC (1)	-0.022	98%	0.0607	0.007	ALL
WMC (2)	-0.086	92%	0.00035	0.024	New-Ext
WMC (3)	-0.027	103%	0.0656	0.0154	DB
WMC (4)	-0.0944	91%	0.0019	0.0467	UI
DIT (1)	-0.485	62%	0.0000	0.0648	ALL
DIT (2)	-0.868	42%	0.0000	0.1314	New-Ext
DIT (3)	-0.475	62%	0.043	0.0187	DB
DIT (4)	-0.29	75%	0.024	0.017	UI
RFC (1)	-0.085	92%	0.0000	0.0648	ALL
RFC (2)	-0.087	92%	0.0000	0.2477	New-Ext
RFC (3)	-0.077	93%	0.0000	0.188	DB
RFC (4)	-0.108	90%	0.0000	0.3624	UI
NOC (1)	3.3848	3000%	0.0000	0.1426	ALL
NOC (2)	3.62	3734%	0.0011	3.6235	New-Ext
NOC (3)	2.05	777%	0.0000	0.0826	DB
CBO (1)	-0.142	87%	0.0000	0.068	ALL
CBO (2)	-0.079	92%	0.017	0.02	New-Ext
CBO (3)	-0.086	92%	0.006	0.034	DB
CBO (4)	-0.284	75%	0.0000	0.17	UI

Table 2: Univariate Analysis - Summary of experimental results.

3.2.3 Multivariate Analysis

The OO design metrics presented in the previous section can be used early in the life cycle to build a predictive model of fault-prone classes. In order to obtain an optimal model, we included these metrics into a multivariate logistic regression model. However, only the metrics that significantly improve the predictive power of the multivariate model were included through a stepwise selection process. Another significant predictor of fault-proneness is the level of reuse of the class (called "origin" in Table 3). This information is available at the end of the design phase when reuse candidates have been identified in available libraries and the required amount of change can be estimated. Table 3 describes the computed multivariate model. Using such a model for classification, the results shown in Table 4 are obtained by using a classification threshold of p(Fault detection) = 0.5 for the probability of detecting a single defect in a given class, i.e., when $p > 0.5$, the class is classified as faulty and otherwise as non-faulty. As expected, classes predicted as faulty contain a large number of faults (250 faults on 48 classes) because those classes tend to show a better classification accuracy.

We now assess the impact of using such a prediction model by assuming, in order to simplify computations, that inspections of classes are 100% effective in finding faults. In that case, 80 classes (predicted as faulty) out of 180 would be inspected and 48 faulty classes out of 58 would be identified before testing. If we now take into account individual faults, 250 faults out of 258 would be detected during inspection. As mentioned above, such a good result stems from the fact that the prediction model is more accurate for multiple-faults classes.

	Coefficient	α
Intercept	3.13	0.0000
DIT	-0.50	0.0004
RFC	-0.11	0.0000
NOC	2.01	0.0178
RFC	-0.13	0.0072
CBO	-0.238	0.0001
Origin	-1.84	0.0000

Table 3: Multivariate Analysis with OO design metrics

Predicted **Actual**	**No fault**	**Fault**
No Fault	90	32
Fault	10 (18)	48.(250)

Table 4: Classification Results with OO Design Metrics. The figures before parentheses in the right column are the number of classes classified as faulty. The figures between the parentheses are the faults contained in those classes.

In order to evaluate the predictive accuracy of these OO design metrics, it would be interesting to compare their predictive capability with the one of the usual code metrics, that can only be obtained later in the development life cycle. Three code metrics, among the ones provided by the Amadeus tool [Amadeus, 1994], were selected through a stepwise regression procedure. Table 5 shows the resulting parameter estimations of the multivariate logistic regression model where: *MaxStatNext* is the maximum level of statement nesting in a class, *FunctDef* is the number of function declarations, and *FunctCall* is the number of function calls. However, based on the whole set of metrics provided by Amadeus, other multivariate models yield results of similar accuracy. This model happens to be, however, the model resulting from the use of a standard, stepwise logistic regression analysis procedure.

	Coefficient	α
Intercept	0.39	0.0384
MaxStatNest	-0.286	0.0252
FunctDef	0.166	0.0010
FunctCall	-0.0277	0.0000

Table 5: Multivariate Analysis with Code Metrics

In addition to being collectable only later in the process, code metrics appear to be somewhat poorer as predictors of class fault-proneness (see Table 6). In this case, 112 classes (predicted as faulty) out of 180 would be inspected and 51 faulty classes out of 58 would be detected. If we now take into account individual faults, 231 faults out of 268 would be detected during inspection. Three more faulty classes would be corrected (51 versus 48) but 32 more classes would have to be inspected (112 versus 80). Moreover, the OO design metrics are better predictors of classes containing large numbers of faults since 19 more faults (250 versus 231) would be detected in that case. Therefore, predictions based on code metrics appear to be poorer. Table 7 confirms that result by showing the values of correctness (percentage of classes correctly predicted as faulty) and completeness (percentage of faulty classes detected). Values between parentheses present predictions' correctness and completeness values when classes are weighted according to the number of faults they contain (classes with no fault are weighted 1).

Predicted Actual	No fault	Fault
No Fault	61	61
Fault	7 (37)	51 (231)

Table 6: Classification Results based on code metrics shown in Table 5

Model Accuracy	OO metrics	Code metrics
Completeness	88% (93%)	83% (86%)
Correctness	60% (92%)	45.5% (86%)

Table 7: Classification Accuracies based on OO and code metrics shown in Table 3 and Table 5

4. Related Work

As far as we know, the only studies attempting to experimentally validate OO metrics are [Lie&Henry, 1993] and [Briand et. al., 1994]. In [Briand et. al. ,1994], metrics for measuring abstract data type (ADT) cohesion and coupling are proposed and are experimentally validated as predictors of faulty ADT's. Further work will consist of verifying that the metrics proposed by [Briand et. al. ,1994] are also applicable to C++ programs, in a context of inheritance.

To the knowledge of the authors, [Lie&Henry, 1993] is the only study which can really be compared to the work we describe in this paper. Li and Henry have proposed a suite of OO design metrics. They validated this suite of metrics by studying the number of changes performed in two commercial systems implemented with an OO dialect of Ada. The suite of OO design metrics used by Li and Henry extends Chidamber&Kemerer's OO metrics with two additional metrics:

- Message Passing Coupling (MPC) which is calculated as the number of send statements defined in a class.
- Data Abstraction Coupling (DAC) which is calculated as the number of abstract data types used in the measured class and defined in another class of the system.

They combined the six Chidamber&Kemerer's OO metrics with these last two metrics in a single least-square regression model. According to the authors, their model was adequate in predicting the size of changes in classes during the maintenance phase. They did not, however, look at the time spent changing a class nor the cause of changes (e.g., corrections, enhancement, etc.). In addition, they assumed that the number of modifications in a class is proportional to the effort spent to change it, which is not necessarily true. Also, we do not believe that the number of changes can be considered as a measure of maintainability since it is not dependent on the modifiability of a class but on the correctness and functional stability of the class.

In this study, we did not consider DAC and MPC because they could not be directly applied in our experimental context (C++ does not provide send statements). Based on the way DAC was defined by Lie&Henry, it cannot be directly used for C++. DAC could, however, be redefined/tailored to our needs, providing another way to calculate coupling across C++ classes. This is, however, beyond of the scope of this paper.

An important difference in our work is that we have used the occurrence of faults in a class to verify whether Chidamber&Kemerer's OO metrics were adequate quality predictors. Of course, many other quality measures of interest could be used in this context, e.g., change productivity. Last, the modeling technique we used (i.e., logistic regression) to predict fault-prone classes is different because of the nature of the dependent variable which is binary in our case. This has led us to use a classification technique.

5. Conclusions and further work

In this experiment, we collected data about defects found in Object-Oriented classes. Based on these data we verified experimentally how much fault-proneness is influenced by internal (e.g., size, cohesion) and external (e.g., coupling) design characteristics of OO classes. From the results presented above, several of Chidamber&Kemerer's OO metrics appear to be adequate to predict class fault-proneness during the early phases of the life-cycle. We also showed that Chidamber&Kemerer's OO metrics are better predictors than "traditional" code metrics on our data set, which, in addition, can only be collected at a later phase of the software development processes.

Our future work includes:

- replicating this study in an industrial setting: a sample of large-scale projects developed in C++ and Ada95 in the framework of the NASA Goddard Flight Dynamics Division (Software Engineering Laboratory). This work should help us better understand the prediction capabilities of the suite of OO metrics described in this paper. By doing that, we intend to:

  - build models and provide guidance to improve the allocation of resources with respect to test and verification efforts,
  - gain a better understanding of the impact of OO design strategies (e.g., simple versus multiple inheritance) on defect density and rework. In this study, because of an inadequate data collection process, we were unable to analyze the capability of OO design metrics to predict rework. We believe that this drawback could be overcome by refining our data collection process in order to capture how much effort was spent on each class individually.

- analyzing OO libraries in order to identify "good" and "bad" OO design patterns. Design patterns have been claimed to be a way to improve reuse and quality of OO software systems [Gamma et. al, 1995]. We intend to use the approach described in this paper to assess organization-specific design patterns, thus providing guidelines about what OO design patterns should be encouraged and which ones should be avoided due to their fault-proneness or their lack of maintainability.
- studying the variations, in terms of metric definitions and experimental results, between different OO programming languages. The fault-proneness prediction capabilities of the suite of OO metrics discussed in this paper can be different depending on the used programming language. Work must be undertaken to validate this suite of OO design metrics across different OO languages, e.g., Ada95, Smalltalk, Eifeil, C++, etc.
- extending the experimental investigation to other OO metrics proposed in the literature (e.g., [Abreu&Carapuça, 1994]) and develop new metrics, e.g., more language specific.

Acknowledgements

This work would not have been possible without the help of Dr. Prem Devanbu from AT&T Labs. We also want to thank Dr. Gianluigi Caldiera for co-teaching the OMT method to the students who participated in this study, and Carolyn Seaman, Barbara Swain and Roseanne Tesoriero for their suggestions that helped improve both the content and the form of this paper.

References

F. B. Abreu and R. Carapuça (1994). "Candidate metrics for object-oriented software within a taxonomy framework". *Journal of System and Software*, 26(1):87–96.

Amadeus Software Research, Inc. (1994). "*Getting Started with Amadeus.*" Amadeus Measurement System.

V. Basili; G. Caldiera; F. McGarry; R. Pajerski; G. Page (1992). "The Software Engineering Laboratory: An Operational Software Experience Factory". In *Proc. of the 14th Int'l Conf. on Software Engineering*.

V. Basili and D. Hutchens (1982). "Analyzing a syntactic family of complexity metrics". *IEEE Trans. on Software Engineering*, SE-9(6):664–673.

L. Briand; S. Morasca; V. Basili (1994). *"Goal-Driven Definition on Product Metrics Based on Properties"*. CS-TR-3346, University of Maryland, Dep. of CS, College Park, MD, 20742.

I. Brooks (1993). "Object-oriented metrics collection and evaluation with a software process". Presented at *OOPSLA'93 Workshop on Processes and Metrics for Object-Oriented Software Development*, Washington, DC.

S. R. Chidamber and C. F. Kemerer (1994). "A metrics suite for object-oriented design". *IEEE Trans. on Software Engineering*, 20(6):476–493.

S. R. Chidamber and C. F. Kemerer (1995). "Authors Reply". *IEEE Trans. on Software Engineering, IEEE Trans. on Software Engineering*, 21(3):265.

N. I. Churcher and M. J. Shepperd (1995). "Comments on 'A Metrics Suite for Object-Oriented Design' ". *IEEE Trans. on Software Engineering*, 21(3):263–265.

P. Devanbu (1992). "GENOA/GENII – a customizable language and front-end independent code analyzer". In *Proc. of the 14th Int'l Conf. on Software Engineering*, Melbourne, Australia.

J. Devore (1991). "*Probability and Statistics for Engineering and Sciences.*" Brooks/Cole Publishing Company.

W. Dillon and M. Goldstein (1984). "*Multivariate Analysis: Methods and Applications.*" Wiley.

W. Harrison (1988). "Using software metrics to allocate testing resources." *Journal of Management Information Systems*. 4(4):93–105.

W. Harrison (1994). "Software measurement: a decision-process approach". *Advances in Computers*, vol 39, pp. 51–105.

G. Heller; J. Valett; M. Wild (1992). *Data Collection Procedure for the Software Engineering Laboratory (SEL) Database*. SEL Series, SEL-92-002.

D. Hosmer and S. Lemeshow (1989). "*Applied Logistic Regression.*" Wiley-Interscience.

N. E. Fenton (1991). "*Software Metrics: A Rigorous Approach*". Chapman&Hall.

E. Gamma; R. Helm; R. Johnson; J. Vlissides (1995). "*Design Patterns: Elements of Reusable Object-Oriented Software*". Addison-Wesley.

W. Lie and S. Henry (1993). "Object-oriented metrics that predict maintainability". *Journal of Systems and Software*. 23(2):111–122.

F. McGarry; R. Pajersk; G. Page; S. Waligora; V. Basili; M. Zelkowitz (1994). "*Software Process Improvement in the NASA Software Engineering Laboratory*". Carnegie Mellon University, Software Engineering Institute, Technical Report, Dec. 1994. CMU/SEI-95-TR-22.

W. Melo; L. Briand; V. Basili (1995). "*Measuring the Impact of Reuse on Quality and Productivity in Object-Oriented Systems.*" Technical Report, University of Maryland, Dep. of Computer Science, Jan. 1995, CS-TR-3395.

J. Rumbaugh; M. Blaha; W. Premerlani; F. Eddy; W. Lorensen (1991). "*Object-Oriented Modeling and Design.*" Prentice-Hall.

B. Stroustrup. "*The C++ Programming Language*". Addison-Wesley Series in Computer Science, 1991. 2nd edition.

D. A. Young (1992). "*Object-Oriented Programming with C++ and OSF/MOTIF.*" Prentice-Hall.

OMIT THIS PAGE

SECTION 5—TECHNOLOGY EVALUATIONS

The technical paper included in this section was originally prepared as indicated below.

- "Generalized Support Software: Domain Analysis and Implementation,", M. Stark and E. Seidewitz, *Addendum to the Proceedings OOPSLA '94*, Ninth Annual Conference, Portland, Oregon, U.S.A., October 1994, pp. 8–13

Page intentionally left blank

GENERALIZED SUPPORT SOFTWARE: DOMAIN ANALYSIS AND IMPLEMENTATION
EXPERIENCE REPORT SUBMITTED TO OOPSLA'94

Mike Stark
Ed Seidewitz

NASA Goddard Space Flight Center
Code 552.3
Greenbelt MD 20771
michael.e.stark@gsfc.nasa.gov / (301)286-5048
ed.seidewitz@gsfc.nasa.gov / (301)286-7631

For the past five years, the Flight Dynamics Division (FDD) at NASA's Goddard Space Flight Center has been carrying out a detailed domain analysis effort and is now beginning to implement Generalized Support Software (GSS) based on this analysis. GSS is part of the larger Flight Dynamics Distributed System (FDDS), and is designed to run under the FDDS User Interface / Executive (UIX). The FDD is transitioning from a mainframe based environment to FDDS based systems running on engineering workstations The GSS will be a library of highly reusable components that may be *configured* within the standard FDDS architecture to quickly produce low-cost satellite ground support systems. The estimates for the first release is that this library will contain approximately 200,000 lines of code.

The main driver for developing generalized software is development cost and schedule improvement. The goal is to ultimately have at least 80 percent of all software required for a spacecraft mission (within the domain supported by the GSS) to be configured from the generalized components.

Domain Analysis

The GSS domain analysis effort originally grew out of a study of the feasibility of generalizing the attitude ground support systems (AGSSs) produced by the FDD for various spacecraft missions. FDD software tends to be similar from mission to mission. An AGSS is used to determine the orientation of a spacecraft from on-board sensor data and to compute maneuvers to change that orientation. It typically has several executable programs that are used for specialized areas such as attitude estimation and sensor calibration. These programs share models to varying degrees. For example, just about every FDD system has an orbit propagator in it. Part of the domain analysis effort is intended to reduce overlap and redundancy between systems.

As part of an ambitious project to re-engineer a majority of the FDD software systems, the domain covered by the analysis was later expanded to also include a number of mission analysis and planning functions. Indeed, at one point plans called for this project to eventually encompass *all* FDD functionality, adding orbit models to the attitude and mission planning functionality.

Project History

The domain analysis effort began by studying the functional specifications of existing AGSSs. These specifications used data flow diagrams, so it was natural to adopt this technique for the generalized domain model. However, the limitations of this approach soon became apparent, especially in the lack of classification techniques crucial to capturing generalizations. Despite the fact that most of the people working on the effort were not particularly familiar with object-oriented approaches, a consensus developed that object-oriented analysis would be a better technique than data-flow diagrams for our purposes. Following this decision, we developed a *Specification Concepts* document [Seidewitz 91] that captured the object-oriented analysis approach used in subsequent analysis.

Unfortunately, budgetary pressures prevented the ambitions re-engineering plans from becoming reality. Further, the expanding scope of the analysis effort became increasingly difficult to handle. Thus, the domain analysis effort was refocused generally to concentrate once again on the attitude support domain. The end effect was that the domain analysis team did not increase as planned, leaving a small team to do the analysis over several years. The effort has specifically proceeded to focus in detail on the analysis of the first two GSS releases: telemetry simulation and real-time attitude determination. We have now completed two versions of the generalized specifications for the first release [Klitsch 93] and work is proceeding on the specifications for the second release [Klitsch 94].

Specification Concepts

The specification products of the domain analysis effort are all based on our standard specification concepts. Actually, these specification concepts have continued to evolve based on our analysis experiences [Seidewitz 93]. Throughout this process there has been a continual tension between keeping the concepts as simple as possible and assuring that they are powerful enough to allow specification of domain functionality without undue complication. The core concepts of the model include the basic object-oriented principles of classes, objects and messages. Additional concepts have been added to this core only when not including the new concept would make it difficult or impossible to clearly specify some specific domain functionality under consideration.

For example, we have used only two levels of classification of objects. Each specific object class belongs to exactly one superclass that represents a general domain category (e.g., a *Sun Sensor* would be in the *Sensor* category). Further, superclasses only specify common interfaces, not common functionality, so there is no inheritance of functionality by subclasses. This restricted approach has allowed us to cleanly and simply introduce the required generalization concepts while maintaining the locality of specification of the functionality of any class. The approach worked well through the first versions of our specifications. However, current work is indicating an increasing number of opportunities where deeper classification hierarchies would be useful, and we may add this to our concepts.

Another restriction in our concepts is that objects are not dynamically created or destroyed. Instead, objects and their interdependencies are specified as part of the *configuration* of an application. Once these objects are created, they exist for the duration of the execution of the application. Data passed between objects is not itself object-oriented, but is instead drawn from a set of standard data types (*Integer*, *Real*, *Vector*, *Matrix*, etc.). This approach provides us with a clear definition of *configuration*, which was a topic of many long discussions. The resolution of these discussions was that the generalized specifications deal exclusively with the definition of classes, while the configuration specifications deal exclusively with the definition of the objects in an application. This philosophy also provides a fundamental connection to our implementation approach.

Besides restrictions in using object-oriented concepts, the specification concepts evolved to eliminate unnecessary and sometimes complex concepts. For example, the original concepts called for modeling separate subsystems that only communicate via data objects. These subsystems were intended to be configured as separate executable programs. This made it hard to specify models (such as estimation algorithms) that are usable in more than one subsystem (such as attitude determination and sensor calibration). The solution was to create a single domain map, and replace the subsystem driver with application categories that provide the same set of actions to the UIX. These application categories also map to separately configured programs, but can draw on classes throughout the domain map, instead of classes contained in a single subsystem.

Lessons Learned

The current specifications are defined with more detail and less ambiguity than the typical FDD specification documents. This has had a positive impact on the development process, since class specifications are generally detailed enough to serve as PDL. However, these specifications are *harder* for the analyst to understand when specifying the configuration of an application program for a given satellite. The generalized specification document is currently weak at showing how an entire application would behave. One reason is that the specification effort has focused the limited resources on producing class specifications to implement, at the expense of producing information that the analysts would use when defining a configuration.

A more important reason is that FDD attitude and orbit analysts don't think in terms of objects, but in terms of algorithms such as a Kalman Filter estimation algorithm. The concept of this algorithm can be expressed to the mathematician in 5 or 6 equations. To understand the GSS specification, the analyst needs to understand how several classes contribute to the processing needed to implement these equations. The specification concepts need to be updated to improve the description of how classes interact to support algorithm. Part of the answer is to complete the intended documentation for each subdomain (major group of categories) to explain these interactions. The concept of "scenarios" or "use cases" (as discussed by e.g. [Jacobson 92]) may be appropriate for describing the overall behavior of an application.

Another key lesson for domain analysis is that developers need to be involved in the process. This is primarily because the class specifications are written at a level of detail that often raises implementation issues such as performance. The GSS project has always had developer involvement in the domain analysis process. This process may be improved by increasing this involvement, perhaps even evolving towards a joint analysis / development team. This is because as more classes are implemented the developers have a greater stake in making sure that new analysis work won't have any negative effects on the existing class library.

Development

The creation of a generalized design is made possible by the standardization of class specifications in the *Specification Concept* document, and by the standardization of the interaction between the UIX and the GSS application [Booth 93]. The UIX drives application processing by calls to three operations provided by the application. These operations allow the user to access and modify operations, or to execute the next action in the application. The application may also send messages to the UIX.

The key feature of a GSS application is that it is built from a library of classes, and can then be configured at run time. The run time configuration process includes allocating the objects for each class, setting the specific dependencies between objects (the generalized specifications define dependencies between *classes*, which are implemented at compile time using the Ada generic parameters), and setting default parameter values.

Implementation

The classes in our generalized specifications are implemented as a set of two Ada packages. A *class package* implements an abstract data type representing the class, and an *object manager package* contains all the objects for a given class. These classes are arranged in a hierarchy with *category packages* implementing the interface for a specified category, and the *Application Interface* package implementing a root object that dispatches to categories the operations to allocate objects, set dependencies and interact with parameters (instance variables). The bodies of the category packages and the Application Interface package implement only dispatching code. All the functionality resides in class and object manager packages.

Ada was chosen as a development language for two reasons. The organizational reason at the beginning of the GSS project the division had experience with several Ada simulators, C++ was not considered mature technology by division management, and no other language met the need for object orientation, support on a wide variety of platforms, and a core of experienced developers in the FDD. The technical reason was the use of generics to add flexibility to the configuration process.

The GSS generic packages use both types (defining the class or category depended on) and subprograms (defining messages sent to the class or category depended on). The configuration process consists of instantiating the generics to set the dependencies between classes and categories and calling dependency operations to set the actual connections between objects. The use of generics allows categories dependencies to be satisfied by classes, bypassing the dispatching code when it is not needed. This fact was important in addressing user concerns that the overhead of dispatching code would hurt run time performance. A class can actually be instantiated using *any* class that provides the operations that are needed to match the generic parameters.

Code Generation (Classgen)

The code for the allocation, dependency and parameter operations is similar in structure from class to class, but each of these operations depends on the specification of the particular class. This means that the implementation code can not be written at the root of the classification tree, but that there is still a lot of tedious repetition to the coding of classes. The development team's solution was to write a code generator (named Classgen) that reads in a concise notation describing class functions, dependencies and parameters. The output of the code generator is the implementation of all the functions specified at the Application Interface level, plus subprogram interfaces and stubs for the constructors and selectors defined in the specification document. This was made possible by the existence of a generalized design that mapped standard specification features into the Ada implementation.

The input language for Classgen also has features corresponding to those defined in the specification concepts, and adds design features such as the error handling. The tool generates a type definition for a class that contains all the parameters, internal data, and dependencies defined for the class, implementation of stubs for the functions in the specification, and implementation of the subprograms needed for allocating objects, setting dependencies, and accessing or modifying parameter values. This code is about 75% of the code needed to implement a class, with one line of Classgen input generating about 10 lines of Ada code. Classgen generates all the code that can be generated based on the standardized specifications and design. The remaining code is the implementation of the functions specified for the class.

Classgen Lessons

Having a code generator has saved time and effort on the GSS project, but it has taken time for the tool to mature. The main reason for this is that the initial concept was for Classgen to be run once per class to generate the code, and editing the created files after that. In practice it was necessary to edit the regenerated code, both because the generalized design evolved and required changes and because the developers used the tool to regenerate files if there were substantial modifications to a class. The problem was that the original version of Classgen required the developer to edit most of the files generated for a class. A notation was defined to mark these changes, but regenerating the class meant having to merge these changes into the new file. Classgen has been modified in stages so that in most cases the only file a developer edits is a separately compiled "subunit" file in which the specified functions are implemented. Changes to the other files still occur, but they are rare enough and small enough that they don't have a major impact.

These changes were generally made by extending the Classgen grammar, but in some cases the generalized design was modified to facilitate code generation. A simple example of this was to move all "with" clauses (which define dependencies between Ada packages) into package specifications, and having all utility packages imported into a class be "with"ed into the Classgen input as well. This sometimes makes packages visible in a larger amount of code than strictly necessary, but it captures the design information in the Classgen input and removes the need to edit files to add the importing of packages.

Process Lessons

The use of standardized, object-oriented specification concepts has had several effects on development. We have already noted that the specifications are complete enough to serve as PDL. The specification of dependencies between classes, together with the generalized design for dependencies, completely captures the system structure typically defined in preliminary design. The development of a build typically starts with detailed design of classes, which is expressed in terms of changes to the specification. Given this shift of "design" work to the domain analysis team, a joint "domain analysis and design" team may be justified. This is particularly true once the class library is populated and changes to the domain may have major effects on the existing code.

Using object-oriented specifications will enable incremental development. However, the flight dynamics domain is one where a substantial number of core classes (integrators, dynamic models, environmental models,...) are needed before anything useful can be done. The builds are still being done incrementally, but a system that is testable by the end user won't be available until the third build is complete. The good news is that once the first application is complete, added capabilities can be created in single builds. For example, the first two releases of GSS will be delivering components to support simulation and real-time attitude estimation in a total of 5 builds. Adding the generalized components for non-real time estimation and for sensor calibration will take one or two additional builds. Similar scale builds can be used to add new models to the existing categories, or to expand into the orbit or maneuver planning areas. Thus "design a little, code a little, test a little" will work for GSS, but only after a base of core classes has been implemented.

The integration of these generalized classes has been easier than for typical projects. This is another benefit of having standard object-oriented specifications that clearly define internal and external interfaces, and a generalized design that standardizes the implementation of dependencies between classes. Together these factors assure that if a class depends on a given operation from another class that class will provide the operation and the two classes will interface correctly.

Summary

The lessons described above have been learned during the specification and the early development of the GSS project. These lessons will be applied to further specification and development work. The initial releases will be complete by the end of 1995, at which point the FDD will start seeing return on the investment in this project.

References

[Booth 93] E. Booth et. al., Flight Dynamics Distributed System (FDDS) Generalized Support Software (GSS) Release 1 (GSSR1) Implementation Description , 552-FDD-93/068R0UD0 (draft), October 1993

[Jacobson 92] I. Jacobson et. al., Object-Oriented Software Engineering, Addison-Wesley 1992

[Klitsch 93] G Klitsch et. al., Flight Dynamics Distributed System (FDDS) Generalized Support Software (GSS) Functional Specifications (Revision 1), 553-FDD-93/046R1UD0, June 1993

[Klitsch 94] G Klitsch et. al., Flight Dynamics Distributed System (FDDS) Generalized Support Software (GSS) Functional Specifications (Revision 1, Update 1), 553-FDD-93/046R1UD1, December 1994

[Seidewitz 91] E. Sedewitz et. al., Combined Operational Mission Planning and Attitude Support System (COMPASS) Specification Concepts, 550-COMPASS-103, May 1991

[Seidewitz 93] E. Seidewitz et. al., Flight Dynamics Distributed System (FDDS) Generalized Support Software (GSS) Specification Concepts (Revision 1), 553-FDD-93/057R1UD0 (draft), August 1993

OMIT TO
END

STANDARD BIBLIOGRAPHY OF SEL LITERATURE

The technical papers, memorandums, and documents listed in this bibliography are organized into two groups. The first group is composed of documents issued by the Software Engineering Laboratory (SEL) during its research and development activities. The second group includes materials that were published elsewhere but pertain to SEL activities.

SEL-ORIGINATED DOCUMENTS

SEL-76-001, *Proceedings From the First Summer Software Engineering Workshop,* August 1976

SEL-77-002, *Proceedings From the Second Summer Software Engineering Workshop,* September 1977

SEL-78-005, *Proceedings From the Third Summer Software Engineering Workshop,* September 1978

SEL-78-006, *GSFC Software Engineering Research Requirements Analysis Study,* P. A. Scheffer and C. E. Velez, November 1978

SEL-78-007, *Applicability of the Rayleigh Curve to the SEL Environment,* T. E. Mapp, December 1978

SEL-78-302, *FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 3),* W. J. Decker, W. A. Taylor, et al., July 1986

SEL-79-002, *The Software Engineering Laboratory: Relationship Equations,* K. Freburger and V. R. Basili, May 1979

SEL-79-004, *Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment,* C. E. Goorevich, A. L. Green, and W. J. Decker, September 1979

SEL-79-005, *Proceedings From the Fourth Summer Software Engineering Workshop,* November 1979

SEL-80-002, *Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation,* W. J. Decker and C. E. Goorevich, May 1980

SEL-80-005, *A Study of the Musa Reliability Model,* A. M. Miller, November 1980

SEL-80-006, *Proceedings From the Fifth Annual Software Engineering Workshop,* November 1980

SEL-80-007, *An Appraisal of Selected Cost/Resource Estimation Models for Software Systems,* J. F. Cook and F. E. McGarry, December 1980

SEL-80-008, *Tutorial on Models and Metrics for Software Management and Engineering*, V. R. Basili, 1980

SEL-81-011, *Evaluating Software Development by Analysis of Change Data*, D. M. Weiss, November 1981

SEL-81-012, *The Rayleigh Curve as a Model for Effort Distribution Over the Life of Medium Scale Software Systems*, G. O. Picasso, December 1981

SEL-81-013, *Proceedings of the Sixth Annual Software Engineering Workshop*, December 1981

SEL-81-014, *Automated Collection of Software Engineering Data in the Software Engineering Laboratory (SEL)*, A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-81-101, *Guide to Data Collection*, V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

SEL-81-110, *Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics*, G. Page, F. E. McGarry, and D. N. Card, June 1985

SEL-81-305, *Recommended Approach to Software Development*, L. Landis, S. Waligora, F. E.1McGarry, et al., June 1992

SEL-81-305SP1, *Ada Developers' Supplement to the Recommended Approach*, R. Kester and L. Landis, November 1993

SEL-82-001, *Evaluation of Management Measures of Software Development*, G. Page, D. N. Card, and F. E. McGarry, September 1982, vols. 1 and 2

SEL-82-004, *Collected Software Engineering Papers: Volume 1*, July 1982

SEL-82-007, *Proceedings of the Seventh Annual Software Engineering Workshop*, December 1982

SEL-82-008, *Evaluating Software Development by Analysis of Changes: The Data From the Software Engineering Laboratory*, V. R. Basili and D. M. Weiss, December 1982

SEL-82-102, *FORTRAN Static Source Code Analyzer Program (SAP) System Description (Revision 1)*, W. A. Taylor and W. J. Decker, April 1985

SEL-82-105, *Glossary of Software Engineering Laboratory Terms*, T. A. Babst, M. G. Rohleder, and F. E. McGarry, October 1983

SEL-82-1306, *Annotated Bibliography of Software Engineering Laboratory Literature*, D. Kistler, J. Bristow, and D. Smith, November 1994

SEL-83-001, *An Approach to Software Cost Estimation*, F. E. McGarry, G. Page, D. N. Card, et al., February 1984

SEL-83-002, *Measures and Metrics for Software Development*, D. N. Card, F. E. McGarry, G. Page, et al., March 1984

SEL-83-003, *Collected Software Engineering Papers: Volume II*, November 1983

SEL-83-007, *Proceedings of the Eighth Annual Software Engineering Workshop*, November 1983

SEL-83-106, *Monitoring Software Development Through Dynamic Variables (Revision 1)*, C. W. Doerflinger, November 1989

SEL-84-003, *Investigation of Specification Measures for the Software Engineering Laboratory (SEL)*, W. W. Agresti, V. E. Church, and F. E. McGarry, December 1984

SEL-84-004, *Proceedings of the Ninth Annual Software Engineering Workshop*, November 1984

SEL-84-101, *Manager's Handbook for Software Development (Revision 1)*, L. Landis, F. E. McGarry, S. Waligora, et al., November 1990

SEL-85-001, *A Comparison of Software Verification Techniques*, D. N. Card, R. W. Selby, Jr., F. E. McGarry, et al., April 1985

SEL-85-002, *Ada Training Evaluation and Recommendations From the Gamma Ray Observatory Ada Development Team*, R. Murphy and M. Stark, October 1985

SEL-85-003, *Collected Software Engineering Papers: Volume III*, November 1985

SEL-85-004, *Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics*, R. W. Selby, Jr., and V. R. Basili, May 1985

SEL-85-005, *Software Verification and Testing*, D. N. Card, E. Edwards, F. McGarry, and C. Antle, December 1985

SEL-85-006, *Proceedings of the Tenth Annual Software Engineering Workshop*, December 1985

SEL-86-001, *Programmer's Handbook for Flight Dynamics Software Development*, R. Wood and E. Edwards, March 1986

SEL-86-002, *General Object-Oriented Software Development*, E. Seidewitz and M. Stark, August 1986

SEL-86-003, *Flight Dynamics System Software Development Environment (FDS/SDE) Tutorial*, J. Buell and P. Myers, July 1986

SEL-86-004, *Collected Software Engineering Papers: Volume IV*, November 1986

SEL-86-005, *Measuring Software Design*, D. N. Card et al., November 1986

SEL-86-006, *Proceedings of the Eleventh Annual Software Engineering Workshop*, December 1986

SEL-87-001, *Product Assurance Policies and Procedures for Flight Dynamics Software Development*, S. Perry et al., March 1987

SEL-87-002, *Ada® Style Guide (Version 1.1)*, E. Seidewitz et al., May 1987

SEL-87-003, *Guidelines for Applying the Composite Specification Model (CSM)*, W. W. Agresti, June 1987

SEL-87-004, *Assessing the Ada® Design Process and Its Implications: A Case Study*, S. Godfrey, C. Brophy, et al., July 1987

SEL-87-009, *Collected Software Engineering Papers: Volume V*, November 1987

SEL-87-010, *Proceedings of the Twelfth Annual Software Engineering Workshop*, December 1987

SEL-88-001, *System Testing of a Production Ada Project: The GRODY Study*, J. Seigle, L. Esker, and Y. Shi, November 1988

SEL-88-002, *Collected Software Engineering Papers: Volume VI*, November 1988

SEL-88-003, *Evolution of Ada Technology in the Flight Dynamics Area: Design Phase Analysis*, K. Quimby and L. Esker, December 1988

SEL-88-004, *Proceedings of the Thirteenth Annual Software Engineering Workshop*, November 1988

SEL-88-005, *Proceedings of the First NASA Ada User's Symposium*, December 1988

SEL-89-002, *Implementation of a Production Ada Project: The GRODY Study*, S. Godfrey and C. Brophy, September 1989

SEL-89-004, *Evolution of Ada Technology in the Flight Dynamics Area: Implementation/Testing Phase Analysis*, K. Quimby, L. Esker, L. Smith, M. Stark, and F. McGarry, November 1989

SEL-89-005, *Lessons Learned in the Transition to Ada From FORTRAN at NASA/Goddard*, C. Brophy, November 1989

SEL-89-006, *Collected Software Engineering Papers: Volume VII*, November 1989

SEL-89-007, *Proceedings of the Fourteenth Annual Software Engineering Workshop*, November 1989

SEL-89-008, *Proceedings of the Second NASA Ada Users' Symposium*, November 1989

SEL-89-103, *Software Management Environment (SME) Concepts and Architecture (Revision 1)*, R. Hendrick, D. Kistler, and J. Valett, September 1992

SEL-89-301, *Software Engineering Laboratory (SEL) Database Organization and User's Guide (Revision 3)*, L. Morusiewicz, February 1995

SEL-90-001, *Database Access Manager for the Software Engineering Laboratory (DAMSEL) User's Guide*, M. Buhler, K. Pumphrey, and D. Spiegel, March 1990

SEL-90-002, *The Cleanroom Case Study in the Software Engineering Laboratory: Project Description and Early Analysis*, S. Green et al., March 1990

SEL-90-003, *A Study of the Portability of an Ada System in the Software Engineering Laboratory (SEL)*, L. O. Jun and S. R. Valett, June 1990

SEL-90-004, *Gamma Ray Observatory Dynamics Simulator in Ada (GRODY) Experiment Summary*, T. McDermott and M. Stark, September 1990

SEL-90-005, *Collected Software Engineering Papers: Volume VIII*, November 1990

SEL-90-006, *Proceedings of the Fifteenth Annual Software Engineering Workshop*, November 1990

SEL-91-001, *Software Engineering Laboratory (SEL) Relationships, Models, and Management Rules*, W. Decker, R. Hendrick, and J. Valett, February 1991

SEL-91-003, *Software Engineering Laboratory (SEL) Ada Performance Study Report*, E. W. Booth and M. E. Stark, July 1991

SEL-91-004, *Software Engineering Laboratory (SEL) Cleanroom Process Model*, S. Green, November 1991

SEL-91-005, *Collected Software Engineering Papers: Volume IX*, November 1991

SEL-91-006, *Proceedings of the Sixteenth Annual Software Engineering Workshop*, December 1991

SEL-91-102, *Software Engineering Laboratory (SEL) Data and Information Policy (Revision 1)*, F. McGarry, August 1991

SEL-92-001, *Software Management Environment (SME) Installation Guide*, D. Kistler and K. Jeletic, January 1992

SEL-92-002, *Data Collection Procedures for the Software Engineering Laboratory (SEL) Database*, G. Heller, J. Valett, and M. Wild, March 1992

SEL-92-003, *Collected Software Engineering Papers: Volume X*, November 1992

SEL-92-004, *Proceedings of the Seventeenth Annual Software Engineering Workshop*, December 1992

SEL-93-001, *Collected Software Engineering Papers: Volume XI*, November 1993

SEL-93-002, *Cost and Schedule Estimation Study Report*, S. Condon, M. Regardie, M. Stark, et al., November 1993

SEL-93-003, *Proceedings of the Eighteenth Annual Software Engineering Workshop*, December 1993

SEL-94-001, *Software Management Environment (SME) Components and Algorithms*, R. Hendrick, D. Kistler, and J. Valett, February 1994

SEL-94-003, *C Style Guide*, J. Doland and J. Valett, August 1994

SEL-94-004, *Collected Software Engineering Papers: Volume XII*, November 1994

SEL-94-005, *An Overview of the Software Engineering Laboratory*, F. McGarry, G. Page, V. Basili, et al., December 1994

SEL-94-006, *Proceedings of the Nineteenth Annual Software Engineering Workshop*, December 1994

SEL-94-102, *Software Measurement Guidebook (Revision 1)*, M. Bassman, F. McGarry, R. Pajerski, June 1995

SEL-95-001, *Impact of Ada in the Flight Dynamics Division at Goddard Space Flight Center*, S. Waligora, J. Bailey, M. Stark, March 1995

SEL-95-003, *Collected Software Engineering Papers: Volume XIII*, November 1995

SEL-RELATED LITERATURE

[10]Abd-El-Hafiz, S. K., V. R. Basili, and G. Caldiera, "Towards Automated Support for Extraction of Reusable Components," *Proceedings of the IEEE Conference on Software Maintenance-1991 (CSM 91)*, October 1991

[4]Agresti, W. W., V. E. Church, D. N. Card, and P. L. Lo, "Designing With Ada for Satellite Simulation: A Case Study," *Proceedings of the First International Symposium on Ada for the NASA Space Station*, June 1986

[2]Agresti, W. W., F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," *Program Transformation and Programming Environments*. New York: Springer-Verlag, 1984

[1]Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," *Proceedings of the Fifth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1981

[8]Bailey, J. W., and V. R. Basili, "Software Reclamation: Improving Post-Development Reusability," *Proceedings of the Eighth Annual National Conference on Ada Technology*, March 1990

[10]Bailey, J. W., and V. R. Basili, "The Software-Cycle Model for Re-Engineering and Reuse," *Proceedings of the ACM Tri-Ada 91 Conference*, October 1991

[1]Basili, V. R., "Models and Metrics for Software Management and Engineering," *ASME Advances in Computer Technology*, January 1980, vol. 1

Basili, V. R., *Tutorial on Models and Metrics for Software Management and Engineering*. New York: IEEE Computer Society Press, 1980 (also designated SEL-80-008)

[3]Basili, V. R., "Quantitative Evaluation of Software Methodology," *Proceedings of the First Pan-Pacific Computer Conference*, September 1985

[7]Basili, V. R., *Maintenance = Reuse-Oriented Software Development*, University of Maryland, Technical Report TR-2244, May 1989

[7]Basili, V. R., *Software Development: A Paradigm for the Future*, University of Maryland, Technical Report TR-2263, June 1989

[8]Basili, V. R., “Viewing Maintenance of Reuse-Oriented Software Development,” *IEEE Software*, January 1990

[13]Basili, V. R., "The Experience Factory and Its Relationship to Other Quality Approaches," *Advances in Computers*, vol. 41, Academic Press, Incorporated, 1995

[1]Basili, V. R., and J. Beane, “Can the Parr Curve Help With Manpower Distribution and Resource Estimation Problems?,” *Journal of Systems and Software*, February 1981, vol. 2, no. 1

[13]Basili, V. R., L. Briand, and W. L. Melo, *A Validation of Object-Oriented Design Metrics*, University of Maryland, Computer Science Technical Report, CS-TR-3443, UMIACS-TR-95-40, April 1995

[13]Basili, V. R., and G. Caldiera, *The Experience Factory Strategy and Practice*, University of Maryland, Computer Science Technical Report, CS-TR-3483, UMIACS-TR-95-67, May 1995

[9]Basili, V. R., G. Caldiera, and G. Cantone, “A Reference Architecture for the Component Factory,”*ACM Transactions on Software Engineering and Methodology*, January 1992

[10]Basili, V., G. Caldiera, F. McGarry, et al., “The Software Engineering Laboratory—An Operational Software Experience Factory,” *Proceedings of the Fourteenth International Conference on Software Engineering (ICSE 92)*, May 1992

[1]Basili, V. R., and K. Freburger, “Programming Measurement and Estimation in the Software Engineering Laboratory,” *Journal of Systems and Software*, February 1981, vol. 2, no. 1

[12]Basili, V. R., and S. Green, "Software Process Evolution at the SEL," *IEEE Software*, July 1994, pp. 58–66

[3]Basili, V. R., and N. M. Panlilio-Yap, “Finding Relationships Between Effort and Other Variables in the SEL,” *Proceedings of the International Computer Software and Applications Conference*, October 1985

[4]Basili, V. R., and D. Patnaik, *A Study on Fault Prediction and Reliability Assessment in the SEL Environment*, University of Maryland, Technical Report TR-1699, August 1986

[2]Basili, V. R., and B. T. Perricone, “Software Errors and Complexity: An Empirical Investigation,” *Communications of the ACM*, January 1984, vol. 27, no. 1

[1]Basili, V. R., and T. Phillips, “Evaluating and Comparing Software Metrics in the Software Engineering Laboratory,” *Proceedings of the ACM SIGMETRICS Symposium/Workshop: Quality Metrics*, March 1981

[3]Basili, V. R., and C. L. Ramsey, “ARROWSMITH-P—A Prototype Expert System for Software Engineering Management,” *Proceedings of the IEEE/MITRE Expert Systems in Government Symposium*, October 1985

Basili, V. R., and J. Ramsey, *Structural Coverage of Functional Testing*, University of Maryland, Technical Report TR-1442, September 1984

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," *Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity, and Cost*. New York: IEEE Computer Society Press, 1979

[5]Basili, V. R., and H. D. Rombach, "Tailoring the Software Process to Project Goals and Environments," *Proceedings of the 9th International Conference on Software Engineering*, March 1987

[5]Basili, V. R., and H. D. Rombach, "TAME: Tailoring an Ada Measurement Environment," *Proceedings of the Joint Ada Conference*, March 1987

[5]Basili, V. R., and H. D. Rombach, "TAME: Integrating Measurement Into Software Environments," University of Maryland, Technical Report TR-1764, June 1987

[6]Basili, V. R., and H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," *IEEE Transactions on Software Engineering*, June 1988

[7]Basili, V. R., and H. D. Rombach, *Towards A Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment*, University of Maryland, Technical Report TR-2158, December 1988

[8]Basili, V. R., and H. D. Rombach, *Towards A Comprehensive Framework for Reuse: Model-Based Reuse Characterization Schemes*, University of Maryland, Technical Report TR-2446, April 1990

[9]Basili, V. R., and H. D. Rombach, "Support for Comprehensive Reuse," *Software Engineering Journal*, September 1991

[3]Basili, V. R., and R. W. Selby, Jr., "Calculation and Use of an Environment's Characteristic Software Metric Set," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

Basili, V. R., and R. W. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, December 1987

[3]Basili, V. R., and R. W. Selby, Jr., "Four Applications of a Software Data Collection and Analysis Methodology," *Proceedings of the NATO Advanced Study Institute*, August 1985

[5]Basili, V. R., and R. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, December 1987

[9]Basili, V. R., and R. W. Selby, "Paradigms for Experimentation and Empirical Studies in Software Engineering," *Reliability Engineering and System Safety*, January 1991

[4]Basili, V. R., R. W. Selby, Jr., and D. H. Hutchens, "Experimentation in Software Engineering," *IEEE Transactions on Software Engineering*, July 1986

[2]Basili, V. R., R. W. Selby, and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," *IEEE Transactions on Software Engineering*, November 1983

[2]Basili, V. R., and D. M. Weiss, *A Methodology for Collecting Valid Software Engineering Data*, University of Maryland, Technical Report TR-1235, December 1982

[3]Basili, V. R., and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering*, November 1984

[1]Basili, V. R., and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives," *Proceedings of the Fifteenth Annual Conference on Computer Personnel Research*, August 1977

Basili, V. R., and M. V. Zelkowitz, "Designing a Software Measurement Experiment," *Proceedings of the Software Life Cycle Management Workshop*, September 1977

[1]Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," *Proceedings of the Second Software Life Cycle Management Workshop*, August 1978

[1]Basili, V. R., and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," *Computers and Structures*, August 1978, vol. 10

Basili, V. R., and M. V. Zelkowitz, "Analyzing Medium Scale Software Development," *Proceedings of the Third International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1978

[13]Basili, V., M. Zelkowitz, F. McGarry, G. Page, S. Waligora, and R. Pajerski, "SEL's Software Process-Improvement Program," *IEEE Software*, vol. 12, no. 6, November 1995, pp. 83–87

Bassman, M. J., F. McGarry, and R. Pajerski, *Software Measurement Guidebook*, NASA-GB-001-94, Software Engineering Program, July 1994

[9]Booth, E. W., and M. E. Stark, "Designing Configurable Software: COMPASS Implementation Concepts," *Proceedings of Tri-Ada 1991*, October 1991

[10]Booth, E. W., and M. E. Stark, "Software Engineering Laboratory Ada Performance Study—Results and Implications," *Proceedings of the Fourth Annual NASA Ada User's Symposium*, April 1992

[10]Briand, L. C., and V. R. Basili, "A Classification Procedure for the Effective Management of Changes During the Maintenance Process," *Proceedings of the 1992 IEEE Conference on Software Maintenance (CSM 92)*, November 1992

[10]Briand, L. C., V. R. Basili, and C. J. Hetmanski, "Providing an Empirical Basis for Optimizing the Verification and Testing Phases of Software Development," *Proceedings of the Third IEEE International Symposium on Software Reliability Engineering (ISSRE 92)*, October 1992

[11]Briand, L. C., V. R. Basili, and C. J. Hetmanski, *Developing Interpretable Models with Optimized Set Reduction for Identifying High Risk Software Components*, University of Maryland, Technical Report TR-3048, March 1993

^{12}Briand, L. C., V. R. Basili, Y. Kim, and D. R. Squier, "A Change Analysis Process to Characterize Software Maintenance Projects," *Proceedings of the International Conference on Software Maintenance*, Victoria, British Columbia, Canada, September 19–23, 1994, pp. 38–49

^{9}Briand, L. C., V. R. Basili, and W. M. Thomas, *A Pattern Recognition Approach for Software Engineering Data Analysis*, University of Maryland, Technical Report TR-2672, May 1991

^{13}Briand, L., W. Melo, C. Seaman, and V. Basili, "Characterizing and Assessing a Large-Scale Software Maintenance Organization," *Proceedings of the 17th International Conference on Software Engineering*, Seattle, Washington, U.S.A., April 23–30, 1995

^{11}Briand, L. C., S. Morasca, and V. R. Basili, "Measuring and Assessing Maintainability at the End of High Level Design," *Proceedings of the 1993 IEEE Conference on Software Maintenance (CSM 93)*, November 1993

^{12}Briand, L., S. Morasca, and V. R. Basili, *Defining and Validating High-Level Design Metrics*, University of Maryland, Computer Science Technical Report, CS-TR-3301, UMIACS-TR-94-75, June 1994

^{13}Briand, L., S. Morasca, and V. R. Basili, *Property-based Software Engineering Measurement*, University of Maryland, Computer Science Technical Report, CS-TR-3368, UMIACS-TR-94-119, November 1994

^{13}Briand, L., S. Morasca, and V. R. Basili, *Goal-Driven Definition of Product Metrics Based on Properties*, University of Maryland, Computer Science Technical Report, CS-TR-3346, UMIACS-TR-94-106, December 1994

^{11}Briand, L. C., W. M. Thomas, and C. J. Hetmanski, "Modeling and Managing Risk Early in Software Development," *Proceedings of the Fifteenth International Conference on Software Engineering (ICSE 93)*, May 1993

^{5}Brophy, C. E., W. W. Agresti, and V. R. Basili, "Lessons Learned in Use of Ada-Oriented Design Methods," *Proceedings of the Joint Ada Conference*, March 1987

^{6}Brophy, C. E., S. Godfrey, W. W. Agresti, and V. R. Basili, "Lessons Learned in the Implementation Phase of a Large Ada Project," *Proceedings of the Washington Ada Technical Conference*, March 1988

^{2}Card, D. N., "Early Estimation of Resource Expenditures and Program Size," Computer Sciences Corporation, Technical Memorandum, June 1982

^{2}Card, D. N., "Comparison of Regression Modeling Techniques for Resource Estimation," Computer Sciences Corporation, Technical Memorandum, November 1982

^{3}Card, D. N., "A Software Technology Evaluation Program," *Annais do XVIII Congresso Nacional de Informatica*, October 1985

[5]Card, D. N., and W. W. Agresti, "Resolving the Software Science Anomaly," *Journal of Systems and Software*, 1987

[6]Card, D. N., and W. W. Agresti, "Measuring Software Design Complexity," *Journal of Systems and Software*, June 1988

[4]Card, D. N., V. E. Church, and W. W. Agresti, "An Empirical Study of Software Design Practices," *IEEE Transactions on Software Engineering*, February 1986

Card, D. N., V. E. Church, W. W. Agresti, and Q. L. Jordan, "A Software Engineering View of Flight Dynamics Analysis System," Parts I and II, Computer Sciences Corporation, Technical Memorandum, February 1984

Card, D. N., Q. L. Jordan, and V. E. Church, "Characteristics of FORTRAN Modules," Computer Sciences Corporation, Technical Memorandum, June 1984

[5]Card, D. N., F. E. McGarry, and G. T. Page, "Evaluating Software Engineering Technologies," *IEEE Transactions on Software Engineering*, July 1987

[3]Card, D. N., G. T. Page, and F. E. McGarry, "Criteria for Software Modularization," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

[1]Chen, E., and M. V. Zelkowitz, "Use of Cluster Analysis To Evaluate Software Engineering Methodologies," *Proceedings of the Fifth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1981

[4]Church, V. E., D. N. Card, W. W. Agresti, and Q. L. Jordan, "An Approach for Assessing Software Prototypes," *ACM Software Engineering Notes*, July 1986

[2]Doerflinger, C. W., and V. R. Basili, "Monitoring Software Development Through Dynamic Variables," *Proceedings of the Seventh International Computer Software and Applications Conference*. New York: IEEE Computer Society Press, 1983

Doubleday, D., *ASAP: An Ada Static Source Code Analyzer Program*, University of Maryland, Technical Report TR-1895, August 1987 (NOTE: 100 pages long)

[6]Godfrey, S., and C. Brophy, "Experiences in the Implementation of a Large Ada Project," *Proceedings of the 1988 Washington Ada Symposium*, June 1988

[5]Jeffery, D. R., and V. Basili, *Characterizing Resource Data: A Model for Logical Association of Software Data*, University of Maryland, Technical Report TR-1848, May 1987

[6]Jeffery, D. R., and V. R. Basili, "Validating the TAME Resource Data Model," *Proceedings of the Tenth International Conference on Software Engineering*, April 1988

[11]Li, N. R., and M. V. Zelkowitz, "An Information Model for Use in Software Management Estimation and Prediction," *Proceedings of the Second International Conference on Information Knowledge Management*, November 1993

[5]Mark, L., and H. D. Rombach, *A Meta Information Base for Software Engineering*, University of Maryland, Technical Report TR-1765, July 1987

[6]Mark, L., and H. D. Rombach, "Generating Customized Software Engineering Information Bases From Software Process and Product Specifications," *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, January 1989

[5]McGarry, F. E., and W. W. Agresti, "Measuring Ada for Software Development in the Software Engineering Laboratory (SEL)," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988

[7]McGarry, F., L. Esker, and K. Quimby, "Evolution of Ada Technology in a Production Software Environment," *Proceedings of the Sixth Washington Ada Symposium (WADAS)*, June 1989

McGarry, F., R. Pajerski, G. Page, et al., *Software Process Improvement in the NASA Software Engineering Laboratory,* Carnegie-Mellon University, Software Engineering Insitute, Technical Report CMU/SEI-94-TR-22, ESC-TR-94-022, December 1994

[3]McGarry, F. E., J. Valett, and D. Hall, "Measuring the Impact of Computer Resource Quality on the Software Development Process and Product," *Proceedings of the Hawaiian International Conference on System Sciences*, January 1985

[3]Page, G., F. E. McGarry, and D. N. Card, "A Practical Experience With Independent Verification and Validation," *Proceedings of the Eighth International Computer Software and Applications Conference*, November 1984

[12]Porter, A. A., L. G. Votta, Jr., and V. R. Basili, *Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment*, University of Maryland, Technical Report TR-3327, July 1994

[5]Ramsey, C. L., and V. R. Basili, "An Evaluation of Expert Systems for Software Engineering Management," *IEEE Transactions on Software Engineering*, June 1989

[3]Ramsey, J., and V. R. Basili, "Analyzing the Test Process Using Structural Coverage," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

[5]Rombach, H. D., "A Controlled Experiment on the Impact of Software Structure on Maintainability," *IEEE Transactions on Software Engineering*, March 1987

[8]Rombach, H. D., "Design Measurement: Some Lessons Learned," *IEEE Software*, March 1990

[9]Rombach, H. D., "Software Reuse: A Key to the Maintenance Problem," *Butterworth Journal of Information and Software Technology*, January/February 1991

[6]Rombach, H. D., and V. R. Basili, "Quantitative Assessment of Maintenance: An Industrial Case Study," *Proceedings From the Conference on Software Maintenance*, September 1987

[6]Rombach, H. D., and L. Mark, "Software Process and Product Specifications: A Basis for Generating Customized SE Information Bases," *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, January 1989

[7]Rombach, H. D., and B. T. Ulery, *Establishing a Measurement Based Maintenance Improvement Program: Lessons Learned in the SEL*, University of Maryland, Technical Report TR-2252, May 1989

[10]Rombach, H. D., B. T. Ulery, and J. D. Valett, "Toward Full Life Cycle Control: Adding Maintenance Measurement to the SEL," *Journal of Systems and Software*, May 1992

[6]Seidewitz, E., "Object-Oriented Programming in Smalltalk and Ada," *Proceedings of the 1987 Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1987

[5]Seidewitz, E., "General Object-Oriented Software Development: Background and Experience," *Proceedings of the 21st Hawaii International Conference on System Sciences*, January 1988

[6]Seidewitz, E., "General Object-Oriented Software Development with Ada: A Life Cycle Approach," *Proceedings of the CASE Technology Conference*, April 1988

[9]Seidewitz, E., "Object-Oriented Programming Through Type Extension in Ada 9X," *Ada Letters*, March/April 1991

[10]Seidewitz, E., "Object-Oriented Programming With Mixins in Ada," *Ada Letters*, March/April 1992

[12]Seidewitz, E., "Genericity versus Inheritance Reconsidered: Self-Reference Using Generics," *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1994

[4]Seidewitz, E., and M. Stark, "Towards a General Object-Oriented Software Development Methodology," *Proceedings of the First International Symposium on Ada for the NASA Space Station*, June 1986

[9]Seidewitz, E., and M. Stark, "An Object-Oriented Approach to Parameterized Software in Ada," *Proceedings of the Eighth Washington Ada Symposium*, June 1991

[8]Stark, M., "On Designing Parametrized Systems Using Ada," *Proceedings of the Seventh Washington Ada Symposium*, June 1990

[11]Stark, M., "Impacts of Object-Oriented Technologies: Seven Years of SEL Studies," *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, September 1993*

[7]Stark, M. E. and E. W. Booth, "Using Ada to Maximize Verbatim Software Reuse," *Proceedings of TRI-Ada 1989*, October 1989

[5]Stark, M., and E. Seidewitz, "Towards a General Object-Oriented Ada Lifecycle," *Proceedings of the Joint Ada Conference*, March 1987

[13]Stark, M., and E. Seidewitz, "Generalized Support Software: Domain Analysis and Implementation," *Addendum to the Proceedings OOPSLA '94*, Ninth Annual Conference, Portland, Oregon, U.S.A., October 1994, pp. 8–13

[10]Straub, P. A., and M. V. Zelkowitz, "On the Nature of Bias and Defects in the Software Specification Process," *Proceedings of the Sixteenth International Computer Software and Applications Conference (COMPSAC 92)*, September 1992

[8]Straub, P. A., and M. V. Zelkowitz, "PUC: A Functional Specification Language for Ada," *Proceedings of the Tenth International Conference of the Chilean Computer Science Society*, July 1990

[7]Sunazuka, T., and V. R. Basili, *Integrating Automated Support for a Software Management Cycle Into the TAME System*, University of Maryland, Technical Report TR-2289, July 1989

[13]Thomas, W. M., A. Delis, and V. R. Basili, *An Analysis of Errors in a Reuse-Oriented Development Environment*, University of Maryland, Computer Science Technical Report, CS-TR-3424, UMIACS-TR-95-24, February 1995

[10]Tian, J., A. Porter, and M. V. Zelkowitz, "An Improved Classification Tree Analysis of High Cost Modules Based Upon an Axiomatic Definition of Complexity," *Proceedings of the Third IEEE International Symposium on Software Reliability Engineering (ISSRE 92)*, October 1992

Turner, C., and G. Caron, *A Comparison of RADC and NASA/SEL Software Development Data*, Data and Analysis Center for Software, Special Publication, May 1981

[10]Valett, J. D., "Automated Support for Experience-Based Software Management," *Proceedings of the Second Irvine Software Symposium (ISS _92)*, March 1992

[5]Valett, J. D., and F. E. McGarry, "A Summary of Software Measurement Experiences in the Software Engineering Laboratory," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988

[3]Weiss, D. M., and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data From the Software Engineering Laboratory," *IEEE Transactions on Software Engineering*, February 1985

[5]Wu, L., V. R. Basili, and K. Reed, "A Structure Coverage Tool for Ada Software Systems," *Proceedings of the Joint Ada Conference*, March 1987

[1]Zelkowitz, M. V., "Resource Estimation for Medium-Scale Software Projects," *Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science*. New York: IEEE Computer Society Press, 1979

[2]Zelkowitz, M. V., "Data Collection and Evaluation for Experimental Computer Science Research," *Empirical Foundations for Computer and Information Science* (Proceedings), November 1982

[6]Zelkowitz, M. V., "The Effectiveness of Software Prototyping: A Case Study," *Proceedings of the 26th Annual Technical Symposium of the Washington, D.C., Chapter of the ACM*, June 1987

[6]Zelkowitz, M. V., "Resource Utilization During Software Development," *Journal of Systems and Software*, 1988

[8]Zelkowitz, M. V., "Evolution Towards Specifications Environment: Experiences With Syntax Editors," *Information and Software Technology*, April 1990

NOTES:

[1] This article also appears in SEL-82-004, *Collected Software Engineering Papers: Volume I*, July 1982.

[2] This article also appears in SEL-83-003, *Collected Software Engineering Papers: Volume II*, November 1983.

[3] This article also appears in SEL-85-003, *Collected Software Engineering Papers: Volume III*, November 1985.

[4] This article also appears in SEL-86-004, *Collected Software Engineering Papers: Volume IV*, November 1986.

[5] This article also appears in SEL-87-009, *Collected Software Engineering Papers: Volume V*, November 1987.

[6] This article also appears in SEL-88-002, *Collected Software Engineering Papers: Volume VI*, November 1988.

[7] This article also appears in SEL-89-006, *Collected Software Engineering Papers: Volume VII*, November 1989.

[8] This article also appears in SEL-90-005, *Collected Software Engineering Papers: Volume VIII*, November 1990.

[9] This article also appears in SEL-91-005, *Collected Software Engineering Papers: Volume IX*, November 1991.

[10] This article also appears in SEL-92-003, *Collected Software Engineering Papers: Volume X*, November 1992.

[11] This article also appears in SEL-93-001, *Collected Software Engineering Papers: Volume XI*, November 1993.

[12] This article also appears in SEL-94-004, *Collected Software Engineering Papers: Volume XII*, November 1994.

[13] This article also appears in SEL-95-003, *Collected Software Engineering Papers: Volume XIII*, November 1995.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
November 1995

3. REPORT TYPE AND DATES COVERED
Technical Memorandum

4. TITLE AND SUBTITLE
Software Engineering Laboratory Series
Collected Software Engineering Papers: Volume XIII

5. FUNDING NUMBERS
Code 551

6. AUTHOR(S)
Flight Dynamics Systems Branch

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS (ES)
Goddard Space Flight Center
Greenbelt, Maryland 20771

8. PEFORMING ORGANIZATION REPORT NUMBER
SEL-95-003

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS (ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
TM—1998–208615

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Unclassified–Unlimited
Subject Category: 82
Report available from the NASA Center for AeroSpace Information,
7121 Standard Drive, Hanover, MD 21076-1320. (301) 621-0390.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Software Engineering Laboratory (SEL) is an organization sponsored by NASA/GSFC and created to investigate the effectiveness of software engineering technologies when applied to the development of application software.

The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document.

14. SUBJECT TERMS
Software Engineering Laboratory,
Application software, Documentation

15. NUMBER OF PAGES
276

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT
Unclassified

20. LIMITATION OF ABSTRACT
UL