Y SERIES

SEL-89-006

D SOFTWARE
PERS: VOLUME VII

ER 1989

SOFTWARE ENGINEERING LABORATORY SERIES SEL-89-006

COLLECTED SOFTWARE ENGINEERING PAPERS: VOLUME VII

NOVEMBER 1989

NASA
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 for the purpose of investigating the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1977 and has three primary organizational members:

NASA/GSFC (Systems Development Branch)

The University of Maryland (Computer Sciences Department)

Computer Sciences Corporation (Systems Development 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. The papers contained in this document appeared previously as indicated in each section.

Single copies of this document can be obtained by writing to

Systems Development Branch
Code 552
NASA/GSFC
Greenbelt, Maryland 20771

TABLE OF CONTENTS

SECTION 1—INTRODUCTION

SECTION 1 - INTRODUCTION

This document is a collection of selected technical papers produced by participants in the Software Engineering Laboratory (SEL) during the period December 1988, through October 1989. 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 seventh 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 seven papers contained here are grouped into three major categories:

- Software Measurement and Technology Studies
- Measurement Environment Studies
- Ada Technology Studies

The first category presents experimental research and evaluation of software measurement and technology; the second presents studies on software environments pertaining to measurement. The last category represents Ada technology and includes research, development, and measurement studies.

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

SECTION 2—SOFTWARE MEASUREMENT AND TECHNOLOGY STUDIES

SECTION 2 - SOFTWARE MEASUREMENT AND TECHNOLOGY STUDIES

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

- Establishing a Measurement Based Maintenance Improvement Program: Lessons Learned in the SEL, H. Rombach and B. Ulery, University of Maryland, Technical Report TR-2252, May 1989
- Maintenance = Reuse-Oriented Software Development, V. Basili, University of Maryland, Technical Report TR-2244, May 1989
- Software Development: A Paradigm for the Future, V. Basili, University of Maryland, Technical Report TR-2263, June 1989

UMIACS-TR-89-54
CS-TR-2252

May, 1989

Establishing a Measurement Based Maintenance Improvement Program: Lessons Learned in the SEL

H. Dieter Rombach[†]
Institute for Advanced Computer Studies
Department of Computer Science

Bradford T. Ulery
Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

The Software Engineering Laboratory (SEL) is a joint venture between NASA's Goddard Space Flight Center, the University of Maryland, and Computer Sciences Corporation. We discuss the use of a goal oriented approach to measurement to establish a maintenance improvement program within the SEL. Differences are found to exist between the initial phase of the program and its routine application. We demonstrate our approach through concrete examples, and summarize lessons we have learned in the establishment of a measurement based, maintenance improvement program.

[†] This work was supported by the NASA grant NSG-5123.

This paper will also appear in the Proceedings of the Conference on Software Maintenance 1989

1. INTRODUCTION

The Software Engineering Laboratory (SEL) is an organization sponsored by the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) and created for the purpose of investigating the effectiveness of software engineering technologies. The SEL was created in 1977 and has three primary organizational members: NASA/GSFC, the University of Maryland, and Computer Sciences Corporation.

NASA/GSFC develops ground control software systems and other support software for satellites. A large number of case studies and controlled experiments have been conducted in the past that have resulted in evolutionary changes to NASA's development practices [Basili85,McGarry85]. Some of the changes include the stricter use of code reading techniques [Basili87], the use of measurement baselines for management purposes [Ramsey88], measurement based recommendations for Ada projects [Brophy87], and most recently the experimental adoption of the CLEANROOM development method [Selby87].

In order to formalize the procedures for investigating software technologies and to organize the resulting experience, three paradigms have been defined for goal oriented measurement:

(1) **Goal/question/metric** [Basili84,Basili85b]. This paradigm is based on the principle that effective measurement procedures should be derived (top-down) from goals. The GQM paradigm suggests that measurement needs to start with a precise specification of the goals, continue with the refinement of each goal into a set of quantifiable questions, and end with the derivation of a set of metrics. This approach yields a rationale for any chosen set of metrics all the way back to the original goals. Therefore, it also provides a basis for the goal oriented interpretation of collected data.

(2) **Evaluation** [Basili84]. This paradigm is based on the additional principle that the measurement process must be designed to fit the production environment. The evaluation paradigm simply extends the GQM paradigm by including the actual measurement

procedures. These procedures must be tailored to the product or process being studied in order to obtain valid results. Previous publications also refer to this as the GQM paradigm.

(3) **Improvement** [Basili85b,Basili88]. This paradigm is based on the principle that improvement is based on continuous learning. The improvement paradigm provides the context for evaluating multiple projects. It emphasizes recording what has been learned through measurement so that this knowledge will be available when it is needed. Knowledge is managed explicitly by modeling the environment and providing feedback from analysis to production.

Experience from several studies has reaffirmed these principles and our belief in the effectiveness of goal oriented measurement whether in a production environment or an experiment. This experience has led to the formulation of specific measurement guidelines and goal templates [Basili88].

A number of improvement programs have been established based on the improvement paradigm [Grady87,Rombach87,Basili85,McGarry85]. Most of the published results, however, address specific studies performed under such programs rather than the establishment of the programs themselves. Examples include industrial case studies [Rombach87,Basili87b,Weiss85,Basili87] as well as academic experiments [Nehmer87,Katz86,Rombach86]. Two noteworthy exceptions address the managerial and technology transfer problems associated with the establishment of such a study [Brelsford88,Grady87b].

An organization's long-term commitment to invest in such a program depends on whether the potential for future payoff can be demonstrated convincingly. We distinguish between the "initial program phase" aimed at establishing an effective program and demonstrating its payoff potential, and the "routine program phase" aimed at applying an accepted program to routine projects.

The purpose of the initial phase is to understand the environment sufficiently to identify high leverage improvement goals and to establish a proper measurement infrastructure (procedures, managerial commitment, tools, personnel, *etc.*). The environment should be modeled explicitly and limited measurement may be required in order to demonstrate the potential leverage of the stated goals and the feasibility of the procedures. It is important to recognize that this initial phase represents an investment. What is learned in this phase provides the foundation for improvements during the routine phase.

This paper reports on our experience from establishing an improvement program for maintenance in the SEL. Section 2 summarizes the approach used to create the initial understanding of the environment, the improvement goals and measurement procedures. Section 3 characterizes the current status of our program. Sections 4 and 5 highlight some important lessons, and outline future SEL maintenance improvement activities.

2. MAINTENANCE IMPROVEMENT APPROACH

Within the SEL, development and maintenance are performed by separate organizational units. In late 1987, measurement of maintenance was included in the scope of the SEL in order to better understand and eventually improve the software life-cycle. At that time there was little documentation of the actual maintenance procedures (beyond some general guidelines) on which to base our initial analysis. Nor was there any explicit feedback to developers about the product maintainability, or the types and amounts of maintenance required.

Based on past experience, we were confident that the guidelines supporting the improvement paradigm would be helpful during the initial program phase. We had, however, only vague ideas as to how these methods should be applied given the lack of explicitly documented experience in the SEL maintenance environment that was available when we began. We expected to learn about the strengths and weaknesses of the improvement approach in a situation where we could not select improvement goals or design measurement procedures based on a mature understanding of the environment, but rather would have to initially bootstrap that understanding.

The improvement program established for maintenance in the SEL is based on a version of the improvement paradigm applied to maintenance [Rombach88]. This paradigm (see Fig. 1) suggests that maintenance can be improved by iterating the following steps for each project: (1) characterize the corporate maintenance environment; (2) state improvement goals in quantitative terms; (3) plan the appropriate maintenance and measurement procedures for the project at hand; (4) perform maintenance, measure, analyze and provide feedback; and (5) perform post mortem analysis and provide recommendations for future projects.

I1. Characterize the corporate maintenance environment

I2. State improvement goals
- a. State improvement goals informally
- b. Specify related measurement goals

I3. Plan maintenance
- a. Plan appropriate maintenance process
- b. Plan appropriate measurement process

I4. Perform maintenance
- a. Perform maintenance process
- b. Perform measurement process
- c. Analyze collected data and provide immediate feedback

I5. Perform post-mortem analysis and provide recommendations for future projects

I6. Return to step I1

Figure 1: The improvement paradigm applied to maintenance.

We applied the principles of the paradigms strictly. However, during the initial phase, our understanding of the environment, goals, and measurement procedures did not develop according to a straightforward sequential application of the first three steps of the improvement paradigm. Nor were all supporting metrics identified by a strictly top–down application of the GQM paradigm. There are two good reasons for not following these steps: (i) we sometimes discover that our knowledge of prior steps is inadequate, so we retrace our steps, or (ii) practical constraints (such as existing data collection forms) preclude a strictly top–down derivation of procedures.

The initial uncertainty in our understanding of the maintenance environment made it necessary to allow for planned and *ad hoc* feedback loops at any time. Such feedback loops resulted in revisions of the goals and measurement procedures. Measurement procedures were validated by actually applying them to real projects on a trial basis. The experience from such trial data collection, validation, and analysis helped us to further improve our understanding of the environment, and provided objective data to demonstrate the existence of suspected maintenance problems. Demonstrating the feasibility of planned measurement procedures on a trial basis has won confidence in their potential to support the improvement goals.

In summary, a number of quick (and sometimes partial) iterations through the improvement paradigm eventually resulted in our current status. Based on this status the SEL has reached a consensus that routinely applying this improvement program to all future maintenance projects is worthwhile.

3. PROGRAM STATUS

This section presents the current status of our maintenance improvement program according to the outline of the improvement paradigm (Fig. 1). Section 3.1 summarizes our current understanding of the SEL maintenance environment (corresponding to improvement step I1). Selected improvement goals and their supporting data collection, and validation procedures are summarized in sections 3.2. and 3.3 (corresponding to steps I2 and I3.b). Initial measurement data are presented and interpreted in sections 3.4 and 3.5 (corresponding to steps I4.b and I4.c). This form of presentation is intended to demonstrate the use of the improvement paradigm during this initial phase.

3.1. ENVIRONMENT

We characterize the SEL maintenance environment in terms of the application, maintained products, and maintenance process.

Application

Two missions in this study are the Cosmic Background Explorer (COBE) and the Gamma Ray Observatory (GRO). They are tentatively scheduled to be launched in July, 1989, and April, 1990, respectively. COBE's scientific mission is to investigate the origins of the universe. GRO will make observations over the energy range from .1 to 30,000 MeV.

The Earth Radiation Budget Satellite (ERBS) was launched from Space Shuttle Challenger in October, 1984. ERBS carried the Earth Radiation Budget Experiment (ERBE) and the Stratospheric Aerosol and Gas Experiment (SAGE)-II. Measurements from these experiments are used to understand the earth's climate and how environmental factors affect it.

Largely because of the Challenger disaster in January, 1986, COBE will be the first mission to which the Flight Dynamics Division of NASA/GSFC has contributed significantly since ERBS.

Maintained Products

To date, we have monitored five projects representing each of the following three major types of systems developed in the SEL environment.

(1) Attitude Ground Support Systems (AGSS) provide operational support for a mission. Their functions include determining spacecraft attitude from telemetry data, verifying the on-board computer's attitude determination and control, supporting star tracking (for guidance), and more.

(2) Attitude Telemetry Data Simulator Systems produce realistic attitude telemetry and engineering data files to exercise the algorithms and processing capabilities of AGSS's. Telemetry data includes essentially everything the spacecraft knows and could report back.

(3) Attitude Dynamics Simulator Systems are analytic tools for testing and evaluating (two subsystems of) the spacecraft simulators. They simulate the environment of the spacecraft, sensor data, the on-board computer's response (actuator commands), and the resulting control torques in order to model the spacecraft dynamics.

In addition to the many numerical algorithms, each of these systems manages a user interface including the control of parameters, reading large data files, and printing tables and plots.

Maintenance Process

In order to understand the role of maintenance in this environment, why changes occur and who could benefit from our observations, and in order to design effective measurement procedures, we have modeled the software life-cycle (Fig. 2). The Flight Dynamics Division of NASA/GSFC is divided into four branches, three of which are included in our model. Note that communications crossing organizational boundaries tend to be more formal and occur less frequently than internal communications.

Project Requirements are received by the Specification Developers. The Specification Developers produce Functional Specifications and Mathematical Derivations for use by the

Requirements Analysts. The Requirements Analysts write the Preliminary Design which includes the high level system architecture. This design is used by the Software Developers who produce Code, a User's Guide, and a System Description. The System Description summarizes the Preliminary and Detailed Designs. The system is then submitted to Acceptance Testing. Upon acceptance, the system is eventually passed to Maintenance. Maintainers are responsible for the system until about three months after launch. If the system is an AGSS, its routine Operation is handled by Operations Support. Dynamics simulators and telemetry simulators are not passed on to Operations Support. Although changes frequently occur immediately after a launch, they are reportedly quite infrequent during Operational Support.

During maintenance, each change is formally defined by an Operational Software Modification Report (OSMR), a form that specifies the change, and then follows it, gathering dates and signatures as the change is approved, implemented, tested, installed, *etc.* Typically there are more outstanding OSMRs than resources. A Project Task Leader is responsible for allocating these resources.

OSMRs may be filed for several reasons. Acceptance Testing may reveal the need for enhancements (corrections are still the responsibility of the Software Developers). Later the Users (same organization unit as maintainers) may request enhancements or identify the need for corrections or adaptations. The Specification Developers may also initiate changes, resulting from ideas about similar forthcoming systems. Or, the Project Office may modify the Project Requirements.

There are three software libraries: Working, Testing, and Operational. Entries in the Working library have not yet been accepted and may not be final. Several changes are made to the Testing library at once. These changes are tested together. The Operational library is very stable. Three months prior to launch, the Operational library is frozen. Maintenance nevertheless continues (the freeze is lifted after launch).

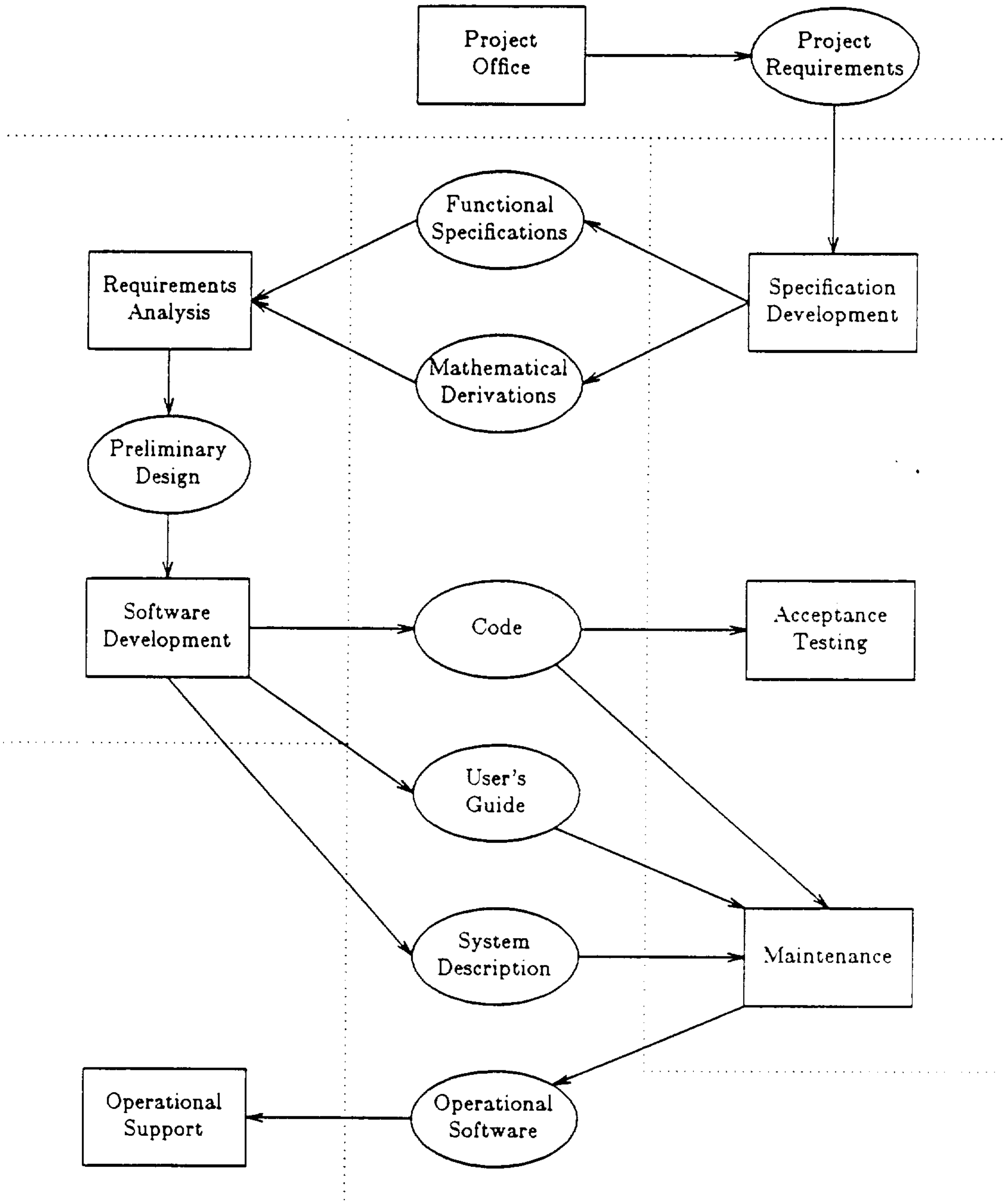


Figure 2. The software life-cycle in the NASA/GSFC Flight Dynamics Division. Organizational units are separated by the dotted lines.

Not all projects conform to the general models, but the models provide a common reference for tailoring procedures to specific projects and making comparisons or generalizations across projects.

3.2. MAINTENANCE IMPROVEMENT GOALS

We have identified a set of important improvement goals and refined them into quantifiable questions and metrics according to the GQM paradigm. This subsection presents some representative goals and questions. We have not included the complete set of goals, questions and metrics according to the GQM templates suggested in [Basili88], but only highlight selected ones. In the questions that refer to specific metrics, the metrics are italicized.

Characterizing maintenance

We study the maintenance process itself to see how maintainers spend their time and what they do. We study the entire software life-cycle to understand how Specification Developers, Software Developers, Users, and Maintainers communicate; why changes are made; and whether the organizational divisions result in the best use of personnel's skills and knowledge.

Currently we are interested in the following questions:

(1) What are the major *maintenance activities*? What are the major *software life-cycle activities*? What are the major *communication links* between activities?

(2) How is productivity related to *types of changes* (corrections, enhancements, adaptations), and characteristics of the product (*type of product, LOC*)? How is effort (*hours*) distributed across various activities?

Characterizing the delivered product

The quality of the delivered product influences both what changes will be performed and the amount of effort that will be required. We therefore characterize the products with the following objectives in mind: to understand how and why the product changes; to understand how the product influences productivity during a change; and to provide historical. baseline data for future projects.

We are currently interested in characterizing each of the three types of software products in terms of the following:

(3) What are the static characteristics of each product (*#LOC, #components, system architecture, programming language, types of documents*)? What are the functional characteristics of each?

(4) What types of changes are made (in terms of how both static and functional characteristics of the system are affected).

Improving maintenance

The maintenance process can be improved by focusing on the maintenance activities themselves, or by improving the entire software life-cycle of which maintenance is a part.

We are currently interested in the following specific possibilities:

(5) Establishing communication from Maintainers to Software Developers for feedback about product maintainability.

(6) Providing management with better mechanisms for monitoring the process.

Improving the developed products

Because of the relatively short maintenance phase, improvements to the products will be directed primarily toward the development of future products.

The following ideas have been proposed for improving the product:

(7) The debug code is not well designed from the Maintainers' perspective. Future designs should allow the Maintainers more control over which messages are turned on.

(8) We are trying to learn more about the relation between system structure and the locality of changes. A significant number of changes affect five or more files.

3.3. DATA COLLECTION AND VALIDATION PROCEDURES

The measurement procedures presented in this subsection support the stated improvement goals within the SEL maintenance environment. These procedures include data collection, their validation, analysis, and feedback.

We routinely monitor the effort associated with various maintenance activities, and other characteristics of the changes. Similar data is available from development. This data will be used to characterize the maintenance process, the types of changes made to the product, and the reasons for making the changes.

Routine data collection is implemented primarily through the use of forms (Fig. 3). At the end of each week, project personnel each complete a Weekly Maintenance Effort Form (WMEF) which briefly summarizes how they spent their time according to type of changes (correction, enhancement, adaptation, or other) and maintenance activity (isolation, implementation, unit test, integration test, other). Upon completion of each change, a Maintenance Change Report Form (MCRF, Fig. 4) is filed. The MCRF summarizes the change from a user's perspective (reason for change and functionality) and from the programmer's perspective (effort spent, parts of the system modified, *etc.*). A history of development (phase dates, effort) and product characteristics (size, number of subsystems, *etc.*) is summarized on a Project Completion Statistics Form (PCSF). This data will be made available at the end of development. It will be compiled through the use of programs which examine the software library and the SEL database.

The data collection forms and procedures reflect our models of the environment, the maintenance process and products, and measurement procedures. For example, the WMEF is filed only by Maintainers (as defined in Fig. 2); Specification Development and Use do not contribute to the hours monitored (although these hours are charged to the project); and Changed Objects (MCRF, section B) refers to specific documents used in this environment.

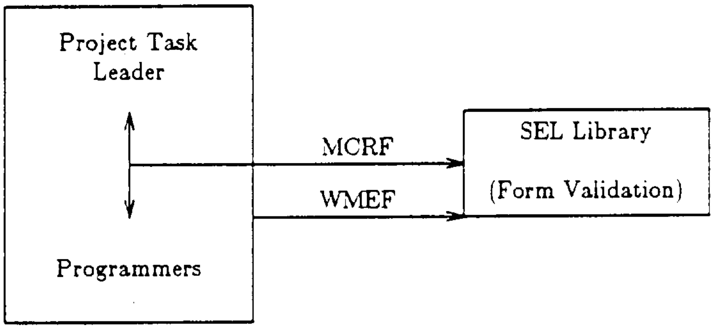
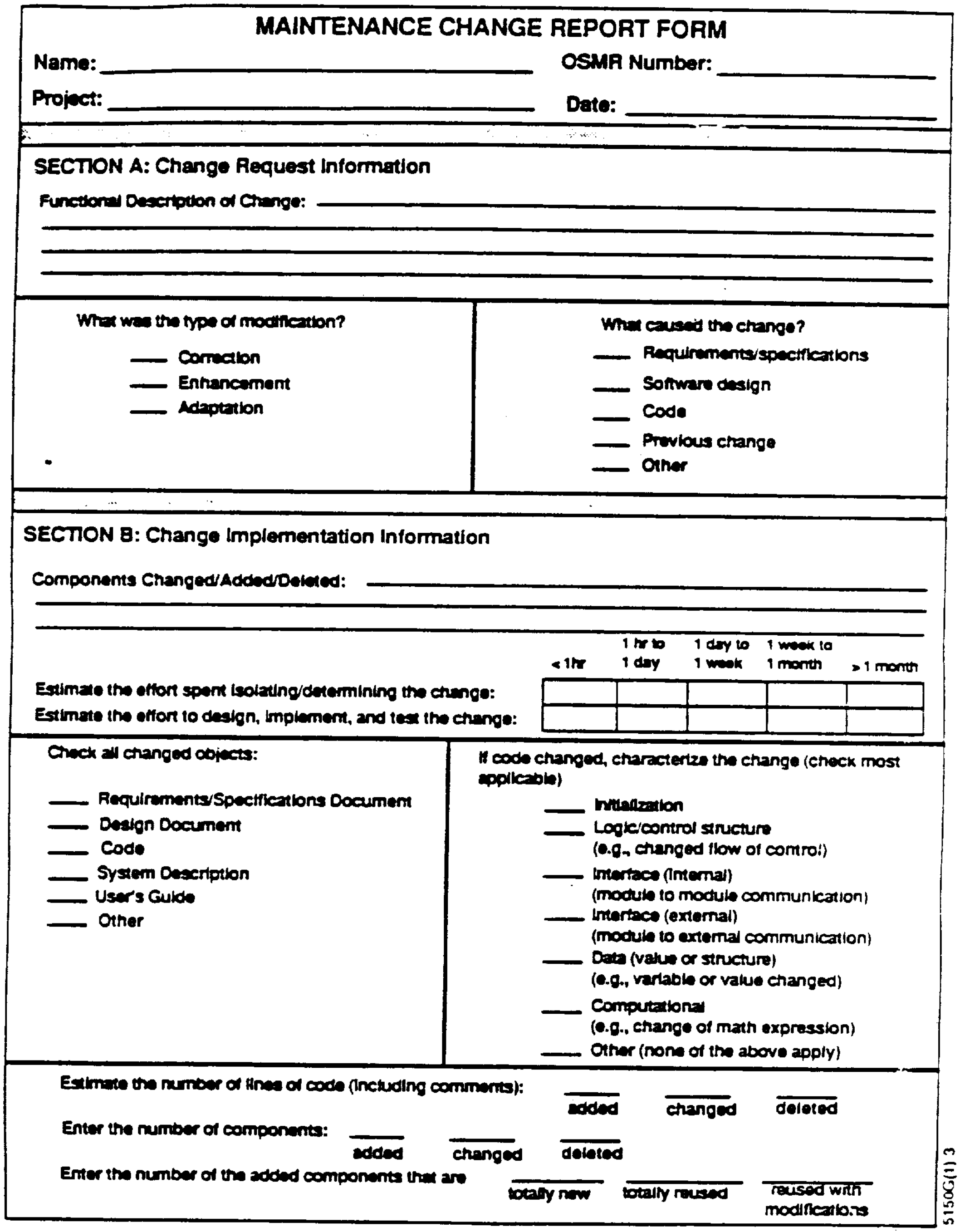


Figure 3. Routine data collection and validation procedures. WMEF is collected weekly from all project personnel (Project Task Leaders. Maintainers. Managers). MCRF is collected once for each change. These forms are validated. analyzed, and stored in the SEL, an independent entity.

ORIGINAL PAGE IS
OF POOR QUALITY



MAINTENANCE CHANGE REPORT FORM

Name: ____________________ OSMR Number: ____________

Project: ____________________ Date: ____________

SECTION A: Change Request Information

Functional Description of Change: ______________________________

____

____

____

What was the type of modification?

___ Correction
___ Enhancement
___ Adaptation

What caused the change?

___ Requirements/specifications
___ Software design
___ Code
___ Previous change
___ Other

SECTION B: Change Implementation Information

Components Changed/Added/Deleted: ______________________________

____

____

	<1hr	1 hr to 1 day	1 day to 1 week	1 week to 1 month	>1 month
Estimate the effort spent isolating/determining the change:					
Estimate the effort to design, implement, and test the change:					

Check all changed objects:

___ Requirements/Specifications Document
___ Design Document
___ Code
___ System Description
___ User's Guide
___ Other

If code changed, characterize the change (check most applicable)

___ Initialization
___ Logic/control structure
(e.g., changed flow of control)
___ Interface (internal)
(module to module communication)
___ Interface (external)
(module to external communication)
___ Data (value or structure)
(e.g., variable or value changed)
___ Computational
(e.g., change of math expression)
___ Other (none of the above apply)

Estimate the number of lines of code (including comments): ____ added ____ changed ____ deleted

Enter the number of components: ____ added ____ changed ____ deleted

Enter the number of the added components that are ____ totally new ____ totally reused ____ reused with modifications

5150G(1) 3

Figure 4. The MCRF. A change report summarizes the change from two perspectives: the functional perspective ("black box"), and the structural perspective ("white box").

3.4. COLLECTED MAINTENANCE DATA

We began monitoring maintenance projects on a trial basis in October, 1987 (Fig. 5). Our initial understanding of the environment reflected many biases from our knowledge of the Software Development process. Prior to October, 1988, the MCRF had emphasized corrective maintenance, did not request separate functional and structural descriptions of changes, and the OSMRs were not monitored. Other minor revisions were also made to the forms and procedures. The latest revision was made in January, 1989. Figure 6 summarizes the number of forms filed in total across the various projects.

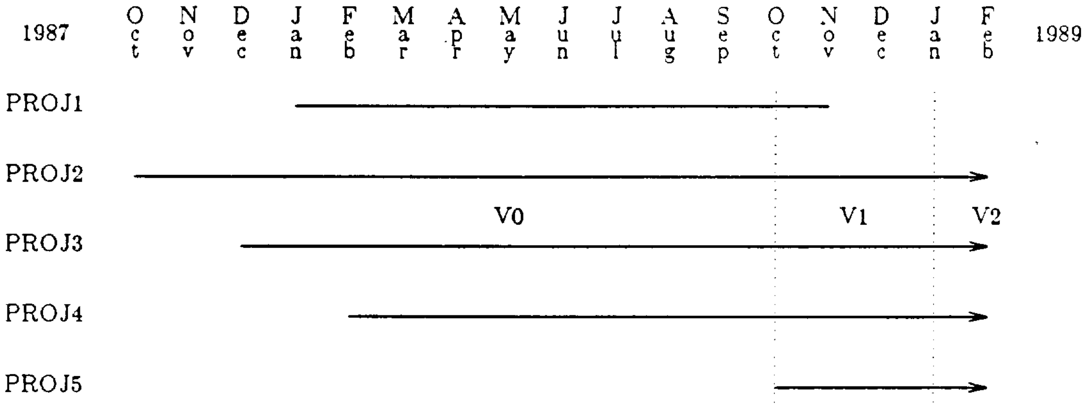


Figure 5. Five maintenance projects have been monitored on a trial basis since October, 1987. Two changes to the data collection forms took place in October, 1988, and January, 1989.

	PROJ1	PROJ2	PROJ3	PROJ4	PROJ5
MCRFs	71	18	12	0	0
WMEFs	266	86	28	15	32

Figure 6. Forms received. PROJ3 and PROJ4 have not been continuously active.

01/08/88	*			****				*
01/15/88	*			****				*
01/22/88	*	*	*	****				*
01/29/88				****				*
02/05/88	*	*	*	****				*
02/12/88	*	*	*	****				*
02/19/88	*		*	***				*
02/26/88	*		*	***				*
03/04/88	*		*	***				**
03/11/88	*			****				*
03/18/88	*	*	*	*****				*
03/25/88	*			****				*
04/01/88	*			***				*
04/08/88	*			****				*
04/15/88	*	*	*	****				*
04/22/88	*	0	*	****				*
04/29/88	**	*		***				*
05/06/88	**	*	*	****				*
05/13/88	*	*	*					*
05/20/88	*	*	*	*****	**	****		*
05/27/88	**	*	**	****	**	****		*
06/03/88	*	*	*	****	**	**		*
06/10/88	*	*	*	****	**	****	****	*
06/17/88	*	*	*	****	**	****	****	*
06/24/88	*	*	*	*****	*	****	****	*
07/01/88	*	*	*	***	***	****	****	0
07/08/88	*	*	0	****	*	****	****	*
07/15/88	0	0	0	****	**	****	****	*
07/22/88	0	0	0	**	****	****	*****	*
07/29/88	*	0	0	*	****	****	*****	*
08/05/88	0		0	**	***	****	****	*
08/12/88	*		0	**	***	****	*****	0
08/19/88	*	0	0	0	***	****	*****	*
08/26/88	*		*	***	**	*	****	*
09/02/88		***	*	**	*	***	****	*
09/09/88			*	0	*	*	***	*
09/16/88			*	*	*	*	****	0
09/23/88			*	*	*	0		*
09/30/88			*		0	0	0	*
10/07/88			*	*	*	0		*
10/14/88			*	0	0	*	0	0
10/21/88			*	*	0	*	0	0
10/28/88			0	0	0	*	0	0
11/04/88			0	***	0	*	0	0
DATE	N1	N2	N3	P4	P5	P6	P7	M8

0	= 0 hours (form submitted)
*	= up to 10 hours
**	= up to 20 hours
***	= up to 30 hours
****	= up to 40 hours
*****	= up to 50 hours

For each week of the project, total expended effort is shown distributed over the various personnel. On this project, Maintenance was contracted out to CSC (CSC manager: M8; CSC programmers: P4 - P7). Although NASA personnel (N1 - N3) were primarily responsible for Specification Development, some hours (meetings, consulting, *etc.*) were attributed to Maintenance.

Figure 7. Weekly Effort Histograms (PROJ1)

Real–time analysis of the data is not yet used by the Project Tasks Leaders or managers, but figure 7 suggests one way effort might be monitored. It shows for each week how much total effort has been invested by the various personnel.

Figures 8 through 10 profile some overall trends. The FORTRAN subroutines in these products are not small, therefore entire components are seldom added or deleted.

MCRF	Total LOC	LOC Added	LOC Deleted	LOC Changed	Total Files	Files Added	Files Deleted	Files Changed
PROJ1	81K	2484	430	1353	?	3	3	335
PROJ2	46K	2323	325	354	433	10	0	107
PROJ3	52K	10	0	55	242	0	0	0
PROJ4	37K	0	0	0	322	0	0	0
PROJ5	176K	0	0	0	?	0	0	0

Figure 8. This table summarizes data from section B of the MCRF. The totals refer to size at delivery ("?" refers to unavailable PCSF data). Files (usually single subroutines) are called "components" on the MCRF.

3.5. INITIAL ANALYSIS RESULTS

In the initial phase of improvement, analysis results frequently reveal limitations in the measurement procedures and forms as well as the naïveté of early goals and questions. Obviously, revealing these limitations and misconceptions is the first step toward improvement. The following examples demonstrate how one's understanding and models develop through the analysis of data. These analyses are based on the current goals, questions, and data.

(1) How expensive is maintenance compared to development?

So far the costs have been low compared to figures often quoted in research literature (Fig. 9). There are two very good reasons for this. First, maintenance, as defined in this environment, does not include Operational Support (during which software changes are presumed infrequent). Second, excepting PROJ1, none of the projects has completed. Also note that time spent in Specification Development (during the Maintenance phase) is not included.

(2) What types of changes are made?

Figure 10 shows the distribution of time spent on various types of maintenance over each of the projects. There are a few significant limitations to this data which make most generalizations premature: 1) the "other type" category was not included on the WMEF until October, 1988; time spent on meetings and management was therefore forced into one of the available categories; 2) there is little total data from some projects; 3) PROJ1 was "maintained" by the original Software Developers; 4) there is no data summarizing those change requests which were not implemented.

	Development Hours	Maintenance Hours
PROJ1	17K	3K
PROJ2	18K	2K
PROJ3	6K	0.2K
PROJ4	12K	0 1K
PROJ5	47K	0.5K

Figure 9. A comparison of total technical and management hours. (PROJ2 was in maintenance before Oct, 1987 when measurement of maintenance began.)

	Total	Correction	Enhancement	Adaptation	Other*
PROJ1	3286	18%	67%	13%	1%
PROJ2	2328	44%	54%	0%	0%
PROJ3	234	58%	35%	3%	3%
PROJ4	132	0%	87%	3%	10%
PROJ5	459	10%	78%	2%	10%

	Total Hours	Isolation	Change Design	Implement	Unit Test	Integration Test
PROJ1	3286	19%	12%	22%	12%	14%
PROJ2	2328	15%	15%	19%	6%	30%
PROJ3	234	53%	12%	23%	6%	4%
PROJ4	132	45%	15%	24%	15%	0%
PROJ5	459	15%	10%	15%	4%	46%

Figure 10. These two tables show the distributions of effort (WMEF) by type of change and activity.

* "Other" was only recently added to section B of the WMEF. Most of the 13% adaptation on PROJ1 actually represents management.

4. LESSONS LEARNED

The lessons learned from our efforts to establish the improvement program for SEL's maintenance environment address (i) why the introduction of measurement is significant, (ii) how well the improvement approach worked, (iii) how we built the credibility of our program, and (iv) automated support.

4.1. WHY IS THE INTRODUCTION OF MEASUREMENT SIGNIFICANT?

It was again apparent from this study that the introduction of measurement into an industrial setting represents not just the introduction of another method. Instead it signals a dramatic change in the organization toward a more engineering oriented software development and maintenance style. Such a change affects all levels of an organization. Most organizations are not ready for this kind of change. As a consequence, the initial phase of the improvement program must be sensitive to the need of selling measurement as a credible and promising mechanism.

Specific lessons learned:

L1: Introducing measurement represents a major shift toward a more engineering oriented software development and maintenance style.

L2: Most environments are not ready for systematic, measurement based improvement.

L3: Special effort must be made to build the credibility of the selected improvement goals and measurement procedures before measurement is attempted on a large scale.

The SEL adopted measurement as a means for routine improvement of its development activities over a decade ago. Still, during the introduction of measurement to the SEL's maintenance activities we encountered the need for selling measurement to a new audience. Initially, it was not clear whether we should aim to reduce the need for maintenance through better Specification Development, making a more maintainable product, or using more stringent acceptance testing. Our current goals were influenced as much by management's receptiveness to our ideas as by our technical understanding of the maintenance process.

4.2. HOW WELL DID THE IMPROVEMENT APPROACH WORK DURING THE INITIAL PHASE?

It became apparent again from this study that the basic principles of the improvement paradigm not only apply to the initial program phase; they are even more important for organizing learning the higher the level of uncertainty is. On the other hand, varying levels of understanding of the environment seem to justify (even require) different procedures for applying the paradigm.

Specific lessons learned:

L4: It is important to distinguish improvement methodologies (which depend on the environment and the maturity of the program) from principles of the improvement paradigm (which emphasize explicit learning through the use of measurement). As expected, the improvement paradigm was extremely helpful during the initial phase; however, the steps taken had to be modified according to the maturity of the program and the need to demonstrate its value.

L5: Most initial learning came from exploratory investigations (e.g., meetings, interviews) and was based on subjective and intuitive data.

In the initial phase, we learned more from attempting to implement the measurement procedures than from the data they provided. For example, many of the environment problems were first revealed to us in exploratory meetings with maintenance personnel. Those meetings provided the focus which enabled us to follow up on some of these issues in a much more goal oriented fashion (including the use of measurement data). Subjective data are very important during the initial program phase when our understanding does not allow for the definition of objective metrics, or when the underlying goal does not require or justify the cost of collecting objective data.

Our application of the improvement paradigm can be characterized as prototyping. It allowed for feedback loops at any time. Sometimes our understanding of the environment was improved when refining goals into questions, metrics, and measurement procedures. An example of this is how we learned about the maintenance libraries. Given the use of formal change requests, we assumed that changes were well defined and that upon completion of a change, a corresponding change form (MCRF) would be filed. Eventually we learned that changes were being made (the hours showed on weekly forms), but we were not receiving MCRFs. This inconsistency was only obvious when a new project started. "Completed" changes were being entered in the working library, but forms were not filed until after several changes were made, transferred to the test library, tested and approved. Although, we sometimes identified interesting metrics based on prior experience or intuition, we always eventually justified such metrics in the context of some improvement goal.

4.3. HOW DID WE BUILD THE CREDIBILITY OF OUR PROGRAM?

Although the actual collection and interpretation of data was not the objective of the initial program phase, we used it on a trial basis in order to design effective measurement procedures and to identify further needs for improvement. It is very hard to convince anyone that you are focusing on the right problems without actually providing some objective evidence that those problems actually exist. It is also difficult to convince someone that a program will be effective without demonstrating that the planned measurement procedures can be implemented in the current environment. The techniques used to establish procedures and demonstrate their credibility are important to the success of the program.

Specific lessons learned:

L6: The environment needs to be modeled at a level of detail that enables us to demonstrate that (i) the chosen goals are justified, (ii) the derived metrics support those goals, and (iii) the planned measurement procedures can be implemented in the given environment.

L7: Trial data collection and validation may be needed to establish confidence that planned measurement procedures are effective and that the identified goals address significant problems.

L8: Depending on the initial level of understanding, data collection may be less accurate, objective, and complete during the initial phase than during routine application.

L9: The participation of all people concerned adds to the credibility of the program.

We involved representatives from several organizational levels. They helped build initial hypotheses based on their insights, and served as reviewers of the results produced. Their comments were solicited, because without their confidence that we were addressing the right issues in a feasible way, we could not expect their cooperation during actual data collection.

4.4. HOW MUCH AUTOMATED SUPPORT iS NECESSARY?

This study also illustrated the need for automated support. During the initial phase, simple tools are needed for storing and analyzing the measurement data. A database system, statistics package, and report generator will become more important as the measurement procedures stabilize and the volume of data increases. Tools for modeling and planning could be very helpful provided they effectively support change.

Specific lessons learned:

L10: Automated support for measurement is less important during the initial phase, but will be required during the routine phase.

L11: Graphical models (*e.g.*, Fig. 2), plans (*e.g.*, relating goals, questions, and metrics), and raw data are unwieldy and numerous. In addition to a database, automated support for designing and managing graphical structures would be extremely useful.

During the initial phase, data collection forms were not stored in a database but simply in folders, and could not be analyzed automatically. As a result it was not always easy to keep track of forms and we might have lost some. This does not cause severe problems during the initial program phase, but would during the routine phase when reliance on this data is greater. We are currently in the process of expanding the SEL development database to include maintenance data.

5. CONCLUSIONS

In this study we have followed the principles of the improvement paradigm while introducing an improvement program to the SEL maintenance environment. We have observed that unlike a well conceived experiment and unlike an environment with a history of using measurement, this maintenance environment required a period of bootstrapping. Several rapid improvement (or learning) cycles were required to create the initial understanding of this environment necessary to identify meaningful goals and to design effective measurement procedures. Using measurement on a trial basis is also important for building the credibility of the improvement approach before attempting to apply it routinely on a large scale.

We have completed the initial program phase in which the goals and measurement procedures presented in this paper have been demonstrated to justify routine application to all maintenance projects within the SEL. The SEL is now providing routine support for improving maintenance, including data collection, form validation, and database support.

Establishing the SEL maintenance improvement program has been mainly a technology transfer problem. We tried to import existing technology, and customize it to the specific needs within the SEL. During the course of this study we have identified several problem areas that cannot be solved with existing technology but require additional research. These problem areas include more formal means for (i) capturing our understanding of an environment, (ii) packaging it into project specific, domain specific and general knowledge, (iii) relating measurement and the objects of measurement (i.e. processes and products), (iv) tailoring existing models to specific needs, and capturing the modeling process itself. Independent of the SEL maintenance improvement program, other research at the University of Maryland is addressing some of these issues: the TAME (Tailoring A Measurement Environment) project [Basili88] focuses on problems (ii) - (iv). The MVP (Multi-View Process Specification) project [Rombach89] focuses on problems (i) and (iii).

6. ACKNOWLEDGEMENTS

We thank Frank McGarry and Jon Valett (NASA/GSFC) for supporting this project from the very beginning, all the NASA and CSC maintenance personnel for putting up with several evolving versions of data collection forms and for giving us valuable feedback, and the librarians and database administrator of NASA/GSFC for handling the logistics of so many data collection forms.

7. REFERENCES

Basili84 V. R. Basili and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering* **SE–10**(6), pp.728–738 (November 1984).

Basili85 V. R. Basili, "Can We Measure Software Technology: Lessons Learned from Eight Years of Trying," *Proceedings Tenth Annual Software Engineering Workshop*, NASA Goddard Space Flight Center (December 1985).

Basili85b V. R. Basili, "Quantitative Evaluation of Software Engineering Methodology," *First Pan Pacific Computer Conference* (September 1985).

Basili87b V. R. Basili and H. D. Rombach, "Tailoring the Software to Project Goals and Environments," *Proc. Ninth International Conference on Software Engineering*, pp.345–357. (March 30–April 2, 1987).

Basili87 V R. Basili and R. W. Selby, Jr., "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering* **SE–13**(12), pp.1278–1296 (December 1987).

Basili88 V. R. Basili and H. D. Rombach, "The TAME Project: Towards Improvement–Oriented Software Environments," *IEEE Transactions on Software Engineering* **SE–14**(6), pp.758–773 (June 1988).

Brelsford88 J. Brelsford, "Establishing a Software Quality Program," *Quality Progress* **21**(11), pp.34–37 (November 1988).

Brophy87 C. Brophy, W. Agresti, and V. R. Basili, "Lessons Learned in Use of Ada Oriented Design Methods," *Proceedings Joint Ada Conference*, pp.231–236 (March 16–17, 1987).

Grady87 R. B. Grady, "Measuring and Managing Software Maintenance," *IEEE Software* **4**(5), pp.35–45 (September 1987).

Grady87b R. B. Grady and D. L. Caswell, *Software Metrics: Establishing a Company-Wide Program*, Prentice–Hall, Inc., Englewood Cliffs, New Jersey (1987).

Katz86 E. E. Katz, H. D. Rombach, and V. R. Basili, "Structure and Maintainability of Ada Programs: Can We Measure the Differences?," *Proceedings 9th Minnowbrook Workshop on Software Performance Evaluation* (August 1986).

McGarry85 F. E. McGarry, "Recent SEL Studies," *Proceedings Tenth Annual Software Engineering Workshop*, NASA Goddard Space Flight Center (December 1985).

Nehmer87 J. Nehmer, D. Haban, F. Mattern, D. Wybranietz, and H. D. Rombach, "Key Concepts of the INCAS Multicomputer Project," *IEEE Transactions on Software Engineering* **SE–13**(8), pp.913–923 (August 1987).

Ramsey88 C. L. Ramsey and V. R. Basili, "Expert Systems for Software Engineering Management: A Summarized Evaluation," in *Expert Systems and Advanced Data Processing*, ed. M. L. Emrich, A. R. Sadlowe, and L. F. Arrowood, Elsevier Science Publishing Co., Inc. (1988).

Rombach86 H. D. Rombach, V. R. Basili, and R. W. Selby, Jr., "The Role of Code Reading in the Software Life Cycle," *Proceedings 9th Minnowbrook Workshop on Software Performance Evaluation* (August 1986).

Rombach87 H. D. Rombach and V. R. Basili, "A Quantitative Assessment of Software Maintenance: An Industrial Case Study," *Proceedings Conference on Software Maintenance-1987*, pp.134–144 (September 21–24, 1987).

Rombach88 H. D. Rombach and B. T. Ulery, "Improving Software Maintenance through Measurement," Technical Report CS-TR-2131, Dept. of Computer Science, University of Maryland, College Park, Maryland (October 1988). Published as an invited paper, IEEE Proceedings, April 1989.

Rombach89 H. D. Rombach and L. Mark, "Software Process & Product Specifications: A Basis for Generating Customized SE Information Bases," *Proceedings Twenty-Second Annual Hawaii International Conference on System Sciences* II (1989).

Selby87 R. W. Selby, Jr., V. R. Basili, and T. Baker, "CLEANROOM Software Development: An Empirical Evaluation," *IEEE Transactions on Software Engineering* **SE–13**(9), pp.1027–1037 (September 1987).

Weiss85 D. M. Weiss and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory," *IEEE Transactions on Software Engineering* **SE–11**(2), pp.157–168 (February 1985).

UMIACS–TR–89–48
CS–TR–2244

May, 1989

Maintenance = Reuse–Oriented Software Development†

Victor R. Basili‡
Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

In this paper, we view maintenance as a reuse process. In this context, we discuss a set of models that can be used to support the maintenance process. We present a high level reuse framework that characterizes the object of reuse, the process for adapting that object for its target application, and the reused object within its target application. Based upon this framework, we offer a qualitative comparison of the three maintenance process models with regard to their strengths and weaknesses and the circumstances in which they are appropriate. To provide a more systematic, quantitative approach for evaluating the appropriateness of the particular maintenance model, we provide a measurement scheme, based upon the reuse framework, in the form of an organized set of questions that need to be answered. To support the reuse perspective, a set of reuse enablers are discussed.

† This research was supported in part by NASA grant NSG–5123 and ONR grant N00014–87–K–0307 to the University of Maryland.

‡ Keynote address, conference on Software Maintenance, Phoenix, AZ, October 1988.

Maintenance = Reuse-Oriented Software Development

Victor R. Basili
Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland

**[Keynote Address, Conference on Software Maintenance,
Phoenix, AZ, October 1988.]**

Abstract

In this paper, we view maintenance as a reuse process. In this context, we discuss a set of models that can be used to support the maintenance process. We present a high level reuse framework that characterizes the object of reuse, the process for adapting that object for its target application, and the reused object within its target application. Based upon this framework, we offer a qualitative comparison of the three maintenance process models with regard to their strengths and weaknesses and the circumstances in which they are appropriate. To provide a more systematic, quantitative approach for evaluating the appropriateness of the particular maintenance model, we provide a measurement scheme, based upon the reuse framework, in the form of an organized set of questions that need to be answered. To support the reuse perspective, a set of reuse enablers are discussed.

Introduction

If we take the view that software should be developed with the goal of maximizing the reuse of prior experience in the form of knowledge, processes, products and tools, then the maintenance process is logically ideally suited to a reuse-oriented software development process. There are a variety of reuse models. The key issue here is which process model is best suited to the particular maintenance problem at hand.

In this paper, we present a high level organizational paradigm for software development and maintenance in which an organization can learn from prior and current development and maintenance tasks and then apply that paradigm to several maintenance process models. The paradigm has associated with it a mechanism for setting goals that can be measured so that the organization can evaluate the process and the product and learn from its experience for future projects or enhancements of the current project.

We begin by identifying three process models that can be used for maintenance. We then present a high level reuse framework that characterizes the object of reuse, the process for adapting that object for its target application, and the reuse object within its target application. Based upon this framework, we offer a qualitative comparison of the three maintenance process models with regard to their strenghts and weaknesses and the circumstances in which they are appropriate. To provide a systematic, quantitative approach for evaluating the appropriateness of the particular maintenance model, we provide a measurement scheme, using the Goal/Question/Metric Paradigm. Since reuse requires a supportive environemnt, a set of environmental reuse enablers are discussed.

Maintenance

The nature of software is that it can be modified without the use of physical tools such as screw drivers and soldering irons. This has lead to the false assumption that maintenance is easy and inexpensive. Clearly nothing could be further from the truth.

Most software systems are complex and modification requires a deep understanding of the functional and non-functional requirements, the mapping of functions to system components, and the interaction of the components. Without good documentation of the requirements, design and code with respect to function, traceability and structure, maintenance becomes a difficult, expensive, error-prone task. As early as 1976, Belady and Lehman reported on the problems with the evolution of OS 360 [7]. The literature is filled with similar experiences and lessons learned [10,12,16,18,20].

Maintenance consists of several different types of activities: correction of faults existing in the system, the adaptation of the system to a changing operating environment, e.g., new terminals, operating system modifications, etc., and changes to the original requirements. The new system is like the old system but different in a specific set of characteristics. One can view the new version of the system as a modification of the old system or a new system which reuses many of the components of the old system. Although these two views have many aspects in common, they are quite different with respect to the process models used and their effects on future environments.

In fact, we can identify at least three process models associated with maintenance depending upon the characteristics of the modification. We will call these (1) the quick fix model, (2) the iterative enhancement model, and (3) the full reuse model. All three models reuse the old system and so are reuse-oriented. Which model should be chosen for any particular modification is a combination of management and technical decisions.

Quick Fix Model. The quick fix model involves taking the existing system, usually just the code, and making the necessary changes to the source code and the accompanying documentation, e.g. requirements, design, and recompiling the system as a new version. This may be as straightforward as a change to some internal component, e.g. an error correction involving a single component or a structural change or even some functional enhancement. Here reuse is implicit.

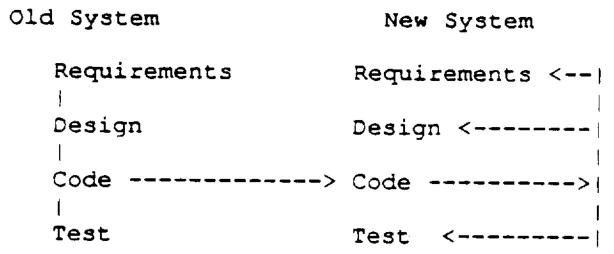


Figure 1. Quick Fix Process Model

Iterative Enhancement Model. Iterative Enhancement [5] is an evolutionary model which was proposed for software development in environments where the complete set of requirements for a system were not fully understood or the author did not know how to build the full system. Although it was proposed as a development model, it is well suited to maintenance. The process model involves:

1. Starting with the existing system requirements, design, code, test and analysis documents
2. Redeveloping starting with the appropriate document based upon analysis of the existing system, propagating the changes through the full set of documents
3. At each step of the evolutionary process, continuing to redesign, based upon analysis.

```
          Old System                 New System

Requirements        --------> Requirements    ---->
|                   |         |               |
Design              |         Design          |
|                   |         |               |
Code                |         Code            |
|                   |         |               |
Test                |         Test            |
|                   |         |               |
Analysis ----------           Analysis -------
```

Figure 2. Iterative Enhancement Model

To view this as a maintenance model, assume the initial implementation is the system in its current state in the evolutionary maintenance process. The process assumes that the maintenance organization has the ability to analyze the existing product, characterize the proposed set of modifications, and redesign the current version where necessary for the new capabilities. Again, reuse is implicit.

Full Reuse Process Model. While iterative enhancement starts with evaluating the existing system for redesign and modification, a full reuse process model starts with the requirements analysis and design of the new system, with the concept of reusing whatever requirements, design and code are available from the old system. The reuse process model involves:

1. Starting the requirements for the new system, reusing as much of the old system as feasible
2. Building a new system using components from the old system or other systems available in the repository developing new components where appropriate.

```
 Old System      Repository       New System

Requirements  --> {Ri} <------> Requirements
|                               |
Design   --------> {Di} <------> Design
|                               |
Code   ----------> {Ci} <------> Code
|                               |
Test   ----------> {Ti} <------> Test
```

Figure 3. Full Reuse Process Model.

Here reuse is explicit, packaging of prior components is necessary and analysis is required for the selection of the appropriate components.

The difference between the last two approaches is more one of perspective. The full reuse model frees the developer to design the solution relative to the available set of solutions of similar systems. The iterative enhancement model takes the last version of the current system and enhances it. Both approaches encourage redesign, but the full reuse model suggests a wider forum and can lead to the development of more reusable components for future systems where the iterative enhancement model suggests the tailoring of an existing system for the given extensions.

A Reuse Framework

The existence of several models for maintenance raises several questions. Which is the most appropriate model for a particular environment? a particular system? a particular set of changes? the task at hand? How do I improve each step in the process model I have chosen? How do I minimize overall cost and maximize overall quality?

In order to answer these questions we need a model of the object of reuse, the process of adapting that object for its target application, and the reused object within its target application. A simple model for reuse is given in figure 4. In that model, an object is any software process or product and a transformation is the set of activities that are performed in reusing that object. Given that the scope of the new application are understood, the steps are:

1. identifying the candidate reusable pieces of the old object
2. understanding them
3. modifying them to our needs
4. integrating them into the process

```
**********************************************************
* --------             context             --------     *
*|  old   |       ----------------        |  new   |    *
*| object | ------>| transformation | ----->| object |  *
* --------    |     ----------------        --------     *
****|********|********************************************
    |        |
  ------------
 | repository |
  ------------
```

Figure 4. A Simple Reuse Model

To flesh out the model, we need a framework for categorizing objects, transformations, and context. The framework should cover various categories e.g. reuse object: process, product. Within each category there are various classification schemes for product: e.g. requirements document, code module, test plan, and process: e.g. cost estimation, risk analysis, design.

There are a variety of approaches to reuse and schemes that classify the object of reuse [8,9,11,13,15,17,19]. We use here a variation of a reuse framework [4] that captures several aspects of the reuse process, product and context.

Object dimensions include:

(1) **Reuse Object Type:** What is a characterization of the candidate reuse object? Sample categories and classifications are: process (e.g. design, test) and product (e.g. application, tool).
(2) **Self-Containedness:** How independent and understandable is the candidate reuse object? Sample categories and classifications are: syntactic independence (e.g. tightly coupled), semantic independence (e.g. similar functionality), and precision of specification (e.g. formal, informal).
(3) **Reuse Object Quality:** How good is the candidate reuse object? Sample categories and classifications are: maturity (e.g. newly developed, used in one application) complexity (e.g. low cyclomatic complexity) reliability (e.g. no failures during prior use).

Context dimensions include:

(1) **Requirements Domain:** How similar are the requirements domains of the candidate reuse objects and current or future projects? Some example categories and classifications are: application (e.g. ground support software for satellites), distance (e.g. same application, similar algorithms/different problem focus).
(2) **Solution Domain:** How similar are the evoloution process which resulted in the candidate reuse objects and the ones used in the current and future projects? Some example categories and classifications are: process model (e.g. waterfall model) design method (e.g. function decomposition) programming language (e.g. FORTRAN).
(3) **Knowledge Transfer Mechanism:** How is information about the candidate reuse objects and their context passed to current and future projects? An example classification is: humans (e.g. subset of the development team doing maintenance, separate team doing maintenance).

Transformation dimensions include:

(1) **Type of Transoformations:** How do we characterize the transformation activities to be performed? Some sample categories and classifications are: Percent of Change (e.g. 0%, 5%), Direction (e.g. general to domain specific, project specific to domain specific), mechanism of modification (e.g. verbatim, parameterization, template-based, unconstrained) and mechanism for identifying (e.g. by name, by functional requirements).
(2) **Activity Integration:** How do we integrate the transformation activities into the new system development? Some sample categories and classifications are: phase activity performed in the new development planning (e.g. cost estimation, risk analysis), construction (e.g. requirements development), analysis (e.g. testing).
(3) **Transformed Quality:** What is the contribution of the reuse object in the context of the new system with respect to the objectives set for it? Sample category and classifications are: reliability (e.g. no failures associated with that component) and performance (e.g. satisfies the timing requirement).

Comparing the Models Using the Reuse Framework

When applying the reuse framework to the maintenance process, we are focusing on a set of reuse objects that are product documents. Let us compare the various models according to the dimensions given.

Consider the reuse object dimensions:

With regard to reuse object type, the object of the quick fix and iterative enhancement models is the set of documents representing the old system: The object of the full reuse model is the repository including the old system.

With regard to self-containedness, all the models depend upon the unit of change. The quick fix model depends upon how much evolution has taken place since entropy may have unstructured the system. In iterative enhancement, the evolved system should be improving for the specific application and for the appropriate set of changes, the unit of change should be more visible. In the full reuse model, the evolved system should be improving with respect to reuse object independence for the general application, depending upon the quality and maturity of the repository.

With regard to reuse object quality, the quick fix model offers little knowledge of the quality of the old object. In iterative enhancement, the analysis phase provides a fair assessment of quality with respect to the particular application. In full reuse, we have an assessment of quality of the reuse object across several systems.

Consider the context dimensions:

With regard to the requirements domain, the quick fix and iterative enhancement model assume the same application, in fact the same project. The full reuse model allows for manageable variation in the application domain, depending upon what is available in the repository.

With regard to the solution domain, the quick fix model assumes the same solution structure exists during maintenance as during development. There is no change in the basic design or structure of the new system. In the iterative enhancement model, because redesign is a part of the model, there is some modification to the solution structure allowed. The full reuse model allows major differences in the solution structure, i.e. a complete redesign is possible going from functional decomposition to object oriented design.

With regard to knowledge transfer mechanism, the quick fix model and iterative enhancement work best with the same people. The full reuse model can compensate for having a different team, assuming we have application specialists and a well documented reuse object repository.

Consider the transformation dimensions:

With regard to type of activities, the quick fix model typically uses a source code look-up, reading for understanding, unconstrained modification and re-compilation approach. Iterative enhancement typically begins with a search through the highest relevant document, changing it and continuing through the subsequent documents using a variety of modification mechanisms. The full reuse is uses a library search, and a variety of modification mechanisms depending upon the type of change. Here modification is done off-line.

With regard to activity integration, in the quick fix model, all activities are performed at same time. Iterative enhancement associates the activities with all the normal development phases. In the full reuse model, identification of the candidate reusable pieces is done during project planning and the other activities are done during development.

With regard to transformed quality, the quick fix model usually works best on small well-contained modifications since their affect on the system can be understood and verified in context. Iterative enhancement is more appropriate for larger changes where the analysis phase can provide better assessment of the full effect of changes. Full reuse is appropriate for large changes and major redesigns. Here, analysis and prior history of the performance of the reuse objects support quality.

Given these differences, we can provide some analysis of the various maintenance process models and recommend where they might be most applicable. But first, let's discuss the relationship between the development and maintenance process models. In some sense development can be considered a subset of maintenance. Maintenance environments differ from development environments with regard to the constraints on the solution, customer demand, timeliness of response, and organization.

Most maintenance organizations are set up for the quick fix model but not for the iterative enhancement or reuse process models. This is because they are responding to timeliness, e.g. a system failure needs to be fixed immediately, or a customer demand, e.g. a modification of the functionality of the system. Clearly these are strengths for the quick fix model. But the weaknesses of the model are that the modification is usually a patch, not well documented, the structure of the system has been partly destroyed which makes future evolution of the system difficult and error-ridden, and it is not compatible with development processes. This model is best used when timeliness and customer need are dominant and there is little chance the system will be modified again.

The iterative enhancement model allows for redesign so the structure of the system evolves and future modification is easier. It focuses on the particular system, making it as good as possible. It is compatible with development process models. The drawbacks are that it is a more costly and possibly less timely approach (in the short run) than the quick fix model and it provides little support for generics or future similar systems. It is a good approach to use when the product will have a long life and evolve over time. In this case, if timeliness is also a constraint, the quick fix model can be used as a patch and the iterative enhancement model can be used for the long term change, replacing the patch.

The full reuse process model provides the maintainer with a broader perspective, focuses on long range development for a set of products and has the side effect of creating reusable components of all kinds for future developments. It is compatible with development process models, and in fact, it is the way we would like such models to evolve. The drawback is that it is more costly in the short run, is not appropriate for small modifications but can be used in conjunction with other models. It is best used when we are living in multiproduct environments or generic development where the product line has a long life.

The assessment given above is informal and intuitive. This is due to the fact that it is a qualitative analysis. To do a quantitative analysis we need quantitative models of the reuse objects, transformations, and context. We need a measurement framework for characterizing via categorization and classification, evaluation, prediction, and motivation to support management and technical decisions. To do this we apply the goal/question/metric paradigm to the models.

The Goal Question Metric Paradigm

The goal/question/metric (GQM) paradigm [1,2,6] represents a systematic approach for setting the project goals (tailored to the specific needs of an organization), defining them in an operational, tractable way by refining them into a set of quantifiable questions that in turn imply a specific set of metrics and data for collection. The tractability of this software engineering process allows the analysis of the collected data and computed metrics in the appropriate context of the questions and the original goal. This context supports feedback (by integrating analytic and constructive aspects) and learning (by defining the appropriate synthesis procedure for lower-level into higher-level pieces of experience).

The process of setting goals and refining them into quantifiable questions is complex and requires experience. In order to support this process, a set of templates for setting goals, and a set of guidelines for deriving questions and metrics has been developed [2].

Goals are defined in terms of purpose, perspective and environment. Different sets of guidelines exist for defining product-related and process-related questions. Product-related questions are formulated for the purpose of defining the product (e.g., physical attributes, cost, changes and defects, context), defining the quality perspective of interest (e.g., reliability, user friendliness), and providing feedback from the particular quality perspective. Process-related questions are formulated for the purpose of defining the process (quality of use, domain of use), defining the quality perspective of interest (e.g., reduction of defects, cost effectiveness of use), and providing feedback from the particular quality perspective.

Application of the Goal Question Metric Paradigm

In applying the goal/question/metric paradigm, we define the goals of the maintenance process and articulate the issues associated with choosing the appropriate process model, providing management with the questions that need to be answered to make intelligent decisions, understand the trade-offs, and perform risk analysis. There are a variety of goals we can generate. For example: to determine which process model should be chosen for a particular product, to improve our performance or evolve a better definition of any of the models for a particular product line.

In what follows we will generate a sample goal for maintenance and provide a partial list of the questions involved. Some of the answers will be obvious, either in the measures they require be taken or the information required form the experts; others will not. Thus a goal for maintenance in the context of the reuse framework might be:

Purpose:
To evaluate the new product requirements in order to reuse as much of the available products as possible.

Perspective:
Examine the cost and future evolution of the development from the point of view of the organization.

Environment:
Along with the standard environmental factors, such as resource factor, problem factors, we would like to pay special attention to the three context dimensions of the reuse framework.

Requirements Domain:
Clearly we are using product objects from the same application domain, although we have the ability to choose candidate components from other application domains.

Solution Domain:

This defines the process models, methods and tools that were used in the development of the existing product. If the same processes are to be used for the evolved project then there is not problem with reuse. However, the reuse model allows us to change the processes (and thus possibly to the product structure) at the cost of reusing less of the prior project. If there are to be changes then we must evaluate the cost of modification of the process and resulting product relative to the gains for process change.

Knowledge Transfer Mechanism:

If the maintenance group is the same as the development group then there is no transfer of knowledge required. If they are different then there are concerns that must be evaluated with respect to application, process and product knowledge of the maintainers and the kinds of documentation available.

Product Definition:

In considering the product, we actually have several. The new product to be built, i.e. the new version of the system, and the old versions plus any other systems that are relevant.

Product Dimensions

New Product:

How many requirements are there in total for the new system?

Old Product:

What is the mapping of requirements to system components?
What is the measure of the complexity of the traceability?
How independent are the components to be modified?
What is the complexity of the system and the individual system components?

Repository:

What candidate components are available in the repository and what are their context, transformation and object classifications?

Difference between new and old:

How many requirements are there that are not in the old system? (Categorize by size, new vs. modification of old vs. deletion of old, etc.)
How many components must be changed, added, deleted? (categorized by size and type of change)

Changes/Defects

How many errors, faults, failures (categorized by class) are associated with the requirements and components that need to be changed?
What is the profile of changes to the original system prior to this change?

Cost

What is the cost of understanding the new requirements?
What is the estimated cost of building a new system, reusing the experience and parts of the old project?

What was the cost of the old system in total?
What was the cost of each version?
What is the estimated cost of modifying the old system to meet the new requirements?

Customer Context

How will the new system be used?
What are the potential future modifications based upon our analysis of customer profiles, past modifications and the state of technologies?

Perspective:
cost and future evolution of the development

Model of Perspective: cost of modification of the design of the system vs. the expected future modifications
Parameters:
the life time of the system
the cost of future evolution of the system
the cost of evolving the old system versus rebuilding from old parts

Feedback:
Is the model appropriate?
How can the model be improved?
How can the estimations be improved?
How can classifications be improved?
How can activities be improved?

The Goal/Question/Metric paradigm allows us to develop other goals for reuse. These can be developed for whether the reuse object is a process or a product. Consider the following examples:

Evaluate the modification activity within the reuse process in order to improve it. Examine the cost and correctness of the resulting object from the point of view of the customer.

Predict the appropriate maintenance process model in order to perform the correct one. Examine its cost with respect to the customer needs and the future evolutions of the system from the point of view of the corporation.

Evaluate the standard corporate design method in order to assess how it should have been tailored for the current project. Examine its effectiveness from the point of view of the designer.

Evaluate the components of the existing product in order to determine whether to reuse them. Examine their independence and functional appropriateness from the point of view of their use in future systems.

Predict the ability of a set of code components to be integrated into the current system from the point of view of the developer.

Motivate the development of a reusable set of components in order to engineer them for reuse. Examine the reward structure from the point of view of the manager and developer.

Reuse Enablers

There are a variety of support mechanisms necessary for achieving maximum reuse that have not been sufficiently emphaisized in the literature. In this paper we have discussed several of these: a set of maintenance models, a mechanism for choosing the appropriate such models based upon the goals and characteristics of the problem at hand, and a measurement and evaluation mechanism. To support these activities there is a need for an improvement paradigm that aids the organization in evaluating, learning and enhancing the software process and product, a reuse-oriented evolution environment that motivates and supports reuse, and automated support for that model as well as the measurement and evaluation process.

The Improvement Paradigm: The improvement paradigm [1] is a high level organizational process model in which the organization learns how to improve their product and process. Within this model the organization should learn how to make better decisions on which process model to use for the maintenance of their future software products based upon learning from past performance. The paradigm is defined as follows:

1. Planning. There are three integrated activities to planning that are iteratively applied:

   (a) Characterize the current project environment. It provides a quantitative analysis of the environment and a model of the project in the context of that environment. In the context of maintenance, the characterization should provide product dimension data, change and defect data, cost data and customer context data for earlier versions of the system to be modified, information about what classes of candidate components are available in the repository for the new system, and any information feedback from prior projects about experience with the different models for the types of modifications required.

(b) Set up goals and refine them into quantifiable questions and metrics using the goal/question/metric paradigm, for successful project performance and improvement over previous project performances. This consists of a top-down analysis of goals that iteratively decomposes high-level goals into detailed subgoals. The iteration terminates when it has produced sub-goals that we can measure directly. For maintenance this involves the development of specific G/Q/Ms as specified in the prior section.

(c) Choose and tailor the appropriate construction model for this project and the supporting methods and tools to satisfy the project goals relative to the characterized environment. Understanding the environment quantitatively allows us to choose the appropriate process model and fine tune the methods and tools needed to be most effective. For example, knowing the effect of prior applications of the various maintenance models and methods in creating new projects from old systems allows us to choose and fine tune the appropriate process model and methods that have been historically most effective in creating new systems of the type required from older versions and component parts in the repository.

2. Analysis. Analyze the data to evaluate the current practices, determine problems, record the findings and make recommendations for improvement. We must conduct data analysis during and after the project. The goal/question/metric paradigm provides traceability from goals to metrics and back. This permits the measurement to be interpreted in context ensuring a focused, simpler analysis. The goal-driven operational measures provide a framework for the kind of analysis needed.

3. Learning and Feedback. This step involves the organization and encoding of the quantitative and qualitative experience gained from the current project into a corporate information base to help improve planning, development, and assessment for future projects. The results of the analysis and interpretation phase can be fed back to the organization to change the way it does business based upon explicitly determined successes and failures. In this way, we can learn how to improve quality and productivity, and how to improve definition and assessment of goals. We can start the next project armed with the experience gained from this and previous projects. For example, understanding the problems associated with each new version of a system, provides insights into the need for redesign and redevelopment.

A Reuse-Oriented Environment: Reuse can be more effectively achieved within an environment that supports reuse [3,8,13]. Software engineering environments provide such things as a project data bases, and support the interaction of people with methods, tools and project data. However, experience is not controlled by the project data base or owned by the organization. Reuse only exists implicitly.

We need to be able to incorporate the reuse process model into the context of development. We need to combine the development and maintenance models in order to maximize the context dimensions. We need to integrate characterization, evaluation, prediction and motivation into the process. We need to support learning and feedback to make reuse viable. We propose that the reuse model can exist within the context of the improvement paradigm, making it possible to support all of the above requirements.

The TAME Project: The improvement paradigm and the reuse oriented process model require automated support for the data base, encoded experience, and the repository of prior projects and reusable components [2,3,14]. We need to automate as much of the measurement process as possible, and provide a tool environment for managers and engineers to develop project specific goals, and generate operational definitions based upon these goals that specify the appropriate metrics needed for evaluation. The evaluation and feedback cannot be done in real time without automated support. Automated support will help in the post mortems analysis.

The goal of the TAME system [2] is to instantiate and integrate the improvement and goal/question metric paradigms and help in the tailoring of the software development process. But it can also support the reuse-oriented process model. The TAME environment model contains basic mechanisms for supporting systematic learning and reuse. To help with systematic learning it provides support for recording experience, off-line generalizing or tailoring of experience, and formalizing experience. To help with systematic reuse it supports mechanisms for using existing experience and on-line generalizing or tailoring of candidate experience. In this way it attempts to integrate both learning and reuse into an overall evolution model.

The application of the TAME system concept to maintenance will provide a mechanism for choosing the appropriate maintenance process model for a particular project and provide data to help us learn how to do a better job of maintenance.

Summary

The approach to maintenance depends on the nature of the problem and the size and complexity of the modification. This paper recommends that we view maintenance as a reuse process. In this way the maintainer is provided with a reuse model and a framework for viewing maintenance that permits a measurement framework to be applied. A new model of a reuse-oriented evolution process can be developed in which the existing models can be defined. Existing models can then be analyzed within this framework, allowing an organization to evaluate the strengths and weaknesses of the different approaches and provides feedback in refining the various process models and creating an experience base from which to support further management and technical decisions.

The approach provides support for defining activities, determining options, and evaluation. If the approach is not adapted then it is difficult for an organization to know which process model to use for a particular project, whether they are evolving the system appropriately, and whether they are maximizing quality and minimizing cost over the life of the system.

References

[1] V. R. Basili, "Quantitative Evaluation of Software Methodology," Dept. of Computer Science, University of Maryland, College Park, TR-1519, July 1985 [also in Proc. of the First Pan Pacific Computer Conference, Australia, September 1986].

[2] V. R. Basili, H. D. Rombach "The TAME Project: Towards Improvement-Oriented Software Environments," IEEE Transactions on Software Engineering, vol. SE-14, no. 6, June 1988, pp. 758-773.

[3] V. R. Basili, H. D. Rombach "Towards a Comprehensive Framework for Reuse: A Reuse Enabling Software Evolution Environment," University of Maryland Computer Science Technical Report, UMIACS-TR-88-92, December 1988.

[4] V. R. Basili, H. D. Rombach, J. Bailey, and B. G. Joo, "Software Reuse: A Framework," Proc. of the Tenth Minnowbrook Workshop on Software Reuse, Blue Mountain Lake, New York, July 1987.

[5] V. R. Basili, A. J. Turner, "Iterative Enhancement: A Practical Technique for Software Development," IEEE Transactions on Software Engineering, vol. SE-1, no. 4, pp. 390-396, December, 1975.

[6] V. R. Basili, D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," IEEE Transactions on Software Engineering, vol. SE-10, no. 6, pp. 728-738, November 1984.

[7] L. Belady and M. Lehman, "A Model of Large Program Development, IBM Systems Journal, vol.15, no.3, 1976.

[8] Ted Biggerstaff, "Reusability Framework, Assessment, and Directions," IEEE Software Magazine, March 1987, pp.41-49.

[9] T. P. Bowen, G. B. Wigle, J. T. Tsai, "Specification of Software Quality Attributes," Technical Report RADC-TR-85-37, Rome Air Development Center, Griffiss Air Force Base, N.Y. 13441-5700, February 1985.

[10] Federal Information Processing Standards, "Guideline on Software Maintenance," U.S. Dept. of Commerce/National Bureau of Standards, FIPS PUB 106, June 1984.

[11] P. Freeman, "Reusable Software Engineering: Concepts and Research Directions," Proc. of the Workshop on Reusability, September 1983, pp. 63-76.

[12] R. B. Grady, "Measuring and Managing Software Maintenance," IEEE Software, Vol. 4, No. 5, September 1987, pp. 35-45.

[13] IEEE Software, special issue on 'Reusing Software', vol.4, no.1, January 1987.

[14] IEEE Software, special issue on 'Tools: Making Reuse a Reality', vol.4, no.7, July 1987.

[15] G. A. Jones, R. Prieto-Diaz, "Building and Managing Software Libraries," Proc. Compsac'88, Chicago, October 5-7, 1988, pp. 228-236.

[16] B. P. Lientz, E.B. Swanson, and G. E. Tompkins, "Characteristics of application software maintenance," Communications of the ACM, Vol 21, No. 6, June 1978, pp. 466-471.

[17] R. Prieto-Diaz, P. Freeman, "Classifying Software for Reusability," IEEE Software, vol.4, no.1, January 1987, pp. 6-16

[18] H. D. Rombach, V. R. Basili, "A Quantitative Assessment of Software Maintenance: An Industrial Case Study," in Proc. Conf. Software Maintenance, Austin, TX, Sept. 1987, pp. 134-144.

[19] Mary Shaw, "Purposes and Varieties of Software Reuse," Proceedings of the Tenth Minnowbrook Workshop on Software Reuse, Blue Mountain Lake, New York, July, 1987.

[20] W. Tracz, "Tutorial on 'Software Reuse: Emerging Technology'," IEEE Catalog Number EHO278-2, 1988.

[21] S. S. Yau, R. A. Nicholl, J. J.-P. Tsai, and S.-S. Liu, "An Integrated Life-Cycle Model for Software Maintenance," IEEE Transactions on Software Engineering, Vol. SE-14, No. 8, August 1988, pp. 1128-1144.

UMIACS-TR-89-57
CS-TR-2263

June, 1989

Software Development: A Paradigm for the Future †

Victor R. Basili
Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

This paper offers a new paradigm for software development that treats software development as an experimental activity. It provides built-in mechanisms for learning how to develop software better and reusing previous experience in the forms of knowledge, processes and products. It uses models and measures to aid in the tasks of characterization, evaluation and motivation. If proposes an organization scheme for separating the project-specific focus from the organization's learning and reuses focuses of software development. It discusses the implications of this approach for corporations, research and education and presents some research activities currently underway at the University of Maryland that support this approach.

† Research supported in part by NASA grant NSG-5123, AFOSR grant 87-0130, ONR grant N00014-87-K-0307, and ITMI. SIEI. (through the Industrial Associates Program of the Department of Computer Science). Keynote address, COMPSAC '89, Orlando, FL, Sept. 1989

1. INTRODUCTION

We have been struggling with the problems of software development for many years [31,64]. Organizations have been clamoring for mechanisms to improve the quality and productivity of software. We have evolved from focusing on the project, e.g. schedule and resource allocation concerns, to focusing on the product, e.g. reliability and maintenance concerns, to focusing on the process, e.g. improved methods and process models [27,33,39,56]. We have begun to understand that software development is not an easy task. There is no simple set of rules and methods that work under all circumstances. We need to better understand the application, the environment in which we are developing products, the processes we are using and the product characteristics required.

For example, the application, environment, process and product associated with the development of a toaster and a spacecraft are quite different with respect to hardware engineering. No one would assume that the same educational background and training, the same management and technical environment, the same product characteristics and constraints, and the same processes, methods and technologies would be appropriate for both. They are also quite different with respect to software engineering.

We have not fully accepted the need to understand the differences and learn from our experiences. We have been slow in building models of products and processes and people for software engineering even though we have such models for other engineering disciplines. Measurement and evaluation have only recently become mechanisms for defining, learning, and improving the software process and product [3,34].

We have not even delineated the differences between such terms as technique, method, process and engineering. For the purpose of this paper we define a technique as a basic technology for constructing or assessing software, e.g., reading or testing. We define a method as an organized management approach based upon applying some technique, e.g., design inspections or test plans. We define a process model as an integrated set of methods that covers the life cycle, e.g., an iterative enhancement model using structured designs, design inspections, etc. We define software engineering as the application and tailoring of techniques, methods and processes to the problem, project and organizational characteristics.

There is a basically experimental nature to software development. We can draw analogies from disciplines like experimental physics and the social sciences. As such we need to treat software developments as experiments from which we can learn and improve the way in which we build software.

2. THE IMPROVEMENT PARADIGM

Based upon our experiences in trying to evaluate and improve the quality in several organizations [5,29,53,58], we have concluded that a measurement and analysis program that extends through the entire life cycle is a necessity. Such a program requires an organization to adopt a long term, quality-oriented, organizational life cycle mode, which we call the Improvement Paradigm [4,19]. The paradigm has evolved over time, based upon experiences in applying it to improve various software related issues, e.g., quality and methodology. In its current form, it has four essential aspects:

1 Characterizing the environment. This involves data that characterizes the resource usage, change and defect histories, product dimensions and environmental aspects for prior projects and predictions for the current project. It involves information about what processes, methods and techniques have been successful in the past on projects with these characteristics. It provides a quantitative analysis of the environment and a model of the project in the context of that environment.

2 Planning. There are two integrated activities to planning that are iteratively applied:

(a) Defining goals for the software process and product operationally relative to the customer, project, and organization. This consists of a top-down analysis of goals that iteratively decomposes high-level goals into detailed subgoals. The iteration terminates when it has produced subgoals that we can measure directly. This approach differs from the usual in that it defines goals relative to a specific project and organization from several perspectives. The customer, the developer, and the development manager all contribute to goal definition. It is, however, the explicit linkage between goals and measurement that distinguishes this approach. This not only defines what good is but provides a focus for what metrics are needed.

(b) Choosing and tailoring the process model, methods, and tools to satisfy the project goals relative to the characterized environment. Understanding the environment quantitatively allows us to choose the appropriate process model and fine tune the methods and tools needed to be most effective. For example, knowing prior defect histories allows us to choose and fine tune the appropriate constructive methods for preventing those defects during development (e.g. training in the application to prevent errors in the problem statement) and assessment methods that have been historically most effective in detecting those defects (e.g., reading by stepwise abstraction for interface faults).

3 Analysis. We must conduct data analysis during and after the project. The information should be disseminated to the responsible organizations. The operational definitions of process and product goals provide traceability to metrics and back. This permits the measurement to be interpreted in context ensuring a focused, simpler analysis. The goal-driven operational measures provide a framework for the kind of analysis needed. During project

development, analysis can provide feedback to the current project in real time for corrective action.

4 Learning and Feedback. The results of the analysis and interpretation phase can be fed back to the organization to change the way it does business based upon explicitly determined successes and failures. For example, understanding that we are allowing faults of omission to pass through the inspection process and be caught in system test provides explicit information on how we should modify the inspection process. Quantitative histories can improve that process. In this way, hard-won experience is propagated throughout the organization. We can learn how to improve quality and productivity, and how to improve definition and assessment of goals. This step involves the organization of the encoded knowledge into an information repository or expereince base to help improve planning, development, and assessment.

- Characterize the current project environment.
- Set up goals and refine them into quantifiable questions and metrics for successful project performance and improvement over previous project performances.
- Choose the appropriate software project execution model for this project and supporting methods and tools.
- Execute the chosen processes and construct the products, collect the prescribed data, validate it, and and analyze the data to provide feedback in real-time for corrective action on the current project.
- Analyze the data to evaluate the current practices, determine problems, record the findings and make recommendations for improvement for future projects. This is an off-line process which involves the structuring of experience so that it can be reused in the future.
- Proceed to step 1 to start the next project, armed with the recorded, structured experience gained from this and previous projects.

FIGURE 1: THE IMPROVEMENT PARADIGM

The Improvement Paradigm is based upon the assumption that software product needs directly affect the processes used to develop and maintain the product. We must first specify our project and organizational goals and their achievement level. This specification helps determine our processes. In other words, we can't define the processes and then determine how we are going to achieve and evaluate certain project characteristics. We must define the project goals explicitly and quantitatively and use them to drive the process.

As it stands, the improvement paradigm is a generic process whose steps need to be instantiated by various support mechanisms. It requires a mechanism for defining operational goals and transforming them into metrics (step 2a). It requires a mechanism for evaluating the measurement in the context of the goals (step 3). It requires a mechanism for feedback and learning (step 4). It requires a mechanism for storing experience so that it can be reused on other projects (steps 1,2b). It requires automated support for all of these mechanisms. In the next three sections, we will discuss mechanisms that have been used to support these activities. In the last half of the paper, we will discuss a proposed organizational structure that allows these activities to be managed and evolve.

2.1. The Goal/Question/Metric Paradigm

The Goal/Question/Metric (GQM) paradigm is a mechanism for defining and evaluating a set of operational goals, using measurement on a specific project. It represents a systematic approach for setting the project goals tailored to the specific needs of an organization, defining them in an operational, tractable way by refining them into a set of quantifiable questions that in turn implies a specific set of metrics and data for collection. It involves the development of data collection mechanisms, e.g., forms, automated tools, the collection and validation of data. It includes the analysis of the collected data and computed metrics in the appropriate context of the questions and the original goals.

The GQM paradigm was originally developed for evaluating defects for a set of projects in the NASA/GSFC environment [28]. The application involved a set of case study experiments. It was then expanded to include various types of experimental approaches, including controlled experiments [4,22,25].

The process of setting goals and refining them into quantifiable questions is complex and requires experience. In order to support this process, a set of templates for setting goals, and a set of guidelines for deriving questions and metrics has been developed [19]. These templates and guidelines reflect our experience from having applied the GQM paradigm in a variety of environments.

Goals are defined in terms of purpose, perspective and environment. Different sets of guidelines exist for defining product-related and process-related questions. Product-related questions are formulated for the purpose of defining the product (e.g., physical attributes, cost, changes and defects, user context), defining the quality perspective of interest (e.g., functionality, reliability, user friendliness), and providing feedback from the particular quality perspective. Process-related questions are formulated for the purpose of defining the process (process conformance, domain conformance), defining the quality perspective of interest (e.g., reduction of defects, cost effectiveness of use), and providing feedback from the particular quality perspective.

The GQM provides a mechanism for supporting step 2(a) of the Improvement Paradigm which requires a mechanism for defining operational goals and transforming them into metrics that can be used for characterization, evaluation,

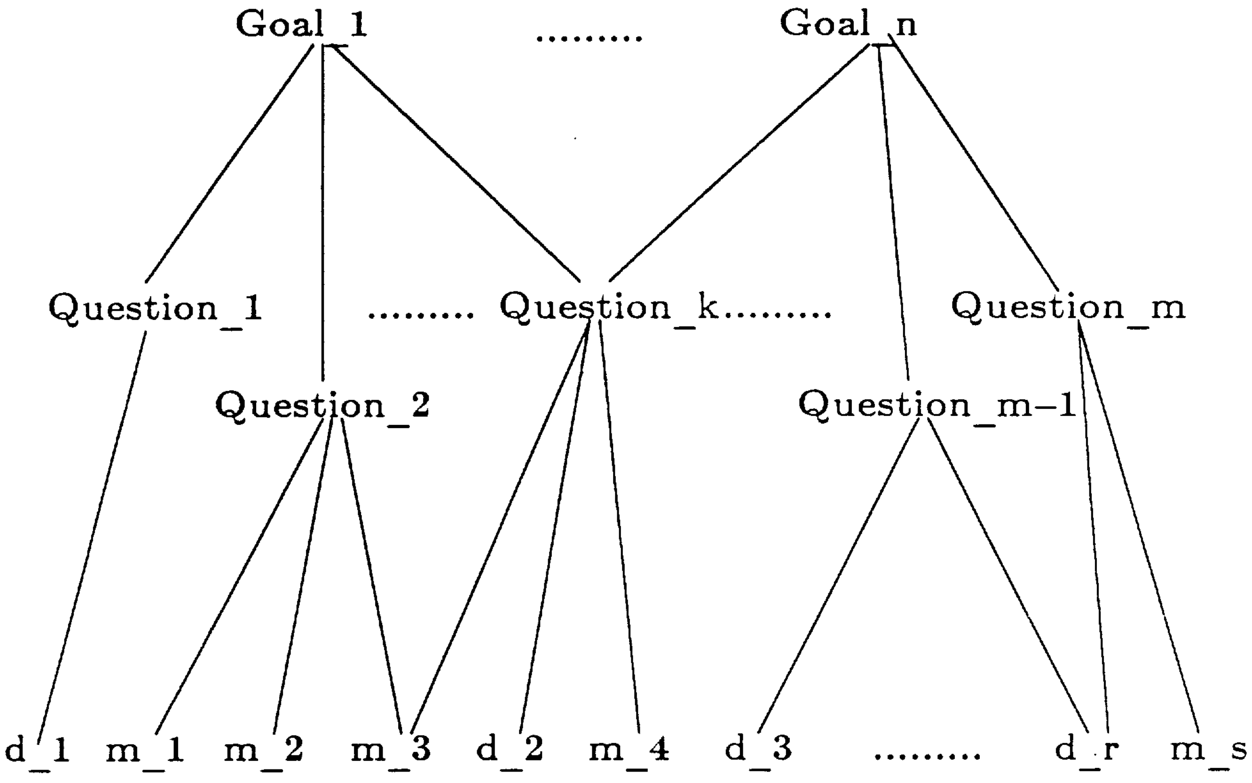


FIGURE 2: THE GOAL/QUESTION/METRIC PARADIGM

prediction and motivation. It supports step 3 by helping to define the experimental context and providing mechanisms for the data collection, validation and analysis activities. It also supports step 4 by providing quantitative feedback on the achievement of goals.

The GQM was originally used to define and evaluate goals for a particular project in a particular environment. In the context of the Improvement Paradigm, the use of the GQM is expanded. Now, we can use it for long range corporate goal setting and evaluation. We can improve our evaluation of a project by analyzing it in the context of several other projects. We can expand our level of feedback and learning by defining the appropriate synthesis procedure for lower-level into higher-level pieces of experience. As part of the IP we can learn more about the definition and application of the GQM in a formal way, just as we would learn about any other experiences.

2.2. The TAME Project

The TAME project [18,19] recognizes the need to characterize, integrate and automate the various activities involved in instantiating the Improvement Paradigm, for use on projects. It delineates the steps performed by the project and creates the idea of an experience base as the repository for what we have learned during prior developments. It recognizes the need for constructive and analytic activities and supports the tailoring of the software development process.

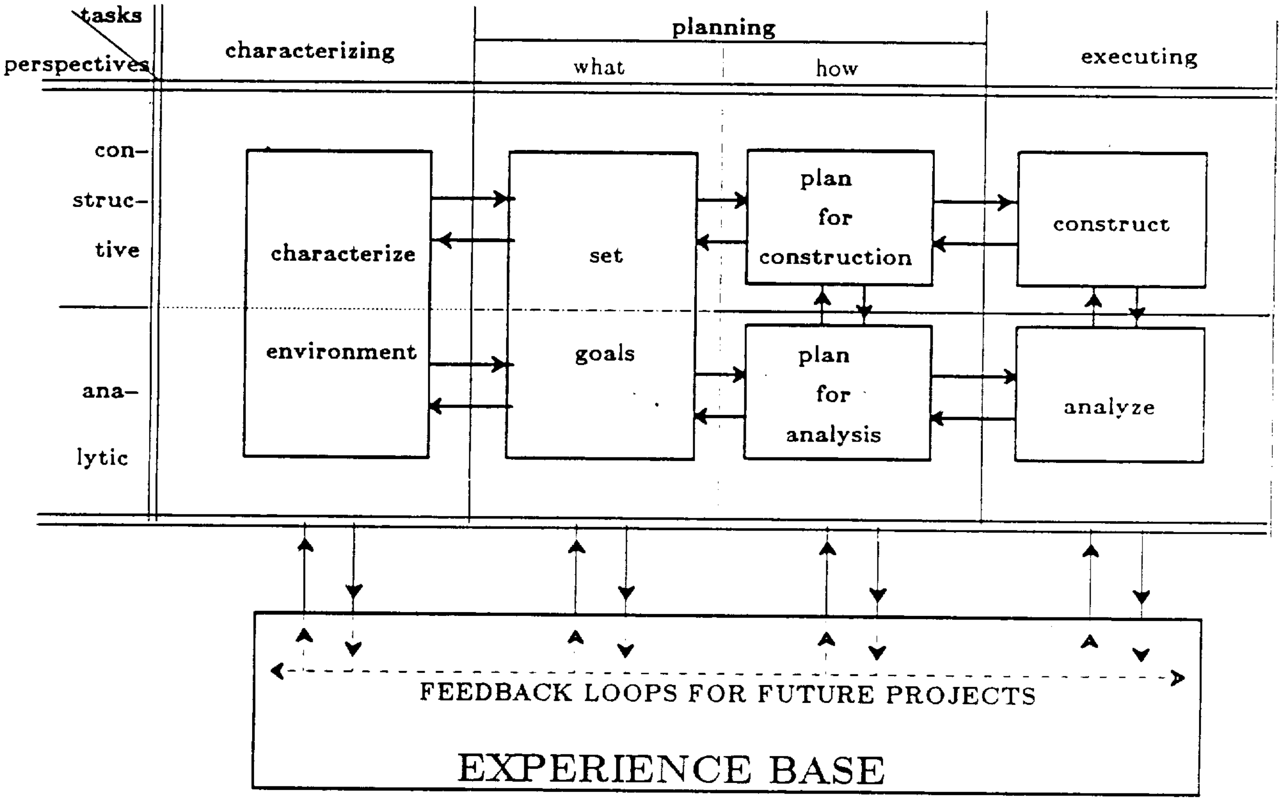


FIGURE 3: THE TAME SYSTEM

The TAME system offers an architecture for a software engineering environment that supports the goal generation, measurement and evaluation activities. It is aimed at providing automated support for managers and engineers to develop project specific goals and specify the appropriate metrics needed for evaluation. It provides automated support for the evaluation and feedback on a particular project in real time as well as help prepare for post mortems.

The Tame project was initiated to understand how to automate as much of the paradigm as possible using whatever current technology is available and to determine where research is needed. It provides a vehicle for defining the concepts in the paradigm more rigorously.

A major goal for the TAME project is to create a corporate experience base which incorporates historical information across all projects with regard to project, product and process data, packaged in such a way that it can be useful to future projects. This experience base would contain as a minimum the historical data base of collected data and interpreted results, the collection of measured objects, such as project documents, and collection of measurement plans, such as GQM models for various projects. It should also contain combinations and synthesis of this information to support future software development and maintenance.

TAME is an ambitious project. It is assumed it will evolve over time and that we will learn a great deal from formalizing the various aspects of the Improvement Paradigm as well as integrating the various sub-activities. It will result in a series of prototypes, the first of which is to build a simple evaluation environment. Building the various evolving prototypes and applying them in a variety of project environments should help us learn and test out ideas.

Tame provides mechanisms for instantiating the Improvement Paradigm by providing an experience base to allow the storing of experience so that it can be used on other projects (steps 1,2a), further defining the various steps to be performed (steps 1,2,3,4), and automating whatever is possible.

3. A REUSE-ORIENTED SOFTWARE ENGINEERING MODEL

The Improvement Paradigm, as instantiated in the TAME system, assumes that improvement can be achieved by iterating planning, execution of plans, and feedback across projects within an organization. Feedback can be viewed as reusing experience from the ongoing or prior projects to improve the planning or execution of ongoing or future projects. Learning can be viewed as the process of accumulating and packaging experience so it can be reused effectively. Thus, the paradigm explicitly recognizes the need to capture and reuse knowledge, products and processes from prior projects.

On the other hand, it should be noted that reuse can be an effective mechanism only if it is paired with learning and viewed as an integral part of an improvement-oriented software evolution process model. If we accept the fact that a better understanding of a process allows for more effective reuse, "reuse orientation" and "improvement orientation" of a process model are identical attributes. Both are supported by experimentation.

In a traditional software process model, learning and reuse only occur because of individual efforts or by accident. They are not explicitly supported and called out as desired characteristics of the development process. As a consequence, this experience is not owned by the organization (via the project database) but rather owned by individual human beings and lost after the project has been completed. A reuse-oriented process model must view reuse, learning and feedback as integral components, and place all experience, including software evolution methods and tools, under the control of an experience base [20].

Since improvement requires the feedback of available experience and feedback is based on learning and reuse activities, a requirement for such a process model is that it support systematic learning and reuse. Systematic learning requires support for the off-line recording, generalizing or tailoring, and formalizing of experience. Systematic reuse requires support for (re-)using existing experience. Off-line activities are performed independent of any particular project in order to improve the reuse potential of existing experience in the experience base.

Project goals are typically directed towards the development of a specific system. Thus off-line activities must have their own organizational structure. They cannot be part of the normal development organization because they require a different focus, a different set of processes, and an independent cost base.

For example, the objective of the recording process is to create a repository of well specified and organized experience. It requires effective mechanisms for collecting, validating, storing and retrieving experience. This should not be part of the project focus. The project can contribute by making its experience available to this independent organization, but cannot itself oversee the recording. It might not even be clear to the project what is worth recording.

The objective of generalizing existing experience prior to its reuse is to make a candidate reuse object useful in a larger set of potential target applications. The objective of tailoring existing experience prior to its potential reuse is to fine-tune a candidate reuse object to fit a specific task or exhibit special attributes, such as size or performance. Clearly a project cannot afford to generalize or tailor experience for another project within its budget constraints. Even worse, it may not have the perspective to do so since objectives and characteristics are different from project to project, and even more so from environment to environment. Generalizing and tailoring require a broader perspective of the organization and the products it develops.

The objective of formalizing existing experience prior to its potential reuse is to encode it in more precise, better understood ways. Off-line tailored or generalized experience needs to be formalized to increase its reuse potential and satisfy general reuse needs within an organization. The more we can formalize experience, the better it can be reused.

Formalization activities include the movement from informal knowledge (e.g., concepts), to structured or schematized knowledge (e.g., methods), or even to completely formal knowledge or automation (e.g., tools). It requires models of the various reuse objects, notations for making the models more precise, notations for abstracting reuse object characteristics, mechanisms for validating these models, and mechanisms for interpreting models in the appropriate context. Clearly the project has neither the budget nor the need to formalize its own experience.

Reuse requires a precise specification of the reuse context including the evolution process that is expected to enable reuse, and the characteristics of the available candidate reuse objects. The objective of a reuse–oriented software evolution process model is to support the use of previously accumulated experience during such reuse activities as: (a) specifying reuse needs in a way that allows matching them with descriptions of available experience, (b) finding and understanding appropriate reuse candidates, (c) evaluating reuse candidates in order to pick the most promising candidate, (d) actually tailoring the reuse candidate if necessary, (e) integrating the reuse candidate into the ongoing software project, and (f) evaluating the software project.

A reuse–oriented software evolution environment is an integral part of the improvement paradigm. The mechanisms supplied by the TAME system to support that paradigm are consistent with the mechanisms needed to support the reuse environment model with its experience base. It provides a mechanism for evaluating the recorded experience, helping us to decide what and how to reuse, tailor and analyze. It captures experience in the form of data from which models can be built to formalize experience. It supports continuous learning.

It is clear that an experience base is a key component of the reuse and improvement paradigms. A project needs help in accessing the reusable experience. If the experience is available (recorded), appropriate (tailored or generalized), and well–packaged (formalized), it can be used by a project. But an experience base is more than a physical entity. It is an organization that must support all the off–line activities that support its creation and use.

4. DIVIDING UP THE RESPONSIBILITIES AND ACTIVITIES

Based upon the prior discussion, the implementation of the Improvement Paradigm would best be served by two separate and distinct organizational structures. One organization is project–oriented. Its goal is to deliver the systems required by the customer. We will call this the Project Organization. The other organization, which we will call the Experience Factory, will have the role of monitoring and analyzing project developments, developing and packaging experience for reuse in the form of knowledge, processes, tools and products, and supplying it to the Project Organization upon request. The Experience Factory represents the experience base discussed above and the various activities associated with building and modifying it, controlling its access, and interfacing to the Project Organization.

Each project in a Project Organization can choose its process model based upon the characteristics of the project, taking advantage of prior experience with the various process models from the experience base in the Experience Factory. It can access information about prior system requirements and solutions, effective methods and tools and even available system components. Based upon access to this prior experience, the project can choose and tailor the best possible process, methods and tools. It can reuse prior products tailored to its needs.

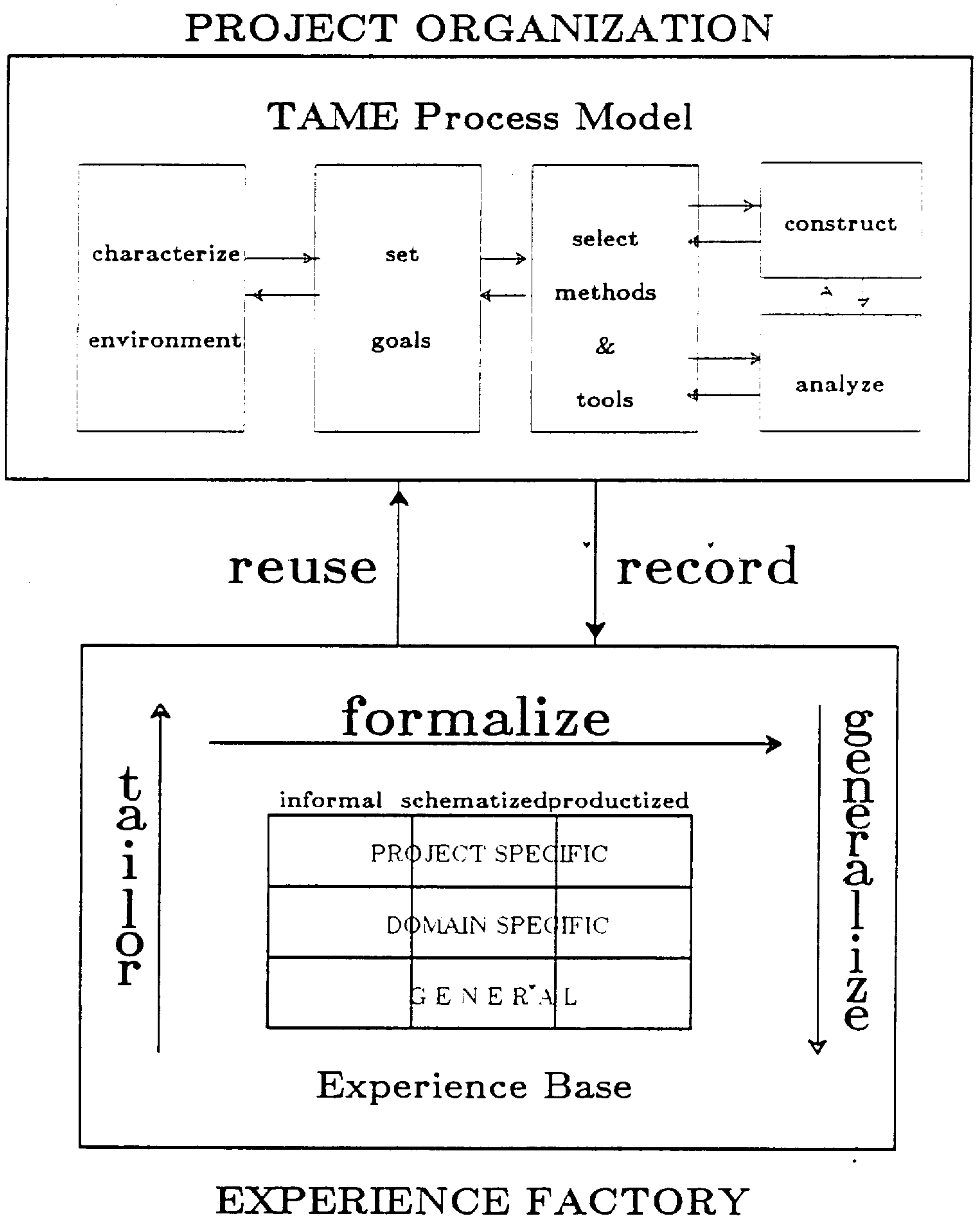


FIGURE 4: IMPROVEMENT AND REUSE ORIENTED-SOFTWARE ENGINEERING MODEL

The Experience Factory analyzes the project development for all systems developed by the corporation. Based upon this it recognizes commonality among projects, generalizes knowledge and packages it for use across all projects. It creates a repository of reusable information. For example, it can develop resource models, defect models, and risk management models and tailor them for the particular projects. It can develop processes, methods, techniques and tools and tailor them based upon the characteristics of the particular project. This can be accomplished based upon the Factory's analysis of the success and failure of the various activities across many projects. It can generate system components, at various levels of the architectural hierarchy based upon its recognition of commonality.

4.1. Some Specific Activities in the Project Organization

Let us consider the activities of the Project Organization with regard to the development of a system and how it might use the Experience Factory while applying the improvement paradigm.

At the start of a project, project management functions consist of activities such as resource and schedule planning, organizing, and staffing. These are covered by the characterizing and planning functions in the Improvement Paradigm.

During the characterizing phase, based upon its needs and characteristics, the project can access the experience base for the information about similar previous projects. This provides the project manager with a context for planning that includes resource estimation and allocation information, personnel experience, software and hardware available for reuse, environmental characteristics of concern and sets of baselines for resources, schedules, defects, etc. The project can store information on its own characteristics back into the experience base for analysis.

During the planning phase, the project can analyze prior goals and use them as defined or tailor them (or have them tailored by the Component Factory) for its needs. It can access the collection of construtive and analytic methods and tools, that have been effective and choose the appropriate ones that will help satisfy its goals. The goals and methods are influenced by the knowledge gained from the characterization phase, specifically with regard to elements of prior systems that can be reused. These elements include data, such as baselines, process models that have been successful, including methods and techniques that have been tailored and tools that support those methods, and components of prior projects such as requirements, design or code that can be adapted for the current project. The goals define the kinds of data that need to be collected as well as the mechanisms needed for collection. This provides the manager with information about what feedback will be provided for the project during development. The goals and process model, as tailored for the project, are stored in the experience base for monitoring the current project and expanding the experience base for future projects.

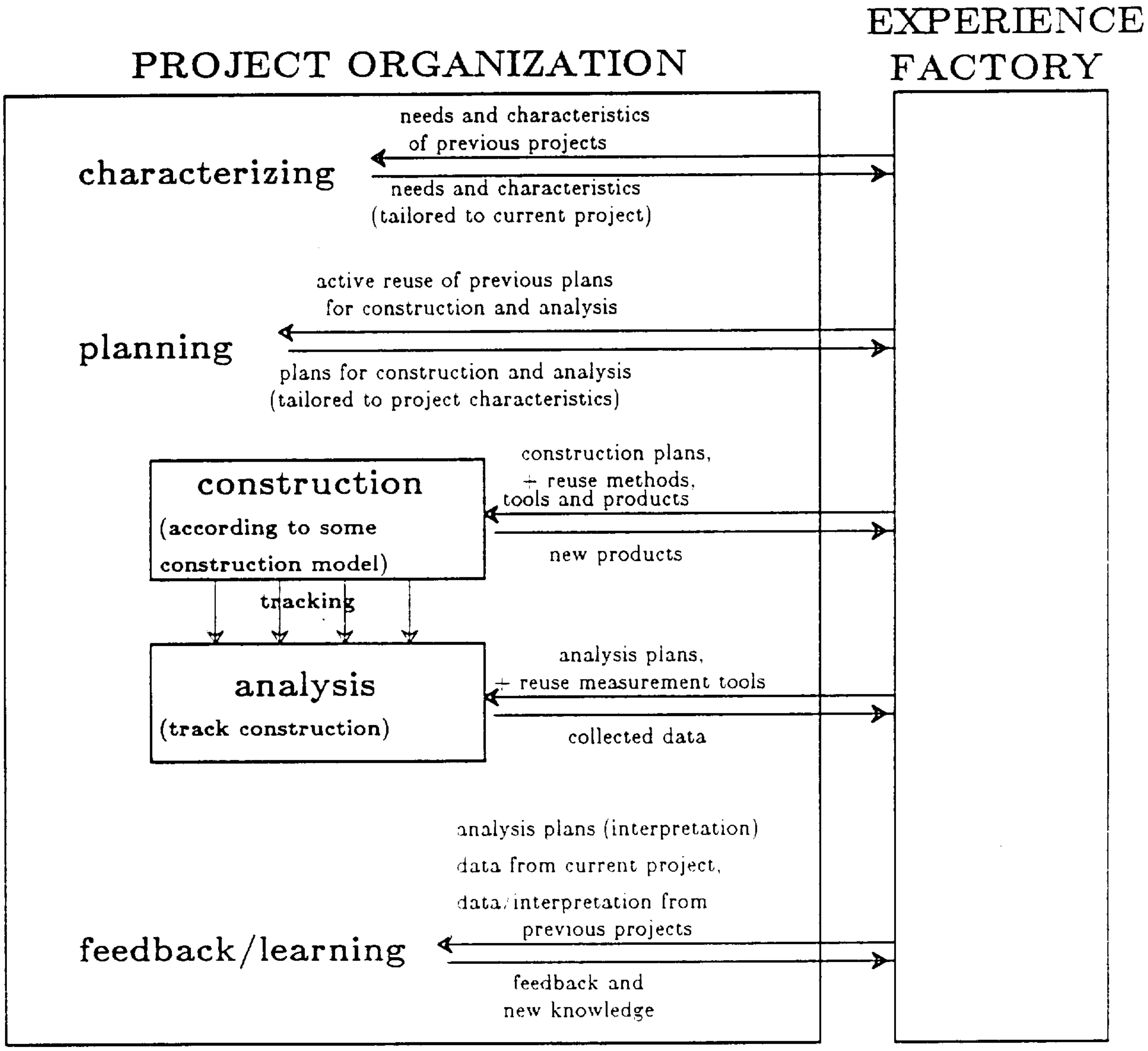

EXPERIENCE FACTORY
PROJECT ORGANIZATION
characterizing
needs and characteristics of previous projects
needs and characteristics (tailored to current project)
planning
active reuse of previous plans for construction and analysis
plans for construction and analysis (tailored to project characteristics)
construction
(according to some construction model)
construction plans, + reuse methods, tools and products
new products
tracking
analysis
(track construction)
analysis plans, + reuse measurement tools
collected data
feedback/learning
analysis plans (interpretation)
data from current project,
data/interpretation from previous projects
feedback and new knowledge

FIGURE 5: ACCESSING THE EXPERIENCE FACTORY

Project execution covers the directing and controlling activities as well as the development activities.

During the execution phase, the project proceeds using the tailored process model, methods, techniques, and tools as specified in the planning phase. It uses prior product parts, supplied by the experience base. Feedback is supplied to project management to support directing and controlling of the project. During execution, project experiences, components and data are returned to the Experience Factory and feedback is provided to the project.

At project conclusion, the overall project is analyzed and the results are fed back to the project as well as packaged and incorporated into the experience base for use on future projects.

4.2. Some Specific Activities in the Experience Factory

The Experience Factory plays several roles. It builds and maintains the experience base, it interfaces with the project in the Project Organization by providing information from the experience base and developing those elements that are requested by the project based upon its current level of expertise, e.g., tailored methods and tools and software components, and it acts as a quality assurance organization, providing feedback to the project with respect to its goals. As such it has several process models associated with it.

In building and maintaining the experience base, the Experience Factory performs the learning and reuse activities of recording, generalizing and tailoring, and formalizing. The degree to which it can perform these activities depends upon the breadth and depth of the information available and the level of technology.

It records information gathered from the various project developments. For example, it saves experiences from the projects it is monitoring, such as code modules, lessons learned on the project from the application of the constructive and analytic processes and measurement data, such as resource and defect data.

It generalizes or tailors the information that it has gathered. For example, it uses the project-specific measurement data across several projects to create baselines such as defect profiles; it develops generic packages from project specific packages or instantiates a generic package for a specific project; it refines a design technology based on the lessons learned from applying it on a specific project; it parameterizes a cost model for a project or uses data from the project to improve the estimation capability of the model.

It formalizes the information in the experience base to enhance its reuse potential. For example, it supplies code modules with their functional specifications and other appropriate documentation such as characterizing attributes, when needed; it makes more precise the steps in applying a method based upon lessons learned from its application; it builds cost models empirically based upon the data available; it develops management support systems based upon the available data and lessons learned; it builds automated support for methods.

In responding to requests from a project, it provides whatever information it has available from the experience base and the people. The level of support clearly depends upon the state of the art in the packaging of experience. The interface with the Project Organization will change over time, starting with small packets of experience and building to higher level ones.

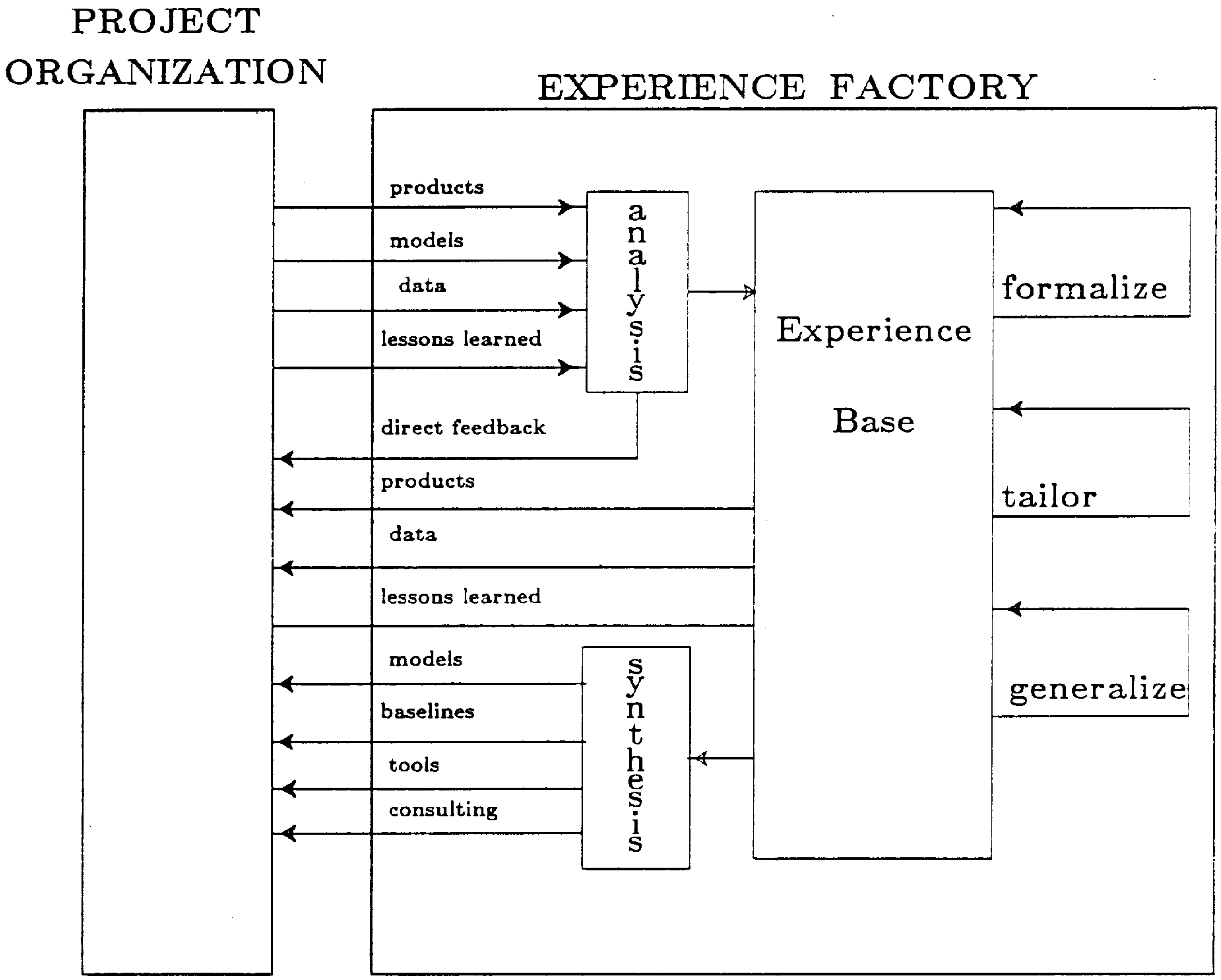


FIGURE 6: ACTIVITIES IN THE EXPERIENCE FACTORY

The actual information supplied depends upon the request and what is currently available in the experience base. For example, during characterization, it provides baselines and estimation models, and information on packaged products, such as requirements templates or code modules. General defect baselines can be tailored to the specific project by limiting the projects considered to those with the same characteristics as the current project, e.g., same application domain, same process model.

During planning it supplies GQM models and process models, methods, tools and techniques. These can be obtained directly from the experience base or tailored for the needs of the project. For example, assuming that inspections are chosen for the project and knowing the classes of faults found in similarly classified projects, the component factory might tailor the reading technology within inspections to concentrate on locating the kinds of faults that tend to occur in this type of project. They can also provide training and consulting on the use of the methods and models.

During project execution, they can act as a contractor supplying various levels of project components. In fact from the Project Organization perspective, any component that can be well specified can be delivered by the Experience Factory. In turn, the Experience Factory can respond to the request by delivering an existing component, modifying an existing component, e.g. instantiating a generic package from the experience base, or developing the component from scratch and adding it to the experience base.

If we view quality assurance as the act of leading, teaching, and auditing the process, then it implies an organizational structure independent but interactive with the projects. (Note that this is different from quality control, which we define as the act of directing, influencing, verifying, and correcting the product, which implies a project controlled organization.) The Experience Factory is an ideal location for the quality assurance activities.

In acting as a quality assurance organization, the Experience Factory audits activities and collects the prescribed data, provides feedback to the project in real time, and offers training in the various planning, constructive, and analytic approaches. The quality assurance activities is consistent with the activities of building and maintaining the experience base and responding to requests from the Project Organization. It also provides an independent chain of command and a corporate perspective with regard to goals, data collection, process and products.

4.3. Viewing the Experience Factory as a Component Factory

As a particular dimension of the Project Organization and the Experience Factory, consider the activities of the Project Organization with regard to the development of a system and how it might use the Factory from the point of view of code development, e.g., as a Component Factory. We can view the project organization within the Project Organization as having the following activities:

Requirements Definition: The system analysts will interact with the customer to determine project requirements. It is assumed that the analysts will know the application domain and what is available in the repository for reuse. They will have access to repository information about what kinds of components are available so they can make tradeoff decisions, negotiating with the customer for function vs. price.

Initially, this negotiation will be limited since the repository will be sparsely populated. This should change over time as the repository fills with components. It should be noted that the system analyst can use Factory components for building and analyzing prototypes of the system.

Specification and Design: The requirements will be turned into a system design and specification for the required components. Those components that can be well specified can be turned over to the Experience Factory and orders will be filled for components.

Initially, the specifications will be for low level components since the Factory will begin bottom up. As time goes on and the repository builds up in terms of components, and the technology for recognizing, specifying and integrating larger pieces of systems develops, larger components can be ordered.

The Experience Factory operates according to several process models. When an order for a component arrives, it can check its repository for the appropriate component or order it externally if it is available from an outside vendor. It can develop it from scratch, using verification technology, based upon the fact that it has the specification and the component it is developing is limited in size. However, given that it has been required to deliver such a component, it can decide whether the component is of general use, from its knowledge of other projects, and can generalize or tailor the component, package it with the necessary attributes for future reuse and store it in the repository.

As an initializing activity, the Factory can analyze prior systems for reusable components and re-engineer them to seed the repository. It can develop components, so they are easy to combine, modify with respect to certain criteria and label and package appropriately.

Integration and Evaluation: The project will have the task of integrating the components into its own specified design. These integrated components might be returned to the Factory for future use. It will then evaluate the system based upon the customer requirements and deliver the system.

5. IMPLICATIONS OF THE NEW LIFE CYCLE ORGANIZATION

5.1. Implications for Corporations

One of the major problems with software development in the past has been that projects have been unable to explicitly reuse experience from prior projects or contribute to the experience base for future projects. This has been due in part to the fact that immediate project delivery goals and the more long-range goals of reuse and learning are distinct and not easily paired. Project schedule often takes precedence over the luxury of passing on learned experience.

The new life cycle organization divides the focus of software development into two separate organizations. It separates the immediate project goals from the long range learning and reuse–oriented goals. In the approach, the Project Organization can focus on the customer needs and has the advantage of access to a knowledgeable support organization in the form of the Experience Factory. The Experience Factory focuses on the organization's goals to learn and reuse. It has the advantage of accumulating experience from a large number of projects which provides it with a broader perspective than any particular project.

This organizational structure has many advantages. It should promote higher quality and productivity because of reuse and learning. It can provide better and more focused education and training for developers and provide better methods and tools for them to use.

It provides the corporation with a corporate asset in the guise of the Experience Factory. The Experience Factory contains everything the organization has learned and developed that is useful for future developments as well as an assessment of the status of corporate quality and productivity. As the Experience Factory grows in its role and assets. the corporation can learn more and more from the various experiences across the corporation.

There will be more emphasis on formalization of all parts of management and development. Formal verification becomes cost effective since the correct units will be used in many systems; it becomes more applicable since we will be applying it to smaller units, at least in the beginning, where the technology is manageable. Formal models of risk assessment can be used since the experience base should provide a broad basis for understanding and comparison.

The organizational scheme has the advantage that it can start small and expand with the growth in technology and the experience base. However. there are several issues that must be dealt with in putting this organization in place. e.g. financial and organizational.

This organization requires separate cost centers for the Project Organization and the Experience Factory. There are several models of how the funding of the Experience Factory might work. For example. it could be funded out of corporate overhead which would grow with the success of the factory or projects could be billed for factory items. The right model will depend upon the company and the organization and politics within that company.

This organization requires a careful definition of management and responsibility structures. It is clear that we do not want to create new conflicts over responsibility for problems with packaged experience.

This organization needs to be motivated and supported. Incentive and reward structures need to be developed. We will need to learn from experience gained from applying different financial and management structures.

5.2. Implications for Research

There are several implications for research based upon this organizational structure. Many of the technologies already developed for programming in the small are applicable in the factory domain. For example, verification technology is already available for factory produced components and it is necessary and cost effective because those units will be reused many times. Research activities can focus on the transfer of these technologies. Therefore, user friendly tools to support verification are needed. Based upon this formalization, we should learn more about the relevant primitives for particular application domains and how to encapsulate them.

There are research activities associated with defining and tailoring models. These include process models, methods and tools; product models of the various products and qualities of those products; and models of information, like goal generation languages, cost, resource allocation, risk, and defect prediction. Models must be defined for the Project Organization and the Experience Factory and must take into account their interface. This involves the definition of languages for defining these models and tool generators, i.e. tools that can be instantiated to support variations of a method.

There are research activities associated with generating larger product units from the Experience Factory. These include defining models of module interconnection languages that scale up, combining specifications and verifying them, and combining test plans to validate integrated components.

There are research activities associated with the building and accessing of the experience base, e.g., mechanisms for encoding lessons learned into a model, tools for generating goals and mapping them onto measures, models that permit the model to learn automatically.

5.3. Implications for Education

The organizational scheme provides a focus for many of the technologies already taught at the University and so makes much of the current education more relevant. Topics that require more emphasis are formalisms of all kinds, e.g., verification technologies, formal requirements and specification notations, formal models of measurement and management. There is a need to teach students how to develop, use and assess methods and tools and deal with access and retrieval of libraries. Reuse and learning technologies need to be made available.

There is a clear entry path for new software engineers through the Component Factory where they can develop small components under careful guidance and tool support and learn from the general experience base. As their experience grows they can be moved into any of the other higher level activities, e.g., the Project Organization, or other parts of the Experience Factory.

6. RESEARCH ACTIVITIES AT MARYLAND THAT SUPPORT THE NEW LIFE CYCLE

The paradigms and organization described in this paper offer a framework for research that focuses on the key issues for improving the software process and product in a context that permit the research to be used and experimented with in an industrial setting. Over the past dozen years, at the University of Maryland, we have been working on several research projects whose goal is to evolve to this framework.

The projects are organized into those dealing with the instantiation of the improvement paradigm in the SEL [5,46], where the concepts of the Project Organization and the Experience Factory have been evolving, the TAME project which is automating support for this framework in a formal way, and a variety of other projects which are attempting to understand, formalize and improve various process and product characteristics.

A major source of activity has been the Software Engineering Laboratory (SEL), a joint venture of the NASA Goddard Space Flight Center, the University of Maryland, and Computer Sciences Corporation. The SEL has informally acted as an Experience Factory that supports project development. The application domain is ground support software for satellites. We have been building models and supplying these models and lessons learned back to projects so they can improve their process and product. This work has been performed via experiments of various kinds, dealing with resource, defect, process, and product models.

In an attempt to better understand the environment we have used data collected during development to build various descriptive models of the SEL environment. In this way we have formalized knowledge from raw data to formal models or baselines and made the results available to the project organization for use in characterizing, planning and evaluating the project.

We have collected data on resource expenditures, applied various existing models [32,47,49,61] and eventually built and tailored models that explicitly described resource allocation in the SEL environment [2,8,10,15,30]. These are used for estimating, planning and evaluating new projects.

We have developed baselines for defects by accumulating defect data over many projects [62]. These defects a classified by phase and type. They vary with different project classifications [16]. They provide insight into the environment, support for project management and evaluation, and point to areas areas that need improvement in the process [18].

We have used various product metrics [41,45] to provide insight into the characteristics of the products being developed as well as evaluating the usefulness of these metrics for the SEL environment [6,11,26,42]. Areas of new technology that have been introduced, like Ada have generated the need for developing new metrics to characterize new product qualities [12,40]. We have used these metrics as baselines to provide the project manager with insights as to what the problems may be with the current development [38].

With regard to process improvement, we have built descriptive and perscriptive models of processes, methods and techniques and experimented with their application. The results of our studies are formalized and reused for future projects within the limits of the technology available.

In some cases, we have performed controlled experiments in which we analyzed the effects of various methods and techniques before recommending them on actual projects. We would then perform case study experiments to evaluate the effect of the method or technology on an actual project development to assure that it scales up and is applicable to the SEL environment. For example, we ran controlled experiments on a set of structured programming methods and techniques [17], various testing and reading techniques [23.54]. object oriented design in Ada [12,40] and the Cleanroom process model [59].

We then apply these approaches to projects within the project organization. We evaluate their effect there, and make recommendations. write lessons learned documents. and refine or change the models to incorporate what we have learned. In this way, the experiences gained from applying a particular model from the experience base is improved based upon the lessons learned from applying the model so that it can be used for future projects. Two case studies currently being run in the SEL, based upon controlled experiments. are the use of object oriented development in Ada [1,14,35] and the application of the Cleanroom process.

In other instances we have developed models and experimented directly on the projects. For example, we have evaluated the test methodology used for acceptance test [51] and the methods used for maintenance [55].

Parts of the data collection process have been automated for the FORTRAN environment [37. 43] and are being automated for the transition to Ada [39]. Other tools have been developed that help support the various technologies used. Parts of the evaluation process have been automated using a knowledge base to create a decision support system [50,60].

The Tame project has focused on the architecture for the measurement and evaluation processes [19]. Work has been done by using studies performed in the SEL to define the process improvement mechanisms [18]. We have devised a resource planning and feedback model that is consistent with the Improvement Paradigm [43].

The Goal/Question/Metric Paradigm has been applied in a variety of environments other than the SEL and has evolved based upon these activities [29.53,58].

We are currently working on supporting the automation of the generation of operational goals in a reasonably complete and consistent manner. A key aspect of the approach is that project personnel can generate goals that can be measured and evaluated. We are working on extending the GQM templates into a goal generation language that will aid the goal writer in articulating questions and metrics based upon the goal and the model of the object of interest. We are currently experimenting with hypertext and attribute grammar technology to develop prototypes of this automated support mechanism.

We are in various stages in the development of three measurement tools for analyzing programs in Ada and C. A source code analyzer for various syntactic metrics, such as cyclomatic complexity and software science metrics, has been developed for Ada (ASAP) [38] and C (CSAP). A structural coverage analyzer (SCA) is under development for Ada [63]. Data bindings analyzers are being developed for Ada and C based on prior versions of the tools for FORTRAN, SIMPL, and PL/C.

We have developed a set of requirements and defined a system architecture for measurement tool generations using a parser generator that retains the parse tree for further transformations [48], an enhancement of YACC and are experimenting with the prototypes of this tool generation system.

With regard to reuse, we have developed a model of a reuse support environment that can exist within the TAME framework [20]. We have applied the model to the maintenance process to show the advantages of viewing maintenance as a reuse process [7].

We are developing a model of reuse consistent with the approach presented in this paper that classifies the objects as they exist in the experience base, the reuse activities and the objects as they are reused [21]. For example, the reusable object can be classified according to the characteristics of the unit itself, its interfaces, and its context. The model recognizes the need to assess the qualities of the reusable object based upon the characteristics of the project in which it will be reused.

We are working on a language and support system that takes elementary processes and generalizes them into more complex processes. Elementary processes correspond to the "basic algorithms" used to perform small tasks, such as the addition of two atomic units. Our goal is to identify useful sets of elementary processes, and then show how they can be combined and extended to perform more complex actions (such as the addition of a stream of atomic units.) Using our language, abstract data structures may be mapped onto particular structures (e.g., the addition process for streams could be mapped onto a process for addition of arrays of numbers), and also composed with other structures (e.g., an array addition task could be composed with a division task in order to create a module for computing means.) Finally the system will package resulting processes into an acceptable language component, whether a procedure or function. Our current langague supports only functional processes, a future step is to support the creation of data abstractions or modules.

To study the issue of code reuse, the LASER project is currently building a system that examines exising systems in order to study and extract code that can be reused to seed a component repository. The system measures the various components in the system and identifies candidate reusable components based upon their lack of complexity, reusability within the existing system, independence, etc. These candidate components are then isolated (made independent) and qualified. The qualification involves the catagorization and classification based upon a number of attributes, and the association of a functional specification with the component.

The approach expressed here provides a focus for further research issues. Some of the questions for which work has begun are:

- How can process models be formally expressed so they can be communicated, analyzed and tailored?
- How can various models be stored so they can be accessed by the GQM tool and help generate the automated collection of the appropriate measures?
- How can a specific process model be developed that satisfies the definition and storage of the prior two questions?
- How can we better capture and reuse experiences in the form of lessons learned from previous efforts?
- What other measurement data can be automatically collected?
- How could the set of measurement tools defined above be developed so that they can be tailored for various types of measures, maximizing the reuse of system components among the tools and the language independence?
- How can we classify experience so it can be appropriately reused?
- Based upon a specification, how can a component be devised quickly from elementary processes?
- How can we transform existing components to make them more independent, and measure the cost of reuse?
- How can we have confidence that the factory-provided modules will do what we want?
- How can we integrate aggregates of modules with their associated attributes so that they can be analyzed, managed, and controlled?
- How can we verify properties of aggregates of modules, not just individual modules?
- How can the test plans for components be combined to provide a test plan and oracle for aggregate application structures?

7. CONCLUSIONS

The approach expressed in this paper has evolved over years of studying and experimenting with software development and maintenance. It provides a compatable and consistent framework for both software development and software engineering research. It recognizes and takes advantage of the experimental nature of software engineering.

It allows us to understand how things are being done and where the problems are by studying the process and product in actual environments. It allows us to formalize models of the process, product and knowledge. These models can then be analyzed. They can be used to form a basis for research and at the same time provide immediate input to project development.

From a research perspective, it provides a focus for research problems based upon problems that need to be solved. It provides a framework to tie together existing pieces of research.

From a corporate perspective, the approach can be applied directly and the organization can grow and build its own experience base. It supports technology transfer in a natural way and it ties the research and development organizations closer together.

REFERENCES

[1] W. Agresti, "SEL Ada Experiment: Status and Design Experience," Proceedings of the Eleventh Annual Software Engineering Workshop, NASA Goddard Space Flight Center, Greenbelt, MD, December 1986.

[2] J. Bailey, V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," Proceedings of the Fifth International Conference on Software Engineering, San Diego, USA, March 1981, pp. 107-116.

[3] V. R. Basili, "Data Collection, Validation, and Analysis," in Tutorial on Models and Metrics for Software Management and Engineering, IEEE Catalog No. EHO-167-7, 1981, pp. 310-313.

[4] V. R. Basili, "Quantitative Evaluation of Software Engineering Methodology," Proc. of the First Pan Pacific Computer Conference, Melbourne, Australia, September 1985 [also available as Technical Report, TR-1519, Dept. of Computer Science, University of Maryland, College Park, July 1985].

[5] V. R. Basili, "Can We Measure Software Technology: Lessons Learned from 8 Years of Trying," Proceedings of the Tenth Annual Software Engineering Workshop, NASA Goddard Space Flight Center, Greenbelt, MD, December 1985.

[6] V. R. Basili, "Evaluating Software Characteristics: Assessment of Software Measures in the Software Engineering Laboratory," Proceedings of the Sixth Annual Software Engineering Workshop, NASA Goddard Space Flight Center, Greenbelt, MD, 1981.

[7] Victor R. Basili, "Software Maintenance = Reuse-Oriented Software Development," in Proc. Conference on Software Maintenance, Key-Note Address, Phoenix, AZ, October 1988 [also available as Technical Report, TR-2244, Dept. of Computer Science, University of Maryland, College Park, July 1985].

[8] V. R. Basili, J. Beane, "Can the Parr Curve help with the Manpower Distribution and Resource Estimation Problems," Journal of Systems and Software, vol. 2, no. 1, 1981, pp. 47 - 57.

[9] V. R. Basili, G. Caldiera, "Reusing Existing Software," Technical Report-2116, Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland, October 1988.

[10] V. R. Basili, K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," Journal of Systems and Software, vol. 2, no. 1, 1981, pp. 47-57.

[11] V. R. Basili, D. H. Hutchens, "An Empirical Study of a Syntactic Measure Family," IEEE Transactions on Software Engineering, vol. SE-9, no. 11, November 1983, pp. 664-672.

[12] V. R. Basili, E. E. Katz, "Metrics of Interest in an Ada Development," Proc. of the IEEE Computer Society Workshop on Software Engineering Technology Transfer, April 1983, pp. 22-29.

[13] V. R. Basili, E. E. Katz, "Examining the Modularity of Ada Programs," Proc. of the Joint Ada Conference, Arlington, Virginia, March 16-19, 1987.

[14] V. R. Basili, E. E. Katz, N. M. Panlilio-Yap, C. Loggia Ramsey, S. Chang, "Characterization of an Ada Software Development," IEEE Computer Magazine, September 1985, pp. 53-65.

[15] V. R. Basili, N. M. Panlilio-Yap, "Finding Relationships Between Effort and Other Variables in the SEL," IEEE COMPSAC, October 1985.

[16] V. R. Basili, B. Perricone, "Software Errors and Complexity: An Empirical Investigation," ACM Communications, vol. 27, no. 1, January 1984, pp. 45-52.

[17] V. R. Basili, R. Reiter, Jr., "A Controlled Experiment Quantitatively Comparing Software Development Approaches," IEEE Transactions on Software Engineering, vol. SE-7, no. 5, May 1981, pp. 299-320.

[18] V. R. Basili, H. D. Rombach, "Tailoring the Software Process to Project Goals and Environments," Proc. of the Ninth International Conference on Software Engineering, Monterey, CA, March 30 - April 2, 1987, pp. 345-357.

[19] V. R. Basili, H. D. Rombach "The TAME Project: Towards Improvement-Oriented Software Environments," IEEE Transactions on Software Engineering, vol. SE-14, no. 6, June 1988, pp. 758-773.

[20] V. R. Basili, H. D. Rombach, "Software Reuse: A Comprehensive Framework," CS-TR-2158, Department of Computer Science, University of Maryland, College Park, Maryland.

[21] V. R. Basili, H. D. Rombach, J. Bailey, and B. G. Joo, "Software Reuse: A Framework," Proc. of the Tenth Minnowbrook Workshop on Software Reuse, Blue Mountain Lake, New York, July 1987.

[22] V. R. Basili, R. W. Selby, Jr., "Data Collection and Analysis in Software Research and Management," Proc. of the American Statistical Association and Biomeasure Society Joint Statistical Meetings, Philadelphia, PA, August 13-16, 1984.

[23] Victor R. Basili, R. W. Selby, "Comparing the Effectiveness of Software Testing Strategies," IEEE Transactions on Software Engineering, Vol. SE-13, No. 12, December 1987, pp. 1278-1296.

[24] V. R. Basili, R. W. Selby, Jr., "Calculation and Use of an Environment's Characteristic Software Metric Set," Proceedings of the Eighth International Conference on Software Engineering, London, UK, August 1985.

[25] V. R. Basili, R. W. Selby, D. H. Hutchens, "Experimentation in Software Engineering," IEEE Transactions on Software Engineering, vol.SE-12, no.7, July 1986, pp.733-743.

[26] V. R. Basili, R. W. Selby, and T.-Y. Phillips, "Metric Analysis and Data Validation Across Fortran Projects," IEEE Transactions on Software Engineering, vol. SE-9, no. 6, November 1983, pp. 652-663.

[27] V. R. Basili, A. J. Turner, "Iterative Enhancement: A Practical Technique for Software Development," IEEE Transactions on Software Engineering, vol. SE-1, no. 4, December 1975.

[28] V. R. Basili, D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," IEEE Transactions on Software Engineering, vol. SE-10, no.6, November 1984, pp. 728-738.

[29] V. R. Basili, D. M. Weiss, "Evaluation of a Software Requirements Document by Analysis of Change Data," Proceedings of the Fifth International

Conference on Software Engineering, San Diego, USA, March 1981, pp. 314–323.

[30] V. R. Basili, M. V. Zelkowitz, "Analyzing Medium Scale Software Development," Proceedings of the Third International Conference on Software Engineering, Atlanta, Georgia, USA, May 1978, pp. 116–123.

[31] B. W. Boehm, "Software Engineering," IEEE Transactions on Computers, vol. C–25, no. 12, December 1976, pp. 1226–1241.

[32] B. W. Boehm, "Software Engineering Economics." Prentice-Hall, Englewood Cliffs, NJ, 1981.

[33] B. W. Boehm, "A Spiral Model of Software Development and Enhancement," ACM Software Engineering Notes, vol. 11, no. 4, August 1986, pp. 22–42.

[34] B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative Evaluation of Software Quality," Proceedings of the Second International Conference on Software Engineering, 1976, pp. 592–605.

[35] C. Brophy, W. Agresti, and V. R. Basili, "Lessons Learned in Use of Ada Oriented Design Methods," Proc. of the Joint Ada Conference, Arlington, Virginia, March 16–19, 1987.

[36] W. J. Decker, W. A. Taylor, "Fortran Static Source Code Analyzer Program (SAP)," Technical Report SEL–82–002, NASA Goddard Space Flight Center, August 1982.

[37] C. W. Doerflinger, V. R. Basili, "Monitoring Software Development Through Dynamic Variables," IEEE Transactions on Software Engineering, vol. SE–11, no. 9, September 1985, pp. 978–985.

[38] D. L. Doubleday, "ASAP: An Ada Static Source Code Analyzer Program," Technical Report, TR–1895, Deptartment of Computer Science, University of Maryland, College Park, August 1987.

[39] M. Dyer, "Cleanroom Software Development Method," IBM Federal Systems Division, Bethesda, Maryland, October 14, 1982.

[40] J. Gannon, E. E. Katz, and V. R. Basili, "Measures for Ada Packages: An Initial Study," Communications of the ACM, vol. 29, no. 7, July 1986, pp. 616–623.

[41] M. H. Halstead, "Elements of Software Science," Elsevier North-Holland, New York, 1977.

[42] D. H. Hutchens, V. R. Basili, "System Structure Analysis: Clustering with

Data Bindings," IEEE Transactions on Software Engineering, August 1985, pp. 749–757.

[43] D. R. Jefferey, V. R. Basili, "Validating the TAME Resource Data Model." Proceedings of the Tenth International Conference on Software Engineering, Singapore, April, 1988, pp. 187–201.

[44] E. E. Katz, H. D. Rombach, and V. R. Basili, "Structure and Maintainability of Ada Programs: Can We Measure the Differences?," Proc. of the Ninth Minnowbrook Workshop on Software Performance Evaluation, Blue Mountain Lake, New York, August 5–8, 1986.

[45] T. J. McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering, December 1976, pp. 308–320.

[46] F. E. McGarry, "Recent SEL Studies," Proceedings of the Tenth Annual Software Engineering Workshop, NASA Goddard Space Flight Center, December 1985.

[47] F. N. Parr, "An Alternative to the Rayleigh Curve Model for Software Development Effort," IEEE Transactions on Software Engineering, vol. SE–6, no. 3, March 1980.

[48] J. Purtilo and J. Callahan, "Parse Tree Annotations", Communications of the ACM, to appear.

[49] L. Putnam, "A General Empirical Solution to the Macro Software Sizing and Estimating Problem," IEEE Transactions on Software Engineering, vol. SE–4, no. 4, April 1978, pp. 345–361.

[50] C. Loggia–Ramsey, V. R. Basili, "An Evaluation of Expert Systems for Software Engineering Management," IEEE Transactions on Software Engineering, Vol. 15, no. 6, June 1989, pp. 747–7597.

[51] J. Ramsey, V. R. Basili, "Analyzing the Test Process Using Structural Coverage," Proceedings of the Eighth International Conference on Software Engineering, London, UK, August 1985.

[52] H. D. Rombach, "Software Design Metrics for Maintenance," Proceedings of the Ninth Annual Software Engineering Workshop, NASA Goddard Space Flight Center, Greenbelt, MD, November 1984.

[53] H. D. Rombach, V. R. Basili, "A Quantitative Assessment of Software Maintenance: An Industrial Case Study," Conference on Software Maintenance, Austin, Texas, September 1987.

[54] H. D. Rombach, V. R. Basili, and R. W. Selby, Jr., "The Role of Code

Reading in the Software Life Cycle," Proc. of the Ninth Minnowbrook Workshop on Software Performance Evaluation, Blue Mountain Lake, New York, August 5-8, 1986.

[55] H. D. Rombach, B. T. Ulery, "Establishing a Measurement-Based Maintenance Environment Program: Lessons Learned in the SEL", Proceedings of the IEEE Conference on Software Maintenance, Miami Beach, October, 1989.

[56] W. W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques," Proceedings of the WESCON, August 1970.

[57] R. W. Selby, Jr., "Incorporating Metrics into a Software Environment." Proceedings of the Joint Ada Conference, Arlington. VA, March 16-19, 1987, pp. 326-333.

[58] R. W. Selby, Jr., V. R. Basili, "Analyzing Error-Prone System Coupling and Cohesion," Technical Report TR-88-46, Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland, June 1988.

[59] R. W. Selby, Jr., V. R. Basili, and T. Baker, "CLEANROOM Software Development: An Empirical Evaluation." IEEE Transactions on Software Engineering, Vol. 13 no. 9, September, 1987, pp. 1027-1037.

[60] J. D. Valett, "The Dynamic Management Information Tool (DYNAMITE):Analysis of the Prototype, Requirements and Operational Scenarios," M.Sc. Thesis, University of Maryland, 1987.

[61] C. E. Walston, C. P. Felix, "A Method of Programming Measurement and Estimation," IBM Systems Journal, vol. 16, no. 1, 1977, pp. 54-73.

[62] D. M. Weiss, V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory," IEEE Transactions on Software Engineering, vol. SE-11, no. 2, February 1985, pp. 157-168.

[63] L. Wu, V. R. Basili, and K. Reed, "A Structure Coverage Tool for Ada Software Systems," Proc. of the Joint Ada Conference, Arlington, Virginia, March 16-19, 1987.

[64] M. Zelkowitz, R. Yeh, R. Hamlet, J. Gannon, and V. R. Basili, "Software Engineering Practices in the U.S. and Japan," IEEE Computer Magazine, June 1984, pp. 57-66.

SECTION 3—MEASUREMENT ENVIRONMENT STUDIES

SECTION 3 - MEASUREMENT ENVIRONMENT STUDIES

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

- Integrating Automated Support for a Software Management Cycle Into the TAME System, V. Basili and T. Sunazuka, University of Maryland Technical Report TR-2289, July 1989
- Towards A Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment, V. Basili and H. Rombach, University of Maryland, Technical Report TR-2158, December 1988

UMIACS–TR–89–75
CS–TR–2289

July, 1989

Integrating Automated Support for a Software Management Cycle into the TAME System†

Toshihiko Sunazuka
NEC Corporation
Tokyo, Japan

Victor R. Basili
Institute for Advanced Computer Studies
Computer Science Department
University of Maryland
College Park, MD 20742

ABSTRACT

Software managers are interested in the quantitative management of software quality, cost and progress. There have been many of models and tools developed, but they are of limited scope. An integrated software management methodology, which can be applied throughout the software life cycle for any number purposes, is required.

The TAME (Tailoring A Measurement Environment) methodology, developed at the University of Maryland, is based on the improvement paradigm and the Goal/Question/Metric (GQM) paradigm. This methodology helps generate a software engineering process and measurement environment based on the project characteristics.

The SQMAR (Software Quality Measurement and Assurance Technology) developed in NEC is a software quality metric system and methodology applied to the development processes. It is based on the feed forward control principle. Quality target setting is carried out before the Plan–Do–Check–Action activities are performed.

These methodologies are integrated to realize goal–oriented measurement, process control and visual management. The Software Management Cycle is a substantiation of these concepts. Based on the TAME process model, development and management environments can be generated. The SQMAT system helps target setting, data analysis and visual display.

In this paper we discuss a metric setting procedure based on the GQM paradigm, a management system called the Software Management Cycle (SMC), and its application to a case study based on NASA/SEL data. A method for evaluation Software Management Cycle process is described. The expected effects of SMC are quality improvement, managerial cost reduction, accumulation and reuse of experience, and a highly visual management reporting system.

KEYWORDS

TAME, improvement paradigm, Goal/Question/Metric paradigm, SQMAR, Plan–Do–Check–Action activities, process control, visual management, software engineering process, goal–oriented measurement, software quality metrics.

† Research supported in part by NASA grant NSG-5123 and NEC (through the Industrial Associates Program of the Department of Computer Science.)

1. Introduction

Management plays a key role in the software development process. In the end, it is management's responsibility to produce and deliver a quality product productively and profitably and to generate corporate credibility with the customer. Thus, effective management methodologies are needed to support management in assessing the current status of the project and achieving delivery of the final system on-time, within budget, and with the specified product qualities. It would also be useful if the methodology supported the improvement of quality and productivity on the current project and on future projects. Many companies are working to provide such methods for their managers.

However, it is difficult to assess the current status of a project precisely because of the lack of visibility of the software during development. It is even more difficult to predict project progress because of the lack of clearly defined goals, the lack of feedback in the achievement of those goals, and the difficulties caused by the variation in personnel.

2. Supporting Methodologies

Thus, requirements for the management methodology include the ability to make the software as visible, quantifiable and objective as possible. Several methodologies and paradigms use metrics to satisfy these management needs during development. There have been many software metrics proposed in the literature that attempt to provide the visibility, quantification and objectivity [Boeh76, McRW77, Muri80, BaKa83].

From a customer perspective of product quality, a comprehensive set of quantifiable software characteristics were proposed by Boehm, et al. [Boeh76] and later refined by McCall and Walters [McRW77]. Based on these studies, Software Quality Metrics (SQM) was developed by Murine (METRIQS Incorporated) as a quantitative software quality assessment technology [Muri80].

SQMAT

Based upon the SQM, the NEC Corporation has developed a Software Quality Measurement and Assurance Technology (SQMAT) [AzSM87, AzSu86, SuAY85] and has been using it as one of the support tools in their software quality control (SWQC) group activities [Mizu82]. Quality control seminars are held periodically for every level of worker; programmer through general manager. The seminars are used to motivate as well as educate everyone with respect to the quality control technologies.

SQMAT is a software quality metric system and methodology applied to the development processes, which takes experimental SQM results into consideration. SQMAT consists of a quality measurement and evaluation method with three levels of quality criteria, and a support tool for a visual display for management. Its most notable feature is that the feed forward control principle is employed in addition to the feedback control principle. That is, quality target setting is carried out before the Plan-Do-Check-Action activities (Deming's PDCA cycle) are performed. SQMAT procedures are defined as follows:

(1) In the TARGET phase, a quality priority ranking is established for the individual quality characteristics, based on the users' requirements and the development policy. It is important to clarify the quality target, i.e., classify the quality characteristics into 3 categories and set the target quantitatively.

(2) In the PLAN phase, Software Quality Measurement Criteria (SQMC), are set up and methods for achieving the target quality are discussed in advance, primarily with the

quality assurance people and managers.

(3) In the DO phase, high quality software is produced by complying with development standards and SQMC as guidelines. Before the formal review, the developer executes a quality self-check.

(4) In the CHECK phase, the software is checked and evaluated against the individual quality criteria set up in the PLAN phase. Quality is measured by a third party. If errors are detected, problem reports are drawn up. After scoring, score sheets and quality graphs are developed, and the achieved quality is judged by comparing it to the target quality level.

(5) In the ACTION phase, corrective action is taken, based on problem reports. Achieving the quality target permits proceeding on to the next phase. SQMAT can be applicable not only to large scale software, but also to small projects. NEC's experience with the approach has had measurable results. For example, based upon comparison with historical data, (1) a number of errors have been eliminated during the design and implementation phases, and (2) productivity (measured by lines-of-source-code/hour) has increased by 10%.

The Improvement Paradigm

The Quality Improvement Paradigm [Bas85a] for software engineering processes is a top level paradigm that is based upon the scientific method as applied to software evaluation. It provides the view of software evolution as an experimental process from which we must learn and improve the current project as well as future projects (Characterize, Set Goals, Choose Methods, Build, Analyze, Learn and Feed Back). It is a meta-life cycle model that aims at improving the software quality and productivity based upon measurement and reuse of experience. It needs to be instantiated for a variety of sub-activities, e.g. specific processes such as testing, product reviews, managing. It consists of six major steps:

(1) Characterize the current project environment.

(2) Set up goals and refine them into quantifiable questions and metrics for successful project performance and improvement over previous project performances.

(3) Choose the appropriate software project execution model for this project and supporting methods and tools.

(4) Execute the chosen processes and construct the products, collect the prescribed data, validate it, and analyze the data to provide feedback in real-time for corrective action on the current project.

(5) Analyze the data to evaluate the current practices, determine problems, record the findings and make recommendations for improvement for future projects.

(6) Package the experience in the form of updated and refined models and other forms of structured knowledge gained from this and previous projects and proceed to step 1 to start the next project.

This paradigm is aimed at providing a basis for corporate learning and improvement [BaRo87] and is based upon experience with measurement and evaluation of software development in a number of companies.

Goal Question/Metric Paradigm

The Goal/Question/Metric (GQM) paradigm [BaWe85, BaSe84] is a mechanism for generating measurement in a goal-directed manner. It represents a systematic approach for setting the project goals (tailored to the specific needs of an organization), defining them in an operational, tractable way by refining them into a set of quantifiable questions that in turn imply a specific set of metrics and data for collection (addresses the aspects related to step 2) of the improvement paradigm). Appropriate metrics are tailored to each project based on the G/Q/M templates and past experience. It includes the development of data collection mechanisms, e.g., forms, automated tools, the collection and validation of data, and the analysis and interpretation of the collected data and computed metrics in the appropriate context of the questions and the original goals.

In order to support the process of setting goals and refining them into quantifiable questions, a set of templates for setting goals, and a set of guidelines for deriving questions and metrics has been developed [BaRo88]. These templates and guidelines reflect our experience from having applied the GQM paradigm in a variety of environments [RoBa87, WeBa84, BaWe81].

Goals are defined in terms of purpose, perspective and environment. Different sets of guidelines exist for defining product-related and process-related questions. Product-related questions are formulated for the purpose of defining the product (e.g., physical attributes, cost, changes and defects, user context), defining the quality perspective of interest (e.g., functionality, reliability, user friendliness), and providing feedback from the particular quality perspective. Process-related questions are formulated for the purpose of defining the process (process conformance, domain conformance), defining the quality perspective of interest (e.g., reduction of defects, cost effectiveness of use), and providing feedback from the particular quality perspective.

The TAME (Tailoring A Measurement Environments) system [BaRo88] is a measurement environment that supports and integrates the Quality Improvement and the Goal Question Metric paradigms.

Based on the work at NEC, the TAME project, and the managerial requirements specified above, a management methodology, called the Software Management Cycle (SMC), has been developed. Its main concepts are goal oriented, process control and visual management. Management procedures, support tools and forms, and an evaluation method are provided as part of SMC.

3. Relationship of the SQM, GQM, and SQMAT

The SQM and the GQM are both mechanisms for measuring software quality. Both models are top-down and characterize quality characteristics at three levels. In the SQM, these levels are Factor, Criteria, and Metric. For example, a high level factor such as correctness is defined by the set of criteria traceability, completeness, and consistency which in turn are defined in terms of a predefined set of metrics.

The GQM model consists of a goal, which is specified by a set of quantifiable questions, which in turn are defined by a set of metrics and data distributions tailored to the specific environment. Thus to define a high level goal like correctness of the final product, we must define a set of questions that characterize the product (with respect to its physical attributes, cost of development, changes and defects, and customer base and operational profile), define a model for correctness (which could include such concepts as traceability, completeness, and consistency), provide insights into the validity of the model and the data within the particular environment, and the results of the model along with some possible substantiation of the model results.

The SQM model predates the GQM model, but the latter is more general. The GQM can be used to characterize, evaluate, predict, or motivate a product, process, model or metric, with respect to a variety of perspectives (e.g. customer, developer, user, manager, etc.) based upon an open ended definition of quality. It takes into account the specific environment in which the product has been developed as well as assessment of such things as an evaluation of how well the particular methods were used, how well the domain of application was understood in order to help interpret the resulting evaluation metrics appropriately. It also involves the feedback of information for future development through learning.

The SQM model is written from the point of view of determining a set of quality characteristics of the final product from the point of view of the customer. It does not measure process for developing that product and since its viewpoint is that of the customer, it provides limited support for learning, feedback and improvement within the development organization. Its measurement process tends to be passive and is not focussed on capturing the causes of the quality problems.

The measurement focus of SQM as used in SQMAT has evolved and widened over time and is currently more consistent with the GQM. This wider view of SQM uses metrics to measure quality of an intermediate product from the point of user, developer and so on.

	GQM		SQM		
			narrow-sense		wide-sense
Objective	Characterize, Assess, Predict, Motivate		Assess (Quality)		
Structure	Goal Question Metric		Factor Criteria Metric		
Usage	Project & Quality Management		Quality Management		
Object	Any Product, Process, Model, or Metric		Product		Any Product Process
Viewpoint	Developer, User, Manager, Corporate		User		(same as GQM)
Establishment manner of GQM or SQM	G	Select or Tailor	F	Select	
	Q	Select or Tailor	C	Select	
	M	Select or Tailor	M	Select	Select or Tailor

Table 1. Features of GQM and SQM.

4. Software Management Cycle (SMC)

The Quality Improvement Paradigm provides a top level organizational perspective on the software development and maintenance process. SMC is the management procedure and support system under that paradigm. It emphasizes three concepts; goal–oriented measurement, process control, and visual management. In response to each concept, several activities are necessary. These activities, performed during the management procedure, make it possible for management to achieve higher quality and productivity.

The management procedure used in SMC consists of the following five steps:

(Step 1) Define system/project characteristics

It is important to define the system characteristics in detail to reflect the user requirements for development. A set of system/project characteristics forms are prepared to gather information on the requirements and the current project status.

This is equivalent to the first step of the Quality Improvement Paradigm. The system engineer is responsible for understanding the customer requirements for the particular project correctly. The development environment should be also clarified. This characterization permits the comparison of the current project with prior projects with similar characteristics. This information is used in the next step.

(Step 2) Select Goals,Questions, and Metrics

To achieve high quality and productivity, it is necessary to set the specific objectives. This is the key step to the success of the project. Unless the goals are appropriate, the project will fail. The GQM paradigm is used to do this. It satisfies the requirement for goal–oriented measurement. It helps both developers and managers clarify the objectives of the project prior to development.

Guidelines and templates are used to establish the particular GQM used. Templates from prior systems can be used or modified for this project. For each metric, measurement instructions are prepared, which include the importance of metric, the collection method and person responsible, the data presentation, the decisions affected etc.

Besides the set of goals and metrics for the particular project, a common set of managerial metrics have been specified to be applied to all projects. We can gather the data for getting the level of quality and productivity through projects and development phases. The metrics from this common set are shown below.

[Quality]

- Number of detected errors at test phase (from integration test through system test)
- Number of detected errors within six months after release

[Productivity]

- Number of specification pages
- Number of non–comment source statement
- Effort at each phase by man–hour

(Step 3) Select activities

Methods to effectively achieve the objective are considered at this time. Appropriate activities for economically producing the software and managing the quality of the project can now be chosen based on the specific objectives laid out in the GQM.

This step is critical to achieve the objectives. Setting goals, without specifying the means to achieve them, is meaningless. Sufficient discussion on the activity selection process is necessary from various viewpoints; how they fit into the development environment, how they integrate with the management methods and training plans, etc. and how the help achieve the objectives, provide focus for the questions and affect the definition of the metrics.

For process control, a review checklist is prepared for each phase of development based on the metrics specified by the GQM model and past history, e.g. prior fault data. Feedback to the process should also be performed as soon as possible after a review. Problems can be easily found using the review checklist. Periodic checks; e.g. monthly, or at the final review of each development phase, are required to monitor the process. The earlier the phase at which monitoring starts, the more effective it is for quality improvement. Audit and configuration management are also process control methods governing quality.

(Step 4) Measure and assess the process and the products

Project data will be collected periodically, at least at the end of each development phase. Based on the metrics selected, the process and the products are measured. The results are assessed by using specific rating criteria. It is helpful for manager to take proper action quickly. Continuous measurement and assessment can produce high quality product.

For visual management, graphical displays of the appropriate management information can be selected based on the graph selection form. The project's current status can be found by using the visual display tool provided by the SMC system. It is helpful for software managers to see the achieved quality level in a concrete form to support such activities as decision making, the management of quality and scheduling of the next workload. For example, graphs provide the manager with time series data indicating process and product changes, as well as comparative data from past projects.

(Step 5) Support corrective action

For low scoring metrics, some action should be taken. A corrective action list is prepared and used to improve both the current and future process and products.

Based on the assessment of results at step 4, proper action is required quickly for problems or the sign of any problems. If necessary, the activity plan can be revised. These experiences are accumulated and used to future projects.

Guidelines necessary to perform project management, based on SMC, are as follows:

- Goal selection
- Question selection
- Metric selection
- Activity selection
- Management graph selection
- Project status diagnosis
- Corrective action recommendation
- Reliability prediction

The SMC helps the manager in the definition of an appropriate software engineering process during the GQM and activities selection phases (steps 2 and 3), by allowing the manager to tailor goals, measures, methods and tools to the specific system/project characteristics. A data base can be defined and built to support the measurement environment during the GQM selection phase and to support both the development and management environments during the activities selection phase. After executing one whole cycle through the SMC process, the results of analyzing the current project data can be fed back to each SMC phase. Updating the database and improving each step of the SMC helps generate a software engineering process for future projects.

The SMC support system is currently a prototype built on top of existing software packages. It consists of (1) a data base, (2) a set of statistical packages, and (3) a set of graphical types (developed using Microsoft Excel), all integrated under a common user interface.

Accumulation of application information in a data base enables the organization to establish guidelines for future projects. Therefore, the relation between the system characteristics and the measurements associated with the particular GQM should be collected and saved in a data base. Emphasis should be on the metrics common across several projects

5. An example G/Q/M

A simplified pair of GQM models, one for product and one for process are given. They are written from the point of view of the manager (which may include some of the concerns of the customer) for evaluating various components to improve quality, cost and usage of methods based upon managerial data.

First we will define some terms and offer a model of the qualities of interest:

DEFINITIONS:

Size (NCSS) = the number of non-commentary source statement (NCSS)

Actual Effort (AEF) = total number of staff hours to develop a component

Estimated Effort (EEF) = estimated number of staff hours based upon the software science metric, E

Actual Errors (AER) = the total number of errors reported

Estimated Errors (EER) = the estimated number of errors based upon the software science metric, B

Actual Error Rate (AERR) = AER / NCSS

Estimated Error Rate (EERR) = EER / NCSS

Changes (CH) = the total number of changes reported

Change Rate (CR) = CH / NCSS

Effort Distribution (PED) = the percent of staff hours for a particular component spent in each phase

Test Efficiency (PTE) = the percent of machine time spent testing a component

Work Rate (WR) = NCSS / AEF

Effort Variance (EFV) = AEF / EEF

Error Variance (ERV) = AER / EER

MODEL:

The objectives for management are cost, quality and the effectiveness of the methods. Evaluation is performed on the basis of improvement over some norm.

Cost can be assessed as the relationship between input, staff effort, and output, the quantity of documentation and program produced. In this case we will consider cost as demonstrated by two factors: work rate (WR), which provides some measure of the cost of production for a line of code, and Effort Variance (EFV), which provides some measure of whether the effort is reasonable relative to some measure of the expected effort.

Quality is assessed in two categories, must-be quality and attractive quality. These terms, must-be quality and attractive quality, are common Japanese quality perspectives. Must-be quality means the fundamental qualities necessary for software to function, i.e., functionality and reliability. Attractive quality means any additional quality characteristics for the software to satisfy the users specific needs, e.g., usability, security, portability. In this case, we will consider quality as demonstrated by two factors: error variance (ERV), which provides some measure of whether the error rate is reasonable relative to some measure of expected errors, and change rate (CR), which provides some measure of the entropy of the system.

Method characteristics are assessed based upon their adherence to a set of standards. Project manager experience is also assessed since the success of a project deeply depends on his ability. In this case, we consider method evaluation using two factors: effort distribution (PED), which will provides us some insight into whether the distribution of the effort was acceptable according to standard baselines of effort distribution, and test efficiency (PTE) which when combined with test time, will provide some insight into the effectiveness of the test process, and therefore the effectiveness of the methods used for development.

Note that the model uses the software science measures, E and B as a basis for estimating, effort and bugs. It assumes these calculated values as basic estimates for the variables effort and errors and uses them as norms when comparing the actual values for effort and errors.

In our proposed model, the values of these variables for any component are then compared to the values for some normal population. All values within 2 sigma variation from the average are considered acceptable. Those values with more than a 2 sigma variation in the "right" direction are considered good; those with more than a two sigma variation in the "wrong" direction are considered as not meeting the target goal. For example, the effort variance (EFV) for a component is considered bad if it is greater than two sigma above the norm determined by the average value of cost for the rest of the population.

In the example given in the next section the baselines are determined by the the rest of the component population in the particular project. In an environment where there is data from a sufficient number of projects, the baselines could be determined by projects with similar characteristics from other projects.

PRODUCT GOAL:

Purpose: Evaluate various software components within a project in order to assess them and recommend areas for improvement.

Perspective: Examine the relative cost and quality from the point of view of the manager.

PRODUCT DEFINITION:

Product Dimensions: A quantitative characterization of the physical attributes of the product.

Q1. What is the size of each component in terms of non-commented source statements (NCSS)?

Q2. What is the value of the software science metrics for each component (E,B)?

Changes/Defects: A quantitative characterization of the enhancements, errors, faults, and failures.

Q3. What is the number of defects associated with each component (AER)?

Q4. What is the number of changes associated with each component (CH)?

Q5. What is the fault rate, change rate (AERR, CR)?

Cost: A quantitative characterization of the resources expended.

Q6. What is the staff effort involved in the development of each component, i.e. design, code, test?

Q7. What is the distribution of effort spent in the design, code and test phase (PED)?

Context: A quantitative characterization of the customer community and their operational profiles.

[No questions for this example]

In general, five viewpoints are necessary for process questions. Two of five, "Effort of Use" and "Effect of Use", are actually used in a case study next section.

PROCESS GOAL:

Purpose: Evaluate the design, code and test processes in order to improve them.

Perspective: Examine the relative cost distribution and test efficiency from the point of view of the manager.

PROCESS QUESTIONS:

Quality of Use: A quantitative characterization of the process and an assessment of how well it is performed.

Q8. How much experience does the team have with respect to the methods and tools used?

Q9. How much experience does the manager have with respect to similar projects?

Domain of Use: A quantitative characterization of the object to which the process is applied and an analysis of the process performer's knowledge concerning this object.

Q10. How understandable are the requirements?

Effort of Use: A quantitative specification of the quality perspective of interest. In this case, a quantitative specification of the costs.

Q6. What is the staff effort involved in the development of each component, i.e. design, code, test?

Q7. What is the distribution of effort spent in the design, code and test phase (PED)?

Q11. What is the machine time spent in the test phase for each component (PTE)?

Feedback from Use: This includes questions related to improving the process relative to the quality perspective of interest.

Q12. What is the input to the design and code methods and tools, and the defect detection methods and tools?

Q13. What should be automated?

6. Case Study

The concepts of SMC can be applied to a variety of project types because of the flexibility of this methodology. Metrics and development methodologies are tailored to each project. In this section, we discuss several general issues in applying SMC and provide a sample application to the management of a specific project based upon models and the goals, questions and metrics of the previous section.

In executing SMC in a project, the software management procedure mentioned previously, the templates, guidelines and some forms are used. A step by step approach based on this procedure is demonstrated. A sufficient budget for managing these activities is required. It is also necessary to establish an organization to support the SQM process. Certainly, a seminar on SMC for both managers and developers would have provided better results. It should be remembered that the more experience the manager and the organization have with SMC, the better they will be able to apply the method. The continuous application of the method provides a better support for quality and productivity.

This example uses the NASA/SEL [McGa85, Bas85b] project data base. Thirteen newly developed components for a particular project were selected. Size range of non-comment source statements is from 60 to 299 LOC. Graphs for project management were made using Microsoft Excel.

In step 1, "System/Project Form" is filled out. This clarify both the software functional requirements and the development environment. The profile of the system and the environment are defined.

In step 2, the project goals are determined based on the system/project characteristics from step 1 and the managerial strategy; e.g. cost, quality level to be achieved, methodologies to be employed, etc. Each goal is extended to questions and metrics by means of the GQM template. The "Quality Target" and "Managerial Metrics" are determined at this step. Cost and quality improvement and better usage of various methods were chosen as goals. To support management, the "Graph Selection Form" is provided. Six graphs were selected; those are for work rate, effort variance, error variance, change rate, effort distribution and test efficiency. Questions to achieve these goals are shown in Chap. 2.

In step 3, the best way to achieve GQM is discussed and appropriate activities are selected. These depend on the pieces of information from previous steps. Development methodologies and quality checkpoints are listed on a specific form. This form is used as a checklist during development.

In step 4, the development process is monitored and managerial data are collected periodically. To make the project status visible, display graphs are very helpful. The graphs used were selected in step 2.

The following table shows the results of statistical analysis on the NASA/SEL project. Six criteria on three categories are chosen. Regression analysis was executed for the "Error Variance" data. Analysis of variance was executed for the rest of data. Based on the graphs and this table, the project's current status can be found. Comments for four of 13 components are described below. Figure 2 shows some sample graphs.

Rules for interpreting the results

For each metric, there exists pattern to interpret the results. Consider the following examples.

[Cost]

(1) Work Rate: Development speed measured by NCSS per man-hour

- In case of a low value, there are several potential problems
  * low quality
  * insufficient development environment
  * loose process control
  etc.

(2) Effort Variance: actual effort vs. estimated effort

- Evaluate the goodness by variation between estimated and actual effort
  * in the case that the actual effort is lower, the work rate is high (or functions could be simple)
  * in the case that actual effort is high, the interpretation of the results are the same as for Work Rate.

[Quality]

(1) Error Variation: the number of actual errors compared with the estimated (a measure of complexity)

Component number	COST		QUALITY		METHOD			
	Work Rate	Effort Variance	Error Variance	Change Rate	Effort Distribution			Test Efficiency
					D	C	T	
c29				O	x	x	XX	XX
c61	O							
c6		X				x	XX	
c5	X	XX	OO		X		x	O
c46			XX					
c7			XX	XX	XX	x		OO
c9								XX
c4			O					
c49				XX		X		
c43	OO	O	OO	O		X		OO
c11		X	XX			X		XX
c63			OO			x		XX
c50								O

D	design phase
C	code phase
T	test phase

Assessment Criteria (except Cost Distribution)

OO :	excellent	(>= AVE +2 O˜)
O :	good	(>= +1 O˜)
X :	poor	(<= -1 O˜)
XX :	bad	(<= -2 O˜)

Assessment Criteria for Cost Distribution

XX :	very high rate	(>= AVE +2 O˜)
X :	high rate	(>= +1 O˜)
x :	low rate	(<= -1 O˜)
xx :	very low rate	(<= -2 O˜)

Table 2. Component Assessment Table.

- It assumes that the greater the complexity, the greater the number of errors.
  * Quality is high if the number of errors is low in comparison with the estimated number based on complexity.

(2) Change Rate: the normalized magnitude of the number of specification changes and error modifications

- In case that the number of specification change is large, there is a problem in the development methods
  * insufficient review
  * less communication with user
  * loose configuration control
- In case that the number of error modification is large,

quality is considered to be low.
- In both cases, degradation of the system can be assumed due to entropy because of change

[Methodology]

(1) Effort distribution: effort ratio of each phase

- Evaluate the percentage of effort in each phase (design / coding / test).
  - * Is the effort in the design phase sufficient? In the case of insufficient effort, the degree of specification completion is considered to be low.
  - * Does it cost too much in coding phase? In case of too much effort, it is assumed that the specification is insufficient or the development environment is not so good.
  - * Is the effort appropriate in test phase? In case of too little effort, it is assumed that testing was insufficient and the system was delivered with errors. In case of too much effort, it is assumed that the test method is not efficient and/or the quality is low so the test phase lasted too long.

(2) Test efficiency : percent of machine time in the test phase

- The ratio is high if the preparation of test cases is sufficient.
- The ratio is high if quality is high so error modification effort is smäll.

Assumed activities in the test phase

Preparation	Machine Test (fixed)	Error modification

The pattern for interpretating of results can be made by combining the above heuristics.

Comments
c5 :

Quality is good, but cost is high. Because of the high cost in design and code phases, product quality must be high. Some changes may have caused the rise of both design and code cost rate.

(good)
Quality is high.

(to be improved)
Work rate is low.

(diagnosis)
It is necessary to monitor the early process to avoid the slide of schedule. Quick feedback and effective

reviews are necessary.

c7 :

There is a problem in the methods for development and management. The number of changes is large. This caused the rate of design and code to be too high. Because of insufficient test instead of high test efficiency, number of errors is also large.

(to be improved)
- Though test efficiency is very high, preparation, interpretation and error correction must be insufficient, because there are still many errors.
- There are many more changes than those of other components.

(diagnosis)
Design or review methodology must be improved. Be more careful in test phase. Try to find out the potential errors based on the test results. More experience and knowledge are required to do so.

c43 :

It's a very good component. The only concern is the percentage effort of the code phase. It is true that the difficulty of this component is low, but both quality and productivity are high.

c11 :

There is a problem in methods for development and management.

(to be improved)
Because of poor design, the code phase costs too much and there are many errors.

(diagnosis)
It is necessary to improve design phase to be able to make a better quality document. The test method should also be reconsidered.

In total, the difference between the goals and results can be evaluated from Table 1.

From the view of COST, only component 5 was well above the standard cost. This component, however, achieved a high quality rating, so its project goal can be considered as achieved.

From the view of QUALITY, four of thirteen components (7,11,46,49) did not realize their quality target. Error analysis indicates that most errors can be reduced by avoiding careless mistakes. Component 7 has an extra problem. An unusually high number of changes extended the design phase and caused many errors. Further investigative action should be taken into the causes of those changes and the manager should be encouraged to minimize change. The quality target has not been achieved for these four components.

From the view of USAGE OF METHODS, two components (6, 29) had too high a cost in test phase. They rated satisfactory for cost and quality however. Four component (9, 11, 29, 63) rated poorly with respect to test efficiency. One component (29) did not meet target in both categories. There are several problems to be solved in test phase.

All three goals can not always be achieved sufficiently. However, avoiding careless mistakes and improving the test method should produce a better product.

In step 5, corrective action is taken based on the collected data and managerial graphs from step 4. For unachieved items in Figs. 4 and 6, the cause of each problem is pursued and an improvement method is discussed and executed. If something is found wrong in a certain step, the activities in that step are improved quickly. In this way, a project can be managed systematically throughout the life cycle.

The expected effects of applying SMC are quality improvement, managerial cost reduction, accumulation and reuse of experience and a highly visible management reporting system.

7. Evaluation of Software Management Cycle

We are interested in evaluating and improving the SMC itself. Data collected at each phase and after release enable us to analyze the effect of the SMC. The followings are the GQM for evaluation of SMC.

Goal: Evaluate the effectiveness of the SMC

Process Conformance:

Q1. How much managerial training was given to the manager?
Q2. How well were the SMC methods applied?

Domain Conformance:

Q3. How well was the SMC procedure understood?
Q4. How well was how to interpret graphs understood?

Cost:

Q5. How many hours were spent to perform SMC?

Effect:

Q6. What was the distribution of the management time?
Q7. Were graphs and forms helpful for the manager?

Feedback:

Q8. What changes need to be made in the methodology to make it more effective?
Q9. What tools or activities would make the use of SMC more effective?

During development, quality/productivity metrics (set Q), methods metrics (set M) and feedback metrics (set F) are necessary. Customer satisfaction metrics (set C) are required after release.

Table 2 shows the classification of data.

Four evaluation methods are provided.

(1) How good are goals?

Based on correlation analysis between Q, M, F and C, it can be judged that a project must be good if the metrics of the project include most of elements of Q, M or F which have high correlation coefficient to C.

(2) How good are activities (methods, feedback)?

Based on correlation analysis between M, F and C, it must be good activity if an element of M or F has high correlation coefficient.

(3) How good are metrics?

Based on regression analysis between C and Q, M, F, the metrics of a project must be good or predictable if the project has high regression coefficient.

(4) How good are products?

Based on significant test of the difference between two population (past projects' C and current C), the current projects' products must be good if the difference is statistically significant.

The results of these analyses help to improve Software Management Cycle and update the knowledge of management database.

8. Conclusion

The concepts and use of Software Management Cycle based on the Quality Improvement Paradigm are described in this paper. This methodology can improve not only product quality but also process quality. Three concepts; goal–oriented measurement, process control and visual management, are important to manage a project effectively, quantitatively and objectively.

Further plans for the SMC include:

(1) its application to a variety of projects, analyzing the processes and accumulating knowledge for different project classes, and

(2) the development of a full management support tool which covers the whole process.

The authors are convinced that this methodology contributes to the building of an appropriate software engineering process for improving both quality and productivity.

9. References

[AzSu86] M.Azuma, T.Sunazuka, "Software Quality Measurement and Assurance Technology (SQMAT)", Quality, Vol.16, No.1, pp.79–84, January 1986, (in Japanese).

[AzSM87] M.Azuma, T.Sunazuka, K.Minomura, "Software Quality Assessment Criteria and Measurement Technology", Standardization and Quality Control, Vol.40, No.8, pp.63–75, August 1987, (in Japanese).

[Bas85a] V.R.Basili, "Quantitative Evaluation of Software Engineering Methodology," Proc. of the First Pan Pacific Computer Conference, Melbourne, Australia, September 1985 [also available as Technical Report, TR–1519, Dept. of Computer Science, University of Maryland, College Park, July 1985].

[Bas85b] V.R.Basili, "Can We Measure Software Technology: Lessons Learned from 8 Years of Trying," Proceedings of the Tenth Annual Software Engineering Workshop, NASA Goddard Space Flight Center, Greenbelt, MD, December 1985.

[BaKa83] V.R.Basili, E.E.Katz, "Metrics of Interest in an Ada Development," Proc. of the IEEE Computer Society Workshop on Software Engineering Technology Transfer, April 1983, pp. 22–29.

[BaRo87] V.R.Basili, H.D.Rombach, "Tailoring the Software Process to Project Goals and Environments," Proc. of the Ninth International Conference on Software Engineering, Monterey, CA, March 30 – April 2, 1987, pp. 345–357.

[BaRo88] V.R.Basili, H.D.Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," IEEE Transactions on Software Engineering, vol. SE–14, no. 6, June 1988, pp. 758–773.

[BaSe84] V.R.Basili, R.W.Selby,Jr., "Data Collection and Analysis in Software Research and Management," Proc. of the American Statistical Association and Biomeasure Society Joint Statistical Meetings, Philadelphia, PA, August 13–16, 1984.

[BaSe85] V.R.Basili, R.W.Selby,Jr., "Calculation and Use of an Environment's Characteristic Software Metric Set," Proceedings of the Eighth International Conference on Software Engineering, London, UK, August 1985.

[BaWe84] V.R.Basili, D.M.Weiss, "A Methodology for Collecting Valid Software Engineering Data," IEEE Transactions on Software Engineering, vol. SE–10, no.6, November 1984, pp. 728–738.

[BaWe81] V.R.Basili, D.M.Weiss, "Evaluation of a Software Requirements Document by Analysis of Change Data," Proceedings of the Fifth International Conference on Software Engineering, San Diego, USA, March 1981, pp. 314–323.

[BoBL76] B.W.Boehm, J.R.Brown, and M.Lipow, "Quantitative Evaluation of Software Quality," Proceedings of the Second International Conference on Software Engineering, 1976, pp. 592–605.

[Grad87] R.B.Grady, "Measuring and Managing Software Maintenance", IEEE Software, Vol.4, No.5, pp.35–45, September 1987.

[GrCa87] R.B.Grady, D.L.Caswell, "SOFTWARE METRICS: Establishing A Company-wide

Program", Prentice-Hall, Englewood Cliffs, NJ, 1987.

[McRW77] J.A.McCall, P.K.Richards, G.F.Walters, "Factors in Software Quality", RADC TR-77-369, 1977.

[McGa85] F.E.McGarry, "Recent SEL Studies," Proceedings of the Tenth Annual Software Engineering Workshop, NASA Goddard Space Flight Center, December 1985.

[Mizu82] Y.Mizuno, "Software Quality Improvement", Proc. 6th compsac 82, 1982.

[Muri80] G.E.Murine, "Applying Software Quality Metrics in the Requirement Analysis Phase of a Distributive System", Proc. Minnow Brook Conference, 1980.

[RoBa87] H.D.Rombach, V.R.Basili, "A Quantitative Assessment of Software Maintenance: An Industrial Case Study," Conference on Software Maintenance, Austin, Texas, September 1987, pp 134-144.

[SuAY85] T.Sunazuka, M.Azuma,N.Yamagishi, "Software Quality Assessment Technology", Proc. 8th International Conference on Software Engineering, London, UK, pp.142-148, August 1985.

[WeBa85] D.M.Weiss, V.R.Basili, "Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory," IEEE Transactions on Software Engineering, vol. SE-11, no. 2, February 1985, pp. 157-168.

UMIACS-TR-88-92
CS-TR-2158

December, 1988

Towards A Comprehensive Framework for Reuse:† A Reuse-Enabling Software Evolution Environment

V. R. Basili and H.D. Rombach
Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

Reuse of products, processes and knowledge will be the key to enable the software industry to achieve the dramatic improvement in productivity and quality required to satisfy the anticipated growing demands. Although experience shows that certain kinds of reuse can be successful, general success has been elusive. A software life-cycle technology which allows broad and extensive reuse could provide the means to achieving the desired order-of-magnitude improvements. This paper motivates and outlines the scope of a comprehensive framework for understanding, planning, evaluating and motivating reuse practices and the necessary research activities. As a first step towards such a framework, a reuse-enabling software evolution environment model is introduced which provides a basis for the effective recording of experience, the generalization and tailoring of experience, the formalization of experience, and the (re-)use of experience.

† Research for this study was supported in part by NASA grant nSG-5123, ONR grant N00014-87-K-0307 and Airmics grant DE-AC05-OR21400 to the University of Maryland.

TABLE OF CONTENTS:

1. INTRODUCTION

The existing gap between the demand and our ability to produce high quality software cost-effectively calls for improved software life-cycle technology. A reuse-enabling software life-cycle technology is expected to contribute significantly to higher quality and productivity. Quality can be expected to improve by reusing proven experience in the form of products, processes and knowledge. Productivity can be expected to increase by using existing experience rather than developing it from scratch whenever needed.

Reusing existing experience is the key to progress in any area. Without reuse everything must be re-learned and re-created; progress in an economical fashion is unlikely. During the evolution* of software, we routinely reuse experience in the form of existing products (e.g., generic Ada components, design documents, mathematical subroutines), processes (e.g., design inspection methods, compiler tools), and domain-specific knowledge (e.g., cost models, lessons learned, measurement data). Most reuse occurs implicitly in an ad-hoc fashion rather than as the result of explicit planning and support. While reuse is less institutionalized in software engineering than in other engineering disciplines, there exist some successful cases of reuse, i.e. product reuse. Reuse in software engineering has been successful whenever the reused experience is self-describing, e.g., mathematical subroutines, or the stability of the context in which the experience is reused compensates for the lack of self-description, e.g., reuse of high-level designs across projects with similar characteristics regarding the application domain, the design methods, and the personnel. In software engineering, the potential productivity pay-off from reuse can be quite high since it is inexpensive to store and reproduce software engineering experience compared to other engineering disciplines.

The goal of research in the area of reuse is the achievement of systematic methods for effectively reusing existing experience to maximize quality and cost benefits. Successful reuse depends on the characteristics of the candidate reuse objects, the characteristics of the reuse process

[*] The term "evolution" is used in this paper to comprise the entire software life cycle (development and maintenance

itself, and the technical and managerial environment in which reuse takes place. Interest in reusability has re-emerged during the last couple of years [4, 9, 11, 12, 13, 14, 15, 16, 17, 19, 20, 21], due in part to the stimulus provided by Ada and in part to our increased understanding of the relation between software processes and products.

Our increased understanding tells us that in order to improve quality and productivity via reuse we need a framework which allows (a) the reuse of all kinds of software engineering experience, i.e., products, processes and knowledge, (b) the better understanding of the reuse process itself, and (c) the better understanding of the technical and managerial evolution environment in which reuse is expected to be enabled.

This paper presents a reuse-enabling software evolution environment model, the first step towards a comprehensive framework for understanding, planning, evaluating and motivating reuse practices and the necessary research activities. Section 2 motivates the necessary scope of a comprehensive reuse framework and the important role of a reuse-enabling software evolution environment model within such a framework. Section 3 introduces the reuse-enabling software evolution environment model and discusses its ability to explicitly model the recording of experience, the generalization and tailoring of experience, the formalization of experience, and the re-use of experience. The TAME model, a specific instantiation of the reuse-enabling software evolution environment model, is presented in Section 4. This specific instantiation is used to more specifically describe the integration of the recording and (re-)use activities into an improvement oriented software evolution process.

Before we proceed, we define some crucial terms that will be used in this paper so the reader understands what we mean by them in the software context. We have tailored Webster's general definitions of these terms to the specific domain of software evolution. *Improvement* means enhancing a software process or product with respect to quality and productivity. *Learning* is the activity of acquiring experience by instruction (e.g., construction) or study (e.g., analysis). *Reuse* is the activity of repeatedly using existing experience, after reclaiming it, with or without

ORIGINAL PAGE IS
OF POOR QUALITY

modification. *Feedback* means returning to the entry point of some process armed with the experience created during prior executions of the process. We use the expression *experience base* to mean a repository containing all kinds of experience. An experience base can be implemented in a variety of ways depending on the type of experience stored. An experience base may consist of one or more of the following: traditional databases containing factual pieces of information, information bases containing structured information, and knowledge bases including mechanisms for deducing new information [5, 24].

2. SCOPE OF A COMPREHENSIVE REUSE FRAMEWORK

Reuse in most environments is implicit and ad-hoc. When it is explicit or planned, it predominantly deals with the reuse of code. In Section 1, we expressed our belief that effective reuse technology needs to be based on (a) the reuse of products, processes and knowledge, (b) a good understanding of the reuse process itself, and (c) a good understanding of the reuse-enabling software evolution environment.

To better justify these beliefs, we will describe and discuss the reuse practice in the Software Engineering Laboratory (SEL) at NASA Goddard Space Flight Center [2, 18]. This is an example where reuse has been quite successful at a variety of levels, albeit predominantly implicit. Ground support software for satellites has been developed for a number of years in FORTRAN. Reused experience exists in the people, methods, and tools as well as in the program library and measurement database.

To explain reuse in this environment we must first explain the management structure. There are two levels of management involved in the technical project management. The second level managers (one from NASA and one from Computer Sciences Corporation, the contractor), have been managing this class of projects for several years. Specific project managers are typically promoted from within the ranks, on either side, from the better developers on prior projects.

This provides a continual learning experience for the management team. Technical review and discussion is informal but commonplace. Lessons learned from experience are used to improve management's ability to monitor and control project developments.

The organizational structure has been relatively constant from project to project There have been minor variations due to improvements in such things as methods and tools which have evolved from experience or been motivated the literature and verified by experimental data analysis on prior projects.

The basic systems have been relatively constant. This permits reuse of the application knowledge as well as the requirements. and design. For example the requirements documents are quite mixed with regard to the level of specificity. In some places they are quite precise but in other cases the are very incomplete. relying on the experience of the people from prior projects

Requirements documents have phrases similar to the following: Capability X for new satellite S2 is similar to capability X for satellite S1 except for the following... This implicitly provides reuse of prior requirements documents as well as implicitly allows for reuse of prior design documents and code.

Systems within a class, all have a similar design at the top level and the interfaces among subsystems are relatively well defined and tend to be relatively error free. Design is implicitly reused from system to system as specified by the experienced high level managers.

Reuse at the code level is more explicit. The software development process used is a reuse oriented version of the waterfall model The coding phase begins by seeding the code library with the appropriately specified elements from the appropriate prior projects. These code components are then examined for their ability to be reused. Some are used as is. others modified minimally. others modified extensively, and yet others are eliminated and judged easier to develop from scratch. This is a reuse approach that has evolved over time and has been quite effective.

A variety of tools have evolved that are quite application specific. These include everything from tools that generate displays needed for testing to application specific system utilities

Knowledge about these tools has been disseminated by guidance from more senior members of the development team.

The SEL environment is a good example of strong reuse at a variety of levels, in a variety of ways as part of the software development process. There has been a pattern of learning and reusing knowledge, processes and products. The use of the measurement database has helped with project control and schedule as well as quality assessment and productivity [2, 18].

NASA is now considering changing to Ada. Several Ada projects have already been completed. This has involved an obvious loss in the reuse heritage at the code level, as was anticipated. But it has also involved a less obvious and unexpected loss of reuse at the requirements and design level, in the organizational structure, and even in the application knowledge area.

The initial impact of Ada was staggering because of the implicit, rather than explicit understanding of reuse in the environment. This understanding of reuse needs to be formalized.

Based upon the concept that reuse is more than just reuse of code and that it needs to be explicitly modeled, we need to reconsider how we measure progress in reuse. The measurements currently used in the SEL are based upon lines of code reused from one project to another. Given this view, progress may not be related at all to the lines of code reused. We need to measure the effects of reuse on the resources expended in the entire software life cycle and on the quality of the products produced using an explicit reuse oriented evolution model. In fact, the process should allow us measure for any set of reuse-related goals [3, 4, 8, 10]. Changing our models and our metrics will help us to better understand the effects of the traditional reuse practices and compare them with the effects of an explicit reuse oriented reuse model.

In summary, we believe that a comprehensive reuse framework needs to include (a) a reuse-enabling software evolution environment model, (b) detailed models of reuse and learning, and (c) characterization schemes for reuse and learning based upon these models.

3. A REUSE-ENABLING ENVIRONMENT MODEL

In the past, reuse has been discussed independent of the software evolution environment. We believe reuse can only be an effective mechanism if it is viewed as an integral part, paired with learning, of a reuse-enabling software evolution environment. None of the traditional engineering disciplines has ever introduced the reuse of building blocks as independent of the respective building process. For example, in civil engineering people have not created "reuse libraries" containing building blocks of all shapes and structures, and then tried to use them to build bridges, town houses, high-rises and cottages. Instead, they devised a standard technology for building certain types of buildings (e.g., town houses) through a long process of understanding and learning. This allowed them to define the needs for certain standard building blocks at well-defined stages of their construction process. In the software arena we have not followed this approach.

If we accept the premise that effective reuse requires a good understanding of the environment in which it is expected to take place, then we must model reuse in the context of a reuse-enabling software evolution environment. Such a context will allow us to learn how to reuse better. The ultimate expectation is that such improvement would lead to an ever increasing usage of generator-technology during software evolution. The ability to automate the generation of products from other products reflects the ultimate degree of understanding the underlying construction processes. Automated processes are easy to reuse. For example, in building compiler front-ends, we rarely reuse components of other compilers; instead, we reuse the compiler generators which automate the entire process of building compiler front-ends from formal language specifications.

In Section 3.1 we discuss how learning and reuse implicitly occur in the context of traditional software evolution environments. In Section 3.2, we discuss how learning and reuse can be explicitly modeled in the context of a reuse-enabling software evolution environment.

3.1. Implicit Learning and Reuse

During a workshop on "Requirements for Software Development Environments", held at the University of Maryland in 1985, a view of a software evolution environment was proposed that consisted of an information system and three information producers and consumers: people, methods, and tools [22]. The information system is defined by a software evolution process model describing the information, the communication among people, methods and tools, and the activity sequences for developing and maintaining software.

The traditional software evolution environment model in Figure 1 is a refinement of this earlier model.

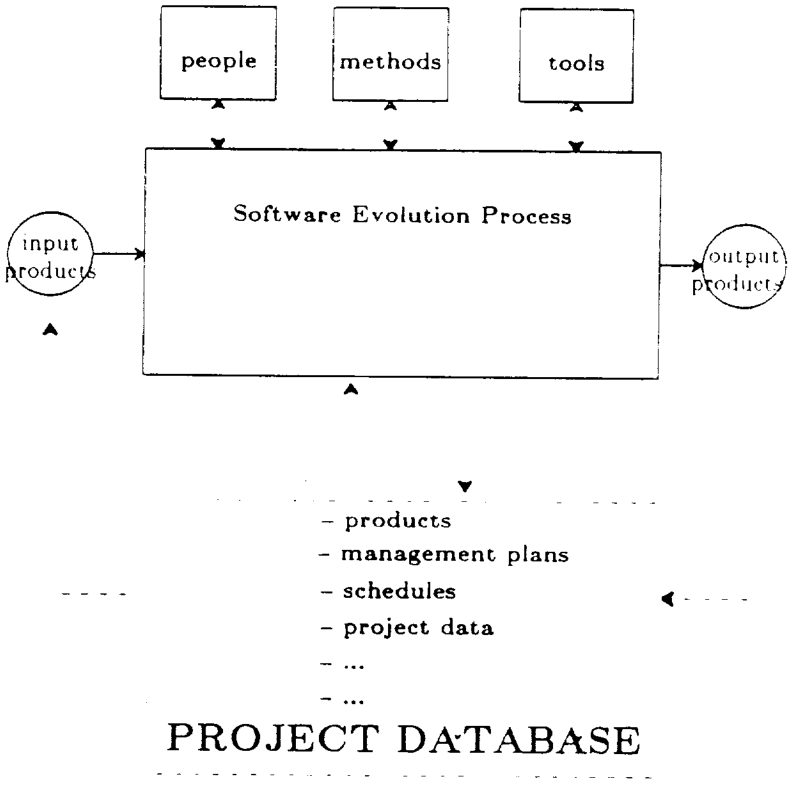


Figure 1: Traditional (non-reuse oriented) Software Evolution Environment Model

The purpose of the software evolution process is to produce output products, e.g., design documents, code, from input products, e.g., requirement documents. People execute this process manually or by utilizing available methods and tools. These methods and tools can be under the control of a project database. All or part of the information produced during this process is stored in a project database, e.g., products, plans such as management plans or schedules, project data.

Typically, support for such a traditional software evolution environment model includes a project database and means for the interaction of people with methods, tools, and the project database during software evolution. The experience of people, as well as some of the methods and tools, is usually not controlled by the project database. As a consequence, this experience is not owned by the organization (via the project database) but rather owned by individual human beings and lost entirely after the project has been completed.

Although the ideas of learning and reuse are not explicitly reflected in the traditional software evolution environment model, they do exist implicitly. The experience of the people involved in the software evolution process and the experience encoded in methods and tools is reused. In many cases, previously developed products are reused as input products. In the same way, products developed during one activity of the evolution process can be reused in subsequent activities of this same process. People learn (gain experience) from performing the activities of the evolution process. Another form of implicit learning occurs whenever products, plans, or project data are stored in the project database.

The basic problem in this traditional environment model is not that learning and reuse can not occur, but that learning and reuse are not explicitly supported and only because of individual efforts or by accident.

3.2. Explicit Modeling of Learning and Reuse

Systematic improvement of software evolution practices requires a reuse-enabling environment model which explicitly models learning, reuse and feedback activities, and integrates them into the software evolution process. Figure 2 depicts such a reuse-enabling environment model

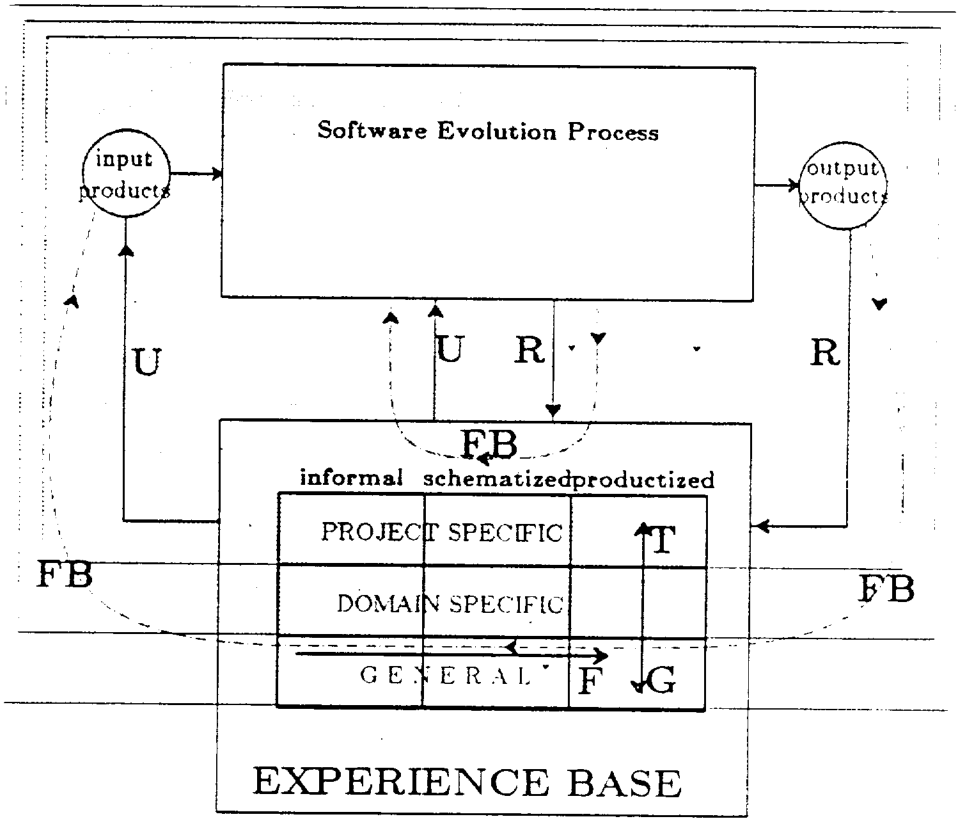


Figure 2: Reuse-Enabling Software Evolution Environment Model

All the potentially reusable experience, including software evolution methods and tools, are under the control of an experience base. Improvement is based on the feedback of existing experience (labeled with "FB" for reuse in Figure 2). Feedback requires learning and reuse. Systematic learning requires support for the recording of experience (labeled with "R" for recording in Figure

2), the off-line[*] generalizing or tailoring of experience (labeled with "G" and "T" for generalizing and tailoring in Figure 2), and the formalizing of experience (labeled with "F" for formalizing in Figure 2). Off-line generalization is concerned with movement of experience from project-specific to domain-specific and general; off-line tailoring is concerned with movement of experience from general to domain-specific and project-specific. Off-line formalization is concerned with movement of experience from informal to schematized and productized. Systematic reuse requires support for (re-)using existing experience (labeled with "U" for use in Figure 2), and on-line[*] generalizing or tailoring of candidate experience (not explicitly reflected in Figure 2, because it is assumed to be an integral part of the (re-)use activity).

Although reuse and learning are possible in both the reuse-enabling and the traditional environment models, there are significant differences in the way experience is viewed and how learning and reuse are explicitly integrated and supported. The basic difference between the reuse-enabling model and the traditional model is that learning and reuse become explicitly modeled and are desired characteristics of software evolution.

3.2.1. Recording Experience

The objective of recording experience is to create a repository of well specified and organized experience. This requires a precise description of the experience to be recorded, the design and implementation of a comprehensive experience base, and effective mechanisms for collecting, validating, storing and retrieving experience. We replace the project database of the traditional environment model by an the more comprehensive concept of an experience base which is intended to capture the entire body of experience recorded during the planning and execution of all software projects within an organization. All information flows between the software evolution process and the experience base reflecting the recording of experience are labeled with "R" in Figure 2.

[*] The attributes "on-line" and "off-line" indicate whether the corresponding activities are performed as part or independent of any particular software evolution project.

Examples of recording experience include such activities as (a) storing of appropriately documented, catalogued and categorized code components from prior systems in a product library, (b) cataloguing of a set of lessons learned in applying a new technology in a knowledge base, or (c) capturing of measurement data related to the cost of developing a system in a measurement database.

In the SEL example of Section 2, code from prior systems is available to the program library of the current project although no code object repository has been developed. Measurement data characterizing a broad number of project aspects such as the project environment, methods and tools used, defects encountered, and resources spent are explicitly stored in the SEL measurement database [2, 8, 18]. Requirements and design documents as well as lessons learned about the technical and managerial implications of various methods and tools are implicitly stored in humans or on paper.

Today it is possible, but not common, to find product libraries. It is even less common to record process-related experience such as process plans or data which characterize the impact of certain methods and tools within an organization. There exist two main reasons why we need to record more process-related experience: (a) it is generally hard to modify existing products efficiently without any knowledge regarding the processes according to which they were created and (b) the effective reuse of process-related experience such as process plans or data could provide significantly more leverage for improvement than just the reuse of products.

3.2.2. Generalizing & Tailoring Existing Experience Prior to its Potential Reuse

The objective of generalizing existing experience prior to its reuse is to make a candidate reuse object useful in a larger set of potential target applications. The objective of tailoring existing experience prior to its potential reuse is to fine-tune a candidate reuse object to fit a specific task or exhibit special attributes, such as size or performance. These activities require a well documented cataloged and categorized set of reuse objects, mechanisms that support the modification process, and an understanding of the potential target applications. Generalization and tailoring are specifically concerned with movement across the boundaries of the "generality" dimension: from general to domain-specific and project-specific and vice versa. Objectives and characteristics are different from project to project, and even more so from environment to environment. We cannot reuse past experience without modifying it to the needs of the current project. The stability of the environment in which reuse takes place, as well as the origination of the experience, determine the amount of tailoring required.

Examples of generalizing and tailoring experience include such activities as (a) developing a generic package from a specific package, (b) instantiating a generic package for a specific type, (c) generalizing lessons learned from a specific design technology for a specific application to any design for that application or any application, (d) or parameterizing a cost model for a specific environment.

In the SEL, requirements and design documents have implicitly evolved to be applicable to all FORTRAN projects in the ground support software domain. Measurement data have been explicitly generalized into domain-specific baselines regarding defects and resource expenditures [2, 8, 18]. Requirements and designs are implicitly tailored towards the needs of a new project based on the manager's experience, and code is explicitly hand-modified to the needs of a new project.

In general, recorded experience is project specific. In order to reuse this experience in a future project within the same application domain, we have to (a) generalize the recorded project specific experience into domain specific or general experience and (b) then tailor it again to the specific characteristics of the new project. We distinguish between off-line and on-line generalizing and tailoring activities:

- **Off-line generalizing and tailoring** is concerned with increasing the reuse potential of existing process and product-related experience before knowing the precise reuse context (i.e., the project within which the experience is being reused). Off-line generalization and tailoring is concerned with movement across the boundaries of the specificity dimension within the experience base: from general to domain-specific and then to project-specific, and visa versa. These activities are labeled with "G" and "T" in Figure 2. An example of off-line generalization is the construction of baselines. The idea is to use project-specific measurement data (e.g., fault profiles across development phases) of several projects within some application domain and to create the application-domain specific fault profile baseline. Each new project within the same application domain might reuse this baseline in order to control its development process as far as faults are concerned. An example of off-line tailoring is the adaptation of a general scientific paradigm such as "divide and conquer" to the software engineering domain.

- **On-line tailoring and generalizing** is concerned with tailoring candidate process and product-related experience to the specific needs and characteristics of a project and the chosen software evolution environment. These activities are not explicitly reflected in Figure 2 because they are integral part of the (re-)use activity. An example of on-line tailoring is the adaptation of a design inspection method to better detect the fault types anticipated in the current project [6]. An example of on-line generalization is the inclusion of project specific effort data from a past project into the domain specific effort baseline in order to better plan the required resources for the current project. Obviously, this kind of generalization could have been performed off-line too.

It is important to find a cost-effective balance between off-line and on-line tailoring and generalization. It can be expected that generalization is predominantly performed off-line, tailoring on-line.

A good developer is capable of informally tailoring general and domain specific experience to the specific needs of his or her project. Performing these transformations on existing experience assumes the ability to generalize experience to a broader context than the one studied, or to tailor experience to a specific project. The better this experience is packaged, the better our understanding of the environment. Maintaining a body of experience acquired during a number of projects is one of the prerequisites for learning and feedback across projects.

A misunderstanding of the importance of tailoring exists in many organizations. These organizations have specific development guidebooks which are of limited value because they "are written for some ideal project" which "has nothing in common with the current project and, therefore, do not apply" [23]. All guidebooks (including standards such as DOD-STD-2167) are general and need to be tailored to each project in order to be effective.

3.2.3. Formalizing Existing Experience Prior to its Potential Reuse

The objective of formalizing existing experience prior to its potential reuse is to increase the reuse potential of a candidate reuse object by encoding it in more precise, better understood ways. This requires models of the various reuse objects, notations for making the models more precise, notations for abstracting reuse object characteristics, mechanisms for validating these models, and mechanisms for interpreting models in the appropriate context. Formalization activities are concerned with movement across the boundaries of the formality dimension within the experience base: from informal to schematized and then to productized. These activities are labeled with "F" in Figure 2.

Examples of formalizing experience include such activities as (a) writing functional specifications for a code module, (b) turning a lessons learned document into a management system that supports decision making, (c) building a cost model empirically based upon the data available, (d) developing evaluation criteria for evaluating the performance of a particular method, or (e) automating methods into tools.

In the SEL, measurement data have been explicitly formalized into cost models [1] and error models enabling the better planning and control of software projects with regard to cost estimation and the effectiveness of fault detection and isolation methods [2, 6, 8, 18]. Lessons learned have been integrated into expert systems aimed at supporting the management decision process [5, 24].

The more we can formalize experience, the better it can be reused. Therefore, we try not only to record experience, but over time to formalize experience from entirely informal (e.g., concepts), to structured or schematized (e.g., methods), or even to completely formal (e.g., tools) The potential for misunderstanding or misinterpretation decreases as experience is described more formally. To the same degree the experience can be modified more easily, or in the case of processes, it may be executed automatically (e.g., tools) rather than manually (e.g., methods).

3.2.4. (Re–) Using Existing Experience

The objective of reusing existing experience is to maximize the effective use of previously recorded experience during the planning and execution of all projects within an organization. This requires a precise characterization of the available candidate reuse objects, a precise characterization of the reuse–enabling environment including the evolution process that is expected to enable reuse, and mechanisms that support the reuse of experience. We must support the (re–)use of existing experience during the specification of reuse needs in order to compare them with descriptions of existing experience, the identification and understanding of candidate, the evaluation of candidate reuse objects, the possible tailoring of the reuse object, the integration of the reuse object into the ongoing software project, and the evaluating of the project's success. All information flows between the experience base and the software evolution process reflecting the (re–)use of experience are labeled with "U" in Figure 2.

Examples of reusing experience include such activities as (a) using code components from the repository, (b) developing a risk management plan based upon the lessons learned from applying a new technology, (c) estimating the cost of a project based on data collected from past projects, or (d) using a development method created for a prior project.

In the SEL, reuse needs are informally specified as part of the requirements document. Matching candidate requirements and design documents are identified by managers who are experienced in this environment. The evaluation of those candidate reuse objects is in part based on human experience and in part on measurement data. They are tailored based on the application-domain knowledge of the personnel. They are integrated into a very stable evolution process based on human experience. All this reuse is implicit except for the reuse of code, which although explicit, is informal. It could only be successful because it evolved within a very stable environment. The recent change from FORTRAN to Ada has resulted in drastic changes of this environment and as a consequence to the loss in the implicit reuse heritage.

Since the key for improvement of products is always improvement of the process creating those products, we need to put equal emphasis on the reuse of product and process oriented experience. Even today, we have examples of reuse of process experience such as process plans (standards such as DOD-STD-2167, management plans, schedules) or process data (error, effort or reliability data that define baselines regarding software evolution processes within a specific organization). In most of these cases the actual use of this information within a specific project context is not supported; it is up to the respective manager to find the needed information, and to make sense out of it in the context of the current project.

4. TAME: AN INSTANTIATION OF THE REUSE-ENABLING ENVIRONMENT MODEL

The objective of the reuse-enabling software evolution environment model of Section 3.2 is to explicitly model the learning and reuse-related activities of recording experience, generalizing and tailoring experience, formalizing experience, and (re-)using experience so that they can be understood, evaluated, predicted and motivated.

In order to instantiate a specific reuse-enabling environment, we need to choose a model of the software evolution process itself. In general, such an evolution process model needs to be capable of describing the integration of learning and reuse into the software evolution process. In particular, it needs to be capable of modeling when experience is created and recorded into the experience base as well as when existing experience is used. It needs to provide analysis for the purpose of on-line feedback, evaluating the application of all reuse experience, and off-line feedback for improving the experience base.

The reuse-enabling TAME environment model depicted in Figure 3 is an instantiation of the reuse-enabling software environment model of Section 3.2, based on a very general improvement oriented evolution process model.

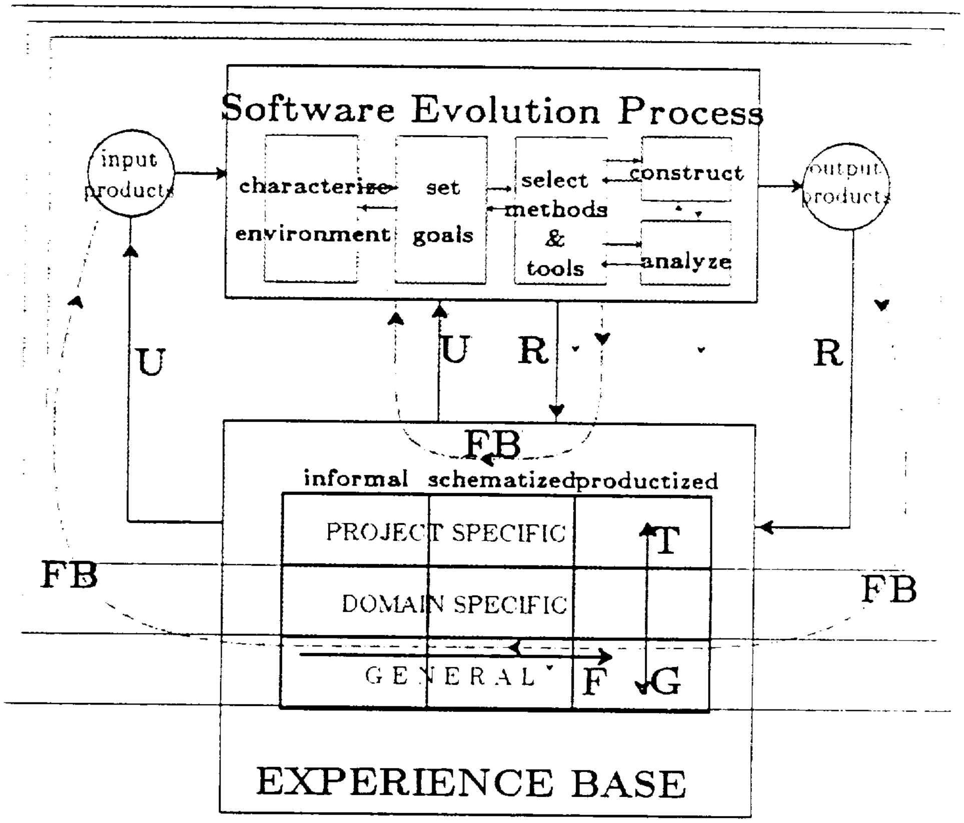


Figure 3: Reuse-Enabling "TAME" Environment Model

Each software project performed according to this improvement oriented evolution process model consists of a planning and an execution stage. The planning stage includes a characterization of the current status of the project environment, the setting of project and improvement goals, and the selection of construction and analysis methods and tools that promise to meet the stated goals in the context of the characterized environment. The execution stage includes the construction of output products and the analysis of these construction processes and resulting output products.

The TAME environment model gives us a basis for discussing the integration of the recording and (re–)use activities into the software evolution process. During the environment characterization stage of the improvement oriented process model we (re–)use knowledge about the needs and characteristics of previous projects and record the needs and characteristics of the current project into the experience base. During the goal setting stage we (re–)use existing plans for construction and analysis from similar projects and record the new plans which have been tailored to the needs of the current project into the experience base. During the method and tool selection stage, we (re–)use as many of the constructive and analytic methods and tools which had been used successfully in prior projects of similar type as feasible and record possibly tailored versions of these methods and tools into the experience base. During construction we apply the selected methods and tools, and record the constructed products into the experience base. During analysis we use the selected methods and tools in order to collect and validate data and analyze them and record the data, analysis results and lessons learned into the experience base.

The TAME environment explicitly supports the capturing of all kinds of experience. The consistent application of the improvement oriented process model across all projects within an organization provides a mechanism for evaluating the recorded experience, helping us to decide what and how to reuse, tailoring and analyzing. TAME supports continuous learning. The explicit and comprehensive modeling of the reuse–enabling evolution environment including the experience base, the evolution process, and the various learning and reuse activities (see Figure 3) allows us to measure and evaluate all relevant aspects of reuse. The measurement methodology used and supported within the TAME environment has been published in earlier papers [7, 8].

5. CONCLUSIONS

In this paper we have motivated and outlined the scope of a comprehensive reuse framework, introduced a reuse-enabling software environment model as a first step towards such a comprehensive reuse framework, and presented a first instantiation of such an environment in the context of the TAME (Tailoring A Measurement Environment) project at the University of Maryland [7, 8].

The reuse-enabling software evolution environment model presented in Section 3 provides a basic environment for supporting the recording of experience, the off-line generalization and tailoring of experience, the off-line formalization of experience, and the (re-) use of existing experience.

Further steps required towards the outlined reuse framework are more specific models of each of these activities that differentiate the components of these activities and serve as a basis for characterization, discussion and analysis. We are currently taking the reuse-enabling software environment model of section 3.2 down one level and developing a model for (re-)using experience. Based on this reuse model we will develop a reuse taxonomy allowing for the characterization of any instance of reuse. The reuse model will provide insight into the other activities of the reuse-enabling environment model only in the way they interact with the (re-)use activity. Corresponding models for each of the other activities need to be developed and integrated into the reuse-enabling software environment model.

The reuse-enabling TAME environment model serves as a basis for better understanding, evaluating and motivating reuse practices and necessary research activities. Performing projects according to the TAME environment model requires powerful automated support for dealing with the large amounts of experience and performing the complicated activities of recording, generalizing and tailoring, formalizing, and (re-)using experience. Indispensable components of such an automated support system are a powerful experience base, and a measurement support system. Many of the reuse approaches in the past have assumed that the developer has sufficient implicit knowledge of the characteristics of the particular project environment, specific needs for reuse, the candidate reuse objects, etc. It is not trivial to have all this information available. The institutionalized learning of an organization and the proper documentation of that knowledge is definitely one of the keys to effective reuse. This leads to even better specification methods and tools (one of the frequently mentioned keys to effective reuse).

As part of the TAME project at the University of Maryland we have been working on providing appropriate support for building such an experience base, and supporting learning and (re-)use via measurement. We have completed several components towards a first prototype TAME system. These components include the definition of project goals and their refinement into quantifiable questions and metrics, the collection and validation of data, their analysis and the storage of all kinds of experience. One of the toughest research problems is to use measurement not only for analysis, but also for feedback (learning and reuse) and planning purposes. We need more understanding of how to support feedback and planning. The TAME system is intended to serve as a vehicle for our research towards the effective support of explicit learning and reuse as outlined in this paper.

6. ACKNOWLEDGEMENTS

We thank all our colleagues and graduate students who contributed to this paper by either working on the TAME or any other reuse-related project or reviewing earlier versions of this paper.

7. REFERENCES

[1] J. Bailey, V. R. Basili, A Meta-Model for Software Development Resource Expenditures," in Proc. Fifth International Conference on Software Engineering, San Diego, USA, March 1981, pp. 107-116.

[2] V. R. Basili, "Can We Measure Software Technology: Lessons Learned from Eight Years of Trying," in Proc. Tenth Annual Software Engineering Workshop, NASA Goddard Space Flight Center, Greenbelt, MD, December 1985.

[3] V. R. Basili, "Quantitative Evaluation of Software Methodology," Dept. of Computer Science, University of Maryland, College Park, TR-1519, July 1985 (also in Proc. of the First Pan Pacific Computer Conference, Australia, September 1986).

[4] Victor R. Basili, "Software Maintenance = Reuse-Oriented Software Development," in Proc. Conference on Software Maintenance, Key-Note Address, Phoenix, AZ, October 1988.

[5] V. R. Basili, C. Loggia Ramsey, "ARROWSMITH-P - A Prototype Expert System for Software Engineering Management," IEEE Proceedings of the Expert Systems in Government Symposium, McLean, VA, October 1985, pp. 254-264.

[6] V. R. Basili, H. D. Rombach, "Tailoring the Software Process to Project Goals and Environments," Proc. of the Ninth International Conference on Software Engineering, Monterey, CA, March 30 - April 2, 1987, pp. 345-357.

[7] V. R. Basili, H. D. Rombach, "TAME: Integrating Measurement into Software Environments." Technical Report TR-1764 (or TAME-TR-1-1987), Dept. of Computer Science, University of Maryland, College Park, MD 20742, June 1987.

[8] V. R. Basili, H. D. Rombach "The TAME Project: Towards Improvement-Oriented Software Environments," IEEE Transactions on Software Engineering, vol. SE-14, no. 6, June 1988, pp. 758-773. (is also available as Technical Report (UMIACS-TR-88-8, CS-TR-1983, or TAME-TR-2-1988), Department of Computer Science, University of Maryland, College Park, MD 20742).

[9] V. R. Basili, H. D. Rombach, J. Bailey, and B. G. Joo, "Software Reuse: A Framework." Proc. of the Tenth Minnowbrook Workshop on Software Reuse, Blue Mountain Lake, New York, July 1987.

[10] V. R. Basili, R. W. Selby, D. H. Hutchens, "Experimentation in Software Engineering," IEEE Transactions on Software Engineering, vol.SE-12, no.7, July 1986, pp.733-743.

[11] V. R. Basili and M. Shaw, "Scope of Software Reuse," White paper, working group on 'Scope of Software Reuse', Tenth Minnowbrook Workshop on Software Reuse, Blue Mountain Lake, New York, July 1987 (in preparation).

[12] Ted Biggerstaff, "Reusability Framework, Assessment, and Directions," IEEE Software Magazine, March 1987, pp.41-49.

[13] P. Freeman, "Reusable Software Engineering: Concepts and Research Directions," Proc. of the Workshop on Reusability, September 1983, pp. 63-76.

[14] R. Prieto-Diaz, P. Freeman, "Classifying Software for Reusability," IEEE Software, vol.4, no.1, January 1987, pp. 6-16.

[15] IEEE Software, special issue on 'Reusing Software', vol.4, no.1, January 1987.

[16] IEEE Software, special issue on 'Tools: Making Reuse a Reality', vol.4, no.7, July 1987.

[17] G. A. Jones, R. Prieto-Diaz, "Building and Managing Software Libraries," Proc. Compsac'88, Chicago, October 5-7, 1988, pp. 228-236.

[18] F. E. McGarry, "Recent SEL Studies," in Proc. Tenth Annual Software Engineering Workshop, NASA Goddard Space Flight Center, Greenbelt, MD, Dec. 1985.

[19] Mary Shaw, "Purposes and Varieties of Software Reuse," Proceedings of the Tenth Minnowbrook Workshop on Software Reuse, Blue Mountain Lake, New York, July, 1987.

[20] T. A. Standish, "An Essay on Software Reuse," IEEE Transactions on Software Engineering, vol. SE-10, no. 5, September 1984, pp.494-497.

[21] W. Tracz, "Tutorial on 'Software Reuse: Emerging Technology'," IEEE Catalog Number EHO278-2, 1988.

[22] M. V. Zelkowitz (ed.), "Proceedings of the University of Maryland Workshop on 'Requirements for a Software Engineering Environment', Greenbelt, MD, May 1986," Technical Report TR-1733, Dept. of Computer Science, University of Maryland, College Park, MD 20742, December 1986 (to be published as a book, Ablex Publ., 1988).

[23] M. V. Zelkowitz, R. Yeh, R. Hamlet, J. Gannon, V. R. Basili, "Software engineering practices in the U.S. and Japan," IEEE Computer Magazine, June 1984, pp. 57-66.

[24] J. Valett, B. Decker, J. Buell, "The Software Management Environment," in Proc. Thirteenth Annual Software Engineering Workshop, NASA Goddard Space Flight Center, Greenbelt, MD, November 30, 1988.

SECTION 4—ADA TECHNOLOGY STUDIES

SECTION 4 - ADA TECHNOLOGY STUDIES

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

- "Evolution of Ada Technology in a Production Software Environment," F. McGarry, L. Esker, and K. Quimby, Proceedings of the Sixth Washington Ada Symposium (WADAS), June 1989
- "Using Ada to Maximize Verbatim Software Reuse," M. Stark and E. Booth, Proceedings of TRI-Ada 1989, October 1989

EVOLUTION OF ADA TECHNOLOGY IN A PRODUCTION SOFTWARE ENVIRONMENT

Frank McGarry

National Aeronautics and Space
Administration
Goddard Space Flight Center
Greenbelt, Md. 20771

Linda Esker

Computer Sciences Corporation
System Sciences Division
10110 Aerospace Rd.
Lanham-Seabrook, Md. 20706

Kelvin Quimby

Computer Sciences Corporation
System Sciences Division
10110 Aerospace Rd.
Lanham-Seabrook, Md. 20706

INTRODUCTION

The Ada programming language and the associated software engineering disciplines have been described as one of the most significant developments in software technology in many years. Although many claims have been made about its advantages and impacts, there have been very few empirical studies that clarify the impact of Ada.

The Software Engineering Laboratory (SEL) is an organization sponsored by the National Aeronautics and Space Administration (NASA) consisting of three principal members: NASA/Goddard Space Flight Center, the University of Maryland, and Computer Sciences Corporation. The SEL was founded in 1976 to carry out studies and measurements related to evolving software technologies [7]. The studies are aimed at understanding both the software development process and the impacts that evolving software practices may have on the software process and product. Since 1976, the SEL has conducted over 65 experiments by applying selected techniques to specific development efforts and measuring the resulting process and product.

In early 1985, the SEL initiated an effort to study the characteristics, applications, and impacts of Ada. Beginning with a relatively small practice problem (6000 source lines of Ada), the SEL has collected detailed development data from a total of eight Ada projects (some of which are still ongoing). The projects range in size from 6000 lines to approximately 160,000 lines of code.

PROJECT BACKGROUND

Development Environment

All Ada projects studied were developed in a DEC environment, using either a VAX 11/780 or a VAX 8600. Both machines are shared with other general users, and the support was average compared to other typical projects developed in FORTRAN. As will be pointed out later, varying degrees of use were made of the available tools and methodologies.

In studying the series of Ada projects, the goal/question/metric (GQM) paradigm [3] was followed. The goal of the study was to determine the impact of Ada on productivity, reliability, reuse, and general product characteristics. A second interest was to study the use of Ada features (such as generics and strong typing) over time.

Project History

Information on six Ada projects was analyzed for this study. The projects were developed over a span of 4-1/2 years starting in late 1984 and ending with two projects that will be completed in 1989. The study categorized the six projects into three groups distinguished solely by approximate start date: the first two are called the first Ada projects, the next two are the second Ada projects, and the most recent are the third Ada projects. The timeline for the six projects is shown in Figure 1.

The first experiences with Ada in the SEL occurred with two projects that were initiated in late 1984 and early 1985. A team of seven programmers was formed in late 1984 and began extensive training in December of 1985. The first target project was a simulator that was required to model an attitude control system of a particular NASA satellite, the Gamma Ray Observatory (GRO). Comparable simulators had been developed in the past by NASA, and this particular Ada project (GRODY) was developed in parallel to the identical project being developed in FORTRAN. The results of that particular comparison are documented by Agresti et al. [1] and McGarry and Nelson [14]. It was estimated that the development of GRODY would probably require from 10 to 12 staff-years of effort, considering previous experiences with similar FORTRAN projects.

The GRODY team had an average of nearly 5 years of experience with software development; however, none had any previous Ada experience. In fact, there had been no earlier Ada experience by anyone in this environment, so no lessons learned and no Ada experts were available to team members. The team was experienced with flight dynamics problems although it was, on the average, less experienced than the typical development teams in this environment.

To prepare for the design and development of GRODY, the team underwent 6 months of extensive

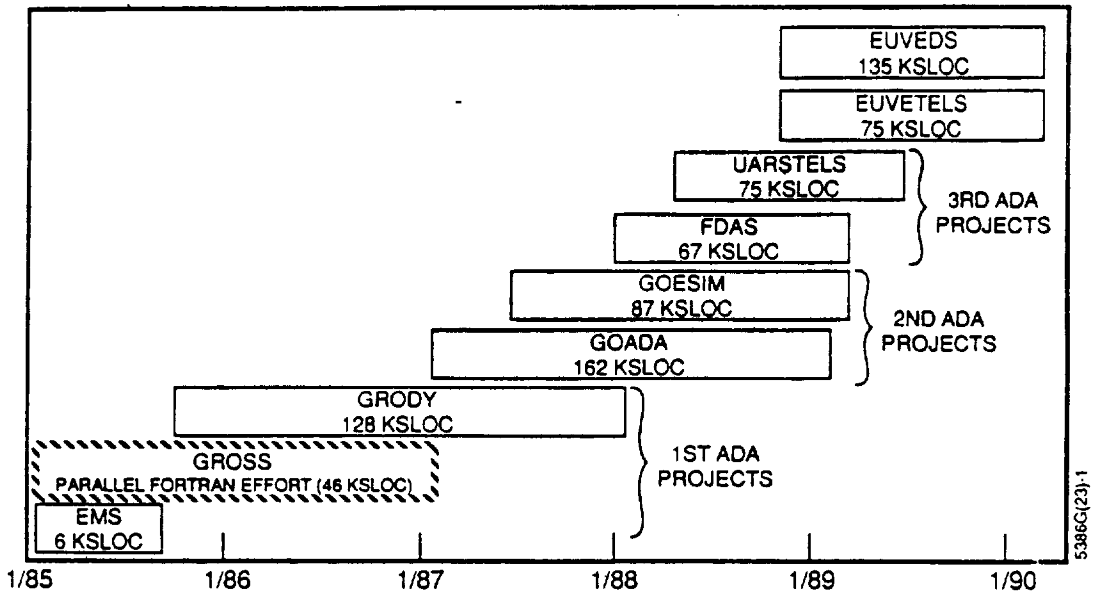


Figure 1. Ada Projects in the Flight Dynamics Division of NASA/Goddard

training, which covered Ada syntax as well as principles of software engineering and detailed design techniques [16]. The training included the following:

- Alsys video tapes with discussion
- Lectures on Ada from University of Maryland staff
- Lectures and workshops on PAMELA [8], object-oriented design, and other design techniques
- Workshops based on Booch's training materials [5]
- Lectures on software engineering principles, including abstraction and information hiding

During the 6 months of training, the electronic mail system (EMS) Ada project was developed. This was the first Ada effort completed by this organization. It was set up as a training problem, but detailed statistics were kept so that the development process and product could be analyzed. When completed, EMS consisted of approximately 6000 lines of Ada code.

These first Ada projects, GRODY and EMS, were developed by the same personnel in a similar software development environment. The EMS project was developed on a VAX 11/780 using the DEC Ada Compilation System (ACS); the GRODY project was developed on the VAX 8600, also using the DEC ACS.

The two projects classified as the second Ada projects both began in early 1987 and were of medium size with complexity typical of other efforts in this environment. Both projects were simulators required to support the GOES mission. GOADA was a dynamics simulator similar to GRODY; GOESIM was a telemetry simulator that would be used in testing the attitude ground support software.

The GOADA team consisted of approximately seven people, some of whom were not assigned full time to this project. Of these seven, three had previous Ada experience and two had experience in the application area. The GOESIM team consisted of four people, one of whom had previous Ada experience and one who had experience with this type of application. The training consisted of lectures and video tapes on Ada, with particular attention to the Ada style guide that had recently been developed; lectures and classes in Ada syntax and Ada concepts were also held. The training lasted approximately 4 weeks for each team.

Both of these second Ada projects used the DEC ACS to complete development. Because there had been previous Ada experience from the first Ada projects, experienced Ada programmers were available to these teams for consultation and guidance. They proved to be heavily used commodities. The second Ada projects spanned approximately 18 months each.

The final two projects analyzed in this study, the third Ada projects, were both started in early 1988. The UARSTELS project had requirements and characteristics similar to GOESIM, one of the second Ada projects. The other project, FDAS, was a much difference type of system, a source code manager used for manipulating flight dynamics software components.

The UARSTELS team consisted of three people, one of whom had previous Ada experience; the FDAS team consisted of four people, none of whom had previous Ada experience. Training for both teams took the form of lectures and workshops based on the Ada style guide; additional training from personnel with previous Ada experience was also provided. All team members attended classes on Ada syntax and Ada development.

The teams supporting the third Ada projects had more personnel available to them who had previous Ada experience, and they also had available several lessons-learned reports that had been developed by the earlier Ada projects. These two projects spanned approximately 15 to 18 months each.

Data Collection

Detailed data for this study were collected for the six projects. As for all software developed in the flight dynamics environment, the data are collected from the following sources:

- Data collection forms
- Tools and accounting records
- Interviews and subjective information

The most extensive and detailed data were provided on a series of data collection forms completed by both the developers and managers and including the following:

- Effort data (hours spent weekly by activity)
- Error data (for all changes and errors)
- Project estimation (managers' estimate of final size, cost, schedules)
- Product origination (characteristics of modules as they are designed)

These data were the major source of information used in the comparisons.

The tools used to record information include the automated accounting system (for records of computer time used, source code changes, and source code size history); configuration control tools (Configuration Management System (CMS), used to record changes to source code); and ASAP, the Ada Static Source Code Analyzer [9], which calculates detailed counts and characteristics of Ada source code. Information was also provided through interviews with team developers and managers (to record lessons learned and general impressions) and through subjective assessments made by senior software engineers (on topics such as methodologies applied). All this information is consistently collected, quality assured, and recorded on a data base where it is used as the basis for study or analysis.

EVOLVING CHARACTERISTICS OF THE SOFTWARE

Design Characteristics

The first Ada project did not automatically assume the reuse of the standard FORTRAN project methodologies and products. During the predesign phase, the project decided on the products, reviews, and methodology to be used. Project personnel decided to conform to the traditional design review process that had been used in the flight dynamics environment for many years [2, 15]. However, project members investigated several design methodologies and eventually applied a modified version of object-oriented design [1, 12, 19].

Use of an object-oriented design required a rethinking of the design, its documentation, and its presentation. This caused some inconsistencies with the traditional approach used in the SEL. Project members were required to develop new design products because the existing design documentation methods for structured and functionally oriented software design did not apply to the object-oriented design [17]. To develop a design notation used in the design documentation, they combined concepts from George Cherry's process abstraction method, PAMELA [8], and Grady Booch's object-oriented design [5]. Design diagrams were presented at the preliminary design review (PDR), and top-level package specifications were developed for the critical design review (CDR). These components were then expanded and compiled during Build 0 of the implementation phase.

The second and third Ada projects adopted and built on the same methodology and design notation. They also conducted design reviews, but the specific products generated at each stage were somewhat modified from the first Ada efforts. For these later projects, package specifications were developed for the PDR, and package bodies and subunit program design language (PDL) were developed for the CDR. In addition, the components were compiled before the CDR. These efforts pointed out the need to redefine the specific products produced at key milestones of the design process; however, the characteristics of the products have yet to be decided. Quimby and Esker [17] present a more detailed analysis of the evolving characteristics of both the design process and the products developed.

Software Size

Traditionally, software size has been described in terms of the lines of code developed for the system. Lines of code can, however, be expressed by many measurements [11], including the following:

- Total physical lines of code (carriage returns)
- Noncomment/nonblank physical lines of code
- Executable lines of code (ELOC) (not including declarations)
- Statements (semicolons in Ada, which include declarations)

Table 1 describes the size of the Ada projects in the flight dynamics environment using these four measurements. For comparison to the Ada projects, typical FORTRAN projects of similar applications are also summarized.

Unless only Ada statements are counted, these figures indicate that using Ada results in many more lines of code than using FORTRAN. The increase in lines of code is not necessarily a negative result; rather, it means that the size of the system implemented in Ada will be larger than an equivalent system in FORTRAN. It is also clear

Table 1. Software Characteristics of Projects

APPLICATION	1ST ADA PROJECTS	2ND ADA PROJECTS		3RD ADA PROJECTS		FORTRAN PROJECTS	
	DYN SIM	DYN SIM	TM SIM	CONFIG	TM SIM	DYN SIM	TM SIM
TOTAL LINES (CARRIAGE RETURNS)	128,000	162,200	87,500	67,700	75,000	45,500	28,000
NONCOMMENT/ NONBLANK	60,000	79,900	42,300	36,000	41,400	26,000	15,000
EXECUTABLE LINES (NO DECLARATIONS)	40,250	49,000	25,500	19,700	22,150	22,500	12,500
STATEMENTS (SEMICOLON – INCLUDES DECLARATIONS)	22,500	29,200	16,300	12,700	15,200	22,300	12,000

5386G(23)-5

Table 2. Effort Distribution (Percent of Total Effort) During Each Life-Cycle Phase of Ada and FORTRAN Projects

PHASE	1ST ADA PROJECTS	2ND ADA PROJECTS	3RD ADA PROJECTS	FORTRAN PROJECTS
REQUIREMENTS ANALYSIS	8	4.3	6.7	12.5
DESIGN	24	30.5	36.0	22.5
CODE	42	52.0	44.0	35.0
TEST	26	13.2	13.3	30

5386G(23)-6

that a precise definition is needed of what constitutes a line of code in Ada and what types of code are included in that measurement.

Many factors contribute to the increased size of the Ada projects. The style of Ada results in code growth because it encourages formatting, blank lines, and longer, readable names for data elements and subunits. The strong typing within Ada also produces more code than in FORTRAN because each data element must be explicitly declared. In addition, the local style guide places further requirements on the format for readability. Among other requirements, the style guide stipulates that each calling argument must be on a separate physical line. All these features have increased the code size, but the increased size also provides advancements in the areas of capability, readability, and understanding.

<u>Effort Distribution by Phase Dates</u>

Effort distributions can be described by the effort expended during the key life-cycle phases of a project and by the effort expended in software development activities. Using the first approach, effort distribution by phase dates, the typical FORTRAN life-cycle effort distribution [15] in the flight dynamics environment shows 12.5 percent of the total effort expended during the predesign or requirements analysis phase, 22.5 percent during the design phase, 35 percent during the code implementation phase, and 30 percent during the system test phase (Table 2).

From the review of literature on Ada [18], it was expected that the effort distributions would be significantly different for the Ada projects due to the modified design and implementation approaches. It had been anticipated that the Ada projects would require more effort during the detailed design phase and less effort during the code and test phases. However, in the flight dynamics environment, significant changes to the life cycle have not been observed. The Ada projects were planned by managers experienced with FORTRAN projects, and perhaps their plans were influenced by the FORTRAN life cycle.

Although the changes are not occurring as quickly as anticipated, the Ada life cycle is changing slightly with each project and may soon show a different life cycle than that expected for a FORTRAN project. The life cycles for the second and third Ada projects are shifting slightly to show more design time required and less system test time. The effort distributions of the Ada

Table 3. Effort Distribution (Percent of Total Effort) for Development Activities Across All Life-Cycle Phases of Ada and FORTRAN Projects

PHASE	1ST ADA PROJECTS	2ND ADA PROJECTS	3RD ADA PROJECTS	FORTRAN PROJECTS
REQUIREMENTS ANALYSIS	3.2	3.7	6.5	6.0
DESIGN	36.5	27.5	35.7	24.0
CODE	42.7	52.0	44.5	45.0
TEST	17.6	16.8	13.3	25.0

5386G(23)-7

projects are showing that the FORTRAN life cycle cannot be automatically assumed for Ada.

These observations probably indicate that the life-cycle definition is not easily changed merely because a different language or technique is applied. The evolution toward the expected characteristics of the new technology is a slow, gradual process.

Effort Distribution by Activity

The second approach to effort distributions is to analyze the effort by activity. In addition to collecting the effort expended between key phase dates (e.g., between the CDR and the start of system test), the SEL also collects detailed effort data independent of phase dates. Effort by phase is time driven and assumes, for example, that all design activities are complete and cease at the end of the design phase. In reality, many activities take place during each life-cycle phase and, therefore, the effort distribution by activity can be quite different from the distribution by phases.

This, indeed, was the case for the first Ada projects. Although only 24 percent of the total effort was allocated to the design phase, 36.5 percent of the total project effort was spent on design activities (Tables 2 and 3). The extra effort needed for Ada design activities is more apparent in the distribution of effort by activity than in the distribution by phase dates. The effort by development activity again reinforces the trend seen above. On the Ada projects, more effort was required for design and less effort for software testing than on the FORTRAN projects.

Use of Ada Features

In an effort to achieve some measurement of the use of the features available in the Ada language, the SEL identified six Ada features to monitor: generic packages, type declarations, packages, tasks, compilable PDL, and exception handling. The SEL then examined the code to see how little or how much these features were used. The purposes of this analysis were, first, to determine to what degree features of Ada were used by the Ada project and, second, to determine whether the use of Ada features "matured" as an environment gained experience with the language. Data on the use of these Ada features were obtained using the Ada Static Source Code Analyzer Program [9]. Analysis of the use of compilable PDL and exception handling did not show any trends. Perhaps it is too early to see results in these areas; however, trends were observed in the use of the other features.

The average size of packages (in source lines of code (SLOC)) for the first Ada projects is much larger than the average size of packages for the second and third Ada projects (Figure 2). This increase is due to a difference in the structuring method between the first Ada projects and all subsequent Ada projects [17]. The first Ada projects were designed with one package at the root of each subsystem, which led to a heavily nested structure. In addition, nesting of package specifications within package bodies was used to control package visibility. Current Ada projects are using the view of subsystems described by Grady Booch [6, Ch. 17] as an abstract design entity whose interface is defined by a number of separately compilable packages, and the only nested Ada packages are generic package instantiations.

The generic package is a major tool in the Ada language contributing to software reusability. Reports have shown the benefits of Ada reusable software [18] and, in the flight dynamics environment, use of generic packages has been increasing from the first to the current Ada projects. More than one-third of the packages on current projects are generic packages. Although more analysis is needed, this higher use of generics possibly reflects both a stronger emphasis on the development of verbatim reusable components and an increased understanding of how to use generic Ada packages effectively in the flight dynamics environment.

The use of strong typing in these software systems was measured by the number of type declarations per thousand lines of code. Although the measure itself is not intuitively meaningful, it provides a method of observing trends in the use of Ada type declarations. In the flight dynamics environment, the amount of typing is increasing over time. This may indicate that the developers are becoming more comfortable with the strong typing features of Ada and are using its capabilities to a fuller extent.

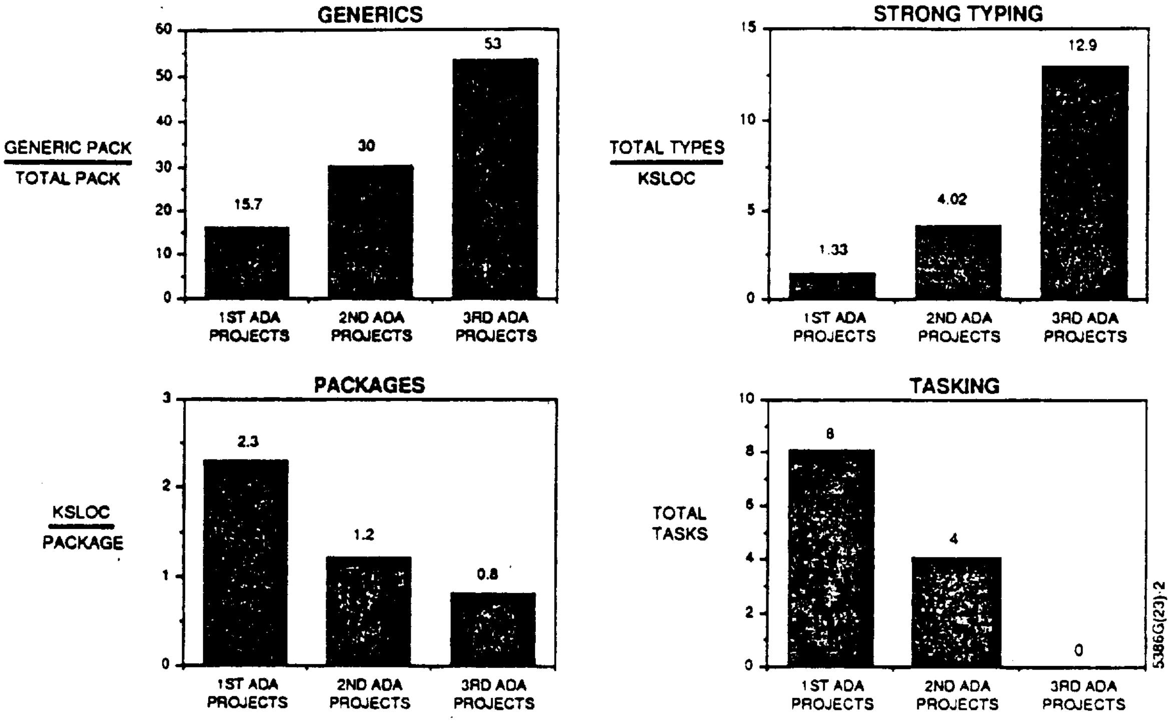


Figure 2. Use of Ada Features

The tasking feature of Ada is used in the flight dynamics environment for the dynamics simulators. Eight tasks were used for GRODY, the first dynamics simulator in Ada. For subsequent dynamics simulators of approximately the same size and functionality as GRODY, design personnel determined that four tasks were sufficient to implement the interactive capability of the system. It is expected that future dynamics simulators will continue to use tasking, however developers are now using tasking more judiciously. The third Ada projects are telemetry simulators, which are sequential systems that do not benefit from the feature of the language, and thus do not use Ada tasks.

COST/RELIABILITY/REUSE

Productivity

Discussions on Ada productivity require careful interpretation because so many definitions exist for software size measures in Ada. Depending on the measurement used, software developers using Ada can be shown to be either as productive as or not as productive as software developers using FORTRAN. Using the total lines of delivered code as a measure, the Ada projects studied show an improving productivity over time, and they show a productivity greater than FORTRAN (Figure 3). However, considering only code statements (semicolons for Ada or excluding all comments and continued lines of code in FORTRAN), the results are different. An increasing productivity trend remains in the Ada projects over time, but the Ada projects have not yet achieved the productivity level of FORTRAN projects.

In the flight dynamics environment, many software components are reused on FORTRAN projects. Because no Ada components existed previously, the first Ada projects were, in fact, developing a greater percentage of their delivered code than the typical FORTRAN project. A past study by the SEL and experience with FORTRAN projects indicated that reused code costs approximately 20 percent of the cost of new code [2]. Using this estimate, reusability can be factored into software size by estimating the amount of developed code. Developed code is calculated as the amount of new code plus 20 percent of the reused code. With software reusability factored in, the productivity for developed statements on Ada projects is approximately the same as that for FORTRAN projects (Figure 4).

Many objections are often made when computing productivity in terms of lines of code: it is affected by style, there are many ways to code the same function, etc. The most intuitive measure to use in computing productivity is cost per "function." Some attempts have been made within the SEL to compare the functionality of projects being compared (e.g., GRODY versus GROSS), and data seem to indicate that comparing "statements" is the closest measure to comparing functionality.

The trends in Ada productivity are very positive in that the overall cost of producing an Ada

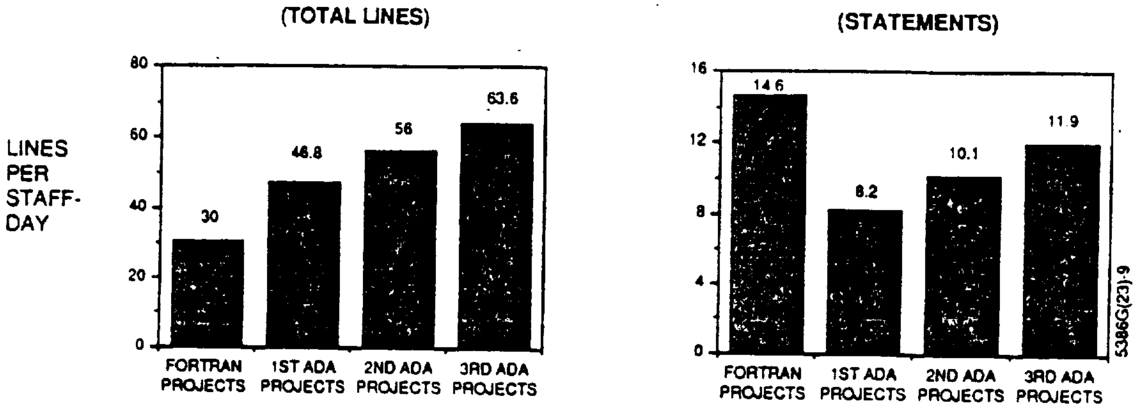


Figure 3. Ada Cost/Productivity of Delivered Code

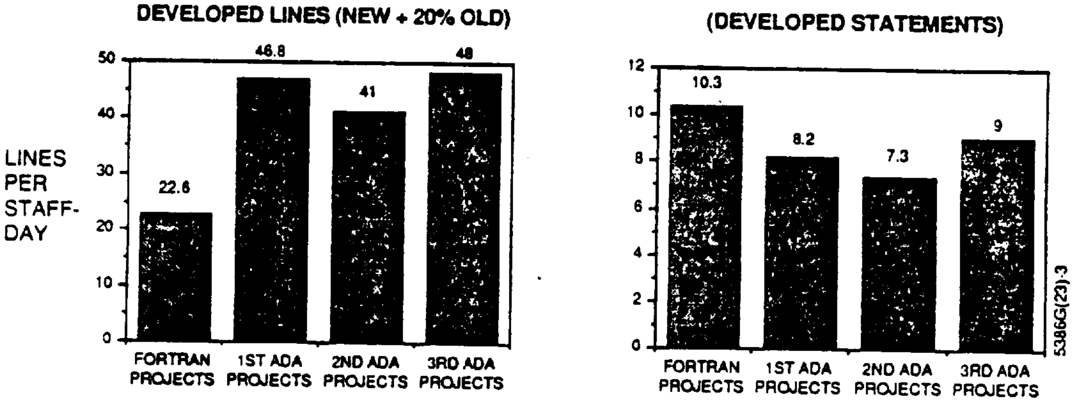


Figure 4. Ada Cost/Productivity of Developed Code

system has quickly become equivalent to that of producing a FORTRAN system. The flight dynamics FORTRAN environment is stable, mature, and built on a long and extended legacy of experience. Although the first Ada project required 30 percent more staff-hours to complete than a similar FORTRAN project, this overhead included effort to develop new practices and processes and to learn a new environment. With experience, the environment is becoming more stable and productivity is increasing.

Reliability

As with productivity, the many ways of measuring software size affect the results of reliability studies. For example, will the error rate normalized by the total system size or by the number of language statements give the most accurate reading of the reliability of Ada software compared to FORTRAN? In the flight dynamics environment, changes to the software made after unit testing when the software is placed under configuration control are formally reported on change report forms. The developer must supply the reason for the change (e.g., error, requirement change) and, if the change is due to an error, the source and type of error.

In a very mature FORTRAN development environment, the GROSS project reported 3.4 errors per thousand lines of source code (KSLOC) in the system. As Table 4 shows, all the Ada projects achieved an error rate lower than the rate on the FORTRAN project. In addition, the error rate on the Ada projects shows a decreasing trend over time.

When the error rate is normalized by the number of language statements, the first and second Ada projects show a slightly higher error rate than the FORTRAN project. However, the error rate again shows a decreasing trend over time. On the third Ada projects, the errors have decreased to a rate as good as, if not better than, the error rate on the FORTRAN project. It is still too early to observe a definite difference from the FORTRAN rates; however, the reliability of the Ada projects appears at least as good as that of FORTRAN projects and is improving with each Ada project.

Classes of Errors

Errors reported are classified according to source and type of error. Sources of errors can be requirements, functional specifications, design, code, or previous changes. Types of errors

Table 4. Ada and Error/Change Rate

	1ST ADA PROJECTS	2ND ADA PROJECTS**		3RD ADA PROJECTS**		FORTRAN PROJECTS
ERRORS/KSLOC *	1.8	1.7	1.4	1.0	1.0	3.4
ERRORS/K STMTS	10.2	9.4	7.5	5.3	5.6	6.9

5386G(23)-8

* SLOC = TOTAL LINES (INCLUDES COMMENTS/REUSED)
** FIGURES BASED ON ESTIMATES

are initialization, logic/control structure, internal interface, external interface, data value or structure, and computational.

On a typical FORTRAN project in the flight dynamics environment, design errors amount to only 3 percent of the total errors on the project (Figure 5). For the first and second Ada projects, 25 to 35 percent of all errors were classified as design errors, a substantial increase. For the third Ada project, however, design errors dropped significantly and are estimated to be approximately 7 percent. This rate is close to that experienced on FORTRAN projects and clearly shows a maturation process with growing expertise in Ada.

The literature on Ada reports that the use of Ada should help reduce the number of interface errors in the software [4]. Although the compiler will catch most calling parameter consistency errors, interface errors can also include errors that will not be detected until run time. Typically, these are errors in string parameters or subtypes with different constraints and errors in calling parameters due to the need for additional or different types of parameters. Using guidelines and examples in the data collection document [13], the errors are classified by the developer reporting and correcting the error.

In the flight dynamics FORTRAN environment, about one-third of all errors on a project are interface errors. On the first and second Ada projects, the percentage of interface errors was not greatly reduced (Figure 5), with approximately one-fourth of the errors being interface errors. With current projects, however, the SEL is now observing a significant change: interface errors are decreasing.

In the SEL, "errors due to a previous change" categorizes errors caused by a previous modification to the software. The first Ada projects showed a large jump in the percentage of these errors compared to projects using FORTRAN (Figure 5). However, all subsequent Ada projects show a rate for these errors that is very similar to the FORTRAN rate. This initial jump in the error rate can probably be attributed to inexperience with Ada, inexperience with Ada design methodologies, and a nested software architecture that made the software much more complex. Again, the error profile is evolving with the maturity of the Ada environment.

Software Reuse

Throughout the years of developing similar systems in FORTRAN in the flight dynamics environment, the average level of software reuse has been between 15 and 20 percent [10, 20]. FORTRAN projects that attained a software reuse rate of 35 percent or higher are rare. After the first Ada projects and with only 5 to 6 years of maturing in the environment, Ada projects have now achieved a software reuse rate of over 25 percent, already greater than the typical FORTRAN project. The UARSTELS project is expected to consist of more than 35 percent reused code. This trend of increasing software reuse is very promising.

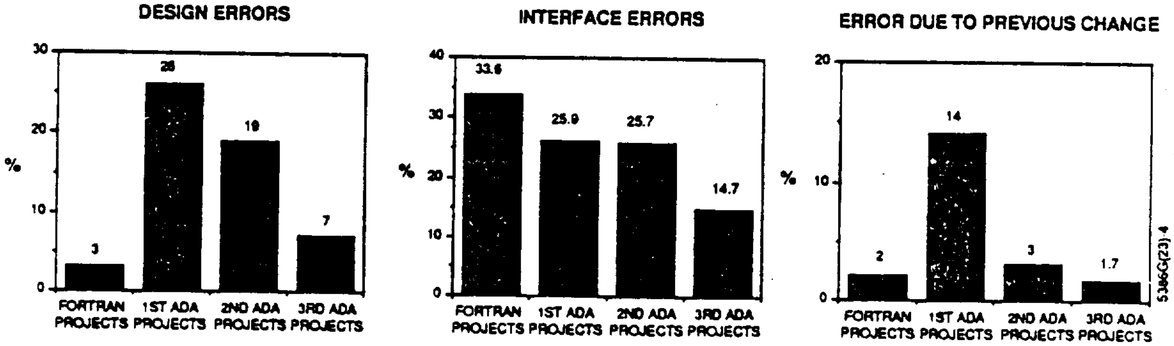


Figure 5. Error Characteristics

CONCLUSIONS/OBSERVATIONS/COMPARISONS

Many aspects of software development with Ada have evolved as our Ada development environment has matured and our personnel have become more experienced in the use of Ada. The SEL has seen differences in the areas of cost, reliability, reuse, size, and use of Ada features.

A first-time Ada project can be expected to cost about 30 percent more than an equivalent FORTRAN project. However, the SEL has observed significant improvements over time as a development environment progresses to second and third uses of Ada.

The reliability of Ada projects is initially similar to that expected in a mature FORTRAN environment. With time, however, improvements can be expected as experience with the language increases.

Reuse is one of the most promising aspects of Ada. The proportion of reusable Ada software on our Ada projects exceeds the proportion of reusable FORTRAN software on our FORTRAN projects. This result was noted fairly early in our Ada projects, and our experience shows an increasing trend over time.

The size of an Ada system will be larger than a similar system in FORTRAN when considering SLOC. Size measurements can be misleading because different measurements reveal different results. Ratios of Ada to FORTRAN range from 3 to 1 for total physical lines to 1 to 1 for statements.

The use of Ada features definitely evolves with experience. As more experience is gained, some Ada features may be found to be inappropriate for specific applications. However, the lessons learned on an earlier project play an invaluable part in the success of later projects.

REFERENCES

1. Agresti, W., et al. 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., McGarry, F., et al. Manager's Handbook for Software Development. Software Engineering Laboratory, SEL-84-001, April 1984.

3. Basili, V. Quantitative Evaluation of Software Methodology. University of Maryland, Technical Report TR-1519, 1985.

4. Basili, V., et al. Use of Ada for FAA's Advanced Automation System (AAS). The MITRE Corporation, April 1987.

5. Booch, G. Software Engineering With Ada. Menlo Park, CA: Benjamin/Cummings Publishing Company, 1983.

6. Booch, G. Software Components With Ada -- Structures, Tools, and Subsystems. Menlo Park, CA: Benjamin/Cummings Publishing Company, 1987.

7. Card, D., McGarry, F., Page, G., et al. The Software Engineering Laboratory. Software Engineering Laboratory, SEL-81-104, February 1982.

8. Cherry, G. Advanced Software Engineering With Ada--Process Abstraction Method for Embedded Large Applications. Language Automation Associates, Reston, VA, 1985.

9. Doubleday, D. ASAP: An Ada Static Source Code Analyzer Program. University of Maryland, Department of Computer Science, Technical Report 1895, August 1987.

10. Esker, L. Software Reuse Profile Study of Recent FORTRAN Projects in the Flight Dynamics Area. Computer Sciences Corporation, IM-88/083(59 253), January 1989.

11. Firesmith, D. Mixing apples and oranges: Or what is an Ada line of code anyway? Ada Letters, September/October 1988.

12. Godfrey, S., and Brophy, C. Assessing the Ada Design Process and Its Implications: A Case Study. Software Engineering Laboratory, SEL-87-004, July 1987.

13. Heller, G. Data Collection Procedures for the Rehosted SEL Database. Software Engineering Laboratory, SEL-87-008, October 1987.

14. McGarry, F., and Nelson, R. An Experiment With Ada--The GRO Dynamics Simulator. NASA/GSFC, April 1985.

15. McGarry, F., Page, G., et al. Recommended Approach to Software Development. Software Engineering Laboratory, SEL-81-205, April 1983.

16. Murphy, R., and Stark, M. Ada Training Evaluation and Recommendations. Software Engineering Laboratory, SEL-85-002, October 1985.

17. Quimby, K., and Esker, L. Evolution of Ada Technology in the Flight Dynamics Area: Design Phase Analysis. Software Engineering Laboratory, SEL-88-003, 1988.

18. Reifer, D. Ada's impact: A quantitative assessment. Proceedings of the 1987 ACM SIGAda International Conference. December 1987.

19. Seidewitz, E., and Stark, M. General Object-Oriented Software Development. Software Engineering Laboratory, SEL-86-002, August 1986.

20. Solomon, D., and Agresti, W. Profile of Software Reuse in the Flight Dynamics Environment (Preliminary). Computer Sciences Corporation, CSC/TM-87/6062, 1987.

USING ADA TO MAXIMIZE VERBATIM SOFTWARE REUSE

Michael E. Stark, NASA/Goddard Space Flight Center
Eric W. Booth, Computer Sciences Corporation

1. INTRODUCTION

The reuse of software holds the promise of increased productivity and reliability. Experience has shown that making even the slightest change to a "reused" piece of software can result in costly, unpredictable errors [Solomon, 1987]. For this reason, the Flight Dynamics Division (FDD) of Goddard Space Flight Center (GSFC) is concentrating effort on developing "verbatim" reusable software components with Ada, where *verbatim* means that no changes whatever are made to the component.

This paper presents the lessons learned on several simulator projects in the FDD environment that exploit features of the Ada language, such as packages and generics, to achieve verbatim reuse. These simulators are divided into two separate, but related, problem domains. A *dynamics simulator* is used by the FDD mathematical analysts to verify the attitude control laws that a spacecraft builder has developed. A *telemetry simulator* generates test data sets for other mission support software. FDD began using Ada in 1985 with the development of the Gamma Ray Observatory attitude dynamics simulator (GRODY). Since that time, six additional simulator projects have been started. With each successive project, a concentrated effort is made to use the lessons learned from previous Ada simulator development projects.

This paper focuses on the concepts used in the projects that have had the most impact on verbatim software reuse in the FDD environment: GRODY, the Upper Atmosphere Research Satellite Telemetry Simulator (UARSTELS), and the Generic Dynamics and Telemetry Simulator (GENSIM). This paper defines underlying design principles, discusses how Ada features support these principles for reuse in the small, and shows how these principles are used to achieve reuse in the large. Finally, this paper presents supporting data from current reusability results.

The FDD has been using a modified version of the General Object-Oriented Development (GOOD) methodology [Seidewitz, 1986; Stark, 1987; Seidewitz, 1988] to develop its Ada software. Three concepts that play a role in GOOD enhance verbatim reuse: abstraction, inheritance, and problem-specific architectures. These concepts support the reuse of successively larger components within successively narrower domains. The next two sections describe how these concepts are applied to simulator projects in the FDD. Abstraction supports "reuse in the small," and inheritance and problem-specific architectures support "reuse in the large." The current practice is to support reuse in the small through component libraries, as is done with a collection of components by Grady Booch [Booch, 1987] and EVB's Generic, Reusable Ada Components for Engineering (GRACE) [Berard, 1989]. The UARSTELS and GENSIM projects are cited to describe how reuse on a large scale is accomplished and to demonstrate the potential in cost savings and/or the ability to solve more complex problems.

2. REUSE IN THE SMALL: USE OF ADA GENERICS

This section presents design and implementation guidelines for using Ada generics. It shows how the design principles mentioned in the previous section are implemented using Ada.

Designing individual generic components is understood to the point that such components are commercially available. The Booch taxonomy of structures creates a family of components that satisfy the same abstraction within the context of different problems, for example, sequential versus concurrent applications.

The design process becomes more interesting when a hierarchy of generic components is needed. This is the case when designing generic subsystems with multiple levels of abstraction. This leveling of a subsystem can be embodied in the Ada code by using generic units and instantiations in the following three ways:

1. Library unit instantiations

2. Nested instantiations

3. Nested generic definitions

This section will define each of these approaches and provide examples to demonstrate their application. Implications for reusability and lessons learned are included for each approach.

Library Unit Instantiations

The first approach is to create an instantiation of a generic that is a library unit. This approach is appealing for practical reasons. The potentially broad scope provided by library unit instantiations may be necessary for Ada compilers that do not implement code sharing[1] [Ganapathi, 1989]. Most implementations currently do not support code sharing. Instead, each instantiation creates a complete object code copy of the generic template. The system-wide scope provided by library units often makes only one copy (instance) necessary.

To describe the first method of library unit instantiations, assume that the generic package A depends on a set of subprograms, P, provided by the generic package B. Instantiating B as a library unit creates a copy of this generic package called Instance_B. The generic package A may then be instantiated by using the subprograms provided by Instance_B as actual parameters. This allows the generic packages A and B to be designed and implemented having no external dependencies, which makes reuse simpler. This approach is depicted in the design diagram[2] shown in Figure 1.

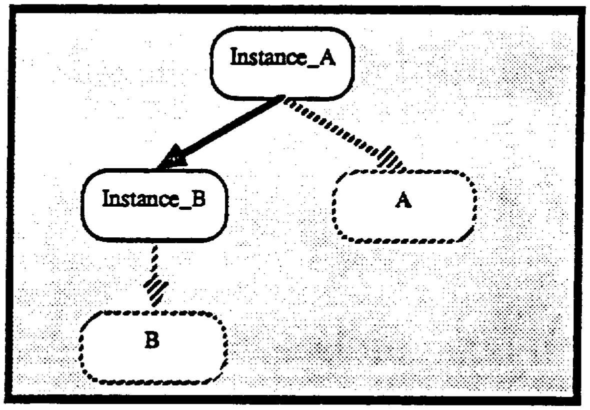


Figure 1.

As previously mentioned, instantiating generics as library units has the advantage that instantiations of A and B may be imported (named in a **with** clause) by any compilation unit in a system.

When full visibility is desired, this technique works well. However, if B were an abstract state machine[3] and the design required that B should be visible only to A, this potentially broad visibility of B is undesirable. It would be far better to use the language to enforce that design decision.

Another drawback of library unit instantiations is that as generic components become more complex, they require a longer list of more problem-specific generic formal parameters. Each instantiation of the generic becomes long and complex. The instantiations can be made simpler by specifying defaults for formal subprograms using the notation "with procedure P1 is <>." Then (assuming in this case that A's only formal parameter is P1), the instance of A can be written as shown in Figure 2.

```
generic
  with procedure P1 is <>;
package A is
  ...
end A;

generic
  ...
package B is
  procedure P1;
  ...
end B;

with A,Instance_B;
use Instance_B;
package Instance_A is new A;
```

Figure 2.

This technique works well when numerous simple functions are used as formal parameters. This use of Ada *simulates* the search of an object code library to resolve external references. Figure 3, part 2 presents an example of mathematical packages being used to instantiate the package of flight dynamics abstract data types[4] shown in part 1. Care should be used with this technique, since the use clause does more than simply make objects and operations directly visible. There are visibility and precedence rules of the language that will affect what defaults will be used [Mendal, 1988].

```
generic
  type REAL is digits <>;
  type RADIANS is digits <>;
  type VECTOR is array ( INTEGER range <>) of REAL;
  type MATRIX is array (  INTEGER range <>,
                          INTEGER range <>) of REAL;
  with function sin ( Angle : in RADIANS)
      return REAL is <>;
  with function cos ( Angle : in RADIANS)
      return REAL is <>;
  with function Floor ( Item: in REAL)
      return REAL is <>;
package Generic_Attitude_Types is
  ...
end Generic_Attitude_Types;
```

Figure 3 (1 of 2).

[1] Code sharing is a technique that allows each instance of a generic to share the same object code. The result is usually a smaller object code size and slower execution speed for the system.

[2] The notation for the design diagrams uses rounded-corner rectangles to represent packages, solid arrows to represent dependencies, broken arrows to represent instantiations, and broken package symbols to represent a generic unit.

[3] A package that is an abstract state machine is a package that maintains state information in the package body [Booch, 1983].

[4] A package that is an abstract data type exports objects, types, and operations but does not maintain state information in the body [Booch, 1983].

```
with  Single_Math_Functions,
      Single_Linear_Algebra;
use   Single_Math_Functions,
      Single_Linear_Algebra;
with  Math_Types,
      Generic_Attitude_Types;

package Single_Attitude_Types is
  new Generic_Attitude_Types (
    REAL      => Math_Types.SINGLE,
    RADIANS   => Single_Math_Functions.RADIANS,
    VECTOR    => Single_Linear_Algebra.VECTOR,
    MATRIX    => Single_Linear_Algebra.MATRIX );
```

Figure 3 (2 of 2).

A second method that uses library unit instances is to instantiate B as a library unit, which is then **with**-ed into the body of generic A (Figure 4). Since the procedures do not need to be passed as actual parameters, this option allows A to have a shorter formal parameter list. However, method often requires the use of common types for A and B. For example, for a flight dynamics application, the generic package A would have to be coupled to the same floating-point types as the generic or instance of package B. As long as this sort of coupling is relatively simple, it can be managed (in the simulator case by having a single package containing the basic floating-point types).

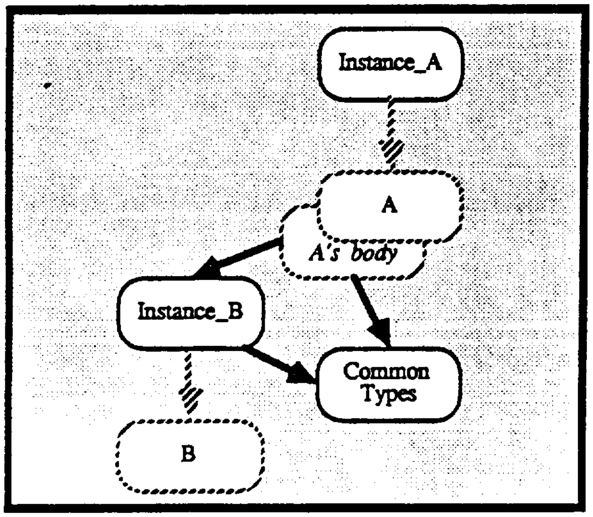


Figure 4.

The disadvantage of this approach is the use of a common types package to implicitly couple A and B. This defeats software engineering principles of information hiding and data abstraction. The following section describes how to exploit these software engineering principles by nesting generic instantiations.

Nested Instantiations

Continuing the same example, the generic package B may be instantiated in the body of generic A (Figure 5) using the generic formals of A. This option is ideal for abstraction and information hiding because it can be extended to a series of nested instantiations. Objects, types, and operations from B may be used as building blocks and specialized to raise the level of abstraction [Stark, 1987]. This is sometimes referred to as re-exported. Importing the package B in this manner allows objects, types, and operations to be hidden in the body of the generic package A but still used to instantiate B.

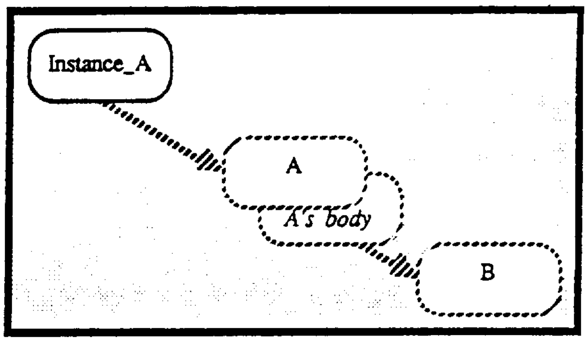


Figure 5.

Using nested instantiations is an appealing technique for software engineering reasons. The limited scope provided by a package boundary increases information hiding and protection. The amount of information that the user of the outer package A needs to know may be limited to the package specification of A. The fact that an instantiation of a lower level generic is used to implement the package is irrelevant to the user. Ramifications of future changes made during the maintenance phase will be much more limited in scope than the library unit instantiation approach.

The disadvantage of using nesting instantiations is the advantage of using library unit instantiations. That is, if the Ada compiler being used does not support code sharing and an instance of a particular generic is necessary in several locations, nesting will result in multiple object code copies. For example, the executable size of GRODY is less than 2 megabytes (Mb). This simulator does not have multiple copies of nested instantiations. UARSTELS, on the other hand, makes extensive use of nested instantiations, which resulted in an executable size of 6 Mb.

On the surface, the implication of these findings seems to suggest using library unit instantiations exclusively. The actual implication, however, is that one should use nested instantiations only when a few copies are necessary. One way to minimize the number of copies is to implement the generic package as an abstract data type rather than an abstract state machine. This approach may be possible when multiple instances of the abstraction use the same types and subprograms as actual parameters but use different objects (Ada allows objects to be passed at run time, while types and subprograms must be passed at instantiation time). This approach allows copies to be created with object declarations instead of generic instantiations; it also has the benefit of increased flexibility, since objects may be declared static at compilation time or created (dynamically) at run time.

When the correct design calls for multiple instances of an abstract state machine the long-term solution is to acquire a compiler that supports code sharing.

Nested Generic Definitions

Nested generic definitions is the third design approach described here. This technique is appealing when the problem calls for a high degree of coupling between generics.

Changing the previous examples, if instantiations of generic package A and generic package B will have common types and subprograms as actual parameters, the following architectures are possible:

1. Make the types and subprograms visible to the generic templates via **with** clauses.
2. Make each generic package a library unit and duplicate the generic formal parameters in the generic part of each.
3. Nest the generic definitions within the specification of another generic package, C. The generic part of C contains the common formal parameters (Figure 6).

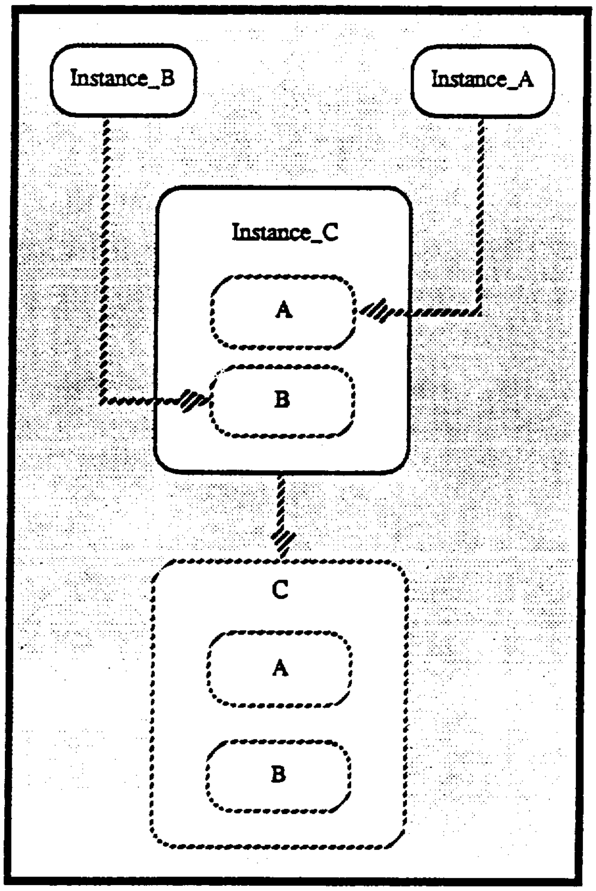


Figure 6.

Although the first option works, it suffers from a high degree of coupling. Future instances of either A or B will always be using the same common types and subprograms. This inflexibility results in limiting the verbatim reusability.

The second option is a large improvement over the first. Future users of packages A and B may now supply their own types and subprograms for the generic formal parameters. However, this architecture becomes tedious and error prone when the common types and subprograms are long and complex. Since generic instantiation becomes a large part of the effort when maximizing verbatim reuse, it is desirable to simplify this activity.

The third option accomplishes the same goals as the second option but with less duplication. This option is useful when the number of nested generics or the number of common generic formal parameters becomes large. It is less error prone because the common actual parameters are supplied only once. The maintenance phase also benefits from the single location of common actual parameters.

Figure 7 shows an example of nested generic definitions from UARSTELS. The generic function FSS Digitize is declared within the generic package Generic Sensor Digitization. The generic formal parameters of the composite package (generic sensor digitization), as well as the generic formal parameters of FSS Digitize, are referenced in the body of FSS Digitize.

```
generic
  type REAL is digits <>;
  type COUNTS is range <>;
  with procedure Log_Error
       (Message : in String) is Text_IO.Put_Line;
package Generic_Sensor_Digitization is

  function Linear_Digitize
    ( Parameter : in REAL;
      Dias : in REAL;
      Scale: in REAL ) return COUNTS;

  generic
    type COUNTER is range <>;
    with function tan (X: REAL) return REAL is <>;
    with function sin (X: REAL) return REAL is <>;
    with function cos (X: REAL) return REAL is <>;
    with function Floor (X: REAL) return REAL is <>;
  function FSS_Digitize
    ( Angle : in REAL;
      Coefficient : in REAL;
      Tolerance : in REAL;
      Maximum_Number_of_Iterations : in COUNTER )
    return COUNTS;
    ...
end Generic_Sensor_Digitization;
```

Figure 7.

Finally, a practical benefit accrues from using the nested generic definition approach. Most compilers, as previously pointed out, do not support code sharing; instead, they expand the generic template at instantiation time. The advantage for the designer needing only 1 generic from a package containing 10 nested generic definitions is that only the object code for the 1 instantiation will be generated.

3. REUSE IN THE LARGE: PROJECT-SCALE REUSE

This section presents the details of the. two projects in FDD that have had the most impact on verbatim reuse: UARSTELS and GENSIM. For both projects, an overview is presented giving the background information, the goals, and the motivating factors involved during development. Each system's architecture is discussed using the concepts and notation from the previous section of this paper. Finally, the lessons learned from each project are discussed with their implications for future development efforts.

UARSTELS Overview

The UARSTELS project was started in February 1988 and was the fourth Ada simulator initiated at FDD. Previous simulators included the GRODY experiment, the Geostationary Operational Environmental Satellite-I (GOES-I) dynamics simulator (GOADA), and the GOES-I telemetry simulator (GOESIM). The GOES-I simulators represent the first operational Ada software developed at FDD.

The GRODY design team exploited the feature of nested units, which resulted in increased information hiding (information protection might be a better phrase). The rationale for using information hiding is increased reliability. Higher reliability should, in turn, increase reusability; that is, if a component is very reliable, it is appealing to reuse. This was the basis for the GRODY design.

The lesson learned from this approach was that extensive use of nested packages actually decreased reusability [Quimby, 1988]. In addition, during the coding and testing phases, the development team observed the high compilation overhead incurred by the nested architecture.

Given these lessons from GRODY, the GOADA and GOESIM design teams developed a non-nested architecture with the twin goals of increasing reusability and reducing compilation overhead. Both these goals were met. Individual software components could be picked up by successive projects and reused with slight modifications. The use of library packages, rather than nested packages, kept the compilation costs to a minimum.

One of the lessons learned from the implementation phases of the GOADA and GOESIM projects was that using Ada was not significantly decreasing the level of effort for integration testing. This was unexpected. It was predicted that the integration test phase would require less effort than past FORTRAN integration test phases. Some Ada developments have claimed that system integration took significantly less effort than for similar, previous non-Ada projects [Hudson, 1988].

Analysis by the UARSTELS design team showed that by unnesting GRODY's packages, more objects and types became visible at a high level in the GOADA design. This increased the number of components to integrate at each level.

UARSTELS Architecture

The architecture of UARSTELS was influenced from the start with the knowledge that another, very similar simulator would follow: the Extreme Ultraviolet Explorer (EUVE) telemetry simulator (EUVETELS). A high level of reuse from UARSTELS to EUVETELS was both desired and thought to be possible because of reused functional specifications. However, because the two spacecraft were themselves different, the telemetry simulators would be different. The design for UARSTELS needed to take into account these spacecraft dependencies and parameterize them.

Each design decision made on UARSTELS attempted to satisfy the following requirements:

1. All UARSTELS requirements

2. Some known requirements from previous systems

3. Some possible future mission requirements

The goals of the UARSTELS team were to maximize *verbatim* reuse and allow the compiler to check system integration as much as possible. To achieve this, the design team took a hybrid approach to the system's architecture. Most packages were developed as library packages, rather than nested units; however, these packages were designed as generic units. This allowed the instantiations of these generic packages to be *nested* in successively higher level packages. The level of nesting (or layering) within UARSTELS is comparable to that of GRODY, with the important difference being the use of generic units in UARSTELS. The generic packages may be picked up and reused just as the nongeneric packages in GOADA or GOESIM, with the important difference again being the use of generics. The nested instantiations allow the language to perform integration checks (whether the programmer wants them performed or not) at each compilation just as in GRODY.

A specific example of this nested instantiation approach is the design of each simulated sensor model within UARSTELS. In Figure 8, the Report_Writer, Data_Set, and Plot_File generic packages are library units. As such, they may be picked up and reused independently of each other. In the case of a sensor model, however, each of these objects is necessary. To provide all three of these abstractions to each sensor model, they are instantiated within the specification of the generic package Sensor_Output. The application-specific parameters are provided to the three low-level generics when they are instantiated. The sensor-specific parameters are generic formal parameters to Sensor_Output. Sensor_Output is then instantiated within the body of the generic sensor model package, with the sensor-specific parameters being provided at that time. The spacecraft-specific parameters are generic formal parameters of the generic sensor model package. The instantiation of the generic sensor model package resolves all the formal parameters, resulting in the Fine_Sun_Sensor object.

As a result of this architecture, UARSTELS differs from previous systems in its recompilation time and executable size. The increased use of generic units had a direct effect on the compilation overhead. It takes longer, in CPU time and elapsed time, to recompile UARSTELS than any of the other simulators. In addition, while UARSTELS is significantly smaller in source lines of code and in number of components, it is also significantly larger in executable image size. This is because the Ada compiler used in FDD does not support code sharing. Instead, it expands generic units for each instantiation.

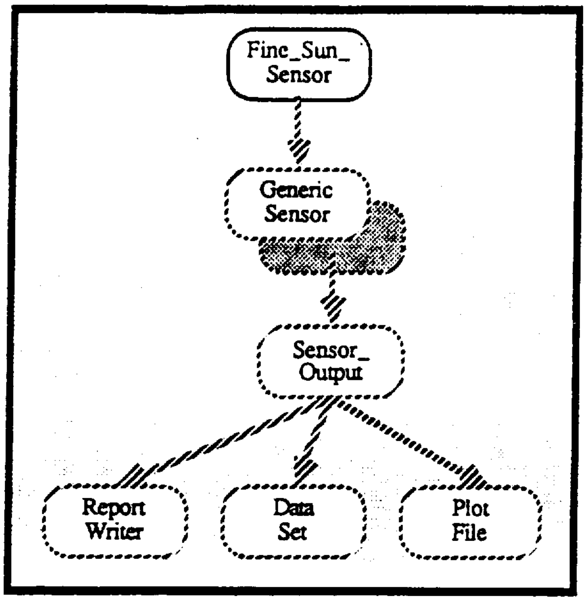


Figure 8.

UARSTELS Lessons Learned

The lessons learned may be summarized as follows:

- Nesting reduces integration testing
- Library units provide reusable software components
- Generic library units provide reusable components and allow information hiding via nested instantiations
- Many Ada implementations incur a large compilation time overhead for the use of nested and generic units
- Many Ada compilers provide a simple implementation of generic units

The nesting feature of Ada must be exploited. The virtue of nesting is information hiding (protection) and higher reliability; this is one of the promises of using Ada. However, like all of Ada's promises, higher reliability does not happen automatically. It must be engineered.

The strong typing feature of Ada must also be exploited. The definition of distinct types that are relevant to the application domain, such as flight dynamics, needs to be engineered. Ada can help eliminate the common mistakes (i.e., mixing radians and degrees or meters and kilometers), but this will not occur automatically either. Through the use of strongly typed objects, reliability can be further improved and integration testing can be further automated.

Strong typing encourages and, in most cases, forces the use of nesting. Operations on private types must be defined within the same scope (package) that defines the type. Since the internal structure of that type is not visible outside this scope, all operations must be defined within the scope or the operations must be imported with a generic instantiation.

Circumventing nesting, strong typing, and generics in order to minimize compilation time is a short-term fix with the long-term ramification of decreased reliability and reusability. If the compilation overhead is unacceptable, then alternative Ada development environments will be required.

GENSIM Overview

The GENSIM project was started in 1986 by a group studying ways to increase the reuse of simulator software and the possibility of integrating the dynamics and telemetry simulation capabilities. This group consisted of both software developers and mathematical analysts, all of whom had simulator project experience. At the same time, reuse studies in the Software Engineering Laboratory (SEL) [Solomon, 1987] showed that reusing code without modification (verbatim reuse) yields a tremendous reduction in development cost. One of the early products of the GENSIM effort was a study that estimated that costs of software development for a dynamics simulator could be cut in half by creating verbatim reusable components.

The GENSIM team believed that the best approach to maximizing verbatim reuse was to reuse products from all phases of the software engineering life cycle. This belief was based on developer experience, rather than any formal software engineering theory. The simulator problem was divided into "modules," each of which models an entity in the problem domain. The products associated with each module include a specification, design documentation, code, and test cases; each follows project-wide standards. A module specification consists of a complete definition of the inputs and outputs needed, the algorithms to be implemented to model the entity, and documentation of the mathematical analysis and assumptions underlying the algorithm. A module design is built according to standard protocols for initialization, computations, and the passing of parameters between modules. These protocols allow the individual modules to be configured into a standard simulator architecture.

To determine the feasibility of a generic simulator, a prototype is being developed and applied to a simplified mission. After the prototype demonstrates the ability to configure a dynamics simulator for different missions, a full set of components and modules will be developed.

GENSIM Architecture

This subsection first describes the GENSIM dynamics simulator architecture, then discusses design issues for both modules and standard subsystems. The next subsection discusses lessons learned from the initial GENSIM design that can be applied to the final version of the system.

Figure 9 shows the standard dynamics simulator architecture. The reusable modules are parts of the spacecraft, hardware, and environment models (SHEM) subsystem. Typical SHEM modules include Sun sensor modeling and geomagnetic field modeling. The spacecraft control subsystem is always mission dependent because a dynamics simulator is intended to test attitude control algorithms for a specific satellite. The only reusable parts in this subsystem are a simulated ground command uplink interface and a module that computes estimation and control errors. The user interface has the obvious capabilities of reading and editing input parameters and producing reports and plots from analysis results. The case interface subsystem is responsible for maintaining simulation cases, including analysis results data, simulation input parameters, and suspended simulation cases. The case interface also standardizes communications between the SHEM modules and the user interface. The simulation executive subsystem has two major purposes: to manage requests to control the simulator and to control the sequencing and timing of module execution. The last subsystem is a utilities subsystem, which consists of several generic units and a set of standard, interrelated instantiations, as is described in Section 2. For example, the instantiation of a generic linear algebra package requires a square root function, which is provided by the instantiation of a general mathematical functions package.

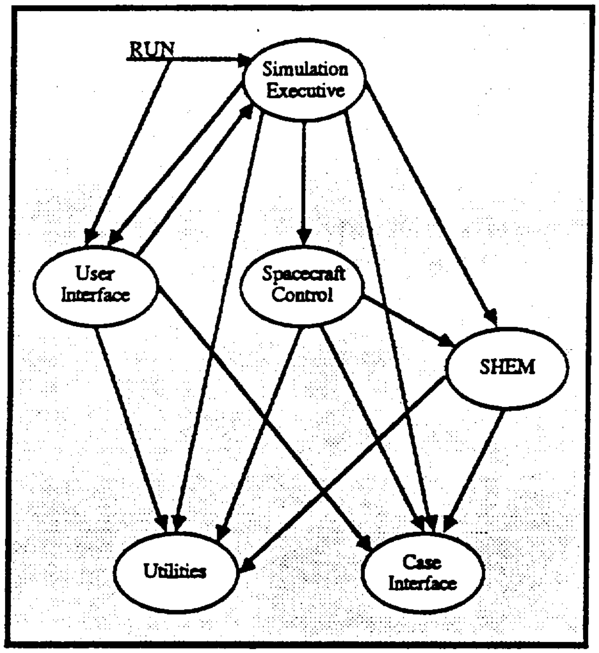


Figure 9.

Figure 10 shows the implementation for a typical SHEM module (Module K). The package Module_K performs the actual modeling. It can be initialized and invoked from the simulation executive, and communicates with other SHEM modules through procedure and function calls. The generic module database and generic module results objects are generic packages that implement the standard communications between a module and the other subsystems. These generics are instantiated with types and values from the Module_K_Types package. These instantiations are called on by the Module_K package when the model itself is being initialized or activated, and they are called on by case interface components whenever parameter or results data is required by some other subsystem. Using this standard approach allows a different set of SHEM modules to be used for each mission.

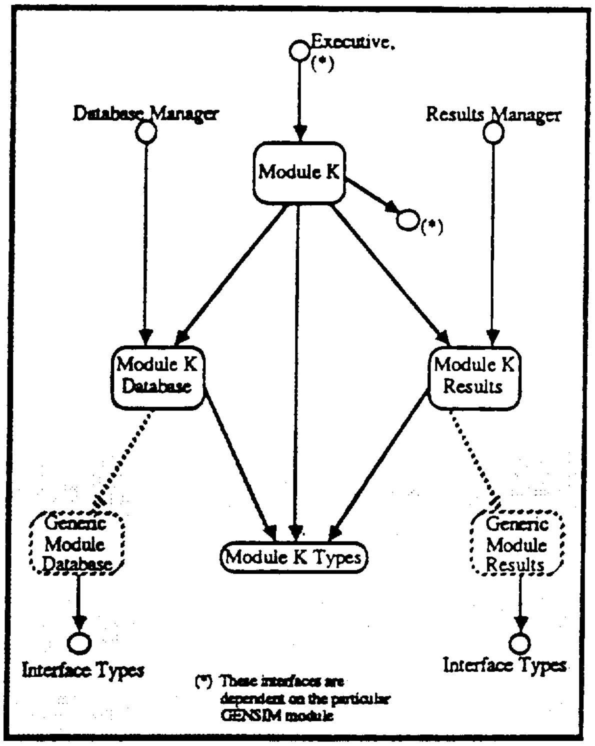


Figure 10.

The GENSIM design just described is a conservative extension of existing simulator designs. In general, the dynamics simulation capability remains the same, but the design has been reworked to be more object oriented. For example, the case interface subsystem was added to treat the concept of simulation case as an object, rather than to distribute those capabilities between the user interface, the SHEM, and the utilities. The utilities subsystem was also changed from one gigantic generic package to several smaller, independent generic units. The main effort in GENSIM has been directed toward generalizing the design to make the components reconfigurable, rather than adding new capabilities.

The user interface, case interface, and the simulation executive subsystems must be implemented as generic subsystems that can be parameterized by the selection of modules for a given mission. The case interface subsystem can be used to demonstrate how a generic subsystem is designed. The discussion of individual modules in the previous paragraph shows how the generic module database and module results packages are used by modules. The case interface subsystem must access these same instances to communicate with the user and to maintain simulation cases.

Figure 11 shows the design for the generic case interface subsystem. The case manager object is responsible for managing simulation cases as a unit; and the ground command interface, parameter interface, and results interface objects manage components (such as analysis results) of a simulation case. The parameter interface and the results interface also manage the communications with SHEM modules. Figure 11 shows that all these packages are interrelated; however, they are used one at a time. For example, a procedure that edits ground commands would use the ground command interface but not the other objects.

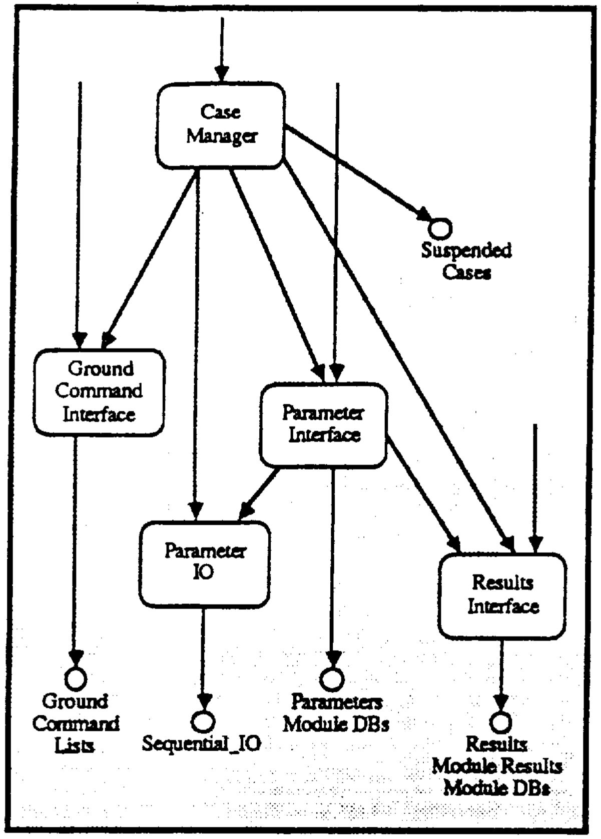


Figure 11.

If these objects were not being implemented as generics, each object in this subsystem would be implemented as a library package, which could be imported independently. With a generic subsystem, each of these packages must be parameterized, and many of the generic formal parameters are common to more than one package. If each object were implemented as a separate generic package, there would be multiple definitions of the same formal parameters with all the maintenance problems such a redundant structure entails. Since the packages in this subsystem are coupled anyway, the case interface subsystem is implemented using the nested generic definitions technique (described in Section 2 of this paper), which allows the common parameters to be placed in the generic part of the composite package.

In the GENSIM design, even the parameters that apply to only one package were placed in the composite package. When this is done, the nested packages are no longer generic. This approach reduces any possible confusion between generic packages and their instances by allowing the user to instantiate the entire subsystem. Then the nested packages can be used without having to instantiate more generics. The cost is that the nested packages are now more highly coupled. In the case interface, this increased coupling is justified because all the coupled packages are part of the abstraction "simulation case." The utilities subsystem consists of independent generic packages, where the coupling is introduced between *instances* of these generics. This approach allows the generic packages to be used outside the context of dynamics simulators. There is no corresponding need to use individual components of case interface outside the context of dynamics simulators because the coupling between the components is defined by the nature of a simulation case. The degree of coupling allowed in the design and implementation of reusable components is one of the key judgments developers must make.

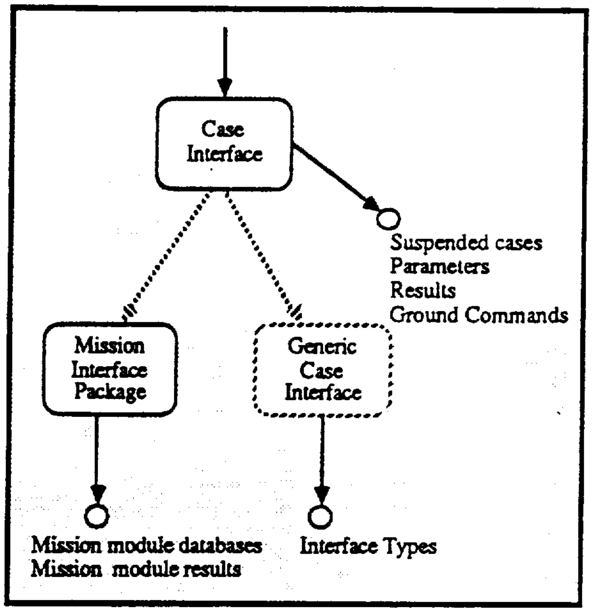


Figure 12.

GENSIM Contributions

The major benefits derived from the GENSIM project are in the categories of (1) gaining experience in the use of advanced Ada features, (2) getting ideas for improvements in simulator design, and (3) producing the reusable components themselves. This subsection will focus on the first two categories. In the first area, the redefinition of the utilities subsystem as a set of *independent* generic packages tied together as a set of *interrelated* instantiations demonstrated the ability to write generic packages as described in Section 2.

When designing a subsystem this way, the developer must make sure that all the generic formal parameters designed are matched by actual parameters provided by some other package. It is also eases development if the code is written and tested bottom up, so that the lower level instantiations needed to instantiate the senior-level generics are tested and, in turn, can be relied on to support the testing of other objects. If care is taken to design a set of generic packages with consistent naming conventions, defaults can be provided by standard instantiations, allowing the rapid writing of instantiations for testing purposes. When all these conditions are met, the techniques described in Section 2 work very well. The decoupled generics/coupled instantiations technique is the best approach to develop packages that provide the ability to use problem domain abstractions rather than predefined Ada constructs.

GENSIM has also contributed to the use of strong typing by using more private types than previous simulators and by beginning to focus on the decision criteria for their use. The criteria for using private types is that they should add the protection of data integrity. For example, the attitude types package defines the private type COSINE_MATRIX. This type is identical in data definition to any other 3-by-3 matrix but has a set of operations that guarantee that a COSINE_MATRIX always represents a rotation. In the case of GENSIM's orbit data types, a private type does not add any such data protection; the effect is to force a user to use operations provided for the data type in precisely the same way as one would use an assignment statement. When this is the case, the type should be made visible in a package specification.

Possible Improvements to GENSIM

The GENSIM prototyping has been successful in generalizing dynamics simulator designs, but some features inherited from past simulators can be improved on. Currently, simulator module state data types are built from individual scalar objects or arrays of scalar objects. A more object-oriented design would define abstract data types (ADTs) for the problem domain entities and then use these ADTs to define module states. This is particularly true when there are multiple objects of a type, as is the case with spacecraft sensors. As an example, the current design of a fine Sun sensor model defines the simplified module state as follows:

```
package body Fine_Sun_Sensor is
   – N is the number of Fine Sun Sensors used for a mission
   type STATE is record
      Alpha_Angles
          : Double_Linear_Algebra.VECTOR(1..N);
      Beta_Angles
          : Double_Linear_Algebra.VECTOR(1..N);
      Alpha_Limit
          : Double_Linear_Algebra.VECTOR(1..N);
      Beta_Limit
          : Double_Linear_Algebra.VECTOR(1..N);
   end record;

   Module_State : STATE;
   ...
end Fine_Sun_Sensor_Module;
```

A better implementation is as follows:

```
with Fine_Sun_Sensor_ADT;
package body Fine_Sun_Sensor_Module is
   type STATE is array of (1..N)
      of Fine_Sun_Sensor_ADT.FINE_SUN_SENSOR;

   Module_State : STATE;
   ...
end Fine_Sun_Sensor_Module;
```

In the second implementation, Fine_Sun_Sensor_ADT .FINE_SUN_SENSOR is an abstract data type that encapsulates all the necessary attributes for a single sensor and provides both selector and constructor operations. One advantage of using abstract data types is the tighter encapsulation of data. If a change is made to the fine Sun sensor modeling, the scope of the change is then restricted to the body of the abstract data type package rather than affecting the entire module.

Another advantage of using abstract data types in this context is the strict separation of the problem domain object itself and its use within a software system. In the first example, the declaration of Alpha_Limit mixes the problem domain concept of a limited sensor field of view with the fact that N sensors are used for a particular mission. The second implementation makes it clear that N refers to the number of sensors and that the abstract data type encapsulates each sensor's angles and limits.

When the problem domain object (or class) is implemented with a distinct Ada library unit, it is possible to use the object-oriented programming concept of inheritance to create a hierarchy of classes and subclasses. Figure 13 shows how this could work when all the details of a fine Sun sensor model are considered. This inheritance tree, which is implemented using nested instantiations, shows four levels of increasing complexity, starting with the superclass FSS_ADT and creating a chain of subclasses from there. Each of these four generics can be instantiated either as a library unit or nested within a module. Each subclass in the chain tailors its superclass by incorporating the models provided by the respective utility packages. Inherited operations can also be specialized and new operations can be added to a package [Stark, 1987]. For example, a fine Sun sensor engineering model needs to decalibrate simulated data so that calibration algorithms can be tested. Since this decalibration is specific to fine Sun sensors, the operation would be added to the generic FSS engineering model package, with the noise, biases, and misalignments being provided by the generic measurement utilities.

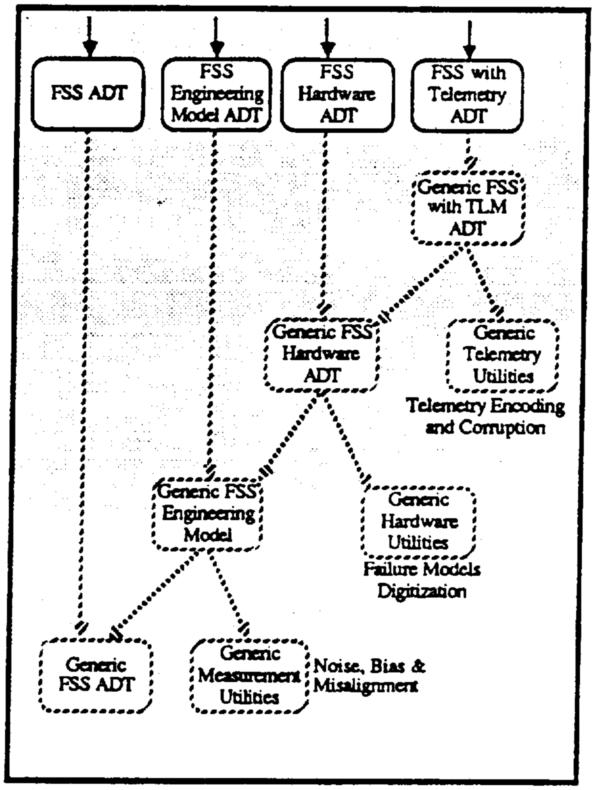

FSS ADT
FSS Engineering Model ADT
FSS Hardware ADT
FSS with Telemetry ADT
Generic FSS with TLM ADT
Generic FSS Hardware ADT
Generic Telemetry Utilities
Telemetry Encoding and Corruption
Generic FSS Engineering Model
Generic Hardware Utilities
Failure Models Digitization
Generic FSS ADT
Generic Measurement Utilities
Noise, Bias & Misalignment

Figure 13.

This approach uses the nested generic instantiations in the same way as UARSTELS. The difference is that all sensor-specific utilities, such as fine Sun sensor decalibration, are part of the sensor abstract state machine, not part of the utility packages.

The use of inheritance allows the selection of an appropriate model for a wider variety of applications. A telemetry simulator would typically pick the telemetry model, a dynamics simulator would pick the hardware model, and error analysis software would use the engineering model. The use of inheritance reduces the redundancy between the different applications, which saves effort in both development and maintenance.

The other area in which the simulator can be made more general is the module types packages. These packages are not defined as generic units, but they contain a mix of mission-specific parameters (such as default initial conditions) and mission independent parameters. The nesting of a generic package within the types package is one possible way to make the system easier to configure. The nested generic would be parameterized by the mission dependencies, with the rest of the types package remaining mission independent. A library instantiation of the nested generic would then be created to define the module's use for a particular mission rather than to extensively modify the types packages. Figure 14 shows how this would work for a fine Sun sensor module. The cost of this is that the other packages in a module now need to import both the types package and the instance of the nested generic, whereas only the types package was needed before. The strict separation of the parameterized part from the consistent part is worth the added complexity.

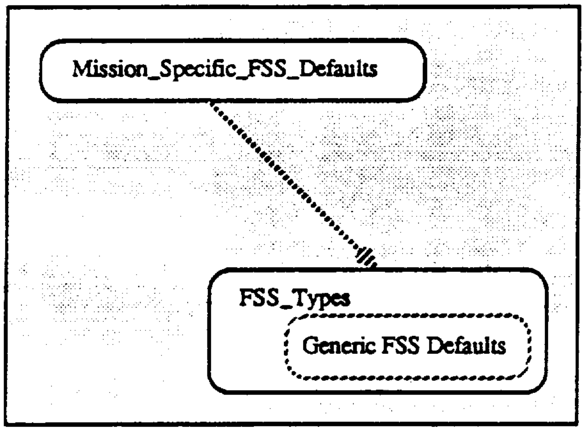


Figure 14.

4. FUTURE DIRECTIONS

The experiences of the UARSTELS and GENSIM projects have demonstrated that the Ada language, and particularly generics, can be used to produce verbatim reusable components that can be fit into more than one architecture. Some other Ada language features need to be examined more closely to see how useful they are for simulation software. There has been a trend to using more strong typing as more experience is gained, but the FDD's Ada software has not gone as far as the Common Ada Missile Packages (CAMP) packages in using distinct types. The CAMP packages use a separate type for each unit of measure, both in generic parameter lists and in nongeneric code [Herr, 1988]. The advantage of using this degree of strong typing is that the compiler is able to catch any dimensionally incorrect computation. The disadvantage is that overloaded operators need to be defined anywhere that two or more different types can be correctly used in a computation. A balance needs to be found between the extremes of CAMP and of using a single floating-point type as the basis of all calculations. To do this, criteria must be defined for the proper use of Ada's typing features. When to use or not to use types, subtypes, derived types, or private types needs clear definition.

This paper's discussion of inheritance focuses on nested generic instantiations as a means of implementing the concept. An alternate approach is to use derived types to simulate inheritance [Perez, 1988]. In the simulators discussed earlier, generics are used for both parameterization and for inheritance. To use derived types for inheritance would require the investigation of the interaction between parameterization and inheritance when different language features are used.

General Concepts for Large-Scale Verbatim Reuse

The lessons learned by the UARSTELS and the GENSIM projects have led us to a general reuse model. This model defines different levels of reuse and which reuse-in-the-small techniques should be applied at which level. Figure 15 shows the leveled reuse model on the left and typical examples on the right. As in most layered models, the higher layers depend on services provided by the lower layers.

	LEVELS	EXAMPLE
ARCHITECTURE LEVELS	• System Templates	Generic_Case Interface
	• Component Templates	Double_Precision _FSS_Module
PROBLEM DOMAIN LEVELS	• Domain Objects and Classes	FSS_ADT
	• Language Extending Objects and Classes	Linear Algebra

Figure 15.

The lowest layer of the model is the *language extension layer*. This layer's purpose is to create a problem-specific language by adding reusable Ada components to the existing capabilities of the Ada language. In the flight dynamics domain, this means defining types and operations for mathematical constructs such as vectors, matrices, and orbits. Applications code can then be developed using the specialized capabilities rather than predefined Ada constructs. This level can be considered the state of the practice for software reuse. The Booch components and the EVB GRACE components are at this level.

The language extension layer itself uses a layered approach. The domain-specific objects are usually built on top of more general objects. The orbit data described above is specific to flight dynamics, but it is represented as two vectors representing position and velocity. When carried into design, an orbit data types package would depend on a more general linear algebra package that exports vector types.

The other important distinction at this level is between entity abstractions and action abstractions. An object with action abstraction is completely described by what it does. A sort package provides operations to sort data; a random number generator generates random numbers. An object with entity abstraction has attributes beyond its set of computations. For example, a queue can be described as a set of homogeneous data that is accessed and modified using a FIFO protocol.

Figure 16 shows how the level of abstraction and level of generality can be used to characterize language extension components. Some typical simulator components are characterized by these two characteristics. The scale from domain specific to general is more continuous than is shown on this diagram. For example, a linear algebra package is specific to the mathematical domain, but it is considered a general-purpose package in the flight dynamics domain. Thus, it would fall somewhere in the middle of the scale. The distinction between entity and action abstraction is more clear cut. If the object has relevant properties beyond the actions it performs, it has entity abstraction. These properties are seen in Ada code as state information that can be retrieved and modified by a package's operations.

	ENTITY ABSTRACTIONS	ACTION ABSTRACTIONS
DOMAIN	Quaternion Orbit	Telemetry Encoding Hardware Failure Models
GENERAL	Stacks, Queues, Vectors, Matrices	Sorts, Integrators, Random Number Generators

Figure 16.

The next level of the model is the *domain level* . This is the level at which the major problem domain entities reside. State-of-the-art reuse libraries such as the CAMP contain components that are reused at this level [Herr, 1988]. Both the domain level and language extension level consist of objects and classes. The difference is that the objects at the domain level *define* the problem domain, and the objects at the language extension level are a means of *expressing* the model for a given problem domain object. The fine Sun sensor abstract data types described in the previous section are all problem domain entities. They are described in terms of vector and matrix algebra, and in terms of standard error sources, telemetry encoding, and sensor failure models. The generic packages for fine Sun sensor data types implement the domain entities *using* capabilities provided at the language extension level.

Figure 13 shows how a mix of domain entities and generic language extensions can be used to build a hierarchy of classes and subclasses. The measurement utilities, hardware utilities, and the telemetry utilities are all language extensions, but they are used in building the problem domain inheritance model.

The next level of reuse is the *component template* level, the level at which generic components are built to fit into a given system architecture. The GENSIM SHEM modules and the UARSTELS sensor models are examples of component templates. Components can be built directly from problem domain objects, or they can provide indirect support. In GENSIM, the SHEM modules will be built around abstract data types, such as those provided for the fine Sun sensors, and the standard module database and module results packages that are instantiated to support the module. In addition to these packages, a standard screen format file is used by the user interface to allow user inputs for each SHEM module. The key distinction is that the component template level defines *all* the components needed to fit a problem domain object into a given system architecture, where the domain level consists of a set of objects that are not constrained by a particular system design, but only by the problem being solved.

The component template level objects are also parameterized, but the emphasis shifts somewhat. The fine Sun sensor abstract data types are parameterized by data types for vectors and matrices and by operations needed to interface with other problem domain objects. The fine Sun sensor module is parameterized by items such as the number of sensors, the default input values, and selections of which inputs a user is allowed to modify. Some values of problem domain parameters may be constrained at this level. Figure 15 gives the example of Double_Precision_FSS_ Module. This module has been constrained to use a particular floating-point data type, but it is still parameterized by the number of sensors and default values.

The top level is the *system template* level. A generic system is a reusable design into which individual components can be fit. Objects at this level are parameterized by the set of components being used in a particular configuration and by any other values that have a system-wide effect. In GENSIM, the parameterization of the generic case interface is related to the particular set of SHEM modules being used. The simulation executive is parameterized both by the set of components being used and by the spacecraft's control modes, which affect how often these components need to be executed.

The two template levels provide the capability of quickly building a software system. Like the language extension and domain levels, the capabilities provided by the lower level are used by the higher one. The key distinction is that the lower two levels give a complete definition of the problem domain, and the upper two levels give a complete definition of a generalized software system architecture. It is important that the problem domain objects be completely independent of particular system architectures. To achieve this, the lower two levels from Figure 15 are grouped as *problem domain* levels and the upper two are grouped as *architecture* levels.

The discussion in this section has focused on design issues, not how Ada should be used to realize these designs. The principles that apply to reuse in the small can be extended to reuse in the large. A developer must still be concerned about a mix of generic packages and their instantiations, and the coupling between components remains a key issue.

In the problem domain level, the only coupling between objects should be defined by the problem. The preferred means of linking objects together is to restrict dependencies to those between library instantiations. One previously mentioned exception to this is the simulation of inheritance. Other relationships can also be simulated through nested generic instantiations or nested generic declarations. An example where nested instantiations are useful is in the case where one object is built from simpler components, as an inertial reference unit (IRU) is built from gyroscopes. The IRU presents a somewhat different interface than a gyroscope, although they are strongly related. The nested generic declarations are useful when alternate models depend on the same objects or types. For example, an orbit types package is parameterized in terms of simple mathematical functions, but they are used by a variety of different models for propagating orbits over time. Rather than nesting instantiations of the orbit types within several different models, the designer can present the models as a set of options that depend on the *same* orbit types.

Importing other library units into generic units is not a problem when used for component templates or system templates. Figure 17 shows where the generic case interface package imports the instantiation of a generic types package. This interface types package provides standard data types for communication between the user, the stored simulation data, and the SHEM modules. The designer should try to minimize this sort of coupling. In GENSIM, only interface types and a common types package are imported into generics in this manner.

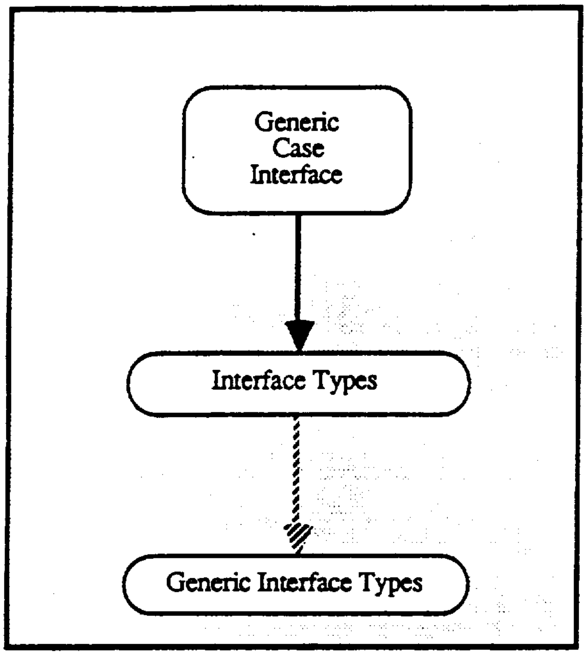


Figure 17.

5. MEASURING THE EFFECT OF LARGE-SCALE VERBATIM SOFTWARE REUSE

This section discusses the impact of the verbatim reuse on project management by describing how costs are affected and the effects of the layered model. A recent SEL study [Solomon, 1987] characterized software components as being new, rebuilt (greater than 25-percent modification), adapted (up to 25-percent modified), and verbatim (unmodified). Expressed as a percentage of the cost of a new component, the costs of the different types of reused components are approximately :

Verbatim	10 % (actually 7.2 %)
Adapted	30 %
Rebuilt	50 %

To make conservative estimates, the 10-percent figure is used for verbatim components, and any nonverbatim component is assumed to be new.

The GENSIM cost study [Mendelsohn, 1988] shows that the current levels of reuse for dynamics simulators save 15 to 20 percent over all-new systems. The study also determined that dynamics simulators have a potential for about 70- to 80-percent verbatim reuse; only the spacecraft control system code is developed from scratch for each mission. These verbatim reuse levels translate to a cost savings of from 60 to 70 percent over an all-new system or at least 50 percent from current systems.

The key to achieving high levels of verbatim reuse is to reuse specifications and design. The analysts who define the requirements for FDD systems developed common mathematical specifications for all systems supporting EUVE and UARS. The current estimate for EUVETELS reuse from UARSTELS is 87 percent, which translates to approximately 80-percent cost savings over a new system. Even the FORTRAN software supporting EUVE has a reuse level of from 60 to 70 percent from UARS, whereas typical levels fall into the 20- to 30-percent range. The increase from reusing the mathematical specifications is much greater than the increase observed as the result of using Ada as the implementation language for simulators [Brechbiel, 1989]. These data confirm the correctness of GENSIM's use of a set of standard mathematical specifications.

In addition to measuring the level of verbatim reuse, the effect of verbatim reuse can be divided into the reuse of problem domain components and the reuse of components at the architecture levels. No FDD simulators have been developed using the proposed reuse model, so the estimate will be based on the fact that the user interface for the dynamics simulators typically contains 40 percent of the source lines of code and no problem domain objects. Since the capabilities of the GENSIM simulation executive and case interface subsystems are currently distributed among other subsystems, 40 percent is a conservative estimate. It is probably correct to assert that the benefits of reusable architectures equal or exceed those of developing reusable problem domain components. It is clear that these benefits are roughly equal.

6. MANAGEMENT RECOMMENDATIONS

The primary management recommendation is to build the problem domain levels first and to build them bottom up. The language extension layer is a means of expression for domain objects and classes. The domain objects and classes serve as the building blocks for reusable system architectures. Another advantage of building the problem domain layers first is the ability to build multiple architectures from the same set of problem domain objects. For simulation applications, this means that the same set of problem domain objects could be used to build a dynamics simulator, a telemetry simulator, or a combined dynamics and telemetry simulator.

The strict separation of problem layers from architecture layers also provides the means of keeping up with technology. The same domain objects would be usable on either an 8086 based computer with a monochrome text screen or on an 80386-based computer with high-resolution graphics. The architecture of the system would be changed, although it would probably not be rebuilt from scratch. The architecture of a system should be driven by technology, and the solution of flight dynamics problems should not be. The separation of these considerations in design makes it easier to manage technological change.

7. CONCLUSIONS

The current state of the art in software reuse is to provide problem domain components and problem domain objects. This paper has demonstrated that designing verbatim reusable components at the architecture level can create approximately the same savings as the current state of the art. The new approach that needs to be applied to future systems is to strictly separate the problem domain objects from the particular system architectures and to build the problem domain layers from the bottom-up. When this approach is used to develop verbatim reusable software, the resources saved can be applied to new problems (extending the problem domain) or to provide better solutions to existing problems by upgrading the architecture.

REFERENCES

Berard, E., "Reusability Tutorial," *Proceedings of the Washington Ada Symposium*, 1989

Booch, G., *Software Engineering With Ada*. Menlo Park, CA: Benjamin Cummings, 1987

Booch, G., *Software Components With Ada*. Menlo Park, CA: Benjamin Cummings, 1987

Brechbiel, F., "Ada and Specification Reuse Versus Software Cost, Reliability, and Reusability in a Flight Dynamics Support Environment," (to be published)

Ganapathi, M., and G. Mendal, "Issues of Ada Compiler Technology," *IEEE Computer*, February 1989, vol. 22, no. 2, pp. 52-60

Herr, C., D. McNicholl, and S. Cohen, "Compiler Validation and Reusable Ada Parts for Real-Time, Embedded Applications," *ACM SIGAda Ada Letters*, September/October 1988, vol. VIII, no. 5, pp. 75-86

Hudson, W., "Ada Compiler Development," *Defense Science*, March 1988, pp. 59-64

Mendal, G., "Three Reasons To Avoid the Use Clause," *ACM SIGAda Ada Letters*, January/February 1988, vol. III, no. 1, pp. 52-57

Mendelsohn, C., "Impact Study of Generic Simulator Software (GENSIM) on Attitude Dynamics Simulator Development Within the Systems Development Branch," (unpublished study of simulator reuse data)

Perez, E., "Simulating Inheritance With Ada," *ACM SIGAda Ada Letters*, September/October 1988, vol. VIII, no. 5, pp. 37-46

Quimby, K., "Evolution of Ada Technology in the Flight Dynamics Area: Design Phase Analysis," Software Engineering Laboratory, SEL-88-003, December 1988

Seidewitz, E., and M. Stark, "Towards a General Object-Oriented Development Methodology," *Proceedings of the First International Symposium on Ada for the NASA Space Station*, June 1986

Seidewitz, E., "General Object-Oriented Software Development With Ada: A Life Cycle Approach," *Proceedings of the CASE Technology Conference*, April 1988

Solomon, D., and W. Agresti, "Profile of Software Reuse in the Flight Dynamics Environment," Computer Sciences Corporation, CSC/TM-87/6062, November 1987

Stark, M., and E. Seidewitz, "Towards a General Object-Oriented Ada Life Cycle," *Proceedings of the Joint Ada Conference*, March 1987

STANDARD BIBLIOGRAPHY OF SEL LITERATURE

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-77-004, A Demonstration of AXES for NAVPAK, M. Hamilton and S. Zeldin, September 1977

SEL-77-005, GSFC NAVPAK Design Specifications Languages Study, P. A. Scheffer and C. E. Velez, October 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 and W. A. Taylor, July 1986

SEL-79-002, The Software Engineering Laboratory: Relationship Equations, K. Freburger and V. R. Basili, May 1979

SEL-79-003, Common Software Module Repository (CSMR) System Description and User's Guide, C. E. Goorevich, A. L. Green, and S. R. Waligora, August 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-80-003, Multimission Modular Spacecraft Ground Support Software System (MMS/GSSS) State-of-the-Art Computer Systems/Compatibility Study, T. Welden, M. McClellan, and P. Liebertz, 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-81-008, Cost and Reliability Estimation Models (CAREM) User's Guide, J. F. Cook and E. Edwards, February 1981

SEL-81-009, Software Engineering Laboratory Programmer Workbench Phase 1 Evaluation, W. J. Decker and F. E. McGarry, March 1981

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 From 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-81-104, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et al., February 1982

SEL-81-107, Software Engineering Laboratory (SEL) Compendium of Tools, W. J. Decker, W. A. Taylor, and E. J. Smith, February 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-81-205, Recommended Approach to Software Development, F. E. McGarry, G. Page, S. Eslinger, et al., April 1983

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 From 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, F. E. McGarry, and M. G. Rohleder, October 1983

SEL-82-806, Annotated Bibliography of Software Engineering Laboratory Literature, M. Buhler and J. Valett, November 1989

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-83-006, Monitoring Software Development Through Dynamic Variables, C. W. Doerflinger, November 1983

SEL-83-007, Proceedings From the Eighth Annual Software Engineering Workshop, November 1983

SEL-84-001, Manager's Handbook for Software Development, W. W. Agresti, F. E. McGarry, D. N. Card, et al., April 1984

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 From the Ninth Annual Software Engineering Workshop, November 1984

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., May 1985

SEL-85-005, Software Verification and Testing, D. N. Card, C. Antle, and E. Edwards, December 1985

SEL-85-006, Proceedings From 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 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, October 1986

SEL-86-006, Proceedings From 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-87-008, Data Collection Procedures for the Rehosted SEL Database, G. Heller, October 1987

SEL-87-009, Collected Software Engineering Papers: Volume V, S. DeLong, November 1987

SEL-87-010, Proceedings From the Twelfth Annual Software Engineering Workshop, December 1987

SEL-88-001, System Testing of a Production Ada Project: The GRODY Study, J. Seigle, L. Ester, 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, Proceeding of the Thirteenth Annual Software Engineering Workshop, November 1988

SEL-88-005, Proceedings of the First NASA Ada User's Symposium, December 1988

SEL-89-001, Software Engineering Laboratory (SEL) Data Base Organization and User's Guide, M. So et al., May 1989

SEL-89-002, Implementation of a Production Ada Project: The GRODY Study, S. Godfrey and C. Brophy, May 1989

SEL-89-003, Software Management Environment (SME) Concepts and Architecture, W. Decker and J. Valett, August 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-RELATED LITERATURE

[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

[7]Basili, V. R., Maintenance = Reuse-Oriented Software Development, University of Maryland, Technical Report TR-2244, May 1989

[1]Basili, V. R., "Models and Metrics for Software Management and Engineering," ASME Advances in Computer Technology, January 1980, vol. 1

[7]Basili, V. R., Software Development: A Paradigm for the Future, University of Maryland, Technical Report TR-2263, June 1989

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

[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

[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

[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

Basili, V. R., and J. Ramsey, *Structural Coverage of Functional Testing*, University of Maryland, Technical Report TR-1442, September 1984

[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 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., 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., and H. D. Rombach, "T A M E: Tailoring an Ada Measurement Environment," *Proceedings of the Joint Ada Conference*, March 1987

[5]Basili, V., and H. D. Rombach, *T A M E: 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

[2]Basili, V. R., R. W. Selby, Jr., and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," *IEEE Transactions on Software Engineering*, November 1983

[3]Basili, V. R., and R. W. Selby, Jr., "Calculation and Use of an Environments'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, Jr., *Comparing the Effectiveness of Software Testing Strategies*, University of Maryland, Technical Report TR-1501, May 1985

[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

[4]Basili, V. R., R. W. Selby, Jr., and D. H. Hutchens, "Experimentation in Software Engineering," *IEEE Transactions on Software Engineering*, July 1986

[5]Basili, V. and R. Selby, Jr., "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, December 1987

[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

[5]Brophy, C., W. Agresti, and V. 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., and W. Agresti, "Resolving the Software Science Anomaly," *The Journal of Systems and Software*, 1987

[6]Card, D. N., and W. Agresti, "Measuring Software Design Complexity," *The Journal of Systems and Software*, June 1988

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

[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., Q. L. Jordan, and V. E. Church, "Characteristics of FORTRAN Modules," Computer Sciences Corporation, Technical Memorandum, June 1984

[5]Card, D., F. McGarry, and G. 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

[5]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

Hamilton, M., and S. Zeldin, *A Demonstration of AXES for NAVPAK*, Higher Order Software, Inc., TR-9, September 1977 (also designated SEL-77-005)

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

[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., and 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

[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

National Aeronautics and Space Administration (NASA), NASA Software Research Technology Workshop (Proceedings), March 1980

[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

[5]Ramsey, C., and V. R. Basili, An Evaluation of Expert Systems for Software Engineering Management, University of Maryland, Technical Report TR-1708, September 1986

[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

[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

[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

[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

[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

[7]Stark, M. E. and E. W. Booth, "Using Ada to Maximize Verbatim Software Reuse," Proceedings of TRI-Ada 1989, October 1989

Stark, M., and E. Seidewitz, "Towards a General Object-Oriented Ada Lifecycle," Proceedings of the Joint Ada Conference, March 1987

[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

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

Turner, C., G. Caron, and G. Brement, NASA/SEL Data Compendium, Data and Analysis Center for Software, Special Publication, April 1981

[5]Valett, J., and F. 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. 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

Zelkowitz, M. V., and V. R. Basili, "Operational Aspects of a Software Measurement Facility," Proceedings of the Software Life Cycle Management Workshop, September 1977

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.