SOFTWARE ENGINEERING LABORATORY SERIES   SEL-88-002

COLLECTED SOFTWARE
ENGINEERING PAPERS: VOLUME VI

NOVEMBER 1988

NASA

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

COLLECTED SOFTWARE ENGINEERING PAPERS: VOLUME VI

NOVEMBER 1988

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
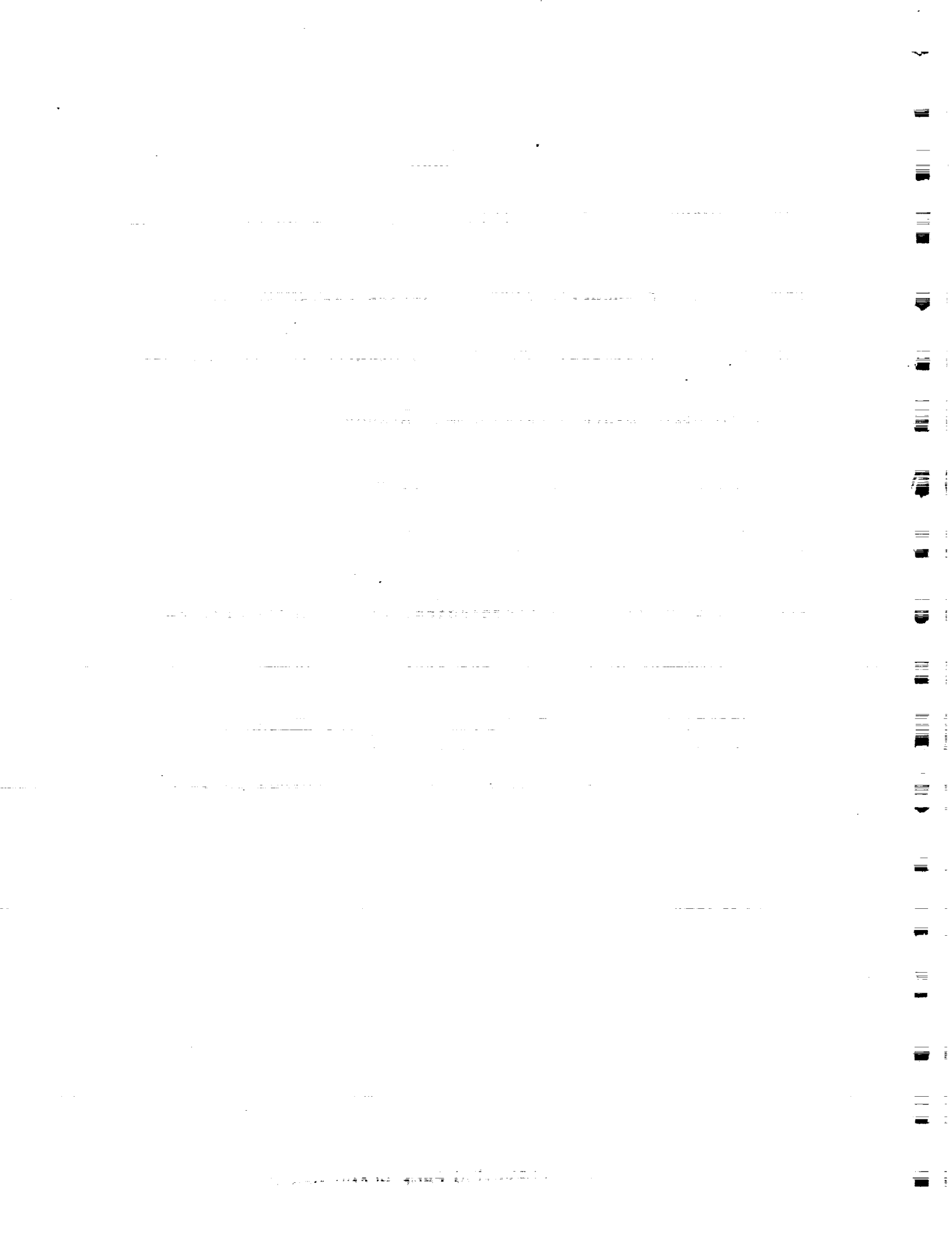   Greenbelt, Maryland 20771

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont'd)

vi

5207

SECTION 1 – INTRODUCTION

SECTION 1 - INTRODUCTION

This document is a collection of technical papers produced by participants in the Software Engineering Laboratory (SEL) during the period June 1, 1987, through January 1, 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 sixth 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 twelve 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.

1-1

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.

- "The Effectiveness of Software Prototyping: A Case Study," M. V. Zelkowitz, Proceedings of the 26th Annual Technical Symposium of the Washington, D.C. Chapter of the ACM, June 1987

- "Measuring Software Design Complexity," D. N. Card and W. W. Agresti, The Journal of Systems and Software, June 1988

- "Quantitative Assessment of Maintenance: An Industrial Case Study," H. D. Rombach and V. R. Basili, Proceedings from the Conference on Software Maintenance, September 1987

- "Resource Utilization During Software Development," M. V. Zelkowitz, The Journal of Systems and Software, 1988

2-1

THE EFFECTIVENESS OF SOFTWARE PROTOTYPING:
A Case Study

Marvin V. Zelkowitz
Department of Computer Science
University of Maryland
College Park, Maryland 20742

ABSTRACT

This paper discusses resource utilization over the life cycle of software development, and discusses the role that the current "waterfall model" plays in the actual software life cycle. The effects of prototyping are measured with respect to the life cycle model. Software production in the NASA environment was analyzed to measure these differences. The data collected from thirteen different projects and one prototype development were collected by the Software Engineering Laboratory at NASA Goddard Space Flight Center and analyzed for similarities and differences. The results indicate that the waterfall model is not very realistic in practice, and that a prototype development follows a similar life cycle as a production system--although, for this prototype, issues like system design and the user interface took precedence over issues such as correctness and robustness of the resulting system.

KEYWORDS: Life cycle, Measurement, Prototyping, Resource utilization, Waterfall chart

1. Introduction

As technology impacts the way industry builds software, there is increasing interest in understanding the software development model and in measuring both the process and product. New workstation technology, new languages (e.g., Ada, requirements and specification languages) as well as new techniques (e.g., prototyping, pseudocode) are impacting how software is built which further impacts how management needs to address these concerns in controlling and monitoring a software development.

In this paper, data are first presented which analyze several fairly large software projects from NASA Goddard Space Flight Center (GSFC) and put the current "waterfall" model in perspective. Data about software costs, productivity, reliability, modularity and other factors are collected by the Software Engineering Laboratory (SEL), a research group consisting of individuals from NASA/GSFC, Computer Sciences Corporation, and the University of Maryland, for research on improving both the software product and the process for building such software [SEL 82]. The Software Engineering Laboratory was established in 1976 to investigate the effectiveness of software engineering techniques for developing ground support software for NASA [BAS 78]. A recent prototyping experiment was conducted and data were collected which compare this prototype with the more traditional way to build software. The paper concludes with comments on the role of prototyping as a software development technique.

The software development process is typically product-driven, and can be divided into six major life cycle activities, each associated with a specific "end product" [WAS 83, ZEL 78]:

(1) *Requirements phase* and the publication of a requirements document.

(2) *Design phase* and the creation of a design document.

(3) *Code and Unit Test phase* and the generation of the source code library.

(4) *System integration and testing phase* and the fulfillment of the test plan.

(5) *Acceptance test phase* and completion of the acceptance test plan.

(6) *Operation and Maintenance phase* and the delivery of the completed system.

In order to present consistent data across a large number of projects, this paper only focuses on the interval between design and acceptance test and involves the actual implementation of the system by the developer group.

In this paper, we will refer to the term *activity* as the work required to complete a specific task. For example, the coding activity refers to all work done in generating the source code for a project, the design activity refers to building the program design, etc. On the other hand, the term *phase* will refer to that period of time when a certain activity is supposed to occur. For example, the Coding Phase will refer to that period of time

7

5207

during a software development when coding activities are supposed to occur. It is closely related to management-defined milestone dates for a project. But during this period, other activities may also occur.

REQUIREMENTS

DESIGN

CODE

INTEGRATION

ACCEPTANCE TEST
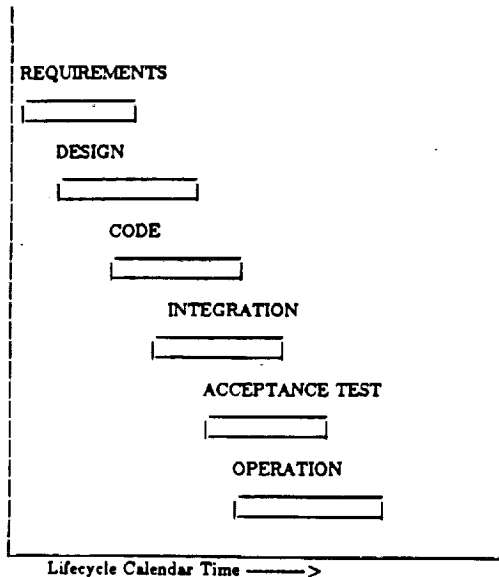
OPERATION

Lifecycle Calendar Time ———>

Figure 1. Typical Life Cycle

The waterfall model makes the assumption that all activity of a certain type occurs during the phase of that same name and phases do not overlap. Once a phase ends, then the next phase begins. Thus all requirements for a project occur during the Requirements Phase; all design activity occurs during the Design Phase. Once a project has a design review and enters the Coding Phase, then all activity is Coding. Since many companies keep data based upon hours worked by calendar date, this model is very easy to track. However, as Figure 1 shows, activities overlap and do not lie in separate phases. We will give more data on this later.

2. The waterfall chart is all wet

In the NASA/GSFC environment that we studied, the software life cycle follows a fairly standard set of activities [SEL 81]:

The *requirements* activity involves translating the functional specification consisting of physical attributes about the spacecraft to be launched into requirements for a software system that is to be built.

The *design* activity can be divided into two subactivities: the *preliminary design* activity and the *detailed design* activity. During preliminary design, the major subsystems are specified, input-output interfaces and implementation strategies are developed. During detailed design, the system architecture is extended to the subrou-

tine and procedure level. Data structures and formal models of the system are defined. These models include procedural descriptions of the system, dataflow descriptions, complete description of all user input, system output, and input-output files, operational procedures, functional and procedural descriptions of each module, and complete description of all internal interfaces between modules.

The *Coding and Unit Test* activity involves the translation of the detailed design into a source program in some appropriate programming language (usually FORTRAN). Each programmer will unit test each module for apparent correctness.

The *System Integration and Test* activity validates that the completed system produced by the coding and unit test activity meets its specifications. Each module, as it is completed, is integrated into the growing system and integration test is performed to make sure that the entire package executes as expected. Functional testing of end-to-end system capabilities is performed according to the system test plan developed as part of the requirements activity.

In the *Acceptance Test* activity, the development team provides assistance to the acceptance test team, which checks that the system meets its requirements.

Operation and Maintenance activities begin after acceptance testing when the system becomes operational. For flight dynamics software at NASA, these activities are not significant to the overall cost. Most software produced is highly reliable. In addition, the flight dynamics software is usually not mission critical in that a failure of the software does not mean spacecraft failure but simply that the program has to be rerun. In addition, many of these programs (i.e., spacecraft) have limited lifetimes of six months to about three years.

Table 1 presents the raw data on the fourteen projects analyzed in this paper. The thirteen numbered projects are all fairly large flight dynamics programs, ranging in size from 15,500 lines of FORTRAN code to 89,513 lines of FORTRAN, with an average size of 57,890 lines of FORTRAN per system. The average work on these projects was 89.0 staff months; thus, all represent significant effort. The last project listed in Table 1 - FDAS - represents a prototype development and will be discussed in more detail later.

In most organizations, phase data are collected weekly so that they are the usual reporting mechanism. However, in the SEL, activity data are also collected. The data that are collected consist of nine possible activities for each component (i.e., source program module) worked on for that week. In this paper, these will be grouped as Design activities, Coding activities (code preparation and unit testing), Integration testing, Acceptance testing and Other. Specific review meetings, such as design reviews, will be grouped with their appropriate activity (e.g., a design review is a design activity, a code walkthrough is a coding activity, etc.). This allows us to look at both phase and activity utilization.

5207

PROJECT SIZE AND STAFF-MONTH EFFORT			
PROJECT NUMBER	SIZE (LINES OF CODE)	TOTAL EFFORT HOUR*	STAFF-MONTHS
1	15,500	17,715	116.5
2	50,911	12,588	82.8
3	61,178	17,039	112.1
4	26,844	10,946	72.0
5	25,731	1,514	10.0
6	67,325	19,475	128.4
7	66,260	17,997	118.4
8	+	+	+
9	55,237	15,262	100.4
10	75,420	5,792	38.1
11	89,513	15,122	99.5
12	75,393	14,508	95.4
13	85,369	14,309	94.1
Average	57,890	13,522	89.0
FDAS	33,967	14,150	93.1

+ - Raw data not available in data base
* - All technical effort including programmer and management time

Table 1. Project Size and Staff-month Effort

The results of this can be briefly summarized by Table 2. According to this, in NASA, 22% of a project's effort is during the design phase, while 49% is during coding. Integration testing takes 16% while all other activities take 12%. (Remember that requirements data are not being collected here. We are simply reporting the percentage of design, coding, and testing activities. A significant requirements activity does occur.)

	Design	Code	Int. Test.	Other
By phase	22	49	16	12
By activity	25	30	15	29

Table 2. Activities performed in each phase (by %)

However, actual activities differ somewhat from simply looking at effort spent between somewhat arbitrary calendar dates set up months in advance. By looking at all design effort across all phases of the projects, design activity is actually 25% of the total effort rather than the 22% listed above. Coding is a more reasonable 30% which means that the coding phase includes many other activities. "Other" increased from 12% to 29%, and include many time-consuming tasks that are not accounted for by the usual life cycle. (Here, Other includes acceptance testing, as well as activities that take a significant effort but are usually not separately identifiable using the standard model. These activities include meetings, training, travel, documentation, and other various activities assigned to the project.)

The situation is actually more complex than shown in Table 2. Although using Phase Date shows that total design effort differs by only 3% from the design phase effort, the distribution of design activity throughout the project is not reflected in the table. These data are presented in Table 3.

Design	Code	Int. Test	Accept. Test
50	29	20	2

Table 3. Design Activity During Life Cycle Phases (by %)

As Table 3 shows, only 50% of all design work occurs during the Design Phase and just under one third of the total design activity occurs during the coding period. Over one fifth (20%+2%) of all design occurs during testing when the system is "supposed" to be finished.

As to coding effort, Table 4 shows that while a major part, or 70% of the coding effort, does occur during the Coding Phase, almost one quarter (16%+7%) occurs during the testing periods. As expected, only a small amount of coding (7%) occurs during the design phase; however, it does indicate that some coding does begin on parts of the system while other parts are still under design.

Design	Code	Int. Test	Accept. Test
7	70	16	7

Table 4. Coding Activity during Life Cycle Phases (by %)

Similarly, Table 5 shows that significant integration testing activities (about 34%) occur before the integration testing period. Once modules have been unit tested, programmers begin to piece them together to build larger subsystems.

Design	Code	Int. Test	Accept. Test
0	34	63	3

Table 5. Integration Activity during Life Cycle Phases

3. Prototyping

As can be seen, programmers readily flow from one activity of a project to another--more like a series of rapids and not as a discrete set of waterfalls. Any model that does not reflect this cannot hope to accurately portray software development. Boehm has proposed a spiral model [BOE 86] of software development which takes some of this into account. In addition, the concept of prototyping has been proposed as an alternative concept. The remainder of this paper will address the prototyping issue.

The current model of software development is becoming even more complex. As new techniques are developed, how do they fit into the life cycle? For example, pseudocode is often written to describe a design. This pseudocode is often iterated in greater detail to evolve into the source program. However, when does pseudocode stop being design and when does it become a source program? Prototyping is another technique which doesn't fit into this model well. In a prototype, the

developer builds some operational aspect of the system and then evaluates the prototype with respect to some criteria. Where does this coding and testing fit? What activity is this in the overall life cycle?

At NASA, a prototype was developed to investigate implementation strategies for a new product. In this section, the role of the prototype will be described and the resulting data collected from building the prototype will be compared with the historical life cycle data presented in the preceding section.

A prototype Flight Dynamics Analysis System (FDAS) was implemented by NASA/GSFC. Data were collected during the development of the system. For typical flight dynamics software, which NASA has considerable experience in building, prototyping would be of limited benefit due to significant knowledge of how previous systems were built. However, in this case, FDAS was to be a source code maintenance system to manage other source code libraries. It would enable NASA analysts to test new spacecraft orbit models by providing a human-engineered common interface which could be used to invoke other flight dynamics packages. Since it was unlike previous NASA projects, and since NASA personnel had limited knowledge of exactly how to build this system, FDAS was a good candidate for prototyping.

The goal of the prototype was to understand the problem domain better. As such, an early decision was made to build the system with every expectation of throwing it away. If part of the source program could be transferred to the final system, then that would be viewed as an unexpected bonus. After the prototype was built, it would be evaluated and from this experience the requirements for a production version of FDAS would be developed. Therefore, the basic idea of the prototype was to learn, and it fits into the life cycle as part of the requirements phase of Figure 2.

This definition of prototyping differs from others that view a prototype as a first release of a system. The goal was clearly to be able to understand the problem and not to generate useable source programs. In another study [BOE 84], prototyping was viewed as an iterative process converging on the final product.

We viewed the prototype as part of the requirements analysis of the problem. However, since the prototype was to execute, it itself had a full development life cycle. As Table 1 previously showed, since FDAS was almost 34,000 lines of code and took about 93 staff months to complete, it was a rather large project by itself.

FDAS was to be an interactive system. That meant that the user interface was crucial. Because of this, it was determined that the prototype should emphasize that aspect of system design.

The prototype was built in FORTRAN for a DEC VAX 11/780. In hindsight it is not clear that such an implementation was the wisest. However, at the start, the problem did not seem that complex, and personnel experience and available hardware and software lent
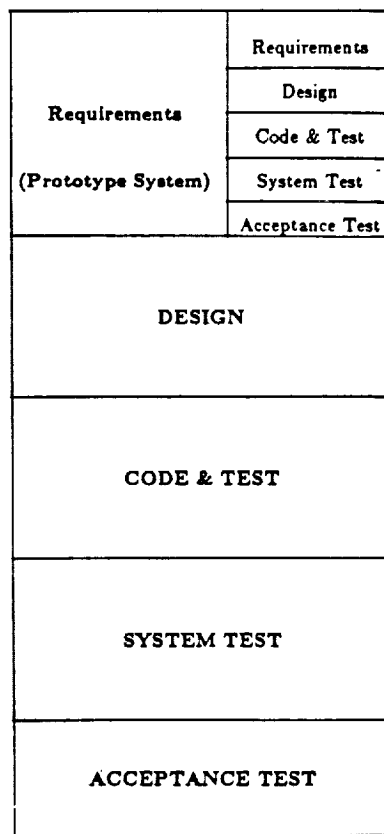
Figure 2. Prototype as part of Software Life Cycle

themselves to a FORTRAN implementation. Since the goal was to give the user a taste of what services the system would provide, a screen simulation applications package (e.g., Rapid/Use [WAS 86]), a very high level simulation, or a 4th generation language might have been adequate.

The use of FORTRAN, however, did have some benefits. For one, it gave the developers experience in using FORTRAN in a type of text-processing application for which they had little previous experience. One of the reasons that the NASA group generally has high productivity is that they have had considerable experience in its application area. By building the prototype in FORTRAN, they were using Brooks' second system property where he advises "plan to throw one away" [BRO 75]. By building a first prototype in FORTRAN, mistakes would undoubtedly be made. By planning on discarding the prototype rather than patching it to correct errors, the ultimate FDAS system should be more reliable and better structured — even if it did not turn out to be cheaper. This by itself is a valuable property, although it is not clear that it is a measurable one on most projects.

A more important aspect of a FORTRAN implementation (at least with respect to this paper) is that the FDAS prototype was a "typical" FORTRAN project. Hence its life cycle characteristics and the data that were collected could be compared with many other projects in the NASA database. This would not have been possible if some other mechanism (e.g., simulation package of some sort) were used.

In the next section, the prototype will be evaluated. However, here are some of our general conclusions. The handling of requirements differed from a production system; FDAS requirements were incomplete when design began. Unlike previous projects, they were not stated precisely because aspects of the system were still an open subject during development [ZEL 84]; even identifying the potential user community and its impact on the user interface and its effect on "assumed computer experience" was still being considered. Dates for completion of each phase were more flexible than in the historical data and milestones were less rigid than in a production development. During other phases, requirements were generally modifiable which in turn affected all activities in each phase.

More time was spent in design, than is usual for a typical project. Unlike other NASA projects, an extensive review process took place almost weekly as design decisions were made and altered. The coding and testing efforts had no formal review. Although status meetings were held almost weekly, the developers placed less emphasis on testing than with a production system; and since the prototype had a very limited lifetime, features that seemed well understood but cumbersome to implement were deleted from the requirements. According to the final report, coding took less time than in previous projects but testing did consume the same amount of effort. Very little effort was spent on acceptance testing, since the effective life of the prototype was short.

4. Evaluation of Prototype

In a manner similar to the 13 other NASA projects, the FDAS project was analyzed by phases and activities using data in the SEL database.

4.1. Phase Analysis

Data collection based on phases is shown in Table 6. The effort expended for design, coding, and testing were comparable, but notice that acceptance testing was only 1.3% of the prototype effort, but 12.7% in the historical data. With a limited lifetime, reliability was a limited feature. As long as the system worked for evaluation, it was adequate. In addition, integration testing took 10% more effort (26% compared to 16%) in the prototype. We believe this was mostly due to "schedule slippage" as the complexity of the prototype caused activities to be delayed until the end.

DEVELOPMENT EFFORT BY PHASE DATE (13 Projects vs Prototype FDAS)				
PROJECT NUMBER	DESIGN (%)	CODE (%)	INTEG. ACT.(%)	ACC.TST (%)
1	20.6	38.6	16.5	24.3
2	16.2	48.4	19.3	16.2
3	21.8	47.9	17.4	12.9
4	35.9	39.5	24.5	0.1
5	18.2	68.8	13.0	0.0
6	16.3	48.6	10.9	24.3
7	19.0	50.4	14.9	15.7
8	22.9	48.4	13.0	15.8
9	22.6	68.3	8.1	1.1
10	24.4	44.6	20.2	10.8
11	22.7	39.4	21.4	16.5
12	16.9	53.1	10.9	19.1
13	28.2	43.5	20.1	8.2
Average	22.0	49.2	16.2	12.7
FDAS	27.0	45.3	26.4	1.3

Table 6. Software Development Effort by Phase

4.2. Activity Analysis

In the previous subsection, we viewed effort by phase date. Table 7 displays the actual activities of design, coding and integration test effort independent of phase. In this case the results differ. Usually during the design phase, coding and testing activities begins on some modules, and in the code and unit test phase, additional design activity continues. Integration testing begins as soon as coding and unit testing of a component completes. Similarly, during the testing phase, any errors that were uncovered might require substantial redesign and recoding. Comparing with Table 6, we discover that most NASA developments have additional design effort later in the life cycle to raise total design effort from 22% to 25.6%. In the FDAS case, total design dropped from 27% to 25%, meaning that activities other than design occurred in the design phase. In both cases, activities other than coding occur during the coding phase since actual coding activity was only 30.5% and 17.6% respectively, as opposed to the 45+% of effort of the coding phase (Table 6).

Comparing FDAS with the 13 other developments, design effort is comparable at 25%, but the code and unit test effort and the integration test effort were different. Due to the wide variability of the "other" category of Table 7, Table 8 presents the same data as relative percent for Design, Code, and Integration testing only. This shows the differences more clearly.

No formal review was performed on the prototype during coding and unit testing. Because of the decision to delete hard-to-build but understood features that did

5207

DEVELOPMENT EFFORT BY ACTIVITY IN ALL PHASES (13 Projects vs Prototype FDAS)				
PROJECT NUM	DESIGN ACT (%)	CODE ACT (%)	INTEG ACT (%)	OTHER ACT (%)
1	17.4	16.4	9.9	56.3
2	30.1	39.4	20.8	9.7
3	26.3	20.3	19.3	34.2
4	27.3	28.7	6.0	38.0
5	31.0	35.5	9.4	24.1
6	14.9	21.8	24.0	39.2
7	20.2	25.9	14.3	39.6
8	11.0	13.9	9.3	65.8
9	31.3	43.5	18.9	6.4
10	38.2	37.3	6.1	18.4
11	29.3	31.0	17.2	22.5
12	23.7	46.5	24.0	5.9
13	32.6	36.3	15.6	15.6
Average	25.6	30.5	15.0	28.9
FDAS	25.0	17.6	25.1	32.3

Table 7. Software Development Effort by Activity

not effect the FDAS evaluation, coding was quite straightforward. Most of the easy coding was completed in a rather short time, and the more difficult coding aspects were simply not implemented. As Table 8 indicates, at 26% coding, FDAS had the lowest relative coding effort of any of the 14 measured projects. The next lowest was 30.8% and the average over all 13 was 42.2%. In addition, while in most projects the design and integration testing efforts were less than the coding activity, in FDAS both were almost 50% greater than for coding (about 37% for each compared to 26% for coding).

PER CENT EFFORT IN EACH PHASE (13 Projects vs Prototype FDAS)			
PROJECT NUM	DESIGN ACT(%)	CODE&UNIT ACT(%)	INTEG. ACT(%)
1	39.9	37.5	22.6
2	33.3	43.7	23.0
3	39.9	30.8	29.3
4	44.0	46.3	09.7
5	40.8	46.8	12.3
6	24.6	35.9	39.5
7	33.5	42.8	23.6
8	32.2	40.7	27.1
10	46.8	45.7	07.5
11	37.8	40.1	22.1
12	25.2	49.4	25.5
13	38.6	43.0	18.4
Average	36.2	42.2	21.6
FDAS	36.9	26.0	37.1

Table 8. Relative Activity

This apparent short circuiting of coding, however, appeared to have a detrimental effect on testing, which took a relative 37.1% of effort as opposed to 21.6% on other projects. Only one other project (6) took as much effort (39%) and from Table 1 project 6 was the most costly, where you might expect an excessive need for testing.

Based on the original productivity rate of 1.4 source lines of code (SLOC) per hour on most NASA projects [BAS 81], FDAS with a size of 33,067 SLOC had a productivity rate of 2.4 SLOC per hour. (Note: the average project size of 57,890 SLOC of Table 1 cannot simply be divided by the average effort of 13,552 hours since most NASA projects reuse some code from previous systems. Table 1 is total system size, and the productivity rate is for new lines of code.)

4.2.1. Design Effort

A true picture of development can be achieved by investigating actual activity during each phase. Although design is supposed to occur principally during the design phase, for both the 13 older projects and the FDAS prototype a comparable one half of the total design effort occurred during the design phase, and equal amounts were distributed through the rest of the life cycle (Table 9). This repeats Table 3 in more detail. Only 2% of the design of FDAS occurred during the acceptance test phase in the prototype, principally because the FDAS acceptance testing phase was so short and the few errors that were found did not get redesigned and corrected. For the historical data, the 6.4% of design occurring during acceptance testing represents errors found in testing that required source code to be redesigned.

DESIGN ACTIVITY EFFORT IN EACH PHASE (13 Projects vs Prototype FDAS)				
PROJECT NUM	DESIGN PHASE(%)	CODE PHASE(%)	INTEG. TEST(%)	ACC.TST. PHASE(%)
1	41.8	33.9	10.0	14.3
2	53.6	31.2	9.2	6.0
3	33.3	37.1	19.7	9.9
4	45.3	32.6	22.0	0.1
5	17.4	69.1	13.5	0.0
6	58.9	30.7	4.3	6.2
7	63.9	15.3	6.8	14.1
8	28.1	56.9	7.1	8.0
9	61.8	38.2	0.0	0.0
10	57.8	27.2	7.0	8.0
11	58.7	13.7	16.67	10.9
12	58.9	32.8	5.9	2.4
13	60.5	24.7	11.9	2.9
Average	49.2	34.1	10.3	6.4
FDAS	49.8	28.9	19.6	1.7

Table 9. Design Activity Effort

4.2.2. Code & Unit Test Effort

The code & unit test activities in the prototype, however, represent a departure from the older projects (Table 10). In most developments, about 7% of the coding is completed during design (although it varied from 0% to 22% in the 13 other projects). Implementation often begins as some components become completely specified. However, with FDAS, due to its greater uncer-

tainty, no coding occurred until the development team really understood the design, i.e., until the coding phase began. For most projects, 70% of the total code and unit test effort is in the coding phase, but in the prototype almost 96% of the effort was during coding. Coding often extends through acceptance testing, but with FDAS's relatively light acceptance test, few critical errors were found so little effort was spent in recoding during test. Coding and testing need to be carried out on the full system for every change or modification of the design, but in the prototype it was not necessary to code the new design.

CODE & TEST ACTIVITY EFFORT IN EACH PHASE (13 Projects vs Prototype FDAS)				
PROJECT NUM	DESIGN PHASE(%)	CODE PHASE(%)	INTEG. TEST(%)	ACC.TST. PHASE(%)
1	1.4	78.3	11.3	9.1
2	0.0	72.8	19.7	7.5
3	22.2	56.2	11.8	9.8
4	16.4	58.5	25.1	0.1
5	21.2	68.7	10.1	0.0
6	0.5	77.3	11.3	10.9
7	1.3	73.9	15.6	9.2
8	14.7	54.7	21.0	9.7
9	5.2	91.1	3.1	0.6
10	0.0	73.0	22.5	4.5
11	2.2	70.5	20.1	7.2
12	0.3	74.8	8.3	16.6
13	4.6	63.6	26.9	4.9
Average	6.9	70.3	15.9	6.9
FDAS	0.0	95.9	4.1	0.0

Table 10. Code & Unit Test Activity Effort

4.2.3. Integration Test Effort

Integration test effort is distributed through all phases in the collected projects with more effort (43%) during the code & unit phase than in either the integration phase (26%) or the acceptance test phase (26%) (Table 11). In general, almost 50% of all integration testing occurs during design and coding phases. In FDAS, this effort was delayed with about two-thirds of all integration activities in the integration phase. This was due to delaying the integration until more pieces of the system were completed.

4.2.4. Other Activity Effort

The Other category consists of activities such as travel, completion of the data collection forms, meetings, or training. While these activities are often ignored in most life cycle studies, the costs are significant. Typically, about 29% of activities are in this category and of the 13 measured projects, "other" consumed more than one-third of the effort on 6 of them (Table 7). FDAS used a comparable 32% "other". As seen in Table 12, the prototype devoted more effort to the design phase, mainly for meeting, traveling, and training due to the extensive unknown quality of the design at the beginning

INTEGRATION ACTIVITY EFFORT IN EACH PHASE (13 Projects vs Prototype FDAS)				
PROJECT NUM	DESIGN PHASE(%)	CODE&UNIT PHASE(%)	INTEG. TEST(%)	ACC.TST. PHASE(%)
1	0.0	17.8	27.4	54.7
2	0.0	45.2	30.1	24.7
3	6.1	53.9	21.1	18.9
4	21.0	39.3	39.7	0.0
5	28.4	71.0	0.6	0.0
6	1.0	40.9	17.6	40.5
7	0.5	54.1	26.3	19.2
8	2.9	33.8	19.2	44.1
9	0.0	66.4	29.2	4.4
10	0.0	23.1	41.5	35.5
11	0.0	36.4	35.1	28.5
12	0.1	32.7	22.4	44.8
13	1.5	49.5	28.8	20.2
Average	4.7	43.4	26.1	25.8
FDAS	0.0	34.5	62.7	2.8

Table 11. Integrating Test Activity Effort

of the task. The acceptance test activity is low for the similar reason that the prototype system had few users of short duration and therefore no detailed tests. On the 13 collected projects, the Other activities are distributed more uniformly during all phases, including the acceptance test where there is a need to test before actually turning the system to the user.

OTHER ACTIVITIES EFFORT IN EACH PHASE (13 Projects vs Prototype FDAS)				
PROJECT NUM	DESIGN PHASE(%)	CODE&TST PHASE(%)	INTEG. TEST(%)	ACC.TST. PHASE(%)
1	23.3	32.2	18.1	26.5
2	0.0	9.1	26.4	64.6
3	21.7	47.8	16.8	13.7
4	46.2	30.2	23.6	0.0
5	11.0	67.7	21.3	0.0
6	18.2	44.2	9.0	28.7
7	14.4	51.6	14.5	19.5
8	26.5	47.7	11.4	14.4
9	15.9	65.5	18.7	0.0
10	12.4	30.2	35.9	21.5
11	21.4	32.2	18.9	27.6
12	47.3	46.6	4.6	1.5
13	42.5	30.0	12.7	14.9
Average	23.1	41.2	17.8	17.9
FDAS	45.1	38.8	15.7	0.3

Table 12. Other Activities Effort

5. Conclusions

In this paper we have collected data on many software projects developed at NASA/GSFC and compared them with a new prototype development. By using data from the SEL database, it appears clear that the software development process does not follow the waterfall life cycle. It also appears that the prototype develop-

13

2-8

5207

ment follows a similar life cycle pattern as other software projects. Although a single data point (the prototype) does not give definitive answers, it does give some trends that are of interest.

Both approaches have similar software life cycles, but the effort distributed over each phase differs. The coding in the prototype was more ad hoc, therefore testing became more involved. Integration testing was harder in the prototype because of the false assumption that reliability was not a central issue. The production developments devote more effort in coding than in testing (Table 7).

While not inexpensive, the prototype appears to be successful. Several design decisions turned out to be partially faulty when the prototype was tested. The human computer interface has been redesigned.

In fact, after completion of the prototype, several screen simulation systems were used to model a user interface, and a more hierarchical menu model was developed. Without the FDAS experience, NASA might have implemented a system where users had no real experience until the large implementation would be too far along to change adequately.

The underlying execution model of FDAS became better understood. As a source code control system, the separation of the FDAS code and the user's flight dynamics application code became clearer. Most user programs would be FORTRAN (at least initially); however, other languages (e.g., Pascal, Ada) would be used in the future, while it would not matter to the user in what language FDAS was itself written.

FDAS included a prototype preprocessor to add abstract data types to FORTRAN. This preprocessor was initially tied directly to the FDAS implementation. It is now somewhat independent to allow for other preprocessors later. The FORTRAN preprocessor, call OPAL, for Object Programming Applications Language [CSC 86], is a more rational extension of FORTRAN with data structures useful for flight dynamics applications, such as vectors, matrices, and quaternions. The decision was also made to move away from FORTRAN, and the system itself is being implemented in Ada, although it will initially process FORTRAN application code.

A new production FDAS implementation would avoid many potential pitfalls discovered via the prototype. Currently the production version of FDAS is under development, and its design has benefited greatly from the earlier development. We will have to wait for completion before fully evaluating this process. It is quite clear, however, that FDAS will be a much better product that if the prototype had not been built.

Prototyping probably increases the cost of the system, but it greatly increases its quality. It gives a flavor to the end user of what the system can do and how it can perform the task, especially in a nonfamiliar environment. It provides the developers a "second system" effect for perfecting a design.

7. References

[BAS 78]
Basili, V. R. and Zelkowitz M. V. "Analyzing Medium-Scale Software Development" 3rd International Conference on Software Engineering, Atlanta, pp. 116-223, (May 1978).

[BAS 81]
Basili, V. R. and Freburger, K., "Programming Measurement and Estimation in the Software Engineering Laboratory" The Journal of System & Software, Vol. 2, pp. 47-57, (1981).

[BOE 84]
Boehm B., Gray, T. and T. See Waldt, Prototyping vs. specifying: a multiproject experience, 7th International Conference on Softwrae Engineering, orlando, FLA, March 1984, pp. 473-484.

[BOE 86]
Boehm B., A spiral model of software development and enhancement, ACM Software Engineering Notes 11, 4, August 1986, pp. 22-42.

[BRO 75]
Brooks, F., The Mythical Man Month Addison Wesley, 1975.

[CSC 86]
Applications Software under the Flight dynamics analysis system (FDAS), Computer Sciences Corporation, CSC/SD-86/6024, November, 1986.

[SEL 81]
McGarry, F. E. and Page, G. and et al., "Standard Approach to Software Development", NASA Goddard Space Flight Center, Greenbelt, MD, (September 1981).

[SEL 82]
McGarry, F. E. and Church, V. and Carl, D. and et al., "Guide to Data Collection", NASA Goddard Space Flight Center, Greenbelt, Md, (August 1982).

[WAS 83]
Wasserman, A., "Software Engineering Environment", *Advances in Computer*, Vol. 22, pp. 110-159, (1983).

[WAS 86]
Wasserman A. I., P. A. Pircher and D. T. Shewmake, Building reliable interactive information systems, *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 1, pp. 147-156 (January, 1986).

[ZEL 78]
Zelkowitz, M.V., "Perspective on Software Engineering", *ACM Computing Surveys*, Vol. 10, NO. 2, pp. 198-216, (June 1978).

[ZEL 84]
Zelkowitz, M. V. and Sukri, J., "Technique for Subjective Evaluation of Prototyping Design", *Proceeding of Ninth Annual Software Engineering Workshop*, NASA Goddard Space Flight Center, Greenbelt, MD, (December 1984).

15

2-10

5207

Measuring Software Design Complexity

D. N. Card and W. W. Agresti

Computer Sciences Corporation, Silver Spring, Maryland

Architectural design complexity derives from two sources: structural (or intermodule) complexity and local (or intramodule) complexity. These complexity attributes can be defined in terms of functions of the number of I/O variables and fanout of the modules comprising the design. A complexity indicator based on these measures showed good agreement with a subjective assessment of design quality but even better agreement with an objective measure of software error rate. Although based on a study of only eight medium-scale scientific projects, the data strongly support the value of the proposed complexity measure in this context. Furthermore, graphic representations of the software designs demonstrate structural differences corresponding to the results of the numerical complexity analysis. The proposed complexity indicator seems likely to be a useful tool for evaluating design quality before committing the design to code.

1. INTRODUCTION

Typically, design is the earliest stage of software development at which the pending software system is fully specified and in which the system structure is clearly defined. Design usually proceeds in two steps—architectural, then detailed design. This study only considers the former. Throughout the following discussion, "design" will refer to architectural design unless otherwise indicated. Assessment of the quality of a software design rates high in the priorities of software developers and managers. However, the multitude of potentially conflicting design objectives, methods, and representations, as well as a lack of appropriate data, have hindered the development of effective measures of software design quality.

One quality attribute, complexity, has been studied extensively. Early investigations [1, 2] focused on the internal organization of individual programs or subprograms rather than on the structure of software systems composed of large numbers of subprograms (or modules). More recently, complexity studies have attempted to consider software systems [3, 4]. Many of these approaches require extensive analysis (usually special tools) to compute values of the complexity measures proposed. Moreover, few of these measures can be computed at design time. The objective of this study was to define some "simple" complexity measures that could easily be derived during early design.

The initial investigation considered many existing models of software complexity but did not find any of them suitable for this application because 1) necessary data were difficult to extract or compute, 2) required information was not available during architectural design, and/or 3) our data data did not support the model. For example, all of these reservations apply to software science [1]; see Card and Agresti [28].

This paper explains a new approach to measuring software design complexity that considers the structure of the overall system as well as the complexity incorporated in individual components. The measures derive from a simple model of the software design process. Analysis of data from eight medium-scale scientific software projects showed that the complexity measures defined in this report provide a good estimate of the overall development error rate, as well as agreeing with a subjective assessment of design quality. Furthermore, differences in design complexity indicated by the complexity measures also demonstrated themselves in design profile graphs.

This analysis relied on data collected by the Software Engineering Laboratory (SEL) from eight spacecraft flight dynamics projects. The SEL is a research program sponsored by the National Aeronautics and Space Administration [5]. It is supported by Computer Sciences Corporation and the University of Maryland. The objectives of the SEL are to measure the process of software development in the flight dynamics environment at Goddard Space Flight Center, identify technology improvements, and transfer this technology to flight dynamics software practitioners.

Address correspondence to David N. Card, Computer Sciences Corporation, 8728 Colesville Road, Silver Spring, MD 20910.

2-11

5207

2. NATURE OF DESIGN COMPLEXITY

Architectural design is the process of partitioning the required functionality and data of a software system into parts that work together to achieve the full mission of the system. Thus, architectural design complexity can be viewed as having two components: 1) the complexity contained within each part (or module) defined by the design, and 2) the complexity of the relationships among the parts (modules). In the following discussion, we will refer to design parts as modules, in the sense that a module is the smallest independently compliable unit of code [6]. Each design part will eventually be implemented as a software module. In the FORTRAN environment of the SEL, modules correspond to subroutines.

Many different approaches or methods achieve the same design result: a high-level architectural design and an integrated set of individual module designs. The detailed design (e.g., PDL) developed to implement the work assigned to a module provides another source of complexity that is not analyzed here. It is not the intent of this paper to address whether specific design methods result in lower-complexity (or better) design products. Rather, its objective is to demonstrate a complexity measurement approach that can be applied to a wide range of such products, regardless of how they were produced. The authors recognize that correct design practice is essential to achieving good designs. Generally, this report shows that the conditions that result in lower values of the complexity measures are consistent with accepted design practices. ·

Of course, any complete design must include nonmodules such as files and COMMON blocks (in FORTRAN). Furthermore, partitioning is not the only design process. This proposed model only attempts to capture a subset of all the possible factors in complexity. As Curtis [7] points out, complexity depends on the perspective from which an object or system is viewed. This paper examines software complexity with respect to the difficulty of producing the designed system (for example, the difficulty of changing the implemented system is not considered). The following discussion is intended to illustrate the line of reasoning followed in developing the model and measures. It should not be construed as a mathematical proof that this model is a necessary and sufficient explanation of complexity.

2.1 A Design Model

One common approach to design is functional decomposition (the basis of structured design [6]). It results in a hierarchical network of units (or modules). For any module, workload consists of input and output items
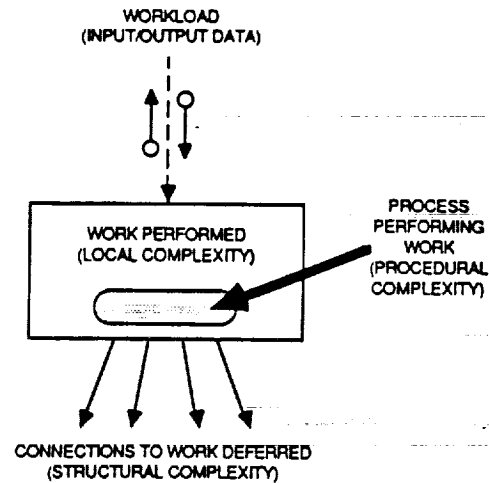

Figure. 1. Decomposition model of software design.

(data couples) to be processed. At each level of decomposition, the designer must decide whether to implement the indicated functionality (perform the work) in the current module or defer some of it to a lower level by invoking one or more other modules (via control couples). Deferring functionality decreases the *local* (intramodule) complexity but increases the *structural* (intermodule) complexity (see Fig. 1). Similar decisions also must be made when following other design approaches (e.g., object oriented [8]).

The internal design of a module (how the work is performed) may contribute *procedural* complexity, but that is outside the scope of this paper. Of course, many early studies of software complexity (e.g., [2]) focused on process construction. The distinction made here between local and procedural complexity parallels the distinction between the specification and the body of an Ada* package.

Thus, architectural complexity is a function of the work performed (within modules) as well as the connections among the work parts (modules). Effective design minimizes work as well as connections. This argument leads to the following formulation for the total complexity of a software design:

$$C{\sim} = S{\sim} + L{\sim} \qquad (1)$$

where

$C{\sim}$ = total design complexity

$S{\sim}$ = structural (intermodule) complexity

$L{\sim}$ = local (intramodule) complexity

---

* Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

That is, the total complexity of a design of given complexity $C\sim$ can be defined as the sum of intermodule plus intramodule complexity. In this simple model, all complexity resides in one or the other of these two components; hence, they are additive. These complexity components correspond to the structured design concepts of module strength (or cohesion) and coupling defined by Stephens et al. [6].

2.2 Relative Complexity

Because projects (and designs) vary greatly in terms of magnitude, a measure of relative complexity ultimately may prove more useful than total complexity. Dividing by the number of modules defined in the design normalizes these complexity measures for size so that designs of different magnitudes may be compared:

$$C = S + L \qquad (2)$$

where

$C = C\sim/n$ (relative design complexity)

$S = S\sim/n$

$L = L\sim/n$

n = number of modules in system

Although individual modules may vary greatly in size in terms of lines of source code, the module, as it is used here, is the unit of design. Hence it is the appropriate normalization factor. The rest of this discussion will concern relative complexity.

3. DEFINITION OF COMPLEXITY MEASURES

The next sections define measures for each of the two components of relative complexity just identified in Equation 2. The measures incorporate counts in the design characteristics (calls, variables, and modules) identified in the model. (Table 1 summarizes some design measures from the modules studied in this analysis). The following sections also discuss methods and consequences of minimizing complexity as defined by this model.

Table 1. Design Measures Summary

	Minimum	Mean	Maximum
Module size	1	66	603
Fanin	1	1.3	16
Fanout	0	2.8	27
I/O variables	1	24	237
Level	2	6.1	11

Note: Based on 1,142 newly developed modules.

3.1 Structural Complexity

Structural complexity derives from the relationships among the modules of a system. The most basic relationship is that a module may call or be called by another module. The structurally simplest system consists of a single module. For more complex systems, structural complexity is the sum of the contributions of the component modules to structural complexity. These potential contributions are occurrences of fanin and fanout as noted by Henry and Kafura [9], as well as by Belady and Evangelisti [3]. (Fanin is the count of calls to a given module. Fanout is the count of calls from a given module.)

In the SEL data analyzed (see Table 1), multiple fanin generally confined itself to modules that were simple mathematical functions reused throughout the system. Consequently, fanin did not prove to be an important complexity discriminator. On the other hand, fanout proved to be highly sensitive, as indicated in a previous study [10]. Counting fanout only also ensures that each connection is counted exactly once. Note that lower fanout indicates less coupling in the sense that there are fewer couples (without regard to their strength [11] or type [6]).

According to this model, a module with a fanout of zero contributes nothing to structural complexity. However, the distribution of fanout within a system also affects complexity. The interconnection matrix representation of partitioning used by Belady and Evangelisti [3] suggests that complexity increases as the square of connections (fanout). All descendents of a given module are connected to each other by their common parent. Then, for a fixed total fanout, a system in which invocations are concentrated in a few modules is more complex than one in which invocations are more evenly distributed. These considerations lead to the following formulation for structural complexity:

$$S = \frac{\Sigma f_i^2}{n} \qquad (3)$$

where

S = structural (intermodule) complexity

f_i = fanout of module "i"

n = number of modules in system

This quantity is the average squared deviation of actual fanout from the simplest structure (zero fanout). Henry and Kafura's term "(fanin * fanout) ** 2" [9] reduces to fanout-squared when fanin is assumed equal to one (the nominal case). Similarly Belady and Evangelisti's measure of complexity [3] is a function of the number of nodes (modules) and edges (fanout) in a system or cluster (partition).

2-13

The fanout count defined here does not include calls to system or standard utility routines, but does include calls to modules reused from other application programs. A reused module must be examined by the designer to determine its appropriateness—as opposed to standard utilities that are well understood by developers.

3.2 Local Complexity

The internal complexity of a module is a function of the amount of work it must perform. The workload consists of data items that are input to or output from higher or parallel modules. This definition is consistent with Halstead's concept [1] of the minimal representation of a program as a function (single operator) with an associated set of I/O variables (operands). This workload measure parallels the idea of actual data bindings as used by Hutchens and Basili [11].

Then, to the extent that functionality (work) is deferred to lower levels, the internal complexity of a module is reduced. Averaging the internal complexities of a systems's component modules produces its local complexity. Most guidelines for decomposition suggest decomposing into units of equal functionality. Assuming, for simplicity, that the workload of a module is evenly divided among itself and subordinate modules leads to the following formulation of complexity:

$$L = \frac{\Sigma \frac{v_i}{f_i + 1}}{n} \qquad (4)$$

where

L = local (intramodule) complexity

v_i = I/O variables in module "i"

f_i = fanout of module "i"

n = number of new modules in system

The "+1" term represents the subject module's share of the workload (incidentally, it prevents the divide-by-zero condition from arising when a module has no fanout). I/O variables include distinct arguments in the calling sequence (an array counts as one variable) as well as referenced COMMON variables. An earlier study [10] indicates that the presence of unreferenced COMMON variables does not affect module quality. Data item complexity is not considered here (only newly developed modules enter into this computation).

Henry and Kafura [9] used the count of source lines of code to represent intramodule complexity. However, as used in Henry and Kafura [9], no matter how large the module, its complexity would be zero if it had no fanout. Basili et al. [12] showed source lines of code (size) to be highly correlated ($r = 0.79$) with the number of I/O variables (operands). Another earlier study [13] shows that high-strength modules [6] tend to be small. Consequently, the local complexity measure may be an indicator of average module strength (or cohesion [6]).

3.3 Minimizing Complexity

Design complexity, as defined in the preceding sections, can be minimized by minimizing its structural and local components. However, these components are not independent. Both measures include fanout. Minimizing structural complexity requires minimizing the fanout from each module. For a given number of both modules and total fanout, structural complexity is minimized when fanout is evenly distributed across all modules (except terminal nodes, of course). On the other hand, local complexity can be minimized by maximizing fanout or minimizing variable repetition.

Repetition occurs whenever a data item appears in more than one module as a calling sequence argument or referenced common variable. Internal uses (including CALLs to other modules) do not count as repetition. In general, minimizing local complexity will produce smaller modules (in terms of executable statements), but is also may increase structural complexity disproportionately. For a given module with a fixed number of I/O variables, the fanout that contributes minimum complexity can be determined as follows:

$$c = f^2 + v/(f + 1)$$

where

c = contribution of given module to total complexity per Equations 2, 3, and 4

then

$$dc/df = 2f - v/(f + 1)^2$$

at minimum

$$0 = 2f - v/(f + 1)^2$$

then

$$v = 2f(f + 1)^2 \qquad (5)$$

Figure 2 shows a plot of Equation 5 as a step function (to reflect the discrete natures of v and f). It identifies the fanout that minimizes complexity for possible counts of I/O variables. For example, in the range from about 100 to 200 I/O variables, complexity is minimized with a fanout of 3. Since very few modules include as many as 200 I/O variables, the plot indicates that the commonly accepted range of values for fanout (up to 7 $\pm$ 2) is much too large. Curtis [7] suggests that the popularity of this bound derives from a misunderstanding of certain psychological studies. This implication is consistent with
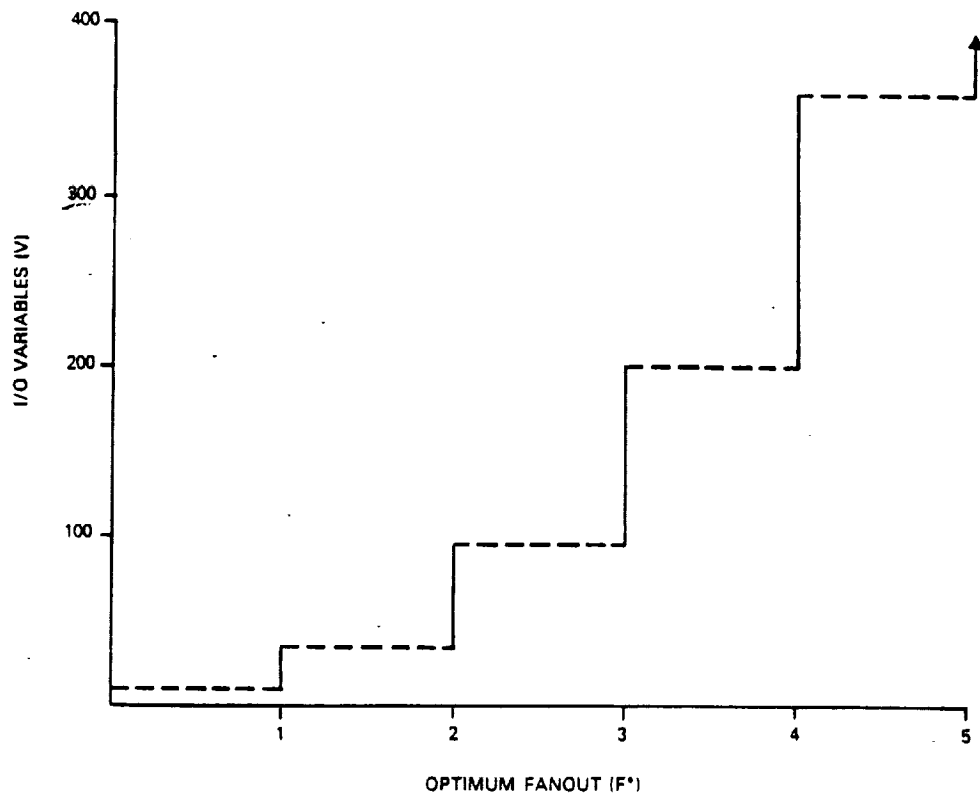
Figure 2. Selecting fanout to minimize complexity.

an earlier study [10]. Furthermore, Constantine [6] observes that most programs can be decomposed effectively into a common structure of three parts: input, process, and output. Larger fanouts may indicate too rapid decomposition. This result suggests than a fanout of one is a reasonable value for modules with few I/O variables.

In addition to the selection of an appropriate fanout, design complexity can also be minimized by reducing variable repetition, i.e., by not including variables where they are not needed. Rigorous application of the principle of information hiding [14] should reduce variable repetition and, hence, local complexity.

Figure 3 shows two design segments of equal structural complexity: The number and distribution of fanouts are identical. Each data couple represents a repetition of the variable "X". Figure 3a traces this variable through a design following strict topdown decomposition rules. "X" appears in the higher level modules (A, B, D) as well as in the lower level modules (C, E). Figure 3b shows an alternative design with a horizontal transfer of data that bypasses the higher level modules (for the case in which modules A, B, and D do not actually use "X"). The local complexity of the intermediate modules (B, D) in the strict top-down configuration (Figure 3a) exceeds

their counterparts in the alternative design (Figure 3b) because their counts of I/O variables are larger.

Parameter transfer between hierarchically adjacent modules (e.g., from B to A) produces a lower complexity than transfer via a global area when that is as far as the data item goes. For a triplet connection (e.g., from B to A to D), the two approaches have the same complexity ("X" counts twice in each). This implication is consistent with the results of an earlier study [10]. Because this model emphasizes the number of data couples rather than the nature of the coupling mechanism, it penalizes "tramp data" (data passed through but not referenced by a module).

Rotenstreich and Howden [15] argue that both horizontal and vertical data flow are essential to good design. Appropriate use of horizontal transfers prevents data flows from violating levels of abstraction. COMMON blocks provide the only mechanism for horizontal data transfers in FORTRAN. Figure 3 shows that horizontal flows can reduce the magnitude of the local complexity measure in some situations.

Of course, a less complex design might also be

2-15

PROC A (   )

PROC B (X,)   PROC D (X,)

PROC C (X,)   PROC E (X,)
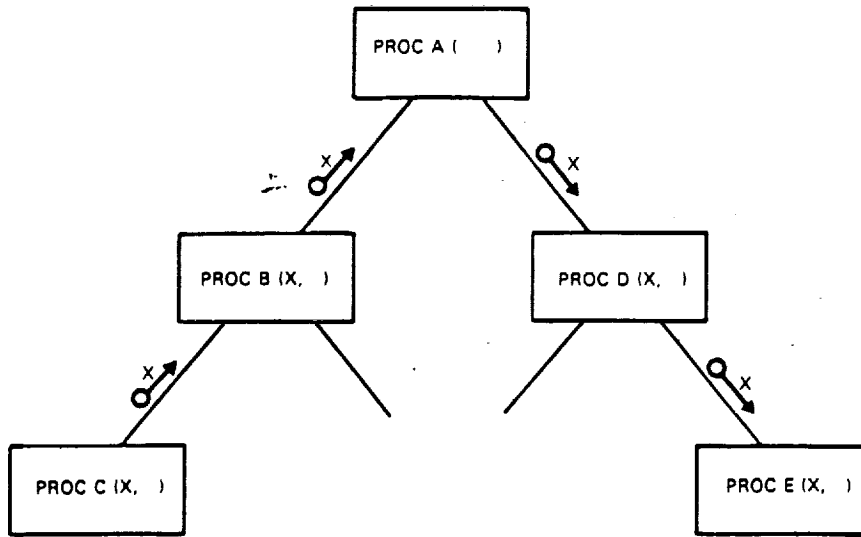
(a) STRICT TOP-DOWN STRUCTURED DESIGN

PROC A (   )

PROC B (   )   PROC D (   )

PROC C (   )   GLOBAL   PROC E (   )
GLOBAL X   AREA   GLOBAL X
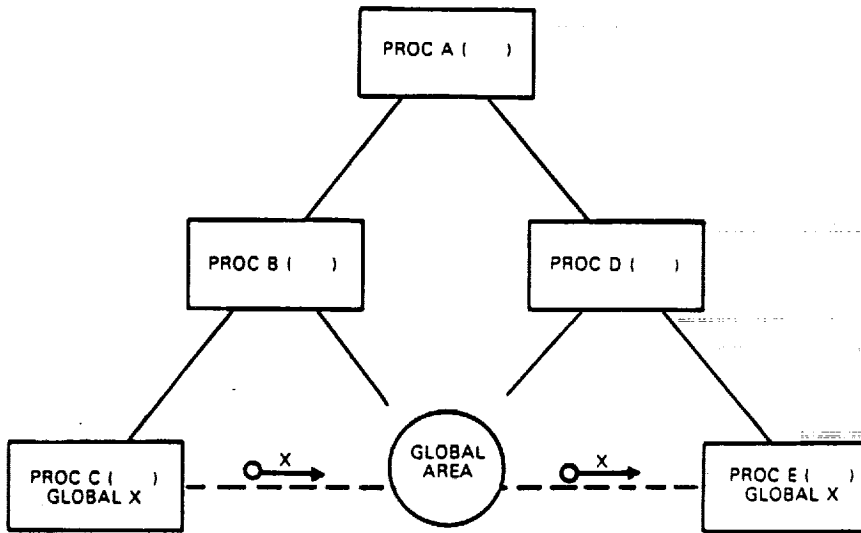
(b) LOWER COMPLEXITY WITH LATERAL TRANSFER

Figure 3. Reducing variable repetition to minimize complexity.

have fanout of one instead of PROC B having fanout of two. Thus, structural complexity diminishes.)

produced by partitioning the work differently and restructuring this design. For example, PROC C could be invoked directly by PROC E (if the nature of the problem permitted). This simpler structure would also be reflected in lower values of the complexity measures defined by this model. (PROCs B and C would each

4. EVALUATION OF COMPLEXITY MEASURES

The value of the complexity measures defined in the preceding sections was evaluated in two ways. First the complexity scores for the eight projects were compared with a subjective rating of design quality using a nonparametric statistical technique. Then the complexity

scores were compared with objective measures of development productivity and error rate. This section presents the results of the two evaluation approaches. Productivity and error rates were computed using the developed lines of code (DLOC) measure as defined by Basili and Freburger [16].

Data for this analysis were extracted from the source code of eight projects by a specially developed analysis tool. However, software developers can easily extract at design time the counts of modules, fanout, and I/O variables necessary to compute these complexity measures. The eight projects studied were ground-based flight dynamics systems for spacecraft in near-earth orbit. Table 2 summarizes some general characteristics of these software systems. The most recent project studied was completed in 1981.

All of these systems were designed and implemented to run under the Graphics Executive Support System (GESS), an interactive graphics interface [17]. Consequently, GESS occupies Level 1 of each design hierarchy. GESS manages most external data interfaces for these systems. It is not included in the complexity calculations.

4.1 Subjective Quality

The eight projects were subjectively ranked in order from best to worst, in terms of design quality, by a senior manager who participated in the development of all eight projects. Then, the four best-rated designs were classified as "good" while the other four were classified as "poor." Table 3 shows the results of that procedure. The table also includes the computed complexity measures. Note that the four designs subjectively rated as "good" also demonstrated the lowest relative complexity. The expert was not provided with specific criteria for "quality," but later reported that perceived "complexity" played a major role in assigning scores.

Table 2. Project Characteristics

Project	Total Modules	Percent Reused[a]	Size (KDLOC[b])	Error Rate[c]	Productivity[d]
A	158	11	50	8.7	3.5
B	203	34	49	8.0	2.9
C	338	32	106	4.5	4.7
D	259	84	37	4.0	4.7
E	327	24	83	4.5	4.8
F	393	47	79	7.1	4.1
G	199	49	57	7.2	2.3
H	245	43	56	6.6	2.4

[a] Percent of total modules.
[b] Thousands of developed lines of code.
[c] Errors per KDLOC.
[d] Developed lines of code per hour.

Table 3. Design Complexity and Quality

	Complexity				
Project	S	L	C[a]	Design Rating[b]	Quality Class
A	24.6	8.2	32.8	5	Poor
B	15.8	9.5	25.3	2	Good
C	11.8	12.1	23.9	3	Good
D	18.4	4.9	23.3	1	Good
E	12.6	10.0	22.6	4	Good
F	22.3	7.3	29.6	6	Poor
G	18.3	10.8	29.1	8	Poor
H	19.2	7.3	26.5	7	Poor

[a] C = S + L as previously defined (Equation 2).
[b] Subjective evaluation (1 = best, 8 = worst).

Although the correspondence between subjective design rating and numerical design complexity is not one-for-one, if the data are viewed as quality classes, they provide persuasive evidence for a relationship. (If one uses the Wilcoxon rank sum statistic the probability is less than 0.02 that the observed good/poor grouping could occur by chance alone.) The objective complexity measure appears to capture much of the information that a human observer includes in a subjective evaluation of design quality.

4.2 Performance Prediction

The other test of the value of these complexity measures is their ability to predict software development performance in terms of the productivity and error rate ultimately realized by the development team. A more complex design will be more difficult to develop into an acceptable system. However, let us first define a few relevant quantities:

Developed lines of code—all newly developed source lines of code plus 20% of reused source lines of code [16].

Errors—conceptual mistakes in design or implementation. An error may result in one or more faults (code changes). These were detected during integration and system testing (after unit testing).

Effort—hours of work by programmers, managers, and support personnel directly attributable to a project.

Productivity—developed lines of code divided by effort (in hours).

Error rate—total errors divided by developed lines of code.

The developed lines of code metric attempts to account for the lower cost and error rate attributable to reused code. Table 2 shows the developed lines of code,
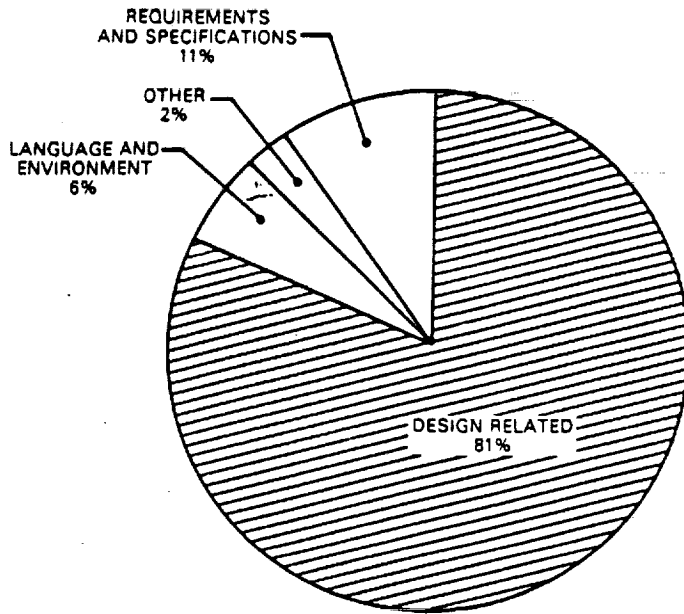
Figure 4. Source of software errors (from Weiss and Basili [19]). Note: Excludes clerical/transcription errors.

productivity, and error rate for the eight projects. Note that together these projects represent more than, 1,000 individual new modules produced by about 50 different programmers.

Designers and researchers commonly assume that higher complexity increases the propensity for error. Potier et al. [18] observe that the implementation process consists largely of translating design specifications into a programming language. It usually does not add complexity to a system. Weiss and Basili [19] show that the bulk (74–82%) of all nonclerical errors reported in three of these projects were related to design, although sometimes at very detailed levels. Figure 4 shows the median distribution of errors for the projects studied by Weiss and Basili [19]. Very few of these errors are true programming errors. Of course, many detailed design and implementation errors are detected during code reading and unit testing (not counted here). In this context, clerical/transcription errors can be regarded as random.

Figure 5 illustrates the relationship between design complexity and error rate. It shows that design complexity effectively predicts the total error rate for development projects. Complexity (as measured here) accounts for fully 60% of the variation in error rate. As seen in Figure 5, all but one of the points lie very close to the regression line. In that case, Project B, the implementation team consisted of an unusually large proportion of junior personnel (although its design team was comparable to those of the other projects). Consequently, it

seems reasonable to find a higher error rate than would be indicated by design complexity alone.

Figure 6 illustrates the relationship between design complexity and productivity. No clear relationship emerges. However, as noted elsewhere [20], many important factors external to the development process (such as computer use and programmer expertise) strongly affect productivity. In this case (consistent with [20]), computer-hours-per-thousand-developed-lines of code correlates strongly with the residuals from the Figure 6 relationship ($r = -0.79$). Computer support was only provided to these projects for detailed design, coding, and testing, so it does measure a different set of activities. However, the small sample size (at the project level) inhibits evaluation of a more complex model incorporating both complexity and computer use.

In this organization, the design team forms the nucleus of the implementation and test teams. Additional personnel join as they are needed. Thus, the complexity measure provides an early indication of the performance of the development team as well as of the quality of the design. A good design team is likely to be a good implementation and testing team, although productivity may be difficult to predict.

5. REPRESENTATION OF DESIGN STRUCTURE

The numerical quantities defining these complexity measures are the number of modules, fanout, and I/O variables. Table 4 shows the distribution of these
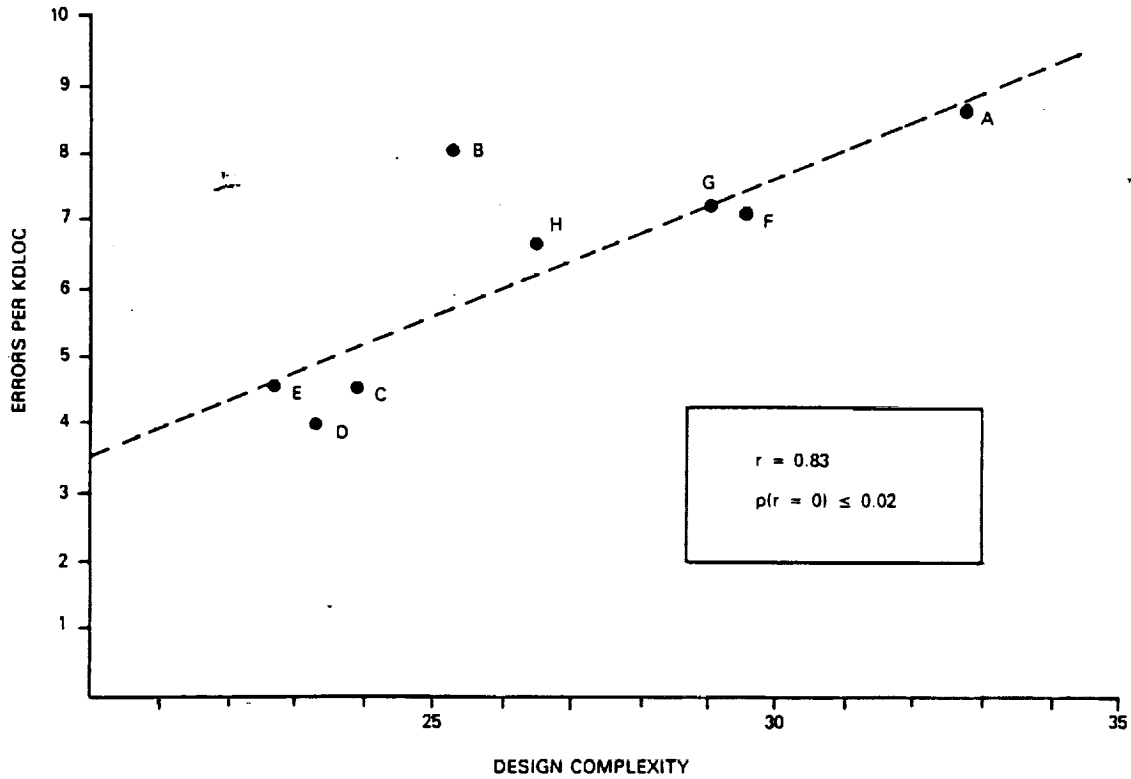
measures by hierarchical level for one project. This design structure can be represented graphically, as shown in Figure 7, by plotting the cumulative percentage of these quantities obtained at each level. Kafura and Henry [21] employed a similar technique to show the effect of design changes on complexity.

In this and subsequent plots, the design structure (or profile) is simplified by combining all utility modules, regardless of where they are invoked, into a single deepest level of the design. That point is not plotted (utility refers to new or reused modules that are invoked from several different points within a design but not to system or standard utilities). Levels greater than or equal to 10 also are combined into a single level to facilitate plotting.

As discussed earlier, the conditions that minimize structural complexity result in an even distribution of fanout. This produces an increasing growth rate in the cumulative percentage of total fanout in the initial levels of the design, followed by a gradual decrease in growth rate as subtrees terminate. The percentage of modules is driven by the fanout at the preceding level (minus calls to utilities). Uneven use of utilities causes the module line to fail to track fanout. Equation 5 showed that I/O

variables should be proportional to fanout in order to minimize local complexity. Together, these conditions define the shape of a good (low relative complexity) design.

Figure 7 illustrates Project E, the design with the lowest relative complexity. It shows three closely fitted "S" shaped curves. Figure 8 illustrates Project A, the design with the highest relative complexity. It shows three separate and irregular lines. Profiles of the other six projects fall in-between these two extremes in correspondence to their measured complexity.

6. CONCLUSIONS

The complexity measures proposed in this report are supported by substantial empirical evidence. The structural complexity component is similar to measures used successfully by Belady and Evangelisti [3] and Henry and Kafura [9] for other languages and application areas. However, neither of these models, as originally formulated, fit the SEL data very well. The new model
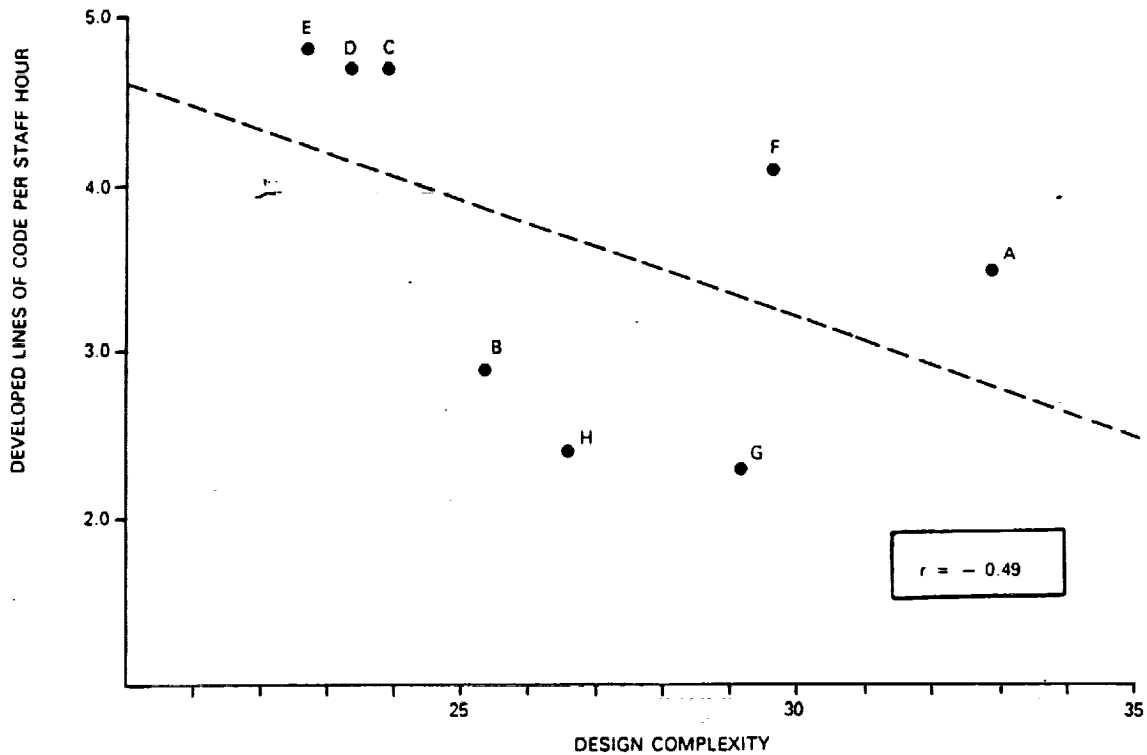
Figure 6. Relationship to productivity.

demonstrated good agreement with subjective assessments of design quality as well as a numerical measure of error rate. Moreover, all relevant measures can be extracted at design time; the Henry and Kafura model includes a code measure.

Table 4. Detailed Design Structure for Project E

Level	Modules	Executable statements	Fanout	Input/output variables
		Module Average		
2	2	91	6.5	45
3	4	37	4.8	9
4	19	59	5.6	29
5	93	67	2.2	26
6	62	59	2.0	24
7	54	59	1.8	20
8	33	37	1.4	14
9	7	19	0.7	8
≥ 10	2	8	0.0	5
Utility	51	90	2.4	21

Many software development methods, e.g., [22], encourage trying design alternatives. Because software developers can easily compute values for these complexity measures at design time, they seem likely to be useful for assessing design quality and comparing design alternatives before committing a design to code. Overall high-complexity designs, as well as individual high-complexity modules, can be identified. These measures could be adapted to support a measures-guided methodology such as that proposed by Ramamoorthy et al. [23].

Of course, complexity is not the only important attribute of software designs. The minimum complexity that can be achieved depends on the nature of the application and the presence of design constraints. Furthermore, design is not a deterministic process. The same design approach or method applied by different individuals can result in different designs. These complexity measures help us to answer the question, "Which is better?" However, it is not enough to produce a design that shows low complexity scores. Following a sensible and well-defined design method ensures that the design problem is responded to while minimizing complexity. Measures play a supporting role in the design process.
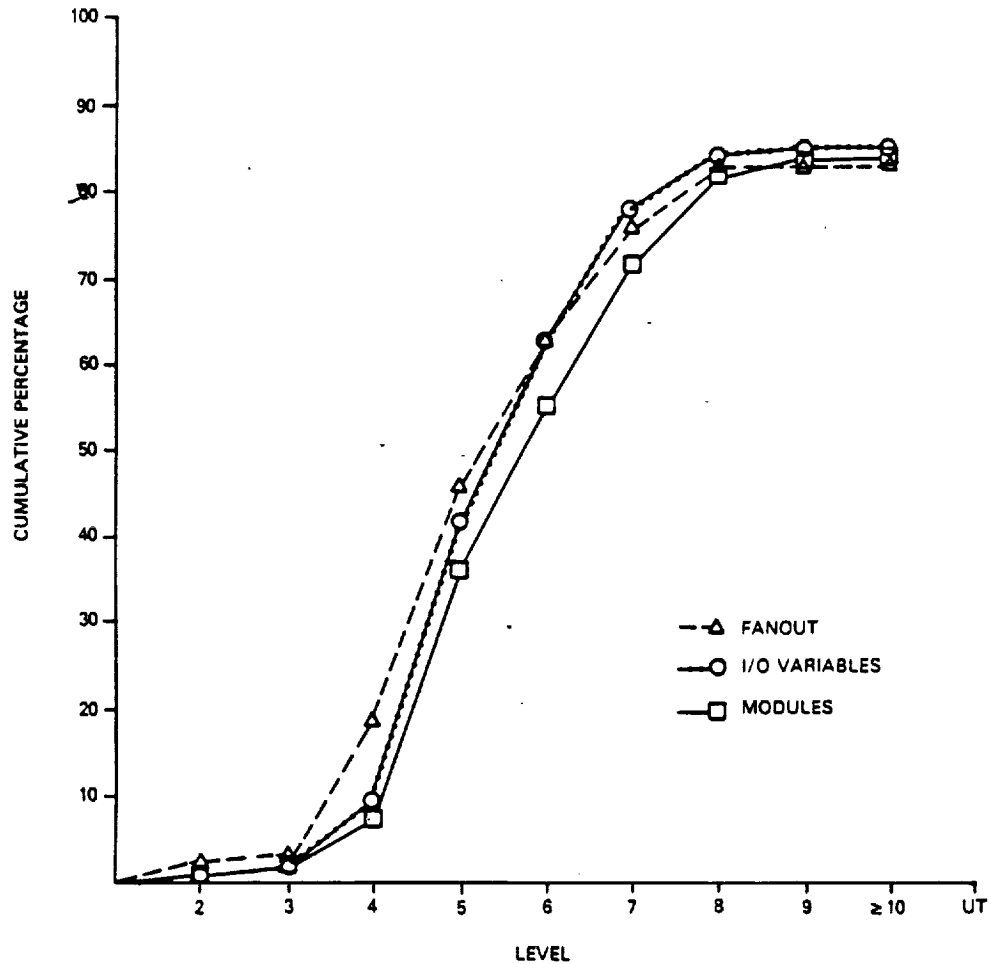
Figure 7. Design profile of Project E (lowest complexity).

As Kearney et al. pointed out [24], ill-founded reliance on complexity measures can degrade the software development process by rewarding poor programming practices. The approach to complexity measurement presented here satisfies the requirements of Kearney et al. [24] for effective complexity measures by clearly identifying the attributes measured, deriving them from a model of the design process, suggesting how they can be used in practice, and empirically testing their validity. Nevertheless, more work remains to be done.

Three aspects of this current complexity measurement approach require additional research. First, methods of incorporating external I/O (e.g., files) into the complexity measures must be developed. In the systems studied, much of the external I/O is handled by the GESS standard interface. Second, the application of the measures

should be extended to designs using different formalisms intended for different implementation languages. "Modules" corresponding to FORTRAN subroutines are not a universal design structure. The SEL has begun to study the application of these measures to Ada design [25]. Third, the existence of two design complexity components suggests that two different types and distributions of the design errors (in addition to programming errors) also exist, as proposed by Basili and Perricone [26]. That needs to be verified empirically.

Finally, Kafura and Reddy [27] showed that similar complexity measures appeared to related to software maintainability. This suggests another new area of investigation.

5207
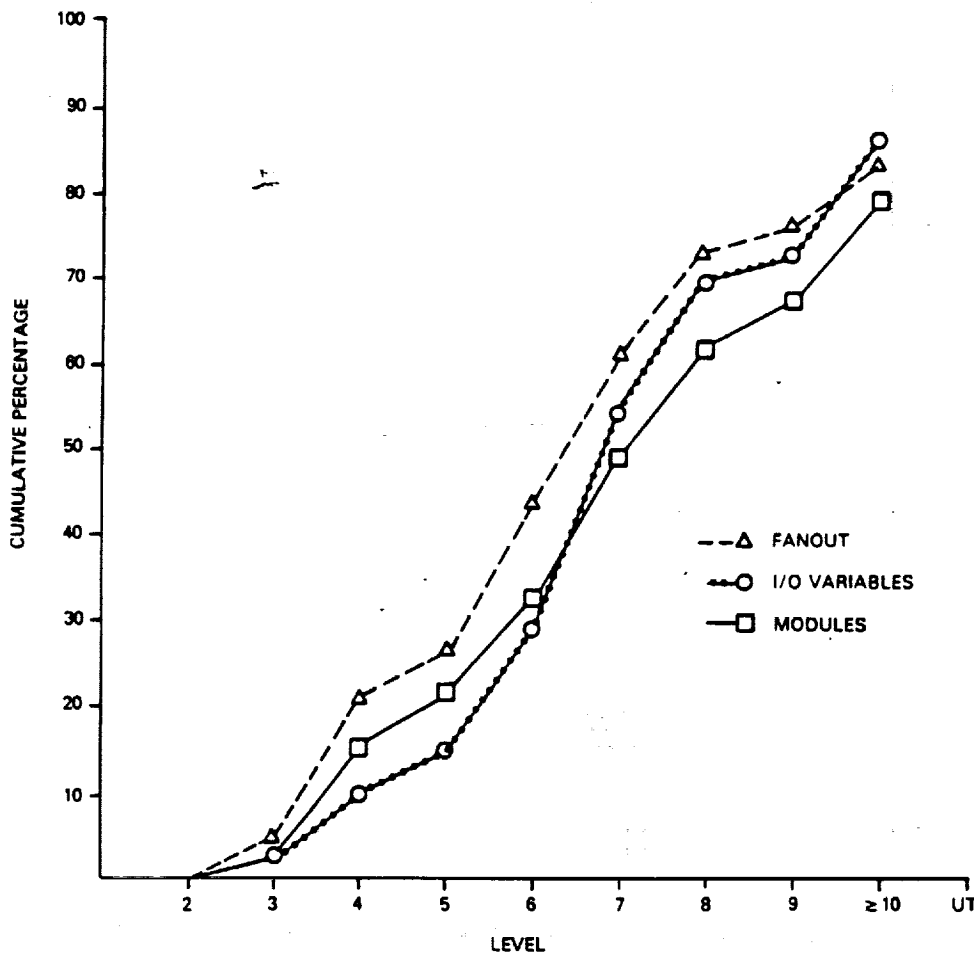
Figure 8. Design profile of Project A (highest complexity).

REFERENCES

1. M. H. Halstead, *Elements of Software Science*, Elsevier Science Publishing, New York, 1977.
2. T. J. McCabe, A Complexity Measure, *IEEE Trans. Software Engineering* 2, 308-320 (1976).
3. L. A. Belady and C. J. Evangelisti, System Partitioning and Its Measure, *J. Systems Software* 2, 23-39 (1981).
4. D. A. Troy and S. H. Zweben, Measuring the Quality of

Structured Designs, *J. Systems Software* 2, 113-120 (1981).
5. D. N. Card, F. E. McGarry, G. T. Page, et al., *The Software Engineering Laboratory*, NASA/GSFC, SEL-81-104, 1982.
6. W. P. Stevens, G. J. Myers, and L. L. Constantine, Structured Design, *IBM Systems J.* 2, 115-139 (1974).
7. B. Curtis, In search of software complexity, in *Proceedings, IEEE Workshop on Quantitative Software Models*, Computer Society Press, New York, 1979, pp. 95-106.
8. G. Booch, Object Oriented Design, *IEEE Trans. Software Engineering* 12, 211-221 (1986).
9. S. M. Henry and D. G. Kafura, Software Structure Metrics Based on Information Flow, *IEEE Trans. Software Engineering* 7, 510-518 (1981).
10. D. N. Card, V. E. Church, and W. W. Agresti, An Empirical Study of Software Design Practices, *IEEE Trans. Software Engineering* 12, 264-271 (1986).
11. D. H. Hutchens and V. R. Basili, System Structure

Analysis: Clustering with Data Bindings, *IEEE Trans. Software Engineering* 11, 749-757 (1985).

12. V. R. Basili, R. W. Selby, and T. Phillips, Metric Analysis and Data Validation Across FORTRAN Projects, *IEEE Trans. on Software Engineering* 9, 652-663 (1983).

13. D. N. Card, G. T. Page, and F. E. McGarry, Criteria for software modularization, in *Proceedings, IEEE Eighth International Conference on Software Engineering*. Computer Society Press, New York, 1985, pp. 372-377.

14. D. L. Parnas, On the Criteria to be Used in Decomposing Systems into Modules," *Commun..ACM* 15, 1053-1058 (1972).

15. S. Rotenstreich and W. E. Howden, Two-Dimensional Program Design, *IEEE Trans. Software Engineering* 12, 377-384 (1986).

16. V. R. Basili and K. Freburger, Programming Measurement and Estimation in the Software Engineering Laboratory, *J. Systems Software* 2, 47-57 (1981).

17. Computer Sciences Corporation, CSC/SD-75/6057, *Graphics Executive Support System User's Guide*, 1975.

18. D. Potier, J. L. Albin, R. Ferreol, and A. Bilodeau, Experiments with computer software complexity and reliability, in *Proceedings, IEEE Sixth International Conference on Software Engineering*. Computer Society Press, New York, 1982, pp. 94-101.

19. D. M. Weiss and V. R. Basili, Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory, *IEEE Trans. Software Engineering* 11, 157-168 (1985).

20. D. N. Card, F. E. McGarry, and G. T. Page, Evaluating Software Engineering Technology, *IEEE Trans. Software Engineering*, 13, 845-851 (1987).

21. D. G. Kafura and S. M. Henry, Software Quality Metrics Based on Interconnectivity, *J. Systems Software* 3, 121—131 (1982).

22. S. Steppel, T. L. Clark, et al., *Digital Systems Development Methodology*, Computer Sciences Corporation, 1984.

23. C. V. Ramamoorthy, W. Tsai, T. Yamaura, and A. Bhide, Metrics guided methodology, in *Proceedings, IEEE Ninth International Conference on Software and Applications*. Computer Society Press, New York, 1985, pp. 111-120.

24. J. K. Kearney, R. L. Sedlmeyer, W. B. Thompson, et al., Software Complexity Measurement, *Commun. ACM* 29, 1044-1058 (1986).

25. W. W. Agresti, V. E. Church, et al., Designing with Ada for satellite simulation: A case study, in *Proceedings of the First International Conference on ADA Applications for the NASA Space Station*, 1986, pp. F.1.3.1-14.

26. V. R. Basili and B. T. Perricone, Software Errors and Complexity: An Empirical Investigation," *Commun. ACM* 27, 42-52 (1984).

27. D. G. Kafura and G. R. Reddy, The Use of Software Complexity Metrics in Software Maintenance, *IEEE Trans. Software Engineering* 13, 335-343 (1987).

28. D. N. Card and W. W. Agresti, Resolving the Software Science Anomaly, *J. Systems Software* 7, 29-35 (1987).

QUANTITATIVE ASSESSMENT OF MAINTENANCE:
An Industrial Case Study

H. Dieter Rombach and Victor R. Basili

University of Maryland
Dept. of Computer Science
College Park, MD 20742, USA
(301) 454-8974

Abstract

In this paper we discuss a study aiming at the improvement of measurement and evaluation procedures used in an industrial maintenance environment. We used a general evaluation and improvement methodology for deriving a set of metrics tailored to the maintenance problems in this particular environment. Some of the required maintenance data were already collected in this environment, others were suggested to be collected in the future. We discuss the general measurement, evaluation and improvement methodology used, the specific maintenance improvement goals important to this environment, the set of metrics derived for quantifying those goals, the suggested changes to the current data collection procedures, and preliminary analysis results based on a limited set of already available data. It is encouraging that based on this limited set of data we are already able to demonstrate benefits of the proposed quantitative approach to maintenance. Finally, we outline ideas for automating the discussed approach by a set of measurement and evaluation tools. This paper emphasizes the steps of introducing such a quantitative maintenance approach into an industrial setting rather than the environment-specific analysis results. The analysis results are intended to demonstrate the practical applicability and feasibility of the proposed methodology for evaluating and improving maintenance aspects in an industrial environment.

1. Introduction

In this paper we present results from a study trying to introduce sound measurement and evaluation procedures into an industrial maintenance environment. The goal of the study has been to investigate the company's needs for quality assessment, and the suitability of the error, change, and effort data already collected in this environment for addressing these quality assessment needs.

First we describe the actual industrial maintenance environment which has been the object of this study including the high-level maintenance assessment and improvement goals as stated by high-level management (section 2) and the goal/question/metric paradigm[1, 5, 7] used in this study for defining and quantifying the maintenance assessment and

improvement goals of interest. The application of this methodology has resulted in a list of clearly defined maintenance assessment and improvement goals and quantifiable questions (section 4) as well as the corresponding data and metrics (section 5). Until now only a subset of these data and metrics required to fully address the stated maintenance goals had been collected (section 6). Based on the needs of the particular industrial environment changes to the data collection and validation process have been suggested for the future (section 7). Preliminary analysis results for a small subset of the questions and goals of interest (depending on the type, amount and quality of data available at the time) are presented (section 8). It is encouraging that based on this limited subset of data we are already able to demonstrate benefits of this quantitative approach to maintenance. Finally, we outline ideas for automating the proposed approach by a set of measurement and evaluation tools (section 9). This paper emphasizes the steps of introducing such a quantitative maintenance approach into an industrial setting rather than the environment-specific analysis results. The analysis results are only included to demonstrate that the proposed approach actually works in this particular environment.

2. Maintenance Environment

The study was conducted in the maintenance environment of a major computer company. The maintenance process from an organizational point of view can be characterized as follows: Customer Support receives maintenance problems (mainly) from customers, evaluates them and, whenever appropriate forwards them in the form of change requests to Product Assurance. Product Assurance evaluates the change requests again and forwards them, whenever appropriate, to Engineering. The eventually changed products are sent back to the customer(s) through the same channels (Product Assurance, Customer Support).

Data are currently being collected during all these different maintenance steps. Customer Support collects data for each single problem concerning scheduling (e.g., time of incoming calls, time of outgoing calls), type of problem (e.g., clarification of documentation, operation request; for a complete list see table 2), priorities of problems, and effort spent on handling the problem. Product Assurance collects data for each single change request concerning scheduling, type of change request, effort spent, and final status (e.g., changed, change postponed, change rejected including the reason for

5207

rejection). Engineering collects data for each change concerning scheduling, change effort, and the type of change performed. Data collection is mandatory in some groups such as Product Assurance; it is done on a voluntary basis in other groups such as Engineering. Based on this fact the completeness and validity of collected data varies across the entire maintenance environment. In general it is true that Customer Support and Product Assurance stress data collection more than Engineering does.

Although this is a very simplified description of the maintenance process it should allow the reader to understand the different needs of these three different maintenance roles as far as assessment needs are concerned.

The data were used for filing status reports concerning the handling of maintenance requests but not (except locally in some groups) for overall quality assessment. The purpose of this study was to find out whether the already collected data are sufficient for assessing the environment specific maintenance problems and, if not, to suggest changes of this data collection process.

The most urgent maintenance assessment and improvement goals were formulated by corporate representatives of the company as follows:

G1: Examine where the bulk of the company's maintenance dollars are being spent and how much is being spent on individual activities.

G2: Identify the best ways of applying the 20/80 rule[*] to get the biggest savings and return on our maintenance dollars.

G3: Identify criteria for when a product is ready for release.

G4: Identify features of product, documentation or support that provide a wider customer satisfaction.

G5: Identify criteria for when a software product should be rewritten rather than maintained.

G6: Identify metrics of customer satisfaction that can be developed based upon existing data.

G7: Develop organizational guidelines for integrating software quality metrics into the company's framework of design, development, and support.

It is obvious that these high-level and complex problems can only be assessed by breaking them down into more and more simple problems. This refinement process, which finally is expected to result in a set of quantitative metrics, is supported by a methodology developed by the authors[1,5,7].

3. The Goal/Question/Metric Paradigm

The approach to quantification of goals is the goal/question/metric paradigm[1,5,6,7]. This paradigm does not provide a specific set of goals but rather a framework for defining goals and refining them into specific quantifiable questions about the software process and product that provide a specification for the data needed to help answering the goals.

The paradigm provides a mechanism for tracing the goals of the collection process, i.e. the reasons the data are being collected, to the actual data. It is important to make clear, at least in general terms, the organization's needs and concerns,

[*] Applying the 20/80 rule means to identify those maintenance problems which can be fixed easily (with twenty percent of the effort of what would be required to fix all maintenance problems) but reduce the maintenance overhead drastically (by eighty percent).

the focus of the current project and what is expected from it. The formulation of these expectations can go a long way towards focusing the work on the project and evaluating whether the project has met those expectations. The need for information must be quantified whenever possible and the quantification analyzed as to whether or not it satisfies the needs. This quantification of the goals should then be mapped into a set of data that can be collected on the product and the process. The data should then be validated with respect to how accurate it is and then analyzed and the results interpreted with respect to the goals.

The actual goal/question/metric paradigm is visualized in figure 1.
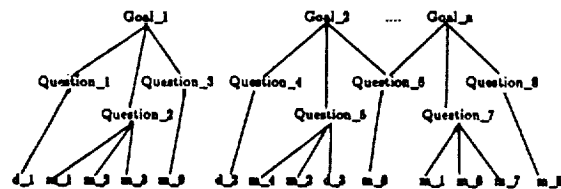
Figure 1: Goal/Question/Metric Paradigm.

Here there are n goals shown and each goal generates a set of questions that attempt to define and quantify the specific goal which is at the root of its goal tree. The goal is only as well defined as the questions that it generates. Each question generates a set of metrics (m_i) or distributions of data (d_i). Again, the questions can only be answered relative to and as completely as the available metrics and distributions allow. As is shown in figure 1, the same questions can be used to define different goals (e.g. Question_6) and metrics and distributions can be used to answer more than one question (e.g. m_1 and m_2). Thus questions and metrics are used in several contexts.

Given the above paradigm, the **process of quantifying improvement goals** consists of three steps:

(1) Generate a set of goals based upon the needs of the organization.
The first step of the process is to determine what it is you want to improve. This focuses the work to be done and allows a framework for determining whether or not you have accomplished what you set out to do. Sample goals might consist of such issues as on how to improve the set of methods and tools to be used in a project with respect to high quality products, customer satisfaction, productivity, usability, or that the product contains the needed functionality.

(2) Derive a set of questions of interest or hypotheses which quantify those goals.
The goals must now be formalized by making them quantifiable. This is the most difficult step in the process because it often requires the interpretation of fuzzy terms like quality or productivity within the context of the development environment. These questions define the goals of step 1. The aim is to satisfy the intuitive notion of the goal as completely and consistently as possible.

(3) Develop a set of data metrics and distributions which provide the information needed to answer the questions of interest.

In this step, the actual data needed to answer the questions are identified and associated with each of the questions. However, the identification of the data categories is not always so easy. Sometimes new metrics or data distributions must be defined. Other times data items can be defined to answer only part of a question. In this case, the answer to the question must be qualified and interpreted in the context of the missing information. As the data items are identified, thought should be given to how valid the data item will be with respect to accuracy and how well it captures the specific question.

In writing down goals and questions, we must begin by stating the purpose of the improvement process. This purpose will be in the form of a set of overall goals but they should follow a particular format. The format should cover the purpose of the process, the perspective, and any important information about the environment. The format (in terms of a generic template) might look like:

- **Purpose of Study:**
  To (characterize, analyze, evaluate, predict, motivate) the (process, product, model, metric) in order to (understand, assess, manage, engineer, learn, improve) it.

- **Perspective of Study:**
  Examine the (cost, effectiveness, correctness, errors, changes, product metrics, process metrics, reliability, user satisfaction, etc.) from the point of view of the (developer, manager, customer, corporate perspective, etc).

- **Environment of Study:**
  The environment consists of the following: process factors, people factors, problem factors, methods, tools, constraints, etc.

- **Process Questions:**
  For each process under study, there are several subgoals that need to be addressed. These include the quality of use (characterize the process quantitatively and assess how well the process is performed, the domain of use (characterize the object of the process and evaluate the knowledge of object by the performers of the process), effort of use (characterize the effort to perform each of the subactivities of the activity being performed), effect of use (characterize the output of the process and the evaluate the quality of that output), and feedback from use (characterize the major problems with the application of the process so that it can be improved).

  Other subgoals involve the interaction of this process with the other processes and the schedule (from the viewpoint of validation of the process model).

- **Product Questions**
  For each product under study there are several subgoals that need to be addressed. These include the definition of the product (characterize the product quantitatively) and the perspective of the evaluation (e.g. reliability or user satisfaction). The definition of the product includes physical attributes (e.g. source lines, number of units, executable lines, control and data complexity, programming lan-

guage features, time space), cost (e.g. effort, time, phase, activity, program), changes (e.g. errors, faults, failures and modifications by various classes), and the context the product is supposed to be used in (e.g. customer community, operational profile). The perspective of the evaluation is relative to a particular quality (e.g. reliability or user satisfaction). Thus the physical characteristics need to be analyzed relative to this quality aspect.

4. Maintenance Goals and Questions

We applied the methodology described in section 3 to specify the high-level quality assessment and improvement goals given to us from a corporate perspective (see section 2) more precisely, and to derive quantifiable analysis questions. Using the template of section 3 proved to be very helpful. The entire process of specifying goals and deriving the evaluation questions was done in very close cooperation with company representatives from Customer Support, Product Assurance, and Engineering.

The seven goals for this study are formulated in terms of the purpose of this study, the perspective of this study, and important information about the company's maintenance environment:

- PURPOSE OF STUDY: Characterize (in the case of goals G1 and G4) and evaluate (G2, G3, and G5) the maintenance methodology and motivate (G6 and G7) the use of metrics for the purpose of better understanding (G1 and G4), management (G2, G3, G5, G6, and G7) and improvement (G2, G3, G5, G6, and G7).

- PERSPECTIVE: Examine the cost (in the case of goals G1, G2, G5, and G7), problems (G2), errors and changes (G1 and G5), product and process metrics (G3, G4, G5, and G6) and the effectiveness (G7) from the point of view of the manager and corporation

- ENVIRONMENT:
  - Maintenance Process: The customer reports problems (by phone) to the Customer Support; if problems cannot be resolved by Customer Support they are forwarded to Product Assurance. Product assurance decides whether the reported problem should be fixed. If approved as a problem to be fixed it is submitted to engineering (to be fixed), gets back to Product Assurance (for fix certification), and is sent back to Customer Support.

  - Maintained Products (for which we had access to data). A retrieval system (called SYS_1 in the following of this paper) and a compiler (called SYS_2 in the following of this paper).

For each process and product under study, there are several subgoals (quality of use, domain of use, effort of use, effect of use, and feedback of use); each subgoal will be addressed by a number of analysis questions (Qi):

(A) PROCESS RELATED QUESTIONS:

- QUALITY OF USE (characterize the company's maintenance process and how well it is performed):

  Q1 What percent of the problems are handled by Customer Support without forwarding them to Product Assurance? What is a distribution of their disposition?

Q2: What percent of change requests forwarded to Product Assurance do not come from the field? What is a distribution by percent of where they come from (engineering, field test, etc) and the reasons they do not come from field? What percent of problems aren't really maintenance problems?

Q3: For change requests rejected by Product Assurance or Engineering: What are the distributions by

1) closure code,
2) organization responsible for rejection, and
3) schedule by closure code by organization?

Q4: What are characteristics of the test plan performed by engineering before release? How effective is this test plan?
More detailed: Is the test suite based upon the new or changed final requirements? Are regression tests performed? Are the tests based upon the importance and complexity of the requirements? What criteria exist for the selection of test cases and test data?

Q5: What are test cases and test data for the beta test? To what extent does it consider the future usage profile? How effective is this test?

Q6: For each fix: How long after the fix is made is it released to the customer?

Q7: What is the distribution of faults or customer problems per organizational unit in total and by various products?

Q8: What is the distribution of faults due to previous changes per organizational unit in total and by various products?

Q9: What are the distributions of change requests by various subclasses (fault/modification, rejected/not rejected, error subclasses, change subclasses)?

• DOMAIN OF USE (characterise the objects of the maintenance process and the knowledge of the people involved in this maintenance process):

Q10: What products are available to

- customer support personnel,
- problem evaluator,
- changer,
- change evaluator, and
- the field support?

Q11: What is the knowledge of the people involved wrt

1) the application,
2) the particular product, and
3) the change methodology?

• EFFORT OF USE (characterise the effort to perform each maintenance activity):

Q12: What is the cost of

- detecting a problem symptom
- understanding the problem,
- isolating the problem causes,
- designing the change,
- implementing the change,
- testing the change, and
- releasing the change

in terms of computer time, people time, by person category and machine category?

Q13: What is the calendar time for

- detecting a problem symptom,
- understanding the problem from a customer's viewpoint,
- understanding the problem from an engineering viewpoint,

- isolating the problem causes,
- designing the change,
- implementing the change,
- testing the change, and
- releasing the change?

[Give the max, min, average and by various types of changes!]

• EFFECT OF USE (characterise the output of the maintenance process and the quality of this output):

Q14: How many and what percent of documents are produced/modified as a result of the maintenance process (patch, user manual, additional technical documents, closure form, patch release information form, advanced technical information form and user letter)?

Q15: How many and what percent of change requests cause a modification?

Q16: How many and what percent of change requests are related to errors, environment adaptations, and requirements changes (= enhancements)?

Q17: How many and what percent of faults are the result of a previous change?

Q18: What is the average cost of a change overall and by type?

Q19: Having categorized changes by function, having made a change in a function: How many future requests do we get for the same function?

Q20: What are characteristics of customer calls over time by type of question?

Q21: What customer categories exist? Do clusters of customer profiles (types of complaints, faults, etc.) match these categorization schemes?

Q22: Is the user satisfied with function, performance, schedule (by a user satisfaction survey)?

• FEEDBACK FROM USE (characterise the problems with the application of the maintenance process so that it can be improved):

Q23: What are the problem areas in the maintenance process by the following categories:

- distribution of changes by various types,
- distribution of problems that are rejected by various types,
- customer types, and
- time distribution (calendar time, effort) by various change types, problem types, or maintenance activities?

(B) PRODUCT RELATED QUESTIONS:

• DEFINITION OF THE PRODUCT (characterise the product quantitatively):
Q24: What are the physical attributes such as

- size (source lines, number of units, executable lines of code),
- complexity (control, data),
- programming language features,
- time to develop,
- memory space, and
- execution frequency?

Q25: What is the cost, e.g., effort (time per phase, activity)?

Q26: What are distributions of changes, e.g., errors, faults, failures, adaptations, and enhancements by various types

GOALS	QUESTIONS																											
	PROCESS																							PRODUCT				
	QUALITY OF USE									DOMAIN OF USE		EFFORT OF USE		EFFECT OF USE														
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22	Q23	Q24	Q25	Q26	Q27	Q28
G1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
G2	x	x	x	x		x		x	x	x	x	x		x	x	x	x	x	x	x	x	x			x	x	x	x
G3		x			x	x	x	x	x	x		x										x		x		x	x	x
G4	x		x	x	x	x	x	x	x	x			x	x		x	x		x	x	x	x		x	x	x		x
G5					x		x	x	x	x	x	x		x			x		x	x				x	x		x	x
G6	x		x			x	x	x	x	x			x	x		x	x		x	x				x		x	x	x
G7	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Table 1: Goal-Question Matrix

Q27: What is the products context, e.g. customer community, operational profile, life cycle model, etc?

Q28: What are the problem areas in the product by the following categories:

- distribution of changes by various types,
- distribution of problems that are rejected by various types,
- customer types, and
- time distribution (calendar time, effort) by various change types, problem types, or maintenance activities?

Each individual evaluation goal is quantifiable via a subset of these 28 evaluation questions. In table 1 the interrelationship is visualized in form of a goal-question matrix.

5. Maintenance Data & Metrics

In this section we discuss the types of maintenance data which has to be collected in order to answer each of the evaluation questions derived in section 4.

The data (Di) are categorized depending on which maintenance aspect (Customer Support, Product Assurance, or Engineering) is mainly affected. For each data it is indicated whether and how it can be retrieved from currently maintained data bases, i.e., whether it is explicitly available (+), it is not explicitly available, but can be derived from other data with reasonable effort (o), a great deal of effort (oo), or it is not available at all (-).

(1) CUSTOMER SUPPORT ORIENTED MAINTENANCE DATA:

For each problem reported by customers (phone calls):

D1 (+): customer identification
D2 (oo): customer type
D3 (+): customer support center identification
D4 (o): problem description
D5 (+): whether a problem resulted in a change request (Y/N)
D6 (oo): connection between customer problem and change request_number
D7 (+): identification of affected system/product
D8 (-): identification of affected product functions

D9 (+): schedules for each activity associated with a customer problem

(2) PRODUCT ASSURANCE ORIENTED MAINTENANCE DATA:

For each problem reported by a change request:

D10 (+): identification of the organization that filled out the change request (customer support, engineering, field test, etc)
D11 (+): identification of system/product affected
D12 (+): customer identification
D13 (-): customer type
D14 (+): identification of Product Assurance center in charge
D15 (o): concise problem description
D16 (o): information whether a change request was rejected (Y/N)
D17 (o): final change request status (= closure code)
D18 (-): information by whom (Product Assurance, Engineering) closure code was set
D19 (+): schedules for each maintenance activity
D20 (+): information whether it is a fault, adaptation, or enhancement

(3) ENGINEERING ORIENTED MAINTENANCE DATA:

For each actually performed change:

D21 (+): identification of the engineering group in charge
D22 (-): information about fault types (for example: control, data, computation, etc)
D23 (o): information whether a fault was caused by a previous change (Y/N)
D24 (o): information which product units (modules) were affected by a change (in terms of lines_of_code or identification of modules)
D25 (-): effort in computer time in total or per phase, change activity
D26 (-): effort in people time in total or per phase, change activity
D27 (+): schedule for each change activity (in calendar days)
D28 (o): percent of code, documents, forms changed
D29 (o): product size
D30 (o): product complexity
D31 (-): memory space

The following question-data matrix (see table 2) shows which of the 31 different types of data are required as a minimum to answer each of the previously listed 28 questions:

138

	DATA/MEASURES																														
QUESTIONS	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11	D12	D13	D14	D15	D16	D17	D18	D19	D20	D21	D22	D23	D24	D25	D26	D27	D28	D29	D30	D31
Q1			x	x			x																								
Q2										x																					
Q3														x		x	x	x													
(Q4)																															
(Q5)																															
Q6																			x												
Q7		x	x			x					x			x	x		x			x	x	x									
Q8		x	x			x					x			x	x		x			x	x		x								
Q9															x	x				x		x									
(Q10)																															
(Q11)																															
Q12																									x	x					
Q13				x				x																				x			
Q14																													x		
Q15																x															
Q16																x			x												
Q17																		x			x										
Q18																x				x				x	x						
Q19					x	x		x																							
Q20				x																											
Q21	x	x		x								x	x				x														
(Q22)																															
Q23																															
Q24																													x	x	x
Q25																									x	x					
Q26														x		x			x		x		x								
(Q27)																															
Q28																															

Table 2: Question-Data Matrix

The questions enclosed in parenthesis have to be answered purely by subjective data.

The complete refinement process from the original goals over questions to the data/metrics can be traced by combining tables 1 and 2.

6. Availability and Validity of Data

In the previous section it was indicated what data are needed for answering the questions of interest. We also included the analysis results to which degree those data are already available inside the company (+,o,-).

Interpreting the question/data matrix together with the availability and validity of the company's data the following conclusions can be drawn:

.- Questions Q6, Q13, Q15, Q16, Q17, Q20 are completely answerable
- Questions (Q4), (Q5), (Q10), (Q11), (Q22) will not be answered based on data collected via regular data collection forms, but by subjective data from interviews.
- Questions Q23 and Q28 require no data, they are answered by interpreting the results of more basic questions
- All questions related to change effort (Q12, Q18, Q25) can not be answered because (at least in the case of SYS_1 and SYS_2) these data were listed as optional on the data collection form and therefore only listed on about 10% of all forms.
- All other questions are (at least partially) answerable

7. Improvement of Data Collection

Based on the company's interests as documented by the high-level problems (see section 2) and the refined set of evaluation questions (see section 4), and the partial lack of valid data available to analyze those questions, the following recommendations for changing the data collection process are being made:

139

2-29

5207

- A uniform data collection method and data base should be defined..

Some data items are interpreted differently by different people. Each organizational unit inside the the maintenance environment has its own data base format. This fact makes it difficult to assess maintenance problems from global views. It is for example difficult to analyze engineering data from various sites, or the complete life cycle of maintenance problems starting at Customer Support throughout Product Assurance and Engineering.

- A maintenance task should be viewed as a single entity in this data base, and it should be traceable through all its phases (Customer Support, Product Assurance, Engineering). Due to the "bottom-up" development of individual data bases, each data base contains only those data important to the individual organization.

The only solution seems to be a central data base that contains all information concerning each maintenance task starting from the first phone call and ending with its final resolution.

- It is mandatory to collect engineering data (effort in staff_hours).

Engineering data are crucial for determining maintenance problems due to product quality problems (e.g., bad structure).

- Development data (errors, changes, tests, etc.) should be collected.

Collection of development data has to start now. As soon as the identification of the maintenance problems is completed, the impact of product quality and development methodology on these problems has to be analyzed. In order to do this, data characterizing the development process are needed.

9. Preliminary Analysis Results

In order to demonstrate the benefits of quantitative assessment of maintenance we used the data collected at the time to answer some of our maintenance questions listed in section 4. We had data available for two commercial systems SYS_1 and SYS_2 (retrieval system and a compiler). We had maintenance data available from the first two quarters of 1985.

In section 6 we outlined the questions which could be answered based on the data available. In the following we present preliminary analysis results of those questions in the context of the originally posted high-level corporate maintenance problems (1) to (5) as listed in section 2.

(G1) Examine where the bulk of the company's maintenance dollars are being spent and how much is being spent on individual activities:

This goal area can be addressed by the following analysis questions (see section 4):

- Question 20: (What are characteristics of customer calls over time by type of question ?) --> Table 3

  The average number of calls per problem is about 4. The most frequent problems are operation questions, capability features, and clarification of documentation (in the case of SYS_1) or operation fault (in the case of SYS_2). The costly problems (in terms of number of calls) are documentation faults, system software, and operation faults (in the case of SYS_1), and clarification of documentation, capability features, operation questions, and pre-sales requests (in the case of SYS_2).

- Question 1 (What percent of problems are not reported as change requests? What is a distribution of their disposition?) --> Table 4

  Overall only about two percent of all problems recorded by Customer Support resulted in change requests (3 out of 177 for SYS_1, 3 out of 152 for SYS_2).

  The disposition of problems not reported as change requests in terms of "type of call" is as follows:

  The bulk of maintenance problems handled by Customer Support is spent for "operation requests" and "operation faults" in the case of SYS_2; in the case of SYS_1 we can identify two additional problem sources: problems due to faults of underlying layers (systems software and hardware) and problems due to bad documentation (almost 20% of all problems !)

call-type	SYS_1			SYS_2		
	calls	problems	calls/problem	calls	problems	calls/problem
unknown type	5	2 (1.1%)	2.5	-	-	-
clarify document	130	35 (19.8%)	3.7	34	7 (4.6%)	4.9
operation question	172	46 (26.0%)	3.7	378	78 (51.3%)	4.8
pre-sales request	7	2 (1.1%)	3.5	9	2 (1.3%)	4.5
capability, feature	88	30 (16.9%)	2.9	84	17 (11.2%)	4.9
other	43	13 (7.3%)	3.3	61	19 (12.5%)	3.2
document fault	7	1 (0.6%)	7.0	-	-	-
operation fault	50	10 (5.6%)	5.0	44	20 (13.2%)	2.2
application SW change request	-	-	-	3	1 (0.7%)	3.0
application SW fault	4	1 (0.6%)	4.0	-	-	-
system SW fault	85	15 (8.5%)	5.7	15	4 (2.6%)	3.8
system SW change request	14	3 (1.7%)	4.7	6	2 (1.3%)	3.0
instruction fault	7	2 (1.1%)	3.5	-	-	-
HW fault	67	17 (9.6%)	3.9	5	2 (1.3%)	2.5
AVERAGE			3.7			4.1

Table 3: All Calls/Problems by Call-Type

call-type	SYS_1			SYS_2		
	calls	problems	calls/problem	calls	problems	calls/problem
unknown type	5	2 (1.1%)	2.5	-	-	-
clarify document	130	35 (19.8%)	3.7	34	7 (4.6%)	4.9
operation question	172	46 (26.0%)	3.7	378	78 (51.3%)	4.8
pre-sales request	7	2 (1.1%)	3.5	9	2 (1.3%)	4.5
capability, feature	88	30 (16.9%)	2.9	84	17 (11.2%)	4.9
other	43	13 (7.3%)	3.3	61	19 (12.5%)	3.2
document fault	7	1 (0.6%)	7.0	-	-	-
operation fault	50	10 (5.6%)	5.0	44	20 (13.2%)	2.2
application SW fault	4	1 (0.6%)	4.0	-	-	-
system SW fault	85	15 (8.5%)	5.7	15	4 (2.6%)	3.8
instruction fault	7	2 (1.1%)	3.5	-	-	-
HW fault	67	17 (9.6%)	3.9	5	2 (1.3%)	2.5
TOTAL	665	174/177 (98.3 %)	3.7	630	149/152 (98 %)	4.1

Table 4: Non-forwarded Calls/Problems by Call-Type

- Question 2 (What percent of problems aren't really maintenance problems?) --> Table 5

Table 5: Portion of Real Maintenance Problems

	SYS_1	SYS_2
Number of total problems	177	152
Number of maintenance problems	80	116
percentage	45.2 %	76.3 %

Not all of the problems reported to Customer Support are really maintenance problems. There are, for example, lots of requests from different divisions inside the company. From a global view, all the effort spent in Customer Support is charged as maintenance effort. In the case of SYS_1, only about 45% of all problems (80 out of 177), and in the case of SYS_2, only about 76% of all problems (116 out of 152) are really maintenance problems.

- Question 3 (What is the distribution of rejected change requests by closure code?) --> Table 6

The distribution of rejected change requests by closure code is as follows:

Table 6: Rejected Change Requests by Closure Code

Closure Code	Systems	
	SYS_1	SYS_2
need additional information	11	11
not reproducible	1	-
no fix scheduled	3	2
already fixed	45	25
forwarded to ...	-	2
works as intended	6	1
works as documented	-	3
incorrect documentation	2	-
operation problem	1	1
document required	1	-
not retrofit	2	8
other	-	2

- Question 12 (What is the cost of ......?)

Because we have no effort data concerning the Product Assurance and engineering aspects of the maintenance process, we only could analyze effort as far as Customer Support was concerned:

The cost for each individual maintenance problem (as far as Customer Support is concerned) can be characterized

call-type	SYS_1			SYS_2		
	time (mins)	problems	time/problem	time (mins)	problems	time/problem
unknown type	52	2	26.0	-	-	-
clarify document	791	35	22.6	247	7	35.3
operation question	1203	46	26.1	3723	78	47.7
pre-sales request	36	2	18.0	211	2	105.5
capability, feature	739	30	24.6	747	17	44.0
other	247	13	19.0	813	19	42.8
document fault	43	1	43.0	-	-	-
operation fault	303	10	30.3	522	20	26.1
application SW change request	-	-	-	20	1	20.0
application SW fault	53	1	53.0	-	-	-
system SW fault	509	15	33.9	78	4	19.4
system SW change request	167	3	55.8	8	2	4.0
instruction fault	13	2	6.5	-	-	-
HW fault	327	17	19.3	33	2	16.6
AVERAGE			26.3 (mins)			43.1 (mins)

Table 7: ON-Line Spent Effort by Call-Type

141

2-31

call-type	SYS_1			SYS_2		
	time (mins)	problems	time/problem	time (mins)	problems	time/problem
clarify document	685	35	19.6	305	7	43.6
operation question	2317	46	50.4	4062	78	52.1
pre-sales request	45	2	22.5	130	2	65.0
capability, feature	1105	30	36.8	855	17	50.3
other	240	13	18.5	1810	19	95.3
document fault	117	1	117.0	.	.	.
operation fault	210	10	21.0	75	20	3.8
application SW fault	330	1	330.0	.	.	.
system SW fault	1125	15	75.0	769	4	192.3
system SW change request	115	3	38.3	335	2	167.5
instruction fault	20	2	10.0	.	.	.
HW fault	780	17	45.9	65	2	32.5
AVERAGE			40.5 (mins)			55.7 (mins)

Table 8: OFF-Line Spent Effort by Call-Type

- by the number of phone calls per problem:

The average number of calls (interactions with the customer) per problem is about 4 (SYS_1: 3.7, SYS_2: 4.1) according to table 4.

The most crucial problems in SYS_1 in terms of number of calls are: documentation faults (7 calls per problem), operation faults (5 calls per problem), and system software faults (5.7 calls per problem). In the case of SYS_2, the most crucial problems are: documentation clarifications (4.9 calls per problem), operation requests (4.8 calls per problem), pre-sales requests (4.5 calls per problem), and capability/feature requests (4.9 calls per problem).

- by the effort spent on-line (time spent talking to the customer on the phone ---> Table 7):

The average effort per problem spent on-line is about 30 minutes.

In the case of SYS_1, most on-line effort is spent for documentation problems (43 minutes per problem), application software faults (53 minutes per problem), and system software faults (56 minutes per problem). In the case of SYS_2 most on-line effort is spent for pre-sales requests (105 minutes per problem)

- by the effort spent off-line (time spent other than talking to the customer on the phone ---> Table 8):

The average effort per problem spent off-line is about 45 minutes.

In the case of SYS_1, the most off-line effort is spent for documentation problems (117 minutes per problem) and application software faults (330 minutes). In the case of SYS_2, the most off-line effort is spent for system software faults (180 minutes per problem).

(G2) Identify the best ways of applying the 20/80 rule to get the biggest savings and return on our maintenance dollars:

Although we have no final results concerning this matter, a careful interpretation of the results related to goal (G1) indicates that for instance better documentation, in the case of SYS_1, could save a big percentage of maintenance problems. In a paper not related to this study an analysis of software maintenance changes is reported[10]; the authors aim at the development of metrics for predicting where those changes might occur. Such metrics might help save dollars by concentrating resources on subsystems or modules which can be expected to require many changes.

(G3) Identify criteria for when a product is ready for release:

This question can only be answered if we know more about the type of problems and effort spent in engineering before release (question Q4) and about the type and problems during field test (question Q5).

(G4) Identify features of product, documentation or support that provide a wider customer satisfaction:

This question can be addressed by designing a customer questionnaire. Some of the technical problems definitely have impact on the customer's satisfaction, such as the high number of documentation-related problems (in the case of SYS_1) or not being able to keep promised dates for calling customers back.

(G5) Identify criteria for when a software product should be rewritten rather than maintained:

Unfortunately there are no data collected indicating explicitly which parts (modules, subsystems) of a product were affected (question Q26) or whether a problem is due to a previous change (question Q8).

The only way to address this question by using the currently available data is to evaluate the actual patch where the actual lines changed are listed. A paper not related to this study indicates that complexity metrics characterizing the locality of changes might be a promising metric for characterizing the suitability of parts of a software system for maintenance purposes[11].

(G6) Identify metrics of customer satisfaction that can be developed based upon existing data:

Based upon the results concerning goal G4 we hope to be able to develop metrics for customer satisfaction. Although it is too early to expect reliable metrics, candidate metrics might include aspects such as ability to keep promised schedules for dealing with maintenance problems or the frequency of similar (at least from the customer's point of view) maintenance problem reports.

(G7) Develop organizational guidelines for integrating software quality metrics into the company's framework of design, development, and support:

142

5207

This goal represents the second step after having understood the maintenance problems and identified possible improvements. Procedures for monitoring quality and productivity have to be established throughout the development and maintenance of software products; the prescribed data and metrics should be used for management and motivation purposes and improved. Before this problem can be addressed in a satisfactory way many more and different analyses have to be performed; in particular, data concerning the development phase of products have to be collected in order to identify the impact of the particular development process on maintainability. In a paper not related to this study interesting approaches for predicting the required customer support for a particular system were presented[*]. The prediction approach utilized development metrics among others.

9. Measurement and Evaluation Tools

In order to apply the proposed quantitative assessment approach practically, data collection and validation procedures as well as evaluation procedures need to be automated. A tool system was proposed integrating many tools already available in this environment. The whole tool system needs to be implemented in a decentralized fashion around a central data base. It has to provide different interfaces to different maintenance groups, limiting each group only to data relevant to their specific task, presenting the data in a helpful way. Independent of this company-specific project, a research project at the University of Maryland is aiming at the development of a comprehensive approach to automating measurement and evaluation in the context of software projects which include support of the generation of goals and questions and the project-specific interpretation of measurement results[2,4].

10. Conclusions

The objective of this study has been to demonstrate the benefits of assessing the software maintenance process in a quantitative way for the purpose of improvement. We have been able to show the applicability of the goal/question/metric paradigm to this complex problem domain and derive first analysis results based on a very limited subset of available data. The long-range benefits can be expected to be much more significant provided the derived set of data are collected in the future and interpreted within the proper context of maintenance questions and goals. In this paper we have not addressed the psychological problems involved in trying to introduce quantitative approaches into a traditional maintenance environment. The interested reader is referred to a book describing Hewlett Packard's experience (including psychological problems of motivating project personnel and higher-level management) from introducing metrics into their daily software production environment[*].

It was even surprising to us, how many characteristics of the maintenance process could be made visible by analyzing the limited set of data available at the time. This visibility of characteristics might be helpful in communicating problems in a more objective and convincing way.

The analysis result underline the importance of viewing software maintenance not as an isolated activity but as integrated into the overall software life cycle. We can improve the effectiveness of maintenance procedures by purely analyzing the maintenance process. However, we will never reduce the overall effort (and money) spent for maintenance below a certain limit if we cannot make sure that software products fulfill certain quality requirements at the time of delivery (start of maintenance). Low quality products will always cause maintenance problems. Accepting this fact will lead us to establish quality criteria for a product to be released to customers and, thereby, entering the maintenance phase. As a consequence, developers could develop guidelines for how to achieve those criteria and metrics to evaluate the degree to which those criteria are actually met. Altogether this would allow us to develop better maintainable products in the first place or, at least, allow us to predict certain maintenance problems at the beginning of maintenance. Additional benefits of collecting maintenance data are to provide a better basis for judging customer satisfaction, the company's image, and marketing

If we want to reduce the overall maintenance effort we need to apply the assessment and improvement procedures introduced in this paper to development as well as maintenance of a product. This requires the availability of development data (as implicated by the evaluation questions in section 4) in addition to maintenance data. As long as we do not assess the overall software life cycle, problems will shift from design to coding, coding to testing, and development to maintenance. It is a well known fact that the really serious maintenance problems originate during the prior development of the product; the identification of these real causes of maintenance problems will result in significant improvements of maintenance.

References

[1] V. R. Basili, "Quantitative Evaluation of Software Engineering Methodology," 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].

[2] V. R. Basili, H. D. Rombach, "TAME: Tailoring an Ada Measurement Environment," Proceedings of the Joint Ada Conference, Arlington, Virginia, March 16-19, 1987, pp. 318-325.

143

5207

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

[4] V. R. Basili, H. D. Rombach, "TAME: Integrating Measurement into Software Environments," submitted for publication to IEEE Transactions on Software Engineering.

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

[6] V. R. Basili, D. M. Weiss, "Evaluating Software Development by Analysis of Changes: The Data from the Software Engineering Laboratory," Technical Report TR-1236, Dept. of Computer Science, University of Maryland, College Park, December 1982.

[7] V. R. Basili, D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," IEEE Transactions on Software Engineering, Vol. SE-10, No.3, November 1984, pp.728-738.

[8] J. Chapin, G. Faidell, "Predicting Software Customer Support," Proc. Conference on Software Maintenance, Washington D.C., 1985, pp. 128-134.

[9] Robert B. Grady, Deborah L. Caswell, "Software Measures; Establishing a Company-Wide Program," to be published as a book by Addison Wesley, 1987.

[10] D. A. Gustafson, A. Melton, and C. S Hsieh, "An Analysis of Software Changes During Maintenance and Enhancement," Proc. Conference on Software Maintenance, Washington D.C., 1985, pp. 92-95.

[11] H. Dieter Rombach, "A Controlled Experiment on the Impact of Software Structure on Maintainability," IEEE Transactions on Software Engineering, Vol. SE-13, No. 3, March 1987, pp.344-354.

144

5207

Resource Utilization during Software Development

Marvin V. Zelkowitz

Department of Computer Science, University of Maryland, College Park, Maryland

This paper discusses resource utilization over the life cycle of software development and discusses the role that the current "waterfall" model plays in the actual software life cycle. Software production in the NASA environment was analyzed to measure these differences. The data from 13 different projects were collected by the Software Engineering Laboratory at NASA Goddard Space Flight Center and analyzed for similarities and differences. The results indicate that the waterfall model is not very realistic in practice, and that as technology introduces further perturbations to this model with concepts like executable specifications, rapid prototyping, and wide-spectrum languages, we need to modify our model of this process.

1. INTRODUCTION

As technology impacts on the way industry builds software, there is increasing interest in understanding the software development model and in measuring both the process and the product. New workstation technology (e.g., PCs, CASE tools), new languages (e.g., Ada, requirements and specification languages, wide-spectrum languages), and techniques (e.g., prototyping, object-oriented design, pseudocode) are affecting the way software is built, which further affects how management needs to address these concerns in controlling and monitoring a software development.

Most commercial software follows a development cycle often referred to as the *waterfall* cycle. While there is widespread dissatisfaction with this as a model of development, there have been few quantitative studies investigating its properties. This paper addresses this problem and whether the waterfall chart is an appropriate vehicle to describe software development. Other models, such as the spiral model and value chaining, have been described, and techniques like rapid prototyping have been proposed that do not fit well with the waterfall chart [1, 2]. This paper presents data collected from 13 large projects developed for NASA Goddard

Space Flight Center that shed some light on this model of development.

Data about software costs, productivity, reliability, modularity, and other factors are collected by the Software Engineering Laboratory (SEL), a joint research project of NASA/GSFC, Computer Sciences Corporation, and the University of Maryland, to improve both the software product and the process for building such software [3]. It was established in 1976 to investigate the effectiveness of software engineering techniques for developing ground support software for NASA [4].

The software development process at NASA, as well as in most commercial development environments, is typically product-drive and can be divided into six major life-cycle activities, each associated with a specific "end product" [5, 6]; requirements, design, code and unit test, system integration and testing, acceptance test, and operation and maintenance. In order to present consistent data across a large number of projects, this paper focuses on the interval between design and acceptance test and involves the actual implementation of the system by the developer.

In this paper, we will use the term "activity" to refer to the work required to complete a specific task. For example, *coding activity* refers to all work performed in generating the source code for a project, the *design activity* refers to building the program design, etc. On the other hand, the term "phase" will refer to that period of time when a certain work product is supposed to occur. For example, *coding phase* will refer to that period of time during software development when coding activities are supposed to occur. It is closely related to management-defined milestone dates between the critical design review (CDR) and the code review. But during this period other activities may also occur. For example, during the coding phase, design activity is still happening for some of the later modules that are defined for the system and some testing activity is already occurring with some of the modules that were coded into the source program fairly early in the process.

331

5207

In the NASA/GSFC environment that we studied, the software life cycle follows this fairly standard set of activities [7]:

1. The *requirements* activity involves translating the functional specification consisting of physical attributes of the spacecraft to be launched into requirements for a software system that is to be built. A functional requirements document is written for this system.

2. A *design* activity can be divided into two subactivities: preliminary design activity and detailed design activity. During *preliminary design*, the major subsystems are specified, and input-output interfaces and implementation strategies are developed. During *detailed design*, the system architecture is extended to the subroutine and procedure level. Data structures and formal models of the system are defined. These models include procedural descriptions of the system; data flow descriptions; complete description of all user input, system output, input-output files, and operational procedures; functional and procedural descriptions of each module; and complete description of all internal interfaces between modules. At this time a system test plan is developed that will be used later. The design phase typically terminates with the CDR.

3. The *coding and unit test* activity involves the translation of the detailed design into a source program in some appropriate programming language (usually Fortran, although there is some movement to Ada). Each programmer will unit test each module for apparent correctness. When satisfied, the programmer releases the module to the system librarian for configuration control.

4. The *system integration and test* activity validates that the completed system produced by the coding and unit test activity meets its specifications. Each module, as it is completed, in integrated into the growing system, and an integration test is performed to make sure that the entire package performs as expected. Functional testing of end-to-end system capabilities is performed according to the system test plan developed as part of preliminary design.

5. In the *acceptance test* activity, a separate acceptance test team develops tests based on functional specifications for the system. The development team provides assistance to the acceptance test team.

6. *Operation and maintenance* activities begin after acceptance testing when the system becomes operational. For flight dynamics software at NASA, these activities are not significant with respect to the overall cost. Most software that is produced is highly reliable. In addition, the flight dynamics software is usually not "mission critical," in that a failure of the software dos not mean spacecraft failure but simply that the program has to be rerun. In addition, many of these programs (i.e., spacecraft) have limited lifetimes of 6 months to about 3 years, so the software is not given the opportunity to age.

The waterfall model makes the assumptions that all activity of a certain type occurs during the phase of that same name and that phases do not overlap. Thus all requirements for a project occur during the requirements phase; all design activity occurs during the design phase. Once a project has a design review and enters the coding phase, then all activity is coding. Since many companies keep resource data based on hours worked by calendar date, this model is very easy to track. However, as Figure 1 shows, activities overlap and do not occur in separate phases. We will give more data on this later.

2. THE WATERFALL CHART IS ALL WET

Table 1 summarizes the raw data on the 13 projects analyzed in this paper. They are all fairly large flight dynamics programs ranging in size from 15,500 lines of Fortran code to 89,513 lines of Fortran, with an average size of 57,890 lines. The average work on these projects was 8.90 staff-months; thus, all represent significant effort.

In most organizations, weekly time sheets are collected as part of cost accounting procedures so that phase data are the usual reporting mechanism. However, in the SEL, weekly activity data are also collected. The data consist of nine possible activities for each component
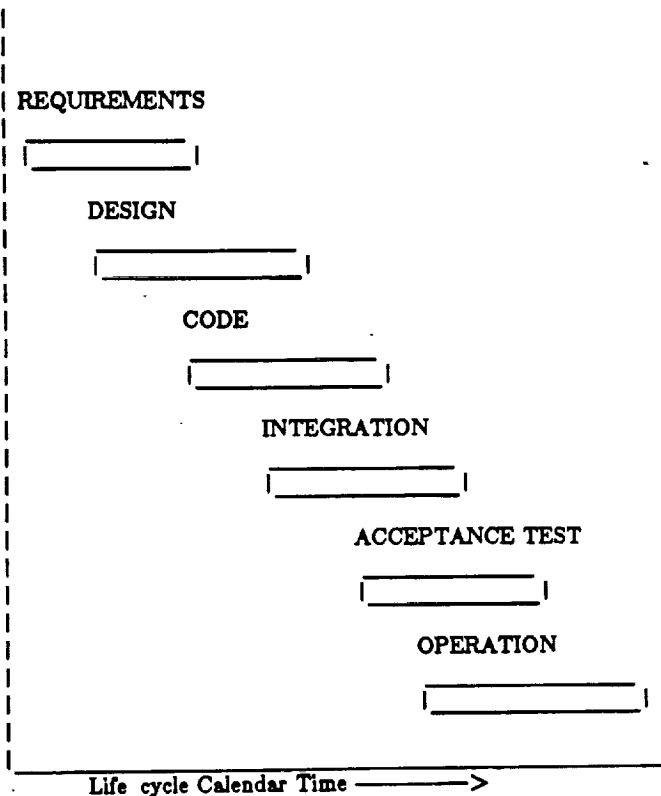
Table 1. Project Size and Staff-Month Effort

Project number	Size (lines of code)	Total effort hours[a]	Staff-months
1	15,500	17,715	116.5
2	50,911	12,588	82.8
3	61,178	17,039	112.1
4	26,844	10,946	72.0
5	25,731	1,514	10.0
6	67,325	19,475	128.4
7	66,260	17,997	118.4
8	—[b]	—[b]	—[b]
9	55,237	15,262	100.4
10	75,420	5,792	38.1
11	89,513	15,122	99.5
12	75,393	14,508	95.4
13	85,369	14,309	94.1
Average	57,890	13,522	89.0

[a] All technical effort, including programmer and management time.
[b] Raw data not available in data base.

REQUIREMENTS

DESIGN

CODE

Figure 1. Typical life cycle.

INTEGRATION

ACCEPTANCE TEST

OPERATION

Life cycle Calendar Time ————>

(e.g., source program module). In this paper, these will be grouped as design activities, coding activities (including unit test), integration activities, acceptance testing activities and other. Specific meetings, such as design reviews, will be grouped with their respective activity (e.g., a design review is a design activity, a code walkthrough is a coding activity, etc.)

Table 2 classifies the data presented in this paper. Each column represents a type of work product (design, code, test). The "by phase" part represents the effort during that specific time period, while the "by activity" part represents the actual amount of such activity. "Other" does not enter into the "by phase" table, since these activities occur during all phases. At NASA, 22% of a project's effort occurs during the design phase, while 49% is during coding. Integration testing takes 16% while all acceptance activities take almost 13%. (Remember that requirements data are not being collected here. We are simply reporting the percentage of design, coding, and testing activities. A significant requirements activity does occur.)

By looking at all design effort across all phases of the projects, we see that design activity is actually 26% of

the total effort rather than the 22% listed above. The coding activity is a more comparable 30% rather than the 49% listed by phase data, which means that the coding phase includes many other tasks. "Other" increased from 12% to 29% and includes many time-consuming tasks that are not accounted for by the usual life-cycle accounting mechanism. Here, "other" includes acceptance testing as well as activities that take a significant effort but are usually not separately identifiable using the standard model. These include corporate (not technical) meetings, training, travel, documentation, and various other tasks assigned to the personnel. The usual model of development does not include an "other," and this is significant since almost one-third of a project's costs are not effective at completing it. More on this later.

The situation is actually more complex, since the distribution of activities across the project is not reflected in Table 2. These data are presented in Tables 3–5. Only 49% of all design work actually occurs during the design phase (Table 3), and one-third of the total design activity occurs during the coding period. Over one-sixth (10.3% + 6.4%) of all design occurs during

5207

Table 2. Development Effort

Project number	Design (%)	Code (%)	Integration act. (%)	Accept. test and other (%)
By Phase				
1	20.6	38.6	16.5	24.3
2	16.2	48.4	19.3	16.2
3	21.8	47.9	17.4	12.9
4	35.9	39.5	24.5	0.1
5	18.2	68.8	13.0	0.0
6	16.3	48.6	10.9	24.3
7	19.0	50.4	14.9	15.7
8	22.9	48.4	13.0	15.8
9	22.6	68.3	8.1	1.1
10	24.4	44.6	20.2	10.8
11	22.7	39.4	21.4	16.5
12	16.9	53.1	10.9	19.1
13	28.2	43.5	20.1	8.2
Average	22.0	49.2	16.2	12.7
By Activity				
1	17.4	16.4	9.9	56.3
2	30.1	39.4	20.8	9.7
3	26.3	20.3	19.3	34.2
4	27.3	28.7	6.0	38.0
5	31.0	35.5	9.4	24.1
6	14.9	21.8	24.0	39.2
7	20.2	25.9	14.3	39.6
8	11.0	13.9	9.3	65.8
9	31.3	43.5	18.9	6.4
10	38.2	37.3	6.1	18.4
11	29.3	31.0	17.2	22.5
12	23.7	46.5	24.0	5.9
13	32.6	36.3	15.6	15.6
Average	25.6	30.5	15.0	28.9

Table 3. Design Activity During Life-Cycle Phases

Project number	Design phase (%)	Coding phase (%)	Integration test (%)	Accept. test phase (%)
1	41.8	33.9	10.0	14.3
2	53.6	31.2	9.2	6.0
3	33.3	37.1	19.7	9.9
4	45.3	32.6	22.0	0.1
5	17.4	69.1	13.5	0.0
6	58.9	30.7	4.3	6.2
7	63.9	15.3	6.8	14.1
8	28.1	56.9	7.1	8.0
9	61.8	38.2	0.0	0.0
10	57.8	27.2	7.0	8.0
11	58.7	13.7	16.67	10.9
12	58.9	32.8	5.9	2.4
13	60.5	24.7	11.9	2.9
Average	49.2	34.1	10.3	6.4

with the other category removed. As can be seen, design took about one-third of the development effort and varied between a low of 25% and a high of 47%—a factor of almost 2. On the other hand, coding took an average of 42% of the relative effort and varied between 36% and 49%—a factor of only 1.36. Testing ranged from a low of 7.5% to a high of 39.5%, with an average of 22%, for a relative factor of over 5.

From Table 2, the "other" category was 29% of the effort on these projects, and of the 13 measured projects, other activities consumed more than one-third of the effort on six of them. The other category consists of activities such as travel, completion of the data collection forms, meetings, and training. While these activities are often ignored in life-cycle studies, the costs are significant. Table 7 presents the distribution of other

testing when the system is "supposed" to be finished. In almost one-third of the projects (4 out of 13), about 10% or more of the design work occurred during the final acceptance testing period.

As to coding effort, Table 4 shows that while a major part (70%) does occur during the coding phase, almost one-quarter (16% + 7%) occurs during the testing periods. As expected, only a small amount of coding (7%) occurs during the design phase; however, the table indicates that some coding does being on parts of the system while other parts are still under design. These data have the widest variability as a range from 0% (project 10) to over 22% (project 3).

Similarly, Table 5 shows that significant integration testing activities (almost one-half) occur before the integration testing period. Once modules have been unit tested, programmers begin to piece them together to build larger subsystems, with almost half (43%) of the integration activities occurring during the coding phase.

Due to the wide variability of the "other" category in Table 2, Table 6 presents the same data as relative percentages for design, coding, and integration testing

Table 4. Coding and Testing Activity During Life-Cycle Phases

Project number	Design phase (%)	Coding phase (%)	Integration test (%)	Accept. test phase (%)
1	1.4	78.3	11.3	9.1
2	0.0	72.8	19.7	7.5
3	22.2	56.2	11.8	9.8
4	16.4	58.5	25.1	0.1
5	21.2	68.7	10.1	0.0
6	0.5	77.3	11.3	10.9
7	1.3	73.9	15.6	9.2
8	14.7	54.7	21.0	9.7
9	5.2	91.1	3.1	0.6
10	0.0	73.0	22.5	4.5
11	2.2	70.5	20.1	7.2
12	0.3	74.8	8.3	16.6
13	4.6	63.6	26.9	4.9
Average	6.9	70.3	15.9	6.9

Table 5. Integration Activity During Life-Cycle Phases

Project number	Design phase (%)	Coding and unit phase (%)	Integration test (%)	Accept. test phase (%)
1	0.0	17.8	27.4	54.7
2	0.0	45.2	30.1	24.7
3	6.1	53.9	21.1	18.9
4	21.0	39.3	39.7	0.0
5	28.4	71.0	0.6	0.0
6	1.0	40.9	17.6	40.5
7	0.5	54.1	26.3	19.2
8	2.9	33.8	19.2	44.1
9	0.0	66.4	29.2	4.4
10	0.0	23.1	41.5	35.5
11	0.0	36.4	35.1	28.5
12	0.1	32.7	22.4	44.8
13	1.5	49.5	28.8	20.2
Average	4.7	43.4	26.1	25.8

Table 7. Other Activities Effort in Each Phase

Project number	Design phase (%)	Coding and testing phase (%)	Integration test (%)	Accept. test phase (%)
1	23.3	32.2	18.1	26.5
2	0.0	9.1	26.4	64.6
3	21.7	47.8	16.8	13.7
4	46.2	30.2	23.6	0.0
5	11.0	67.7	21.3	0.0
6	18.2	44.2	9.0	28.7
7	14.4	51.6	14.5	19.5
8	26.5	47.7	11.4	14.4
9	15.9	65.5	18.7	0.0
10	12.4	30.2	35.9	21.5
11	21.4	32.2	18.9	27.6
12	47.3	46.6	4.6	1.5
13	42.5	30.0	12.7	14.9
Average	23.1	41.2	17.8	17.9

activities across all phases. While such effort varies widely from project to project, no general trends can be observed, except that it does take a significant effort as a percent of total costs.

3. CONCLUSIONS

Using data from the SEL database, it seems that the software development process does not follow the waterfall life cycle but appears to be more a series of rapids as one process flows into the next. Significant activities cross phase boundaries and do not follow somewhat arbitrary milestone dates. The classical product-driven model has many shortcomings.

In the SEL environment, as well as elsewhere, other classes of activities take a significant part of a project's resources. At almost one-third of the total effort, it

Table 6. Relative Activity

Integration act. (%)	Project number	Design act. (%)	Coding and unit act. (%)
1	39.9	37.5	22.6
2	33.3	43.7	23.0
3	39.9	30.8	29.3
4	44.0	46.3	9.7
5	40.8	46.8	12.3
6	24.6	35.9	39.5
7	33.5	42.8	23.6
8	32.2	40.7	27.1
10	46.8	45.7	7.5
11	37.8	40.1	22.1
12	25.2	49.4	25.5
13	38.6	43.0	18.4
Average	36.2	42.2	21.6

might be part of an explanation of why software is typically over budget. Estimating procedures often use a work breakdown structure where the system is divided into small pieces and estimates for each piece are summed up. Inclusion of a significant "other" usually does not occur.

Newer technology is affecting this traditional model even more. In one NASA experiment, a prototype of a project was developed as part of the requirements phase [8]. In this case, 33,000 lines of executable Fortran were developed at a cost of 93.1 staff-months—already a significant project in this SEL environment. When viewed as a separate development, the prototype had a life cycle typical of the data presented here, but if viewed as only a requirements activity it puts a severe strain on existing models.

Current models do not handle executable products as part of requirements. Other questions arise: Are Ada package specifications design or code? Are executable specification languages specification or design? When does testing start?

It is clear that our current product-driven models need to be updated. Other models, such as the spiral model, which is an iterative sequence of risk-assessment decisions, or value chaining, which addresses value added by each phase, are alternative approaches that need to enter our vocabulary and be further studied for effectiveness.

5207

M. V. Zelkowitz

REFERENCES

1. B. Boehm, A Spiral Model of Software Development and Enhancement, *ACM Software Eng. Notes* 11(4) 22–42, 1986.
2. B. Boehm, Improving Software Productivity, *Computer* 20(9) 43–57, 1987.
3. F. E. McGarry, et al., Guide to Data Collection, NASA Goddard Space Flight Center, Code 552, Greenbelt, MD, August 1982.
4. V. R. Basili and M. V. Zelkowitz, Analyzing Medium-Scale Software Development, 3rd International Conference on Software Engineering, Atlanta, pp. 116–223, 1978.
5. A. Wasserman, Software Engineering Environment, *Adv. Comput.*, 22, 110–159, 1983.
6. M. W. Zelkowitz, Perspective on Software Engineering, *ACM Comput. Surv.*, 10(2), 198–216, 1978.
7. F. E. McGarry, G. Page, et al., Standard Approach to Software Development, NASA Goddard Space Flight Center, Code 552, Greenbelt, MD, September 1981.
8. M. V. Zelkowitz, The Effectiveness of Software in Prototyping: A Case Study, ACM Washington Chapter 26th Tech. Symposium, Gaithersburg, MD, pp. 7–15, 1987.

2-40

SECTION 3 – MEASUREMENT ENVIRONMENT STUDIES
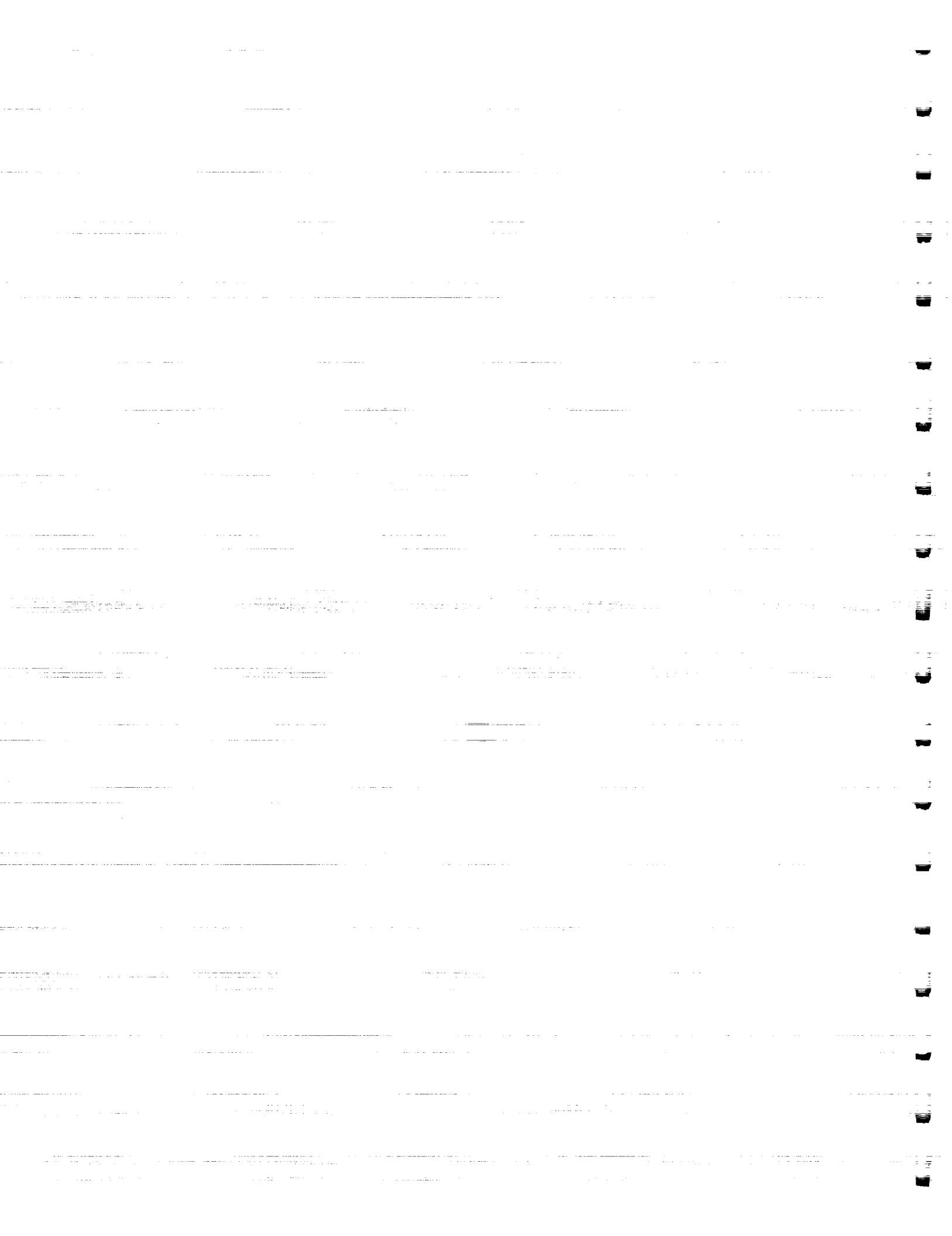
SECTION 3 - MEASUREMENT ENVIRONMENT STUDIES

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

- "Generating Customized Software Engineering Information Bases from Software Process and Product Specifications," L. Mark and H. D. Rombach, <u>Proceedings of the 22nd Annual Hawaii International Conference on System Sciences</u>, January 1989

- "Software Process and Product Specifications: A Basis for Generating Customized SE Information Bases," H. D. Rombach and L. Mark, <u>Proceedings of the 22nd Annual Hawaii International Conference on System Sciences</u>, January 1989

- "The TAME Project: Towards Improvement-Oriented Software Environments," V. R. Basili and H. D. Rombach, <u>IEEE Transactions on Software Engineering</u>, June 1988

- "Validating the TAME Resource Data Model," D. R. Jeffery and V. R. Basili, <u>Proceedings of the 10th International Conference on Software Engineering</u>, April 1988

Generating Customized Software Engineering Information Bases
from Software Process and Product Specifications

Leo Mark and H. Dieter Rombach

Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

Software engineering is a challenging new application area for information bases. The new challenges are twofold: how software engineering processes and products can be properly modeled; and, how such processes and products can be mirrored naturally within an information base. Meeting these challenges requires software engineering and information base research. Our "Meta Information Base for Software Engineering" project at the University of Maryland represents such a joint research effort. The idea of our approach is to generate customized software engineering information bases from formal specifications of software engineering processes and products. The three central research topics are: (i) develop a software process and product specification language which permits all the information necessary to understand, control and improve any given software engineering process; (ii) develop a meta information base schema which automatically generates an information base structure given a software process and product specification; and (iii) develop a mapping between the software engineering oriented and information base oriented models. The generator approach acknowledges the fact that software engineering changes not only from environment to environment, but also from project to project. If an information base is expected to truly mirror and support a given software engineering project, it needs to be tailorable to the changing characteristics of the software project itself. The generator based approach suggested by our project seems to be the natural approach to satisfy this important need.

This paper presents the information base oriented part of our joint project. It discusses how to represent a set of software process and product type specifications in a database and how to use these to automatically generate database support for process executions and product instances.

Introduction

When we began our research on a Meta Information Base for Software Engineering one of the first future research topics we identified was object-oriented database systems. However, as the reader may already have noticed, the words object-oriented were not mentioned in the title or abstract of this paper. While a number of object-oriented database systems have been proposed in the past few years [Dittrich 86], there seems to be

little consensus as to what such a system should be. Although our research will eventually lead to our version of an object-oriented database system, we are currently using and extending existing relational database technology, trying to find out how far it will take us. Others, following the same approach, have extended the relational model with more semantics [Codd 79], provided better support for complex objects [Dadam 86], and extended the data definition capabilities with support for type inheritance [Borgida 88]. The common goal of these efforts, best described in [Carey 88], is to use and extend existing relational technology, but to retain $<$ a powerful non-procedural query language.

Our joint project is based on the following framework for a Software Engineering Environment [Rombach 88]:
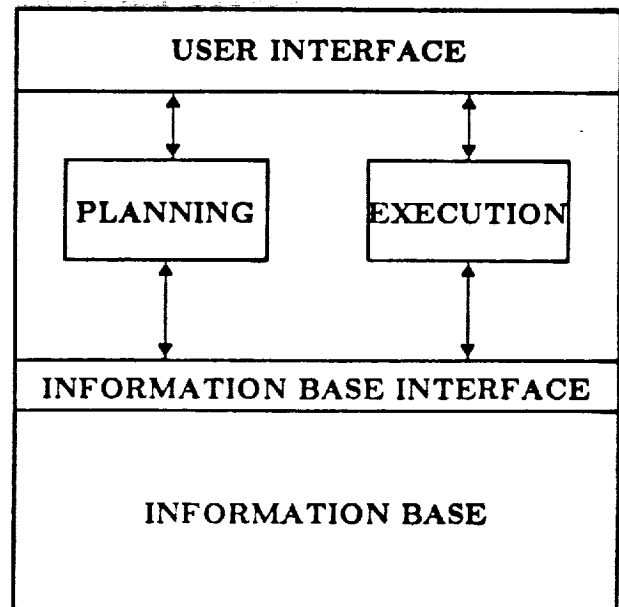
Framework for SEE

This paper concentrates on the following two information base issues:

- the representation of a formal specification of a set of software engineering process and product type descriptions using an extended relational data model, and

---

- the automatic generation of information base support for software process executions and product instances based on their type descriptions.

As a basis for addressing the first issue we shall, in section 2, briefly summarize the set of requirements for a software process and product specification language (as described in our companion paper [Rombach 88]). In section 3, we introduce a graphical formalism for the extended relational model. In section 4, we show the representation of formal specifications. The notion of a Self-Describing Database System [Mark 85], briefly described in section 5, is the basis for the generator approach discussed in section 6.

Requirements for a Software Process and Product Specification Language

We distinguish between very general requirements which acknowledge the basic nature of software processes and more specific requirements [Rombach 88].

In this section we shall restrict ourselves to listing the specific requirements for a software process and product specification language from a planning perspective and from an execution perspective.

Since we want formal specifications in this language represented in our software engineering information base, the specific requirements, from a planning perspective, have direct bearing on the schema design of the information base. Similarly, since we want to automatically generate information base support for software process executions and product instances based on the formal specifications, the specific requirements, from execution perspective, also have direct bearing on the schema design.

From a planning perspective, the specific requirements for the specification language include the ability to specify:

1. process, product, and constraint types

2. produce (output) and consume (input) relationships between product and process types

3. control flow relationships between process types (sequence, alternation, iteration, and parallelism)

4. structural relationships between product types (sequence, alternation and iteration)

5. dependency relationships between process types (Process P1 is dependent of process P2 if every execution of P2 triggers a simultaneous execution of P1. Typically, measurement processes are dependent on the construction processes which they are supposed to monitor.)

6. pre-conditions and post-condition relationships between constraint and process/product types (A pre-condition of a process is a constraint imposed upon initiation of this process; a post-condition of a process is a constraint imposed upon termination of this process; a condition of a product is a constraint imposed upon this product.)

7. aggregation and decomposition of process and product types

8. generalization and specialization of process and product types

9. constructive as well as analytic (measurement-oriented) product and process types

10. different roles (Different roles are performed in a software project such as design role, test role, quality assurance role, or management role. Roles define views or perspectives of (a subset of) the processes and products relevant to a particular project. Type and number of roles may change from project to project.)

11. time (relative and absolute) & space (software structure, versions, configurations) dimensions

12. dialogues between processes (including human beings)

In section 4 we design the schema for the meta information base to meet most of these requirements.

The specific requirements from an execution perspective for a software process and product specification language include the ability to handle

1. the instatiation (creation of objects) of process, product and constraint types

2. long-term, nested transactions (Many software engineering processes such as designing may stretch over weeks or months; in addition, they may contain nested activities.)

3. varying degrees of persistence (Some information needs to be kept forever, some only for the duration of the project, and others only until a new instance (e.g., product version) has been created.)

4. tolerance of inconsistency (Because of the long-term nature of software engineering processes, it might be necessary to store intermediate information that does not yet conform with the desired consistency criteria.)

5. dynamic types (type hierarchies) (an object of type product (e.g., a compiler developed during one project) may be used as an object of type process during a future project.)

6. non-determinism due to user interaction

7. dynamic changes of process specification types (It is impossible to plan for all possible (non-deterministic) results produced by human beings in advance. However, we would like to react to those situations by dynamically re-planning during execution. Although it is not a problem to change a specification during the planning stage, it might be a problem to change the specifications during execution while preserving the current execution state.)

8. back-tracking due to execution failures

9. the organization of historical sequences of product objects and process executions

10. the enormous amounts of interaction between parallel activities

11. the role-specific interpretation of facts (The same process and product facts might require different interpretation in the context of different roles.)

12. the triggering of actions (based on pre-conditions and post-conditions)

The list of execution-point-of-view requirements is heavily influenced by the results of a working group during the 4th International Workshop on Process Specification (Moretonhampstead, UK, May 1988), chaired by Tom Cheatham [IWSPS 88]. Some of these requirements will be met by our automatically created information base, however many of them define topics for future research on information bases.

Graphical Formalism for an Extended Relational Model

To create a schema for a meta information base that completely and precisely mirrors the fundamental software engineering concepts, we need a powerful data model. A data model consists of a language for defining data structures, a language for defining constraints, and a language for data manipulation and query processing. We shall primarily concentrate on the data structure language in this paper. This language is an extended version of the relational model with a number of concepts borrowed from semantic data models and object oriented data models.

The language supports two kinds of uniquely named **domains**: lexical and non-lexical.

Non-lexical domains (full circles) model object-sets and lexical domains (broken circles) model object-name-sets. We shall almost entirely be using non-lexical domains in the specification. The reason is that we concentrate on modeling the fundamental software engineering concepts and their relationships, and postponing the aspects of how the concepts are lexically described, represented, and referenced. In an implementation the non-lexical domains will be represented by surrogates [Hall 76], which are system generated, internal, unique identifiers for objects.

Since surrogate values are internal to the system and invisible to the user we are primarily modeling the invisible part of the meta information base, and ignoring the visible part. Several important observations are related to the use of surrogates:

1. Surrogates allow us to model aggregate objects fairly easily while preserving normalized relation representations. Non-procedural query languages, like relational calculus, can be used for query processing without change. If a nested relation representation [Thomas 86] was used, allowing attributes with set values, we would violate first normal form representations and we would be forced to use a powerset calculus.

2. Surrogates allow us to model generalization in a straight forward manner [Codd 79], however a slight generalization of relational calculus is needed if we want to utilize inheritance in queries.

3. As far as a database system is concerned, anything stored as an instance of a lexical object type is primitive, and all the database system can do is insert, delete, or retrieve it. Surrogates are ideal for modeling the structure of new user defined object types, providing a means for extensibility. However, the complete structure of an object must be explicitly represented if the user wants to use the relational calculus to manipulate and answer questions about the structure of the object.

4. Breaking down everything to obtain an explicit representation of the internal structure of objects may result in inefficiency from a system point-of-view. However, current research on view cache and incremental computation models show very promising results [Roussopoulos 87]. Inefficiency from a user point-of-view can basically be ignored because relational views can be used to define a higher level query interface when needed.

An arrow between two domains represent an **is-a** relation type. In the example below, the object type O has subtypes O_1 and O_2. An is-a relation type represent a total function from the subtype to the supertype. The set of is-a relation types define a directed acyclic graph on the set of domains. Various rules for inheritance may be adopted, however, inheritance from multiple supertypes is hard to define properly [Borgida 88].
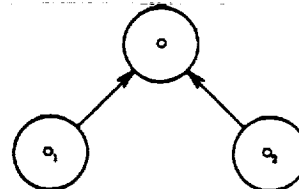
Relation types are uniquely named and are represented by the notation below. **Attributes** model the roles of the corresponding domains in **relations**. Attribute names may be omitted, in which case the corresponding domain name is used. However, attribute names must be unique within relations.

Identifier constraints (double headed arrow under an attribute combination) model partial functions from an attribute combination to each of the other attributes in the relation.
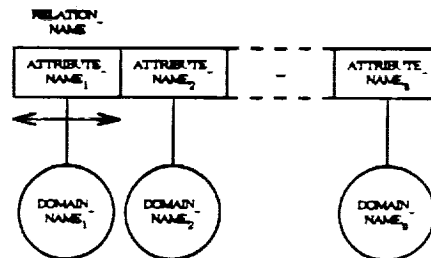
Rather than using relational normalization, we aim at identifying atomic facts. Multiple atomic facts may later be combined into larger relations while preserving at least Boyce-Codd Normal Form (BCNF). As is customary in object-role data models we shall model all concepts in terms of domains. The role of the relations is therefore reduced to capture the aggregates that form the concepts and to relate the concepts. An important advantage of our specification language over the traditional relational data definition language is that it clearly indicates that only attributes over the same domain can be used as a basis for entity joins between relations. The relational model traditionally only supports domains of primitive types and does not support a strong typing concept.

Although some types of aggregate objects, e.g. abstract syntax trees, could be conveniently represented by recursively defined relation types, whatever that is, we have not considered such an extension of our model because databases have a hard time managing instances that are not all of the same structure and size.

Our approach clearly allows us to use the **relational calculus** for data manipulation and query processing.

A significant advantage of this is that a powerful **constraint definition capability** may be based on the relational calculus.

Although important, we shall postpone further discussion of the query language and the constraint capability to a later paper.

Representation of Process and Product Specifications

Before we start, let it be perfectly clear that we are dealing with three levels of information. What we are about to design in this section is a schema - or rather a meta-schema - that describes all process and product descriptions that can be defined in the specification language introduced in [Rombach 88].

The data stored under this schema are, in other words, process and product descriptions and can in turn be interpreted as the schema for process executions and product instances under these descriptions. This issue is discussed in sections 5 and 6

The two fundamental concepts in the specification language are **process descriptions** and **product descriptions**.

Process descriptions and product descriptions are modeled by the two domains shown below.

Instances of process descriptions and product descriptions are tied to their type through the insert operation. Therefore, a "member" relationship need not be modeled explicitly.

We use the concept process recursively in two ways. First, a process description may be an aggregate of a set of component process descriptions. In an aggregation we form a concept from existing concepts. The phenomena that are members of the new concept's extension are composed of phenomena from the extensions of the existing concepts. Second, a process description may be a generalization of a more specific process description. In a generalization we form a new concept by emphasizing common aspects of existing concepts, but out special aspects. The phenomena that are members of the existing concepts are all members of the new concept, and they therefore inherit all the attributes of the members of the new concept. Aggregation and generalization are classical themes in object oriented databases [Smith 77].

The **aggregate process descriptions** are modeled below. A process description may be reused in many aggregate process descriptions. Aggregate process descriptions may have multiple levels, but cannot be defined recursively, (i.e. a process description cannot contain itself as a component at any level). This constraint is not modeled below.
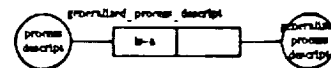
As modeled by the second relation type below, some but not all process descriptions may have names. We have modeled these names to be universally unique. Other models are of course possibile.
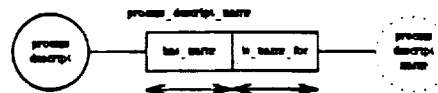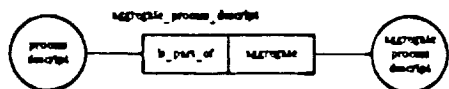
Within aggregate process descriptions, the component process descriptions may be **sequential, alternative, parallel**, or **iterated**. Only process descriptions that are parts of an aggregate process description can be used in any of these ordering schemes. Since process descriptions may be reused in many aggregate process descriptions, the ordering must be aggregate process description specific. Our approach is to model the restrictions imposed by the ordering schemes. Since parallelism is not a restriction we need not model it. Sequence is, for convenience, assumed to be represented in a relation where the tuples are ordered on the aggregate process descriptions and subsequently on the component process descriptions. The order of the component elements will, of course, depend on the lexical representation of their names since it makes no sense to order on the non-lexical surrogate values. Iteration will simply be modeled as a "goto".
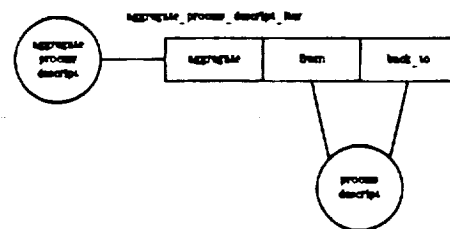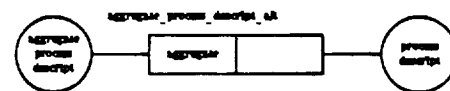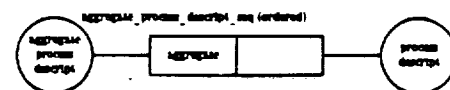
The **generalized process descriptions** are modeled below. Notice that a process description may be in more that one generalization, (i.e., we model a generalization net rather than a generalization hierarchy). However, the generalization net cannot contain cycles; this constraint is not modeled below. This model will provide the information needed to support any inheritance scheme we may want to adopt.
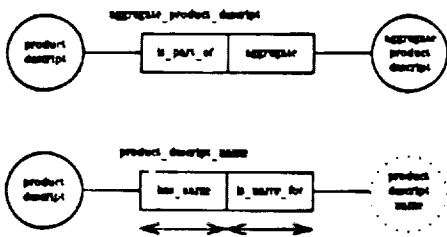
To complete the two recursive definitions, we must model the fact that an aggregate process description and a generalized process description are themselves process descriptions.
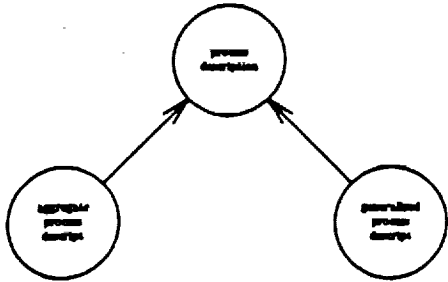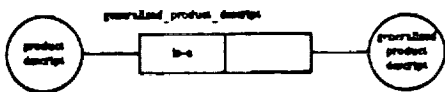
This model will provide the information needed to support any inheritance scheme we may want to adopt.

We use the concept product recursively in two ways. First, a product description may be an aggregate of a set of component product descriptions. Second, a product description may be a generalization of more specialized product descriptions.

As modeled by the second relation type below, some but not all product descriptions may have names. We have modeled these names to be universally unique. Other models are of course possibile.
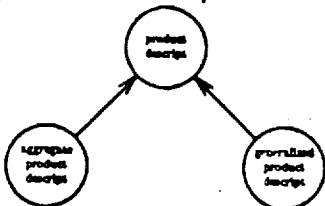
We model the **generalized product descriptions below.** As with generalized process descriptions, a product description may be reused in more that one generalization, (i.e., we model a generalization net rather than a generalization hierarchy). However, the generalization net cannot contain cycles; this constraint is not modeled below.

This model will provide the information needed to support any inheritance scheme we may want to adopt.

To complete the two recursive definitions, we must model the fact that an aggregate product description and a generalized product description are themselves product descriptions.

To summarize, we have now modeled how aggregate and generalized process and product descriptions can be defined from other process and product descriptions. Since process and product description instances are tied to their respective type by insertion, we can summarize our complete model (the dashed arrows indicate "member" relationships that are maintained through insertion).
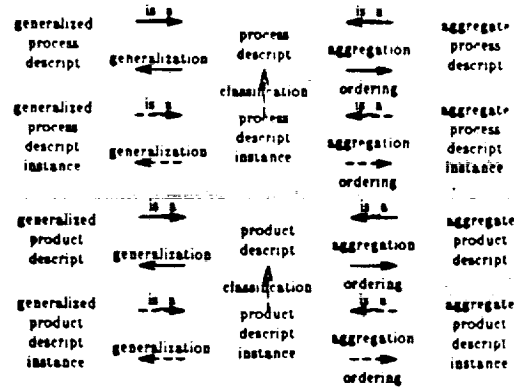
All the non-lexical domains are represented by surrogates. Any information about the objects modeled by these surrogates, including their lexical representation, will be connected to the surrogates.

The fundamental relationship between process executions and product instances is that a **process execution uses a set of product instances as input and produces a set of product instances as output.** To model this at the process and product description level, we need the following relation. The i/o domain consists of the values {i, o, io}.
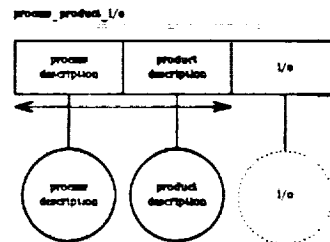
Some software methodologies require detailed i o information for each element in a document rather than for the document as a whole. This requirement is supported by our model through the use of the recursive definition of process and product descriptions.

The concept of mapping is introduced to allow process and product descriptions in a project using one software methodology to be compared to process and product descriptions in a project using a different software methodology. We must provide data structures that help the software engineer define mappings between process and product descriptions in different software methodologies. We model a rudimentary mapping definition capability below.
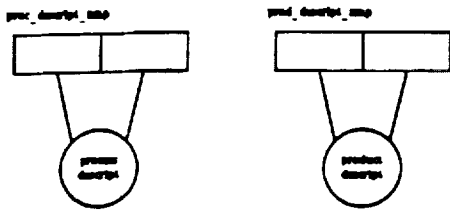
Before a mapping can be defined, we may have to use the recursive process and product definition capability in order to bring the concepts we want to compare to the same level of abstraction. Once this is done we can use the mapping definition capability.

The notion of user views is very important. User views are needed for managers, designers, programmers, etc. In general, a user view is defined as a consistent collection of product and process descriptions together with a collection of product instances and data about process executions that conform to the descriptions and are relevant to a particular project.
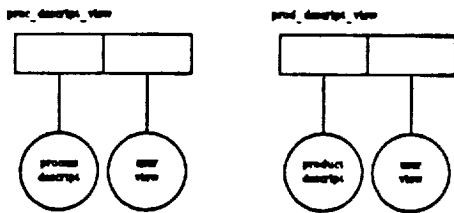
The notion of a process pre–constraint and post–constraint on a database is as important as the notion of a control mechanism in software engineering. Since different pre–constraints and post–constraints may apply to the same process decription used in different aggregate process descriptions, we have to tie the relationship between process description and constraints to a particular aggregate process description. A simple model is illustrated below.
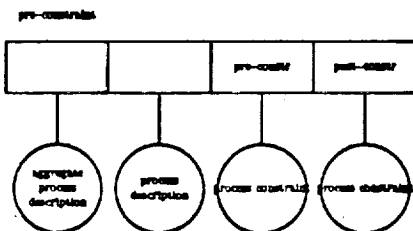
Like static constraints in a database, we think about the notion of product constraint as something independent from the processes that use and produce the product. We therefore model product constraints as follows:
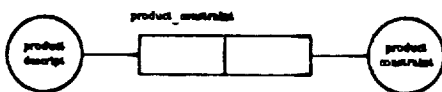
We have introduced a large number of database constraints between the model of process and product descriptions and the instances of these descriptions. The most natural way of maintaining consistency between the surrogates in an aggregation and generalization hierarchy is through the use of a well defined set of operations for insertion and deletion.

Maintaining consistency between the lexical representations is a much more complicated problem. Fortunately, part of this problem has a very elegant solution.

To control the consistency of lexical representations we only store the lexical representations of the atomic process and product descriptions, an object is atomic if it is not defined as an aggregate or a generalization. Lexical representations of aggregate and generalized process and product descriptions should merely refer to the other aggregate and generalized process and product descriptions and to the atomic process and product descriptions directly used in their descriptions. To avoid storing multiple almost identical copies of atomic process and product descriptions, we shall investigate incremental file representation techniques where a new file which is an almost identical copy of an existing file is represented by a pointer to the existing file plus a file differential. Techniques of this nature are discussed in [Roussopoulos 87].

The lexical representation of non–atomic objects can be materialized through the use of relational views.

Based on the above discussion we can now model the storing of lexical representations of atomic process and product descriptions. What these lexical representations look like, will of course depend on which language we choose for their representation.

Currently available database management systems do not directly support the storing of large, variable size, unstructured lexical objects. A possible but not very desirable solution is to develop a program that stores these objects on files under operating system control and stores addresses of the files under database control.

We model the lexical representation of atomic objects as follows:
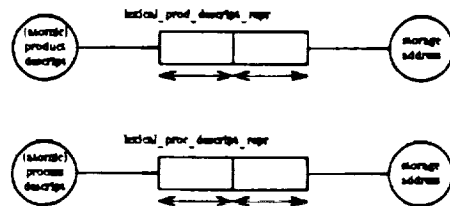
A version normally refers to an object that is almost identical to another object. In our model, the concepts of process description and product description can be used to model the notion of version, and we shall not introduce versions as a separate concept.

A configuration normally refers to a collection of versions. Again, the concept of configuration will not be introduced as a separate concept because it can be modeled by the concepts already defined.

The concept of measurements has recently been the subject of considerable attention in software engineering. Measurement can be perceived as a product instance or a process execution. A measurement can be part of a product instance or a product instance in its own right, or measurement can be part of a process execution or a process execution in its own right. A measurement can therefore be described by or as part of a product description, or it can be described by or as part of a process description. Therefore, we shall not introduce measurement as a new concept.

Process executions and product instances have several time attributes associated with them. Examples are the actual start and end times of process executions and the actual time of creation of product instances. Examples of time attributes for process and product descriptions are time of creation and last time executed and instantiated. Other time attributes are defined on a relative time scale, (e.g., one process execution must preceed another one).

Time attributes are, however, examples of measurements, and we shall therefore not introduce the time concept explicitly at this stage.

The purpose of this section has been to provide a formal schema definition that completely and correctly mirrors fundamental concepts in our process and product specification language independently of their lexical representation.

The next step is to define the lexical object-name-sets that will allow us to reference and represent the concepts. It is very important to understand that the information base is completely blind with respect to the internal structure of the object-names; it cannot see, use, or maintain any internal structure of object-names, (e.g. an object-name-set may consist of a set of Ada programs, but they all look like text strings to the information base). This implies that the maintenance of any structure of or constraints between object-names is the sole responsibility of the users and software tools accessing the information base.

Self-Describing Database Systems

A Self-Describing Database System is unique in that it provides an active and integrated data dictionary as part of the database management system. Such a data dictionary system is essential in our system.

The architecture of a Self-Describing Database System is illustrated below, [Mark 85]. This architecture has recently been adopted by the ANSI SPARC [Burns 86] as the basis for a new Reference Model for database management systems, and it is the basis for current work in the ISO.
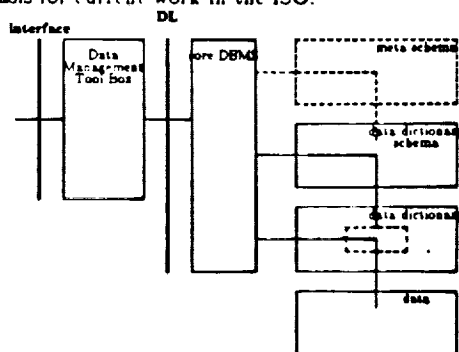
Architecture of a Self-Describing Database System

The core DBMS supports the well-known point-of-view dimension of data description which consists of internal, conceptual, and external schemata. In addition, it supports and enforces the intension-extension dimension of data description. The intension-extension dimension has four levels of data description. Application data are stored as data. The application schemata, describing and controlling the use of the application data, are stored in the data dictionary. The rules for defining, managing, and controlling the use of the application schemata are stored in the data dictionary schema. A fundamental set of rules for defining schemata, (i.e. a description of the data models supported by the Self-Describing Database System), is defined in the meta-schema. The set of rules in the meta-schema will allow the management strategies represented in the data dictionary schema to evolve in accordance with changing data management policies. Each level of data description in the intension-extension dimension is the extension of the level above it, and the intension for the level below it. The meta-schema is self-describing, (i.e. it is one of the schemata it describes).

The core DBMS can be thought of as a DBMS stripped to the bones. It supports the Data Language, DL, which is the only language used to retrieve and change data and data descriptions at any level in the intension-extension dimension. The DL provides a set of primitive operations on any data element or data description element at any level in the intension-extension dimension of data description. Any compound operations needed must be implemented as a tool in the Data Management Tool Box using the primitive operations of the DL. Data Management Tools are plug-compatible with the core DBMS through the DL.

The basic idea behind our generator approach is to make the schema designed in section 4 part of the data dictionary schema above. By doing this, the process and product description instances created through this schema will be stored as data in the data dictionary. These data may in turn be interpreted as an application schema controlling process executions and product instances in a specific software engineering project. The data describing these process executions and product instances will therefore be stored as part of the application data.

To make this work, the semantics of insert operations used through the data dictionary schema must guarantee that: 1) process and product descriptions are inserted in the data dictionary, and 2) data structures are created at the application data level to hold process execution and product instances conforming to these descriptions.

The Generator Approach

To understand the philosophy behind the generator approach we will consider the data dictionary schema (catalog) of a self-describing database system.

One of the most important things contained in the data dictionary schema is a relation of relations. Simplified, it looks something like this:

As can be seen, the first few tuples in the relation of relations contain a definition of that relation itself (its is self-describing) and other relations describing the relational data model. The next set of tuples define the first two relations we defined in section 4, namely "aggregate_process_descript" and "process_descript_name".
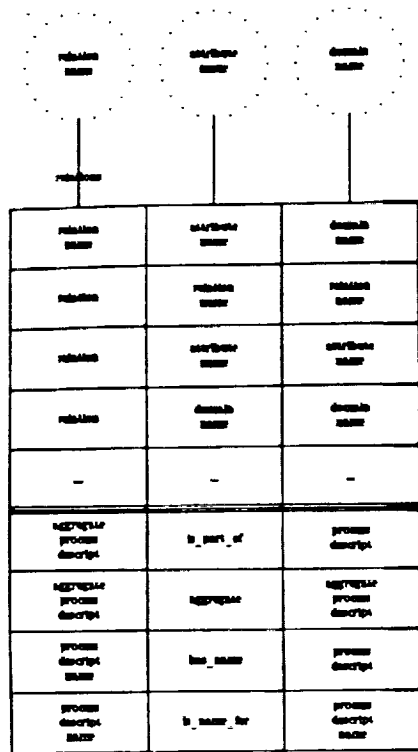
Whenever, a set of tuples is inserted into this relation of relations, the semantics of the insert operation further trigger the creation of an empty structure to hold the extension (data) of the defined relation. This means that the insertion of the tuples defining "aggregate_process_descript" and "process_descript_name" in the above example will result in the creation of an empty structure in the data dictionary to hold the extension of these relations.

Let us now turn our attention to these empty structures. When we insert tuples in them we are inserting data that in turn can be interpreted as defining an application schema.

In the illustration above we have inserted a set of tuples that constitute an aggregate process description. The aggregate process description defines the development (dev) process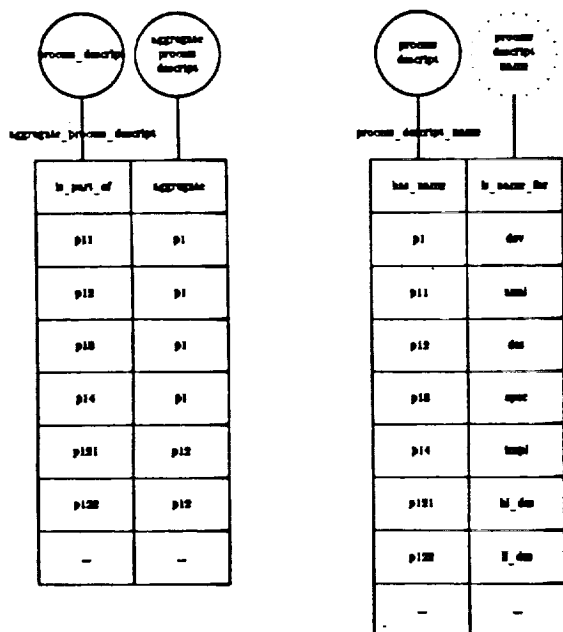 to consist of analysis (anal), design (des), specification (spec), and implementation (impl), and it defines the design to consist of high-level design (hl_des) and low-level design (ll_des). The p-values are surrogates produced by the system.

These aggregate process description will result in the creation of two empty structures at the application data level, one for the aggregate development process and one for the aggregate design process. Into these empty structures we can store data about specific executions of the defined development and design processes.

We could continue the example by 1) inserting into the relation "aggregate_ process_ descript_ seq" tuples defining the sequence of the processes in the development process. 2) inserting into the relation "aggregate_ product_ descript" tuples defining the products relevant to the development process, and 3) inserting into the relation "process_ product_ i,o" tuples defining which processes use and produce which products.

However, the example we have given is hopefully sufficient to illustrate the idea.

Conclusions and Future Research

Ideally, the information base for software engineering described in this paper will provide support for the automatic generation of an information base from a formal specification of a set of process and product descriptions. In order to further develop this idealized information base, more research in the areas of software engineering and databases is required. Although we only list the major database research issues, we strongly believe that success in this research area will depend on the tight cooperation between the two areas. (The software engineering research issues are listed in [Rombach 88]).

Future Database Research Issues

There is currently no data model, let alone a database management system, capable of supporting a meta information base for software engineering. One of the goals of our research is to develop the concepts and tools that are missing. For now, we are taking a very conservative approach, trying to use and extend existing relational database technology to see how far it will take us. There are especially two things from existing relational technology that we would like to preserve: a non-procedural calculus query language, and a constraint definition capability based on this calculus. We see no conflict between preserving these and at the same time providing a more object oriented data manipulation interface between the software engineering oriented model and the database model on which we have concentrated in this paper.

> Providing an object oriented data manipulation interface between the software engineering oriented model and the database oriented model will be our next major research topic.
>
> We plan to use the insert, delete, and update operations provided in the relational calculus to program transactions that will allow us to create, aggregate, decompose, generalize, specialize, and delete process and product descriptions in a consistent way.

An information base for software engineering must ideally be adaptable to meet the needs for continuously tailoring software engineering processes and products to changing project needs and characteristics of the project environment and the organization.

An important research issue is therefore the handling of data when its corresponding schema changes. The self-describing database system provides an ideal framework for investigating this issue.

Given a formal specification of a programming language, a document form, etc., it is theoretically possible to automatically produce the schema needed to explicitly represent the internal structure of all objects produced according to the formalism, is it practical?

A major research question is where the "invisible" part of the database ends and the "visible" part begins. Our approach to this question is to try to push the existing database technology as far as possible.

However, we will eventually have to face the problem of complex lexical object types.

A major research problem is therefore the support of extensibility which allows for user defined lexical object types.

Two possible solutions, representing the main streams in object oriented database research, are to provide tool access to complex lexical objects through the query language or to store user defined operations on complex lexical objects in the database. The difference between the solutions is minor.

A long list of additional research problem can be derived from the list of specific execution requirements presented in section 2. Many of these research problems are discussed in [Bernstein 87] and are not repeated here.

References

[Bernstein 87]
P. A. Bernstein, "Database System Support for Software Engineering," Proceedings of the Ninth International Conference on Software Engineering, Monterey, CA, March 30 –April 2, 1987, pp. 166–178.

[Codd 79]
Codd, E.F., "Extending the Database Relational Model to Capture More Meaning," ACM TODS 4, No. 4, December 1979.

[Hall 76]
Hall, P., Owlett, J., Todd, S.J.P., "Relations and Entities," In G.M. Nijssen (ed.), "Modeling in Data Base Management Systems," North-Holland 1976.

[Mark 85]
Mark, L., "Self-Describing Databases – Formalization and Realization," TR 1484 Department of Computer Science, University of Maryland, April 1985.

[Roussopoulos 87]
Roussopoulos, N., "Incremental Computation Models," Department of Computer Science, University of Maryland, 1987

[Rombach 88]
H.D. Rombach, L. Mark, "Software Process and Product Specifications: A Basis for Generating Costumized Information Bases," Submitted for HICSS-22. 1988.

[Dittrich 86]
K. Dittrich, "Object-Oriented Database Systems: The Notion and the Issues," Proceedings International Workshop on Object-Oriented Database Systems, Pacific Grove, CA, Sept. 1986.

[Dadam 86]
P. Dadam et al., "A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View of Flat Tables and Hierarchies," Proceedings SIGMOD Conference, Washington, DC, 1986.

[Borgida 88]
Alexander Borgida, "Modeling Class Hierarchies with Contradictions," Proceedings SIGMOD Conference, Chicago, 1988.

[Carey 88]
M. Carey, D. DeWitt, S. Vanderberg. "A Data Model and Query Language for EXODUS," Proceedings SIGMOD Conference, Chicago, 1988

[Thomas 86]
S.J. Thomas, P.C. Fischer. "Nested Relational Structures," The Theory of Databases, P.C. Kannellakis, ed., JAI Press, 1986.

SOFTWARE PROCESS & PRODUCT SPECIFICATIONS:

A Basis for Generating Customized SE Information Bases

H. Dieter Rombach and Leo Mark

Department of Computer Science
and
Institute for Advanced Computer Studies (UMIACS)
University of Maryland
College Park, MD 20742

Abstract

Software Engineering is a challenging new application area for information bases. The new challenges are twofold: software engineering specific (how can we model software engineering processes and products properly?) and information base specific (how can we mirror such processes naturally within an information base?). Meeting these challenges requires joint software engineering/information base research. The Meta Information Base project at the University of Maryland represents such a joint research effort. This project aims at generating customized software engineering information bases from formal specifications of software engineering processes and products. The three central research topics are to develop (i) a software process specification language which allows us to capture all the information necessary to understand, control and improve any given software engineering process, (ii) an object oriented information base schema language which allows us to model the mirroring information base structure for any such software engineering process, and (iii) a mapping between the software engineering oriented and information base oriented models. If an information base is truly expected to mirror a given software engineering process, it needs to be tailorable to the changing characteristics of the software process itself. The generator-based approach suggested in our project seems to be the natural approach to satisfy this important need. Software process and product specifications are expected to have not only an impact on generating customized software engineering environment components (such as information bases). Systematic improvement of software processes and products - learning about software engineering approaches and reusing software engineering related experience - can not be achieved without having a specification of the objects we want to improve. This paper discusses general requirements for software process specification languages, presents a first prototype software process specification language, demonstrates the application of this language and derives software engineering related requirements for a supporting information base. The actual efforts aimed at implementing these information base requirements are briefly mentioned in the conclusions.

1. Introduction

Lessons learned from having monitored the software development and maintenance process over a decade [1, 11] suggest a high-level improvement oriented software engineering model consisting of planning, execution, and learning & feedback stages [4]:

- **Planning** the software engineering process is aimed at defining plans for developing quality a priori. It includes choosing the appropriate overall process model as well as the specific methods and tools supporting this process model. It involves tailoring each of them for the project specific goals and the characteristics of the project environment and the organization. Process models, methods and tools need to be planned for construction as well as learning and feedback. The effectiveness of this planning process depends on the precision in the specification of the process models, methods and tools (formal is better than heuristic) and the experience concerning their effects. The entire planning process as well as the tailoring process need to be formalized.

- **Execution** of the software engineering processes follows the plans derived during planning; the existence of construction guidelines helps in assuring that process models, methods and tools are being used as intended. It should be noted that execution includes the construction of the traditional project documents (e.g. requirements, design, code) and all other kinds of information prescribed by the planning process (e.g., test results, schedule, effort data), as well as the analysis of the construction processes and resulting products from various (during planning prescribed) perspectives.

- **Learning and feedback** follows the plans defined during planning. Learning is in part based on the analysis results derived during execution of processes (e.g., regarding the use of process models, methods and tools) as well as products. We compare the actual results with the desired results, and feed the lessons learned back into the ongoing project (which might result in iterating the project plans) or into the planning of future projects. Feedback is important to engineers and managers. An effective feedback mechanism is especially crucial for supporting the complex management decision process.

Software engineering processes need to possess the attributes tailorable and tractable. Tailorability is required in order to plan the software engineering process for the project specific goals and project environment characteristics. Tractability is required in order to specify processes in an understandable way, construct products according to these plans, and monitor the construction for the purpose of feedback and learning. The TAME (Tailoring A Measurement Environment) project at the University of Maryland aims at the development of a measurement, feedback and planning environment for software engineering [4]. Part of this project is to develop a software engineering information base. The development of a process and product specification language (although necessary) is not part of the current scope of the TAME project.

Our Meta Information Base project project at the University of Maryland represents a joint software engineering/information base research effort. The basic idea of this approach is to generate customized software engineering information bases from formal specifications of software engineering processes. The three central research topics are to develop (i) a software process specification language which allows us to capture all the information necessary to understand, control and improve any given software engineering process, (ii) an object oriented information base schema language

which allows us to model the mirroring information base structure for any such software engineering process, and (iii) a mapping between the software engineering oriented and information base oriented models. The generator approach acknowledges the fact that software engineering processes change from environment to environment, but also from project to project. If an information base is truly expected to mirror a given software engineering process, it needs to be tailorable to the changing characteristics of the software process itself. The generator-based approach suggested by our project seems to be the natural approach to satisfy this important need. Generating customized software information bases is not the sole application of software process specifications. We are also investigating the benefits of software specifications for the purpose of better understanding, planning and improvement of software engineering related aspects. We believe that learning about software engineering and reusing software engineering related experience can not be done in a systematic way without specifying the objects of learning and reuse - the software processes - themselves. In order to do a good job of learning and reuse, measuring and analyzing the software processes and their effects seems to be a very helpful mechanism. We therefore suggest not only to model the construction oriented software engineering aspects, but also the analysis oriented ones.

Based on our improvement oriented TAME software process model, we anticipate the following framework for supporting software engineering processes (see figure 1):
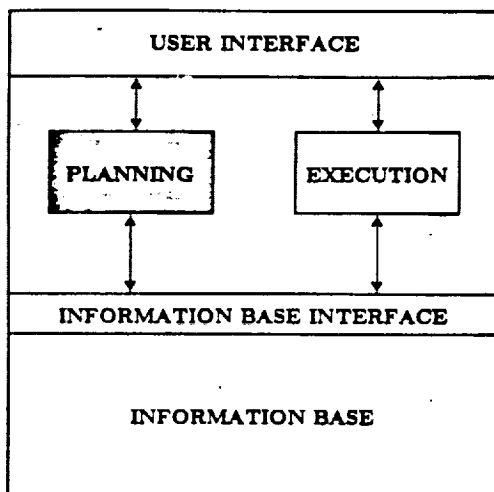
```
+-------------------------------------------------+
|                                                 |
|              USER INTERFACE                     |
|                                                 |
|   +----------------+    +----------------+      |
|   |                |    |                |      |
|   |   PLANNING     |    |   EXECUTION    |      |
|   |                |    |                |      |
|   +----------------+    +----------------+      |
|                                                 |
+-------------------------------------------------+
|                                                 |
|        INFORMATION BASE INTERFACE               |
|                                                 |
+-------------------------------------------------+
|                                                 |
|            INFORMATION BASE                     |
|                                                 |
+-------------------------------------------------+
```

Figure 1: Framework for SE Process Support

Each software engineering project consists of a planning and execution stage. During the planning phase plans (specifications) of all project relevant processes and products get developed; the execution stage consists of conducting the project according to these plans. The underlying information base stores all process and product plans as well as the information derived during execution of these plans. The plans themselves provide the basis for structuring the execution-derived information. Storing such information across projects results in historical information bases. Improvement can then be achieved by structuring this information appropriately (based on process plans), and reusing it during the planning and execution phase of future projects after tailoring it to the specific characteristics of these future projects. Figure 1 suggests that we need to specify software processes and products for different purposes: to support the planning activities at the user interface, to allow the internal representation of plans, and to support the storage and retrieval of plans and information derived during exe-

cution according to plans. In our project, we expect to use three different (but compatible) specification languages in order to satisfy the different needs of each perspective.

This paper presents the software engineering oriented part of our joint project . It discusses general requirements for software process specification languages, presents first prototype software specification languages (one to support the planning activities at the user interface, one to represent plans internally), demonstrates the application of these prototype languages, and derives software engineering related requirements for a supporting information base. The information base related work of our project, aimed at implementing these software engineering oriented information base requirements, is not part of this paper.

2. Requirements for Software Process & Product Specification

We distinguish between very general requirements which acknowledge the basic nature of software processes, and more concrete requirements whose relative importance depends on the purpose of software process and product specifications.

General requirements for a software process specification language include the ability to

1. specify all aspects that seem to be important within a given software project (and not to be limited to a specific set of aspects): This requirement acknowledges the fact that there exist no commonly accepted software process models today.
2. specify with varying degrees of detail and to refine initial specifications in the future as we learn: This requirement acknowledges the fact that our understanding of some processes is insufficient, of others is pretty precise.
3. deal with creative and mechanical aspects of software processes in different ways (e.g., behavioral specifications for creative aspects and algorithmic specifications for mechanical aspects): This requirement acknowledges the fact that software processes include both creative and mechanical aspects, and that we must deal with both in a natural way.
4. easily modify process specifications: This requirement acknowledges the constant need for tailoring process specifications to changing project or environment needs.

Specific requirements (from a planning perspective) for a software process and product specification language include the ability to specify

1. process, product, and constraint types
2. use (input) and produce (output) relationships between process and product types
3. (pre- and post-) condition relationships between constraint and process/product types (A pre-condition of a process is a constraint imposed upon initiation of this process; a post-condition of a process is a constraint imposed upon termination of this process; a condition of a product is a constraint imposed upon this product.)
4. control flow relationships between process types (sequence, alternation, iteration, and parallelism)
5. structural relationships between product types (sequence, alternation and iteration)
6. dependency relationship between process types (Process P1 is dependent on process P2, if every execution of P2 triggers simultaneous execution of P1. Typically, measurement processes are dependent on the construction processes they are supposed to monitor.)
7. aggregation and decomposition of process and product types

---

* We present another paper discussing the information base oriented part of our project during the 'Database Formalisms, Software & Systems' session of this same conference [10].

8. generalization and specialization of process and product types
9. constructive as well as analytic (measurement oriented) product and process types
10. different roles (Different roles are performed in a software project such as design role, test role, quality assurance role, or management role. Roles define views or perspectives of (a subset of) the processes and products relevant to a particular project. Type and number of roles may change from project to project.)
11. time (relative and absolute) & space (software structure, versions, configurations) dimensions
12. non-determinism due to user interaction

Specific requirements (from an execution perspective) for a software process and product specification language include the ability to handle

1. the instantiation (creation of objects) of process, product and constraint types
2. long-term, nested transactions (Many software engineering processes such as designing may stretch over weeks or months; in addition, they may contain nested activities.)
3. varying degrees of persistence (Some information needs to kept forever, some only for the duration of the project, and others only until a new instance (e.g., product version) has been created.)
4. tolerance of inconsistency (Because of the long-term nature of software engineering processes, it might be necessary to store intermediate information that does not yet conform with the desired consistency criteria.)
5. dynamic types (type hierarchies) (an object of type product (e.g., a compiler developed during one project) may be used as an object of type process during a future project.)
6. dialogues between processes (including human beings)
7. dynamic changes of process specification types (It is impossible to plan for all possible (non-deterministic) results produced by human beings in advance. However, we would like to react to those situations by dynamically re-planning during execution. Although, it is no problem to change a specification during the planning stage, it might be a problem to do it during execution and preserve the current execution state.)
8. back-tracking due to execution failures
9. the organization of historical sequences of product objects and process executions
10. the enormous amounts of interaction between parallel activities
11. the role-specific interpretation of facts (The same process and product facts might require different interpretation in the context of different roles.)
12. the triggering of actions (based on pre- and post-conditions)

The list of execution-point-of-view requirements is heavily influenced by the results of a working group during the 4th International Workshop on Process Specification (Moretonhampstead, UK, May 1988), chaired by Tom Cheatham [12].

3. Prototype Process & Product Specification Languages

Several research projects are working towards improving the software development process from various perspectives: Arcadia [13], TAME [2, 3, 4], GENESIS [15], and others [12]. No consensus seems to be reached as to what an appropriate specification language should look like in order to be both capable of describing the important process and product aspects and acceptable to the intended user.

We believe that no single specification language will satisfy the needs of software engineers as well as the designers of the information base. Based on our SEE model in figure 1, we believe that there is a need for at least three different language representations:

- the application level language, which is used to support the task of specifying the relevant process and product aspects during the planning stage (at the user interface of our SE process model in figure 1). This type of specification language should accommodate the needs of its potential users (e.g., software engineers, managers).
- the intermediate level language, which is used to represent the results of the planning stage. This type of specification language should emphasize completeness, consistency, and preciseness. Complete in this context means executable, independent of whether this execution requires user interaction or not.
- the information base level language, which is used to formulate the storage and retrieval needs of software processes and products. These needs encompass the process and product specifications themselves as developed during the planning stage, as well as the information accumulated during the execution of those plans during the execution stage. This kind of language is usually referred to as schema language.

In addition, we need to provide for transformations between adjacent language levels. The application level language representation of a particular software engineering process or product (e.g. the design process) eventually needs to be transformed into the appropriate information base level language representation (schema). This transformation must preserve consistency. The intermediate level language representation can be looked at as a reference representation acceptable to both the software engineering and information base perspective. The separation provides independence of application and information base representations and it allows us to to separate the entire research area into two clearly distinguished but connected (via the intermediate level) areas. Ideally, these transformations should be automated; this would allow us to completely hide the information base view from the software engineer and vise-versa.

In the following two subsections, we introduce first prototype languages for the application and intermediate level.

3.1. A Prototype Application Level Specification Language

Our prototype process and product specification language for the application level is graphically oriented. At this point it provides graphical elements satisfying the first eight specific planning oriented requirements listed in section 2:

1. **Three kinds of object types: process types (represented as boxes), product types (represented as circles), and constraint types (represented as rhombs).**
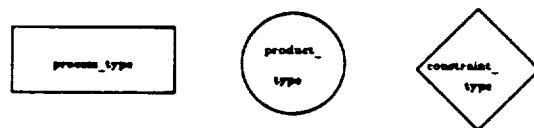
Figure 3.1(a): Object Types

The concept *process* is used for all kinds of software engineering activities. It comprises the elements of our high-level software engineering model (planning, construction, learning and feedback), overall software process models such as the "water fall" [7, 16], "iterative enhancement" [5] or "spiral" [8] model, complex methodologies such as the "Cleanroom" [9] methodology, particular methods and tools such as "top-down design" or "Jackson design", and even individual statements of an automated tool.

The concept *product* is used for all kinds of software engineering information. It comprises the plans for construction and learning and feedback produced by the planning process of our high-level software engineering model, deliverable products produced by

the construction process such as requirements, design, code, but also test data, schedule, resources, and all kinds of measurement data.

The concept *constraint* is used to represent all kinds of software engineering conditions (pre-conditions for the execution of a process and post-conditions which are checked at process termination time). Constraints may also be imposed on products. Constraints are used to model schedules, completeness criteria or any other kind of quality or productivity characteristic. Constraints are basically expressed as boolean expressions.

2. **Two kinds of relations between process and product types: the use relation (represented as a solid arrow connecting a product and a process type) and the produce relation (represented as a solid arrow connecting a process and a product type)**
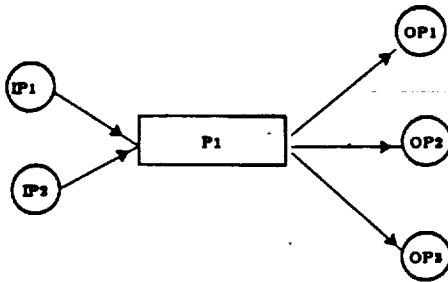

Figure 3.1(b): Use/Produce Relations

In figure 3.1(b), process of type P1 uses products of type IP1 and IP2 and produces products of type OP1, OP2, and OP3.

The relations *use* and *produce* are used to explicitly express all kinds of information needed for executing a process and resulting from its execution. Used information can range from experience (for example, in the form of historical data), to products produced during the same project by other processes, products produced during prior projects, and characteristics of the project and project environment. Produced information can range from deliverable products (e.g. design or code documents) to measurement data or even new process and product descriptions based on learning.

3. **Three kinds of relationships between constraint types and process or product types: pre-condition, post-condition, condition (represented as solid double arrows).**
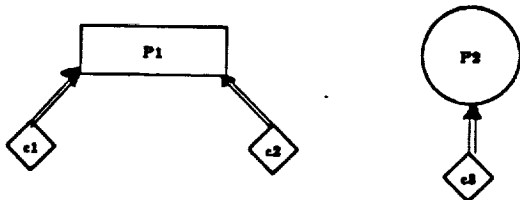

Figure 3.1(c): Constraint Relations

In figure 3.1(c), constraint of type c1 is a pre-condition for a process of type P1; constraint of type c2 is a post-condition for a process of type P1; constraint of type c3 is a condition of a product of type P2.

The *constraint* relationships are used to explicitly express all conditions that need to be fulfilled before start or after termination of a process, but cannot be expressed via use/produce relationships or explicit control flow relationships between processes.

Examples are schedule, and all kinds of quality and performance requirements. In addition, constraint relationships are used to express expected characteristics of a product; e.g., maximum complexity.

4. **Four kinds of control flow relations between process types: the sequence, alternation, iteration and parallelism relations (represented as solid arrows between process types; parallel control flow is indicated through the augmentation of the corresponding arrows with "||").**

The semantics of sequential control flow is obvious. The semantics of alternate control flow is to execute exactly one of the alternatives. The selection criterion can be expressed in terms of a pre-condition on each of the alternative processes. Alternation is completely deterministic if each of the alternate processes possesses a pre-condition and all pre-conditions are mutually exclusive. It is possible to have nondeterministic alternation (no constraints) or incomplete alternation (no alternative applies under certain circumstances). The semantics of iterative control flow is to execute some process repeatedly. The negated termination criterion is provided in form of a pre-condition to the iterated process. It is possible to specify indefinite iteration (no termination constraint). The semantics of parallel control flow is to execute all parallel processes independent of each other. However, all of them must be completed in order to satisfy the parallel control flow relation.
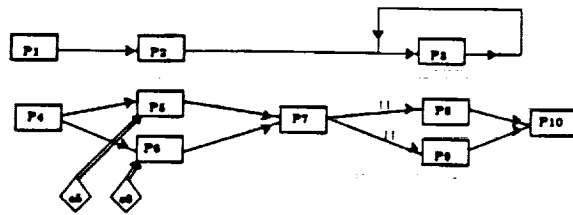

Figure 3.1(d): Control Flow Relations

In figure 3.1(d), process of type P1 is in sequence with process of type P2, processes of type P5 and P6 are alternatively executed after P4, process of type P3 is iteratively executed, and processes of type P8 and P9 are executed in parallel (independently). The decision whether to execute P5 or P6 can be based on two mutually exclusive pre-conditions C5 and C6.

Note: The graphical symbols for data and control flow are distinguished by their context. Arrows representing data flow connect processes and products, whereas arrows representing control flow connect just processes.

5. **Three kinds of structural relations between product types: the sequence, alternation, and iteration relations (represented in the same way as control flow between process types).**

The relation 'sequence' indicates the sequential composition of two products; the relation 'alternation' indicates the alternate inclusion of either of two products, and the relation 'iteration' indicates the repeated occurrence of a product (0 or more times). We use the same relation names to express the control flow composition of process types and the structural composition of product types to minimize the number of concepts.

6. **A dependency relationship between processes (represented as dotted double arrows between two processes).**
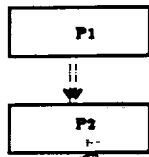
Figure 3.1(e): Dependency Relation

In figure 3.1(e), P1 is dependent on P2.

The dependency relationship is used to express a very tight form of parallelism between processes. The relation is directed and defines a master-slave relationship in the sense that whenever the master process is in execution, the slave process gets executed too. This means more than just to start end terminate at the same time; it means absolutely synchronized execution. This concept allows us to model the measurement of software processes. For example, if we have a design process and we would like to collect all the effort spent on designing, we model the design process as the master process and the effort measurement process as the slave process.

7. A relation between process or product types allowing for decomposition and aggregation: the is_part_of relation (represented as dashed arrows augmented with the relation name)
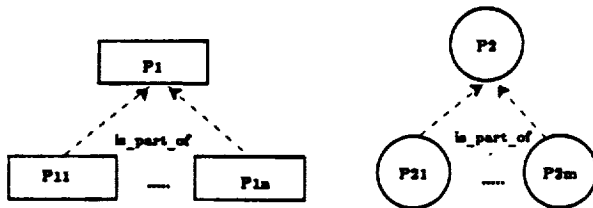

Figure 3.1(f): Decomposition/Aggregation Relations

In figure 3.1(f), process type P1 is decomposed into (and completely substituted by) process types P11 to P1n. Product type P2 is similarly decomposed into product types P21 to P2m.

We need to allow for *decomposition* and *aggregation* of process and product types. The decomposition is necessary to describe the refining of some process or product into more precise (less abstract) processes or products.

Note: Decompositions are level complete. This means, if a process type P1 is decomposed in process types P11, P12, P13, and P14 (see (6)), then these four processes together make up the entire functionality of P1 (they entirely substitute P1)!

For example, the overall process "development" might be refined into "requirements analysis", "design", "coding", etc.; similarly, we can refine the product "deliverables" into products "requirements document", "design document", "source code documents", etc.

Decomposition is also necessary in order to reflect the hierarchy of product structure. For example, "system" might be recursively decomposed into "subsystems", "components" and "modules".

We use the concept *process* recursively in two ways. Each process type can be *decomposed* into lower-level process types or can be included into the *aggregation* of higher-level process types. This use of the term process can reduce the difference between an informal method and a concrete automated tool supporting this method to a difference in the degree of formalism in the specification. Whereas the method might be described in informal English, the tool might be the complete algorithmic formalization of the same process. The second possibility of using pro-

cess types recursively is *specialization* and *generalization*. In the case that one method can be automated by a variety of tools alternatively, we can view the tools as specializations of the method, or the method as a generalization of those specific tools.

We use the concept product recursively in the same two ways as processes.

The relations *sequence*, *alternation*, *iteration* and *parallelism* (see 4.) are used in the context of decomposing and aggregating process or product types.

The semantics of these relations in the context of a process type decomposition is as follows: Each decomposed process type either (a) inherits the entire set of use and produce relations of the aggregated process types, (b) inherits parts of the use and produce relations of the aggregated process types, (c) uses product types produced by a different decomposed process type and produces product types to be used by a different decomposed process type, or all possible combinations of (a), (b), and (c). According to (2), each decomposed process type requires at least one product type for use and production. The functionality of the aggregated process type is identical to the functionality achieved by all decomposed process types if executed according to their control flow relationships.

8. A relation between process or product types allowing for specialization and generalization: the is_a relation (represented as dashed arrows augmented with the relation name)
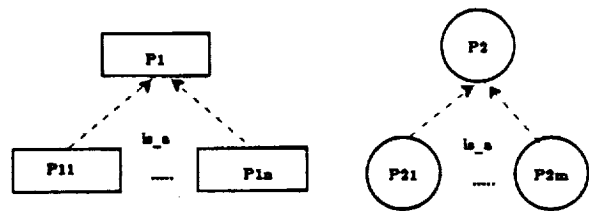
Figure 3.1(g): Specialization/Generalization Relations

In figure 3.1(g), each of the process types P11 to P1n is a specialization of process type P1, and each of the product types P21 to P2m is a specialization of product type P2. P1 is a generalization of each of the P11 to P1n and P2 is a generalization of P21 to P2m.

We need to allow for *specialization* and *generalization* of process and product types. Generalization of a set of process and product types allows to group them according to some common aspect.

For example, we can generalize compilers for all kinds of languages to a general compiler process that translates an algorithmic source code document into object code. Another example is viewing all tools supporting a specific method alternatively as specializations of that method.

In addition we satisfy the specific planning oriented requirements 9 and 10 as follows:

9. We encourage the awareness of constructive and analytic process and product aspects. Again, the control flow and structural relations defined for the graphical notation allow for representing all kinds of decompositions and aggregations. The success of software projects depends on a sound integration of constructive and analytic aspects as indicated by our high-level software engineering model. This fact does not mean that we should not view them as different aspects.

The *constructive* aspects are concerned with generating products, while the *analytic* aspects are concerned with secondary informa-

tion derived from monitoring and analyzing constructive processes and products.

10. We allow for the definition of different roles. Roles are defined as projections onto the set of process and product types defined for some project. They define specific views or perspectives. Different views may include the same process or product type. For example, the design role and the quality assurance role may both be interested in the design product, but from very different perspectives. Whereas the design role is interested in how he can build the design product best, the quality assurance role is emphasizing the adherence of the design product to stated quality requirements. One role may be performed by several people, or one person may execute several roles. The number of roles is not predefined, but rather project specific. Roles are defined explicitly. In practice, different roles will very often be specified by people with different project experience.

We believe that the concepts and principles presented in this section provide a promising basis for building process and product specification languages. The objective is to be able to specify all aspects of a software process or product (completely), according to a set of unifying principles (consistently), and to the level of detail possible due to the nature of the problem and our understanding (precisely).

3.2. A Prototype Intermediate Level Spec. Language

The intermediate level specification language is specified in BNF-style. Appendix (A) contains the syntax rules necessary to specify process types. The necessary context rules are not included in Appendix (A).

This language allows us to specify a given process type (or product type) at any desirable and possible level of detail. Each process type specification consists of a *process_heading* and a *process_body*. The process_heading describes the unique process_type_name, whereas the process_body contains the actual specification. The process_body consists of a *process_specification_part*, a *role_specification_part*, and a *resource_assignment_part*. The process_specification_part contains an *interface_part*, a *refinement_part*, and an *implementation_part*. The interface_part characterizes the used and produced product types and the attached constraint types. The refinement_part describes the refinement of this process type into lower level processes (includes also the refinement of the related products and constraints) and defines their connections at this lower level. The implementation part contains the final algorithmic implementation of a process type. Refinement and implementation parts exclude each other; either a process type gets refined further or it is at its final level of detail. Refinement and implementation parts are optional. The role_specification_part defines all roles. Roles can be viewed as 'super' processes. The resource_assignment_part assigns resources to processes and/or roles. The resources are specified like product types. This includes the ability to refine them. If we have several organizations of people involved in executing a certain process, we can model each organization as a resource consisting of people resources. The *product body* of product type specifications consists only of a *refinement_part* and *implementation_part*.

This prototype language allows us to satisfy the planning oriented specific requirements listed in section 2. The current language definition is by no means final. We plan on using it as an experimental vehicle allowing us to validate whether the chosen concepts are satisfactory for specifying all kinds of process and product related software engineering aspects.

4. Application of the Prototype Specification Languages

The validity and usefulness of our software engineering process model depends on whether we are able to (a) generate specifications for all kinds of process and product types using our languages, (b) make project personnel use the specification languages during planning as well as the generated specific models during execution and learning and feedback, and (c) generate an information base supporting the planning of process and product types (store and reuse) and the execution of instantiations of process and product types (e.g., instantiation itself, storing information accumulated during execution).

So far we have been able to specify a number of process and product types using our graphical notation. The specified process types include a variety of existing project models (e.g. [17]) as well as specific development methods. The completely automated version of a process type is a tool. We would be able to represent any structured implementation of a tool using our control flow relations.

The answer to part (b) requires more work. It seems that our process model and languages will be useful during planning for describing aspects of construction and learning and feedback as well as the consumed and produced products completely, precisely and as formal as possible. It should also help execution and learning and feedback in that it should be easy to follow these kinds of complete, consistent and precise plans. The degree to which execution can be supported will depend on the degree to which we will be able to satisfy the specific, execution oriented requirements listed in section 2.

Our initial answer to part (c) is presented in [10].

In this section we will apply our two prototype specification languages to a small example. We will introduce the example in section 4.1, demonstrate the use of the graphical prototype application level language in section 4.2, and show how the final plans can be represented using our prototype intermediate level language in section 4.3.

4.1. An Example

The example we have chosen to demonstrate the applicability of the two prototype languages is a subset of the design related aspects out of the context of a larger project.

The example can be characterized as follows:
Specify a process type for the design phase (named 'design') that consumes a requirements product type ('r') and produced a design product type ('d'). The design process type consists of two sequential design sub processes for high-level design ('hl_design') and low-level design ('ll_design'). We want to use methods for high-level design ('yourdon') and low-level design ('pdl'). The design process will start on date t_1, and has to be completed by date $t\_3 = t\_1+3$ High-level design should be completed by date $t\_2 = t\_1+1$. In addition, the actual effort spent for high-level design ('hleff') and low-level design ('lleff'), the number of low-level design errors ('llerr'), and McCabe's complexity of the low-level design products ('v') must be measured for quality assurance purposes. A low-level design product will not be accepted if its complexity value exceeds 20. The design process will be performed by five people. One person is assigned to perform the high-level design. Three people, including the person who performed the high-level-design, are assigned to perform the low-level design. A fourth person is assigned to perform the quality assurance activities; a fifth person is assigned to manage the project.

4.2. Specification of the Example (Application Level)

We apply our graphical notation to specify all aspects of the example described in section 4.1. except the assignment of people. The sequence of specification steps is not predefined. We have chosen to

3-16

specify the example according to the three identifiable roles (designer, quality assurance, manager):

1. DESIGN ROLE:

1.1. Specification of the use and produce relationships associated with process type 'design':
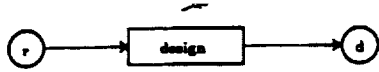

Figure 4.2(a): Specification of design process and products

Figure 4.2(a) describes our initial specification of the example described in section 4.1.

1.2. Decomposition of process type 'design' into 'hl_design' and 'll_design' (with sequential control flow) and decomposition of product type 'd' into 'hld' and 'lld' (with sequential structure):
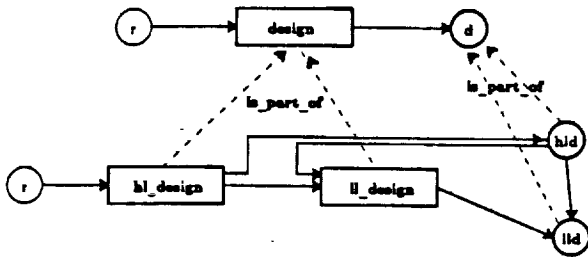

Figure 4.2(b): Decomposition of 'design' and 'd'

Figure 4.2(b) describes the decomposition of process type 'design' into process types 'hl_design' and 'll_design'. The control flow between 'hl_design' and 'll_design' is sequential; the aggregation of product types 'hld' and 'lld' into 'd' is sequential.

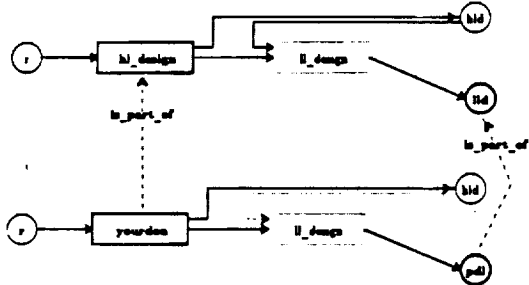1.3. Decomposition of 'hl_design' into 'yourdon' and 'lld' into 'pdl':

Figure 4.2(c): Decomposition of 'hl_design' and 'll_design'

Figure 4.2(c) describes the decomposition of process type 'hl_design' into 'yourdon' and product type 'lld' into 'pdl'. The decomposition relation is used to describe this refinement; in addition, we could also use the specialization relation to indicate that 'yourdon' is a specific instance of 'hl_design' and 'pdl' of 'lld'.

2. QUALITY ASSURANCE ROLE:

2.1. Specification of measurement oriented process and product types:
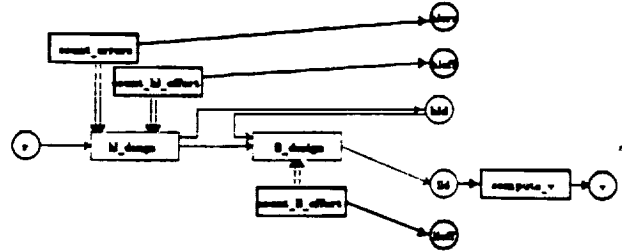
Figure 4.2(d): Specification of measurement processes

Figure 4.2(d) describes the measurement of 'hlerr' via process type 'count_errors', 'hleff' via process type 'count_hl_eff', 'lleff' via process type 'count_ll_eff', and the McCabe complexity 'v' via process type 'compute_v'. The process types 'count_errors', 'count_hl_effort' and 'count_ll_effort' are dependent on process types 'yourdon', 'yourdon' and 'll_design', respectively.

2.2. Specification of constraints:
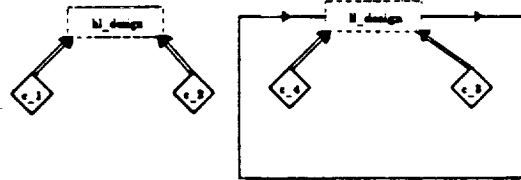
Figure 4.2(e): Specification of process and product constraints

Figure 4.2(e) describes how the constraint types c_1, c_2, and c_3 (which use the boolean expressions 'calendar_time = t1', 'calendar_time <= t2', and 'calendar_time <= t3') are assigned as pre-condition to 'yourdon', post-condition to 'yourdon', and post_condition to 'll_design' respectively. The constraint type c_4 which uses the boolean expression 'v (pdl) > 20' is assigned as a pre-condition to 'll_design' to indicate another iteration.

3. MANAGEMENT ROLE:

4.1. Specialization of 'hl_design' and 'lld':
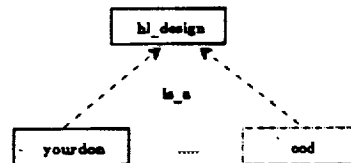
Figure 4.2(f): Specialization of 'hl_design'

Figure 4.2(f) describes how the specific design method 'yourdon' can be categorized as a specialization of the process 'hl_design'. There exist other possible specializations, e.g., the object oriented design method 'ood'.

4.3. Specification of the Example (Intermediate Level)

In this section we give an example as to how the specification information produced using the application level language in section 4.2 is represented internally using the intermediate level specification language. Each of the objects (process and product types) mentioned in section 4.2 is represented by a separate intermediate level specification. However, each specification will combine all the information that relates to a specific object completely, independent of

5207

the sequence in which it had been created.

As an example, we give the specification of the process type 'design'. This specification includes the refinement of 'design' into 'hl_design' and 'll_design' (step 1.2 in section 4.2.), and all related quality assurance aspects (see steps 2.1. and 2.2. in section 4.2.). The intermediate specification of this scenario is contained in Appendix (C).

The interface_part contains product types 'r' and 'd' as well as constraint types 'c_1' and 'c_3'. The next refinement level is described in terms of *decomposed types* (see decomposition_part), *imported process, product, and constraint type specifications* (see use_part), and *relations between all those types* (see connection_part). The '*' is used in the decomposition_part to indicate that a non-determined number of processes of type 'll_design' needs to be instantiated (for each module one). Each of these instantiations will produce a product of type 'lld'. The role_specification_part identifies all roles according to the way information was provided at the user interface level (see section 4.2.). In the resource_assignment_part, *people resources* (p_1, ..., P_5) are assigned to execute certain roles ('quality_assurance_role', P_5) are assigned to execute certain roles ('quality_assurance_role', 'management_role') and/or process types ('hl_design', 'll_design').

4.4. Specification of the Example (Information Base Level)

The example in APPENDIX (C) is only one of the specifications to be stored in a supporting software engineering information base. A complete list of specification objects according to our example is contained in Appendix (B). Remember, these objects comprise only the planning part of what needs to be stored in an information base. In addition, the information base must be capable of storing all the information derived during execution of these process type specifications.

5. Deriving Information Base Requirements

The role of software engineering information bases is to mirror the software processes and products relevant to a project or entire environment. Assuming our improvement oriented software process model (consisting of planning and execution stages for each project), a supporting information base needs to be capable of storing the process plans as well as the execution derived information. In our case, plans are the intermediate process specifications introduced in section 3.2 and demonstrated in section 4.3. These plans could also provide the necessary information for organizing the execution derived information. Obviously, in the case of our example in section 4, we would like to see all the plans listed in Appendix (B) stored in an information base.

A list of important requirements for designing an information base interface are identical with the requirements (general and specific) listed in section 2. Additional requirements can be found in [6, 14]. The specific planning oriented requirements that are expected to allow us to specify all aspects of software processes and products seem not to be the problem as far as the information base is concerned. It is not clear at this point, whether all the execution oriented specific requirements can be easily satisfied with state-of-the-art database technology. It is not even clear, whether all these execution-oriented requirements should be dealt with inside a persistent database at all. Our first approach to generating a software engineering information base from process and product specifications is described in [10].

The specification research goal of our project is to develop a formal language for specifying all aspects of software processes and products in a complete, consistent and precise way. We do not believe that all aspects can be formalized in an algorithmic manner. However, we believe that even those creative aspects can be described as integrated into the overall software development and maintenance process; this integration would make them accessible to control to a certain degree. We have developed first prototype language definitions and have them manually applied to specify small but realistic software engineering scenarios. This limited experience seems to indicate that the concepts chosen for our languages are promising. We need to further experiment with these languages and refine them. We believe that feedback from a variety of people is essential in order to improve them incrementally. It is however, not realistic to expect other people to apply our languages manually. Therefore, the most important next step is to prototype both languages. Out of the list of twelve planning oriented specific requirements, we are least satisfied with our solution to representing the relationship between time and space dimensions. Our current specification approach seems to be too static. It is not possible to convey to a user the fact that, e.g., in the case of our example in section 4, we have to instantiate the low-level design process for each module that has been identified during high-level design (or even for each person and module?). Other important research aspects are related to the execution-related specific requirements listed in section 2. Most of all, we have to come up with a good mechanism for instantiating process and product types.

The information base research goal of our project is to develop an (eventually) object oriented information base interface supporting the planning and execution stages of software projects. The design of this information base interface is inspired by the software engineering oriented requirements. Eventually, we would like to generate customized information bases from process and product specifications. We have developed a first approach for mapping process and product specifications into a information base schema [10]. We have implemented a first prototype information base (based on relational database technology). This prototype will be used as a vehicle for validating and improving our approach. For the future, it is planned to integrate this prototype into the prototype of the measurement and evaluation system TAME [2, 3, 4].

The major future research issues besides refining and automating our prototype specification languages are to effectively support the reuse of process specifications, their tailoring to new project needs, the different roles of a single process specification (role specific interpretation of facts), and all the execution oriented specific requirements listed in section 2.

7. Conclusions

We are aware of the huge dilemma between the need for specifying software processes & products and the unsatisfactory degree of knowledge how to do it properly. Understanding software processes better is necessary for making progress in software engineering. Being able to specify a process is the fundamental basis for sound understanding, training, execution, control and improvement as well as generating appropriate automated support.

Our two prototype specification languages reflect our current understanding of how to capture the important process and product related aspects. We believe strongly that the only way of improving our current understanding is experience from practical application. This requires us to have some initial language notation. This statement should clarify the fact that we do not view

3-18

these initial language definitions as being final. They represent a vehicle for further learning. The initial applications (one of which is described in this paper) have already helped us in understanding important process specification issues as well as in giving us a sense of the potential and limitations of our approach.

We will continue refining our languages based on experience. We are especially interested in using such process specifications as a basis for generating customized environment components, e.g. software engineering information bases [10]. We hope that this paper will inspire other groups involved in process specification research as well as result in feedback from those groups regarding our initial approach.

8. Acknowledgements

9. References

[1] V. R. Basili, "Can We Measure Software Technology: Lessons Learned from 8 Years of Trying," Proc. Tenth Annual Software Engineering Workshop, NASA Goddard Space Flight Center, Dec. 1985.

[2] V. R. Basili, H. D. Rombach, "TAME: Tailoring an Ada Measurement Environment," Proc. Joint Ada Conference, Arlington, VA, pp. 318-325, March 1987.

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

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

[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, Dec. 1975.

[6] P. A. Bernstein, "Database System Support for Software Engineering," Proc. Ninth International Conference on Software Engineering, Monterey, CA, pp. 166-178, March 1987.

[7] B. W. Boehm, "Software Engineering," IEEE Transactions on Computers, vol. C-25, no. 12, pp. 1226-1241, Dec. 1976.

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

[9] M. Dyer, "Cleanroom Software Development Method," IBM Federal Systems Division, Bethesda, Maryland, Oct. 1982.

[10] Leo Mark & H. Dieter Rombach, "Generating Customized Software Engineering Information Bases from Software Process and Product Specifications," Proc. HICSS-22, January 1989.

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

[12] Proc. Fourth International Workshop on Software Process Specification, Moretonhampstead, UK, May 1988 [to be published in ACM Software Engineering Notes].

[13] L. Osterweil, "Software Processes are Software Too," Proc. Ninth International Conference on Software Engineering. Monterey, CA, pp. 2-13, March 1987.

[14] M. H. Penedo & E. D. Stuckle, "PMDB - A Project Master Database for Software Engineering Environments," Proc. Eighth International Conference on Software Engineering, London, UK, pp. 150-157, Aug. 1985.

[15] C. V. Ramamoorthy, Y. Usuda, W. Tsai, and A. Prakash, "GENESIS: An Integrated Environment for Supporting Development and Evolution of Software," Proc. COMPSAC, 1985.

[16] W. W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques," Proc. WESCON, Aug. 1970.

[17] W. W. Royce, "Managing the Development of Large Software Systems," Proc. Ninth International Conference on Software Engineering, Monterey, CA, pp. 328-338, March 1987.

APPENDIX (A): Intermediate Level Language Definition:

```
——> START 'LANGUAGE GRAMMAR':
SR_1: <plan> ::= <process_plan> | <product_plan>
——> START 'PROCESS_PLAN GRAMMAR':
SR_2: <process_plan> ::= <process_heading> <process_body>
——> START 'PROCESS_PLAN_HEADING GRAMMAR':
SR_3: <process_heading> ::= <process_type_name> PROCESS_PLAN <comment>
——> START 'PROCESS_PLAN_BODY GRAMMAR':
SR_4: <process_body> ::= <process_specification_part> <role_specification_part>
        <resource_specification_part> <resource_assignment_part>
——> START 'PROCESS_PLAN_SPECIFICATION_PART GRAMMAR':
SR_5: <process_specification_part> ::= PROCESS_SPECIFICATION_PART:
        <comment> <process_interface_part>
        <process_refinement_part> <process_implementation_part>
——> START 'PROCESS_SPECIFICATION_INTERFACE_PART GRAMMAR':
SR_6: <process_interface_part> ::= INTERFACE_PART:
        <comment> <consume_part> <produce_part>
        <constraint_part>
SR_7: <consume_part> ::= CONSUMES: <product_type_name_list>
SR_8: <produce_part> ::= PRODUCES: <product_type_name_list>
SR_9: <constraint_part> ::= CONSTRAINTS: <constraint_type_name_list>
——> START 'PROCESS_SPECIFICATION_REFINEMENT_PART GRAMMAR':
SR_10: <process_refinement_part> ::= REFINEMENT_PART:
        <comment> <spec_use_part>
        <decomposition_part>
        <connection_part>
SR_11: <spec_use_part> ::= USE_PART: <spec_use_part_body>
SR_12: <spec_use_part_body> ::= <library_type> | <library_type>, <spec_use_part_body>
        | <null>
SR_13: <decomposition_part> ::= DECOMPOSITION_PART: <decomposition_part_body>
SR_14: <decomposition_part_body> ::= <process_decomposition_part>,
        <product_decomposition_part>,
        <constraint_decomposition_part> | <null>
SR_15: <process_decomposition_part> ::= <process_decomposition> |
        <process_decomposition>, <process_decomposition_part>
SR_16: <process_decomposition> ::= <process_type_name> DECOMPOSES_INTO
        <process_construct>
SR_17: <process_construct> ::= <SUB_LAN_1>
SR_18: <product_decomposition_part> ::= <product_decomposition> |
        <product_decomposition>, <product_decomposition_part>
SR_19: <product_decomposition> ::= <product_type_name> DECOMPOSES_INTO
        <product_construct>
SR_20: <product_construct> ::= <SUB_LAN_1>
SR_21: <constraint_decomposition_part> ::= <constraint_decomposition> |
        <constraint_decomposition>, <constraint_decomposition_part>
SR_22: <constraint_decomposition> ::= <constraint_type_name> DECOMPOSES_INTO
        <constraint_construct>
SR_23: <constraint_construct> ::= <SUB_LAN_1>
```

5207

SR_24: <connection_part> ::= CONNECTION_PART: <connection_part_body>

SR_25: <connection_part_body> ::= <product_process_connection_list>;
                      <process_process_connection_list>;
                      <process_constraint_connection_list>;
                      <product_constraint_connection_list>;
                      <process_process_dependency_list> | <null>

(****Only the first 2 nonterminals are refined here!********)

SR_26: <product_process_connection_list> ::= <product_process_connection>
                      | <product_process_connection>,
                      <product_process_connection_list>

SR_27: <process_process_connection_list> ::= <process_process_connection>
                      | <process_process_connection>,
                      <process_process_connection_list>

SR_28: <product_process_connection> ::= <product_type_name> IS_INPUT_TO
                      <process_type_name> |
                      <product_type_name> IS_OUTPUT_FROM <process_type_name> |
                      <null>

SR_29: <process_process_connection> ::= <SUB-LANGUAGE_1>

————> START 'PROCESS_SPECIFICATION_IMPLEMENTATION_PART GRAMMAR':

SR_30: <process_implementation_part> ::= IMPLEMENTATION_PART:
                      <implementation_part_body>

SR_31: <implementation_part_body> ::= <SUB_LANGUAGE_2> | <null>

————> START 'PROCESS_PLAN_ROLE_SPECIFICATION_PART GRAMMAR':

SR_32: <role_specification_part> ::= <comment> <role_part>

SR_33: <role_part> ::= <role> | <role>; <role_part> | <null>

SR_34: <role> ::= <role_header> <role_body>

SR_35: <role_header> ::= <role_type_name>: ROLE

SR_36: <role_body> ::= <role_interface_part> <role_action_part>

SR_37: <role_interface_part> ::= INTERFACE: <interface_list>

SR_38: <interface_list> ::= <library_type> | <library_type>, <interface_list>

SR_39: <library_type> ::= <process_type_name> | <product_type_name> |
                      <constraint_type_name>

SR_40: <role_action_part> ::= <process_type_list>

————> START 'PROCESS_PLAN_RESOURCE_ASSIGNMENT_PART GRAMMAR':

SR_41: <resource_assignment_part> ::= <comment> <resource_use_part>
                      <resource_assignment_part>

SR_42: <resource_use_part> ::= USE_PART: <resource_use_part_body>

SR_43: <resource_use_part_body> ::= <resource_type_name_list> : RESOURCE | <null>

SR_44: <resource_assignment_part> ::= ASSIGNMENT_PART:
                      <resource_assignment_part_body>
SR_45: <resource_assignment_part_body> ::= <resource_assignment> |
                      <resource_assignment>
                      <resource_assignment_part> | <null>

SR_46: <resource_assignment> ::= <resource_type_name> IS_ASSIGNED_TO
                      <role_type_name> |
                      <resource_type_name> IS_ASSIGNED_TO <process_type_name>

(***)
(***)

SR_47: <comment> ::= COMMENT: <text>
SR_48: <process_type_name_list> ::= <process_type_name> | <process_type_name>;
                      <process_type_name_list> | <null>
SR_49: <product_type_name_list> ::= <product_type_name> | <product_type_name>;
                      <product_type_name_list> | <null>
SR_50: <constraint_type_name_list> ::= <constraint_type_name> | <constraint_type_name>;
                      <constraint_type_name_list> | <null>
SR_51: <resource_type_name_list> ::= <resource_type_name> | <resource_type_name>;
                      <resource_type_name_list> | <null>
SR_52: <role_type_name_list> ::= <role_type_name> | <role_type_name>;
                      <role_type_name_list> | <null>
SR_53: <process_type_name> ::= <text>
SR_54: <product_type_name> ::= <text>
SR_55: <constraint_type_name> ::= <text>
SR_56: <resource_type_name> ::= <text>
SR_57: <role_type_name> ::= <text>

SR_58: <SUB_LANGUAGE_1> ::= Regular expression language for process and product type
                      names (using SEQUENCE, ALTERNATION, ITERATION, and PARALLELISM)

SR_59: <SUB_LANGUAGE_2> ::= Any kind of formal or informal language representation
                      allowing the description of a process implementation.
SR_60: <null> ::=

APPENDIX (B): Process and Product Specification Types
(Example of section 4.1.):

design;	count_errors;	c_1;
hl_design;	count_hl_effort;	c_2;
yourdon;	count_ll_effort;	c_3;
ll_design;	compute_v;	c_4;
r;	hlerr;	p_1;
d;	hleff;	p_2;
hld;	lleff;	p_3;
lld;	v;	p_4;
pdl;		p_5;

APPENDIX (C): Process Specification of process 'design'
(Example of section 4.1.):

design (name): PROCESS_PLAN
     COMMENT: ***********

PROCESS_SPECIFICATION_PART:
     COMMENT: ************;

  INTERFACE_PART:
  COMMENT: ***********

     CONSUMES: r;
     PRODUCES: d;
     CONSTRAINTS: c_1, c_3;

  REFINEMENT_PART:
     COMMENT: **********

     USE_PART:
        count_errors, count_hl_effort, count_ll_effort, compute_v: PROCESS_PLAN;
        hlerr, hleff, lleff, v: PRODUCT_PLAN;
        c_3, c_4(name): CONSTRAINT
     DECOMPOSITION_PART:
        design DECOMPOSES_INTO hl_design, ll_design*;
        d DECOMPOSES_INTO hld, lld;
     CONNECTION_PART:
        hl_design USES r,
        hl_design PRODUCES hld,
        ll_design PRODUCES lld,
        count_errors PRODUCES hlerr,
        count_hl_effort PRODUCES hleff,
        count_ll_effort PRODUCES lleff,
        compute_v USES lld,
        compute_v PRODUCES v,
        hl_design SEQ (PARALL (ll_design*)),
        count_errors DEPENDS_ON hl_design,
        count_hl_effort DEPENDS_ON hl_design,
        count_ll_effort DEPENDS_ON ll_design,
        ITER ((ll_design,c_4)),
        c_1 IS_PRE_CONDITION_FOR hl_design,
        c_3 IS_POST_CONDITION_FOR ll_design,
        c_2 IS_POST_CONDITION_FOR hl_design,
        c_4 IS_PRE_CONDITION_FOR ll_design;

  IMPLEMENTATION_PART:
  COMMENT: ***********

ROLE_SPECIFICATION_PART:
     COMMENT: ***********

design_role: ROLE
        INTERFACE: r, hld, lld, c_1, c_2, c_3, c_4
        ACTIONS: hl_design, ll_design

quality_assurance_role: ROLE
        INTERFACE: c_1, c_2, c_3, c_4, hlerr, hleff, lleff, v
              hl_design, ll_design, lld
        ACTIONS: count_errors, count_hl_effort, count_ll_effort,
              compute_v

management_role: ROLE
        INTERFACE:
        ACTIONS:

RESOURCE_ASSIGNMENT_PART:

  COMMENT: ***********

  USE_PART:
        p_1, p_2, p_3, p_4, p_5: RESOURCE

  CONNECTION_PART:
        p_1 IS_ASSIGNED_TO hl_design,
        p_1 IS_ASSIGNED_TO ll_design,
        p_2 IS_ASSIGNED_TO ll_design,
        p_3 IS_ASSIGNED_TO ll_design,
        p4 IS_ASSIGNED_TO quality_assurance_role,
        p5 IS_ASSIGNED_TO management_role,

The TAME Project: Towards Improvement-Oriented Software Environments

VICTOR R. BASILI, SENIOR MEMBER, IEEE, AND H. DIETER ROMBACH

Abstract—Experience from a dozen years of analyzing software engineering processes and products is summarized as a set of software engineering and measurement principles that argue for software engineering process models that integrate sound planning and analysis into the construction process.

In the TAME (Tailoring A Measurement Environment) project at the University of Maryland we have developed such an improvement-oriented software engineering process model that uses the goal/question/metric paradigm to integrate the constructive and analytic aspects of software development. The model provides a mechanism for formalizing the characterization and planning tasks, controlling and improving projects based on quantitative analysis, learning in a deeper and more systematic way about the software process and product, and feeding the appropriate experience back into the current and future projects.

The TAME system is an instantiation of the TAME software engineering process model as an ISEE (Integrated Software Engineering Environment). The first in a series of TAME system prototypes has been developed. An assessment of experience with this first limited prototype is presented including a reassessment of its initial architecture. The long-term goal of this building effort is to develop a better understanding of appropriate ISEE architectures that optimally support the improvement-oriented TAME software engineering process model.

Index Terms—Characterization, execution, experience, feedback, formalizing, goal/question/metric paradigm, improvement paradigm, integrated software engineering environments, integration of construction and analysis, learning, measurement, planning, quantitative analysis, software engineering process models, tailoring, TAME project, TAME system.

I. INTRODUCTION

EXPERIENCE from a dozen years of analyzing software engineering processes and products is summarized as a set of ten *software engineering* and fourteen *measurement principles*. These principles imply the need for software engineering process models that integrate sound planning and analysis into the construction process.

Software processes based upon such *improvement-oriented software engineering process models* need to be *tailorable* and *tractable*. The tailorability of a process is the characteristic that allows it to be altered or adapted to suit

a set of special needs or purposes [64]. The software engineering process requires tailorability because the overall project execution model (life cycle model), methods and tools need to be altered or adapted for the specific project environment and the overall organization. The tractability of a process is the characteristic that allows it to be easily planned, taught, managed, executed, or controlled [64]. Each software engineering process requires tractability because it needs to be planned, the various planned activities of the process need to be communicated to the entire project personnel, and the process needs to be managed, executed, and controlled according to these plans. Sound tailoring and tracking require *top-down measurement* (measurement based upon operationally defined goals). The goal of a *software engineering environment (SEE)* should be to support such tailorable and tractable software engineering process models by automating as much of them as possible.

In the *TAME (Tailoring A Measurement Environment) project* at the University of Maryland we have developed an improvement-oriented software engineering process model. The *TAME system* is an instantiation of this TAME software engineering process model as an ISEE (Integrated SEE).

It seems appropriate at this point to clarify some of the important terms that will be used in this paper. The term *engineering* comprises both development and maintenance. A software engineering *project* is embedded in some *project environment* (characterized by personnel, type of application, etc.) and within some *organization* (e.g., NASA, IBM). Software engineering within such a project environment or organization is conducted according to an overall software engineering *process model* (one of which will be introduced in Section II-B-3). Each individual software project in the context of such a software engineering process model is exeucted according to some *execution model* (e.g., waterfall model [28], [58], iterative enhancement model [24], spiral model [30]) supplemented by *techniques (methods, tools)*. Each specific instance of (a part of) an execution model together with its supplementing methods and tools is referred to as *execution process* (including the construction as well as the analysis process). In addition, the term *process* is frequently used as a generic term for various kinds of activities. We distinguish between *constructive* and *analytic* methods and tools. Whereas constructive methods and tools are concerned with building products, analytic

5207

method and tools are concerned with analyzing the constructive process and the resulting products. The body of experience accumulated within a project environment or organization is referred to as *experience base*. There exist at least three levels of formalism of such experience bases: *database* (data being individual products or processes), *information base* (information being data viewed through some superimposed structure), and knowledge base (knowledge implying the ability to derive new insights via deduction rules). The project personnel are categorized as either *engineers* (e.g., designers, coders, testers) or *managers*.

This paper is structured into a presentation and discussion of the improvement-oriented software engineering process model underlying the TAME project (Section II), its automated support by the TAME system (Section III), and the first TAME system prototype (Section IV). In the first part of this paper we list the empirically derived lessons learned (Section II-A) in the form of software engineering principles (Section II-A-1), measurement principles (Section II-A-2), and motivate the TAME project by stating several implications derived from those principles (Section II-A-3). The TAME project (Section II-B) is presented in terms of the improvement paradigm (Section II-B-1), the goal/question/metric paradigm as a mechanism for formalizing the improvement paradigm (Section II-B-2), and the TAME project model as an instantiation of both paradigms (Section II-B-3). In the second part of this paper we introduce the TAME system as an approach to automatically supporting the TAME software engineering process model (Section III). The TAME system is presented in terms of its requirements (Section III-A) and architecture (Section III-B). In the third part of this paper, we introduce the first TAME prototype (Section IV) with respect to its functionality and our first experiences with it.

II. SOFTWARE ENGINEERING PROCESS

Our experience from measuring and evaluating software engineering processes and products in a variety of project environments has been summarized in the form of lessons learned (Section II-A). Based upon this experience the TAME project has produced an improvement-oriented process model (Section II-B).

A. Lessons Learned from Past Experience

We have formulated our experience as a set of software engineering principles (Section II-A-1) and measurement principles (Section II-A-2). Based upon these principles a number of implications for sound software engineering process models have been derived (Section II-A-3).

1) Software Engineering Principles: The first five software engineering principles address the need for developing quality *a priori* by introducing engineering discipline into the field of software engineering:

(P1) We need to clearly distinguish between the role of constructive and analytic activities. Only improved construction processes will result in higher quality software. Quality cannot be tested or inspected into software. An-

alytic processes (e.g., quality assurance) cannot serve as a substitute for constructive processes but will provide control of the constructive processes [27], [37], [61].

(P2) We need to formalize the planning of the construction process in order to develop quality *a priori* [3], [16], [19], [25]. Without such plans the trial and error approach can hardly be avoided.

(P3) We need to formalize the analysis and improvement of construction processes and products in order to guarantee an organized approach to software engineering [3], [25].

(P4) Engineering methods require analysis to determine whether they are being performed appropriately, if at all. This is especially important because most of these methods are heuristic rather than formal [42], [49], [66].

(P5) Software engineers and managers need real-time feedback in order to improve the construction processes and products of the ongoing project. The organization needs post-mortem feedback in order to improve the construction processes and products for future projects [66].

The remaining five software engineering principles address the need for tailoring of planning and analysis processes due to changing needs form project to project and environment to environment:

(P6) All project environments and products are different in some way [2], [66]. These differences must be made explicit and taken into account in the software execution processes and in the product quality goals [3], [16], [19], [25].

(P7) There are many execution models for software engineering. Each execution model needs to be tailored to the organization and project needs and characteristics [2], [13], [16], [66].

(P8) We need to formalize the tailoring of processes toward the quality and productivity goals of the project and the characteristics of the project environment and the organization [16]. It is not easy to apply abstractly defined methods to specific environments.

(P9) This need for tailoring does not mean starting from scratch each time. We need to reuse experience, but only after tailoring it to the project [1], [2], [6], [7], [18], [32].

(P10) Because of the constant need for tailoring, management control is crucial and must be flexible. Management needs must be supported in this software engineering process.

A more detailed discussion of these software engineering principles is contained in [17].

2) Software Measurement Principles: The first four measurement principles address the purpose of the measurement process, i.e., why should we measure, what should we measure, for whom should we measure:

(M1) Measurement is an ideal mechanism for characterizing, evaluating, predicting, and providing motivation for the various aspects of software construction processes and products [3], [4], [9], [16], [21], [25], [48], [56], [57]. It is a common mechanism for relating these multiple aspects.

(M2) Measurements must be taken on both the soft-

ware processes and the various software products [1], [5], [14], [29], [38], [40], [42]-[44], [47], [54]-[56], [65], [66]. Improving a product requires understanding both the product and its construction processes.

(M3) There are a variety of uses for measurement. The purpose of measurement should be clearly stated. We can use measurement to examine cos. effectiveness, reliability, correctness, maintainability, efficiency, user friendliness, etc. [8]-[10], [13], [14], [16], [20], [23], [25], [41], [53], [57], [61].

(M4) Measurement needs to be viewed from the appropriate perspective. The corporation, the manager, the developer, the customer's organization and the user each view the product and the process from different perspectives. Thus they may want to know different things about the project and to different levels of detail [3], [16], [19], [25], [66].

The remaining ten measurement principles address metrics and the overall measurement process. The first two principles address characteristics of metrics (i.e., what kinds of metrics, how many are needed), while the latter eight address characteristics of the measurement process (i.e., what should the measurement process look like, how do we support characterization, planning, construction, and learning and feedback):

(M5) Subjective as well as objective metrics are required. Many process, product and environment aspects can be characterized by objective metrics (e.g., product complexity, number of defects or effort related to processes). Other aspects cannot be characterized objectively yet (e.g., experience of personnel, type of application, understandability of processes and products); but they can at least be categorized on a quantitative (nominal) scale to a reasonable degree of accuracy [4], [5], [16], [48], [56].

(M6) Most aspects of software processes and products are too complicated to be captured by a single metric. For both definition and interpretation purposes, a set of metrics (a metric vector) that frame the purpose for measurement needs to be defined [9].

(M7) The development and maintenance environments must be prepared for measurement and analysis. Planning is required and needs to be carefully integrated into the overall software engineering process model. This planning process must take into account the experimental design appropriate for the situation [3], [14], [19], [22], [66].

(M8) We cannot just use models and metrics from other environments as defined. Because of the differences among execution models (principle P7), the models and metrics must be tailored for the environment in which they will be applied and checked for validity in that environment [2], [6]-[8], [12], [23], [31], [40], [47], [50], [51], [62].

(M9) The measurement process must be top-down rather than bottom-up in order to define a set of operational goals, specify the appropriate metrics, permit valid

contextual interpretation and analysis, and provide feedback for tailorability and tractability [3], [16], [19], [25].

(M10) For each environment there exists a characteristic set of metrics that provides the needed information for definition and interpretation purposes [21].

(M11) Multiple mechanisms are needed for data collection and validation. The nature of the data to be collected (principle M5) determines the appropriate mechanisms [4], [25], [48], e.g., manually via forms or interviews, or automatically via analyzers.

(M12) In order to evaluate and compare projects and to develop models we need a historical experience base. This experience base should characterize the local environment [4], [13], [25], [34], [44], [48].

(M13) Metrics must be associated with interpretations, but these interpretations must be given in context [3], [16], [19], [25], [34], [56].

(M14) The experience base should evolve from a database into a knowledge base (supported by an expert system) to formalize the reuse of experience [11], [14].

A more detailed discussion of these measurement principles is contained in [17].

3) Implications: Clearly this set of principles is not complete. However, these principles provide empirically derived insight into the limitations of traditional process models. We will give some of the implications of these principles with respect to the components that need to be included in software process models, essential characteristics of these components, the interaction of these components, and the needed automated support. Although there is a relationship between almost all principles and the derived implications, we have referenced for each implication only those principles that are related most directly.

Based upon our set of principles it is clear that we need to *better understand* the software construction process and product (e.g., principles P1, P4, P6, M2, M5, M6, M8, M9, M10, M12). Such an understanding will allow us to *plan* what we need to do and improve over our current practices (e.g., principles P1, P2, P3, P7, P8, M3, M4, M7, M9, M14). To *make those plans operational*, we need to specify how we are going to affect the construction processes and their analysis (e.g., principles P1, P2, P3, P4, P7, P8, M7, M8, M9, M14). The *execution* of these prescribed plans involves the *construction* of products and the *analysis* of the constructive processes and resulting products (e.g., principles P1, P7).

All these implications need to be integrated in such a way that they allow for sound *learning and feedback* so that we can improve the software execution processes and products (e.g., principles P1, P3, P4, P5, P9, P10, M3, M4, M9, M12, M13, M14). This interaction requires the integration of the constructive and analytic aspects of the software engineering process model (e.g., principles P2, M7, M9).

The components and their interactions need to be formalized so they can be supported properly by an *ISEE*

(e.g., principles P2, P3, P8, P9, M9). This formalization must include a *structuring of the body of experience* so that characterization, planning, learning, feedback, and improvement can take place (e.g., principles P2, P3, P8, P9, M9). An ideal mechanism for supporting all of these components and their interactions is *quantitative analysis* (e.g., principles P3, P4, M1, M2, M5, M6, M8, M9, M10, M11, M13).

B. A Process Model: The TAME Project

The TAME (Tailoring A Measurement Environment) project at the University of Maryland has produced a software engineering process model (Section II-B-3) based upon our empirically derived lessons learned. This software engineering process model is based upon the improvement (Section II-B-1) and goal/question/metric paradigms (Section II-B-2).

1) Improvement Paradigm: The improvement paradigm for software engineering processes reflects the implications stated in Section II-A-3. It consists of six major steps [3]:

(I1) Characterize the current project environment.

(I2) Set up goals and refine them into quantifiable questions and metrics for successful project performance and improvement over previous project performances.

(I3) Choose the appropriate software project execution model for this project and supporting methods and tools.

(I4) Execute the chosen processes and construct the products, collect the prescribed data, validate it, and provide feedback in real-time.

(I5) Analyze the data to evaluate the current practices, determine problems, record the findings, and make recommendations for improvement.

(I6) Proceed to Step I1 to start the next project, armed with the experience gained from this and previous projects.

This paradigm is aimed at providing a basis for corporate learning and improvement. Improvement is only possible if we a) understand what the current status of our environment is (step I1), b) state precise improvement goals for the particular project and quantify them for the purpose of control (step I2), c) choose the appropriate process execution models, methods, and tools in order to achieve these improvement goals (step I3), execute and monitor the project performance thoroughly (step I4), and assess it (step I5). Based upon the assessment results we can provide feedback into the ongoing project or into the planning step of future projects (steps I5 and I6).

2) Goal/Question/Metric Paradigm: The goal/question/metric (GQM) paradigm is intended as a mechanism for formalizing the characterization, planning, construction, analysis, learning and feedback tasks. It represents a systematic approach for setting project goals (tailored to the specific needs of an organization) and defining them in an operational and tractable way. Goals are refined into a set of quantifiable questions that specify metrics. This paradigm also supports the analysis and integration of

metrics in the context of the questions and the original goal. Feedback and learning are then performed in the context of the GQM paradigm.

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. These templates and guidelines reflect our experience from having applied the GQM paradigm in a variety of environments (e.g., NASA [4], [17], [48], IBM [60], AT&T, Burroughs [56], and Motorola). We received additional feedback from Hewlett Packard where the GQM paradigm has been used without our direct assistance [39]. It needs to be stressed that we do not claim that these templates and guidelines are complete; they will most likely change over time as our experience grows. 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.

• **Templates/Guidelines for Goal Definition:**

Purpose: To (characterize, evaluate, predict, motivate, etc.) the (process, product, model, metric, etc.) in order to (understand, assess, manage, engineer, learn, improve, etc.) it.

Example: To evaluate the system testing methodology in order to improve it.

Perspective: Examine the (cost, effectiveness, correctness, defects, changes, product metrics, reliability, etc.) from the point of view of the (developer, manager, customer, corporate perspective, etc.)

Example: Examine the effectiveness from the developer's point of view.

Environment: The environment consists of the following: process factors, people factors, problem factors, methods, tools, constraints, etc.

Example: The product is an operating system that must fit on a PC, etc.

• **Guidelines for Product-Related Questions:**

For each product under study there are three major subgoals that need to be addressed: 1) definition of the product, 2) definition of the quality perspectives of interest, and 3) feedback related to the quality perspectives of interest.

Definition of the product includes questions related to *physical attributes* (a quantitative characterization of the product in terms of physical attributes such as size, com-

plexity, etc.), *cost* (a quantitative characterization of the resources expended related to this product in terms of effort. computer time, etc.), *changes and defects* (a quantitative characterization of the errors, faults, failures, adaptations, and enhancements related to this product), and *context* (a quantitative characterization of the customer community using this product and their operational profiles).

Quality perspectives of interest includes, for each quality perspective of interest (e.g., reliability, user friendliness), questions related to the *major model(s) used* (a quantitative specification of the quality perspective of interest), the *validity of the model for the particular environment* (an analysis of the appropriateness of the model for the particular project environment), the *validity of the data collected* (an analysis of the quality of data), the *model effectiveness* (a quantitative characterization of the quality of the results produced according to this model), and a *substantiation of the model* (a discussion of whether the results are reasonable from various perspectives).

Feedback includes questions related to *improving the product relative to the quality perspective of interest* (a quantitative characterization of the product quality, major problems regarding the quality perspective of interest, and suggestions for improvement during the ongoing project as well as during future projects).

• **Guidelines for Process-Related Questions**

For each process under study, there are three major subgoals that need to be addressed: 1) definition of the process, 2) definition of the quality perspectives of interest, and 3) feedback from using this process relative to the quality perspective of interest.

Definition of the process includes questions related to the *quality of use* (a quantitative characterization of the process and an assessment of how well it is performed), and the *domain of use* (a quantitative characterization of the object to which the process is applied and an analysis of the process performer's knowledge concering this object).

Quality perspectives of interest follows a pattern similar to the corresponding product-oriented subgoal including, for each quality perspective of interest (e.g., reduction of defects, cost effectiveness), questions related to the *major model(s) used*, and *validity of the model for the particular environment*, the *validity of the data collected*, the *model effectiveness* and the *substantiation of the model*).

Feedback follows a pattern similar to the corresponding product-oriented subgoal.

• **Guidelines for Metrics, Data Collection, and Interpretation:**

• The choice of metrics is determined by the quantifiable questions. The guidelines for questions acknowledge the need for generally more than one metric (principle M6), for objective and subjective metrics (principle M5), and for associating interpretations with metrics (principle M13). The actual GQM models generated from these templates and guidelines will differ from project to project and organization to organization (principle M6). This reflects their being tailored for the different needs in different projects and organizations (principle M4). Depending on the type of each metric, we choose the appropriate mechansims for data collection and validation (principle M11). As goals, questions and metrics provide for tractability of the (top-down) definitional quantification process, they also provide for the interpretation context (bottom-up). This integration of definition with interpretation allows for the interpretation process to be tailored to the specific needs of an environment (principle M8).

3) Improvement-Oriented Process Model: The TAME software engineering process model is an instantiation of the improvement paradigm. The GQM paradigm provides the necessary integration of the individual components of this model. The TAME software engineering process model explicitly includes components for (C1) the characterization of the current status of a project environment, (C2) the planning for improvement integrated into the execution of projects, (C3) the execution of the construction and analysis of projects according to the project plans, and (C4) the recording of experience into an experience base. The learning and feedback mechanism (C5) is distributed throughout the model within and across the components as information flows from one component to another. Each of these tasks must be dealt with from a constructive and analytic perspective. Fig. 1 contains a graphical representation of the improvement-oriented TAME process model. The relationships (arcs) among process model components in Fig. 1 represent information flow.

(C1) Characterization of the current environment is required to understand the various factors that influence the current project environment. This task is important in order to define a starting point for improvement. Without knowing where we are, we will not be able to judge whether we are improving in our present project. We distinguish between the constructive and analytic aspects of the characterization task to emphasize that we not only state the environmental factors but analyze them to the degree possible based upon data and other forms of information from prior projects. This characterization task needs to be formalized.

(C2) Planning is required to understand the project goals, execution needs, and project focus for learning and feedback. This task is essential for disciplined software project execution (i.e., executing projects according to precise specifications of processes and products). It provides the basis for improvement relative to the current status determined during characterization. In the planning task, we distinguish between the constructive and analytic as well as the "what" and "how" aspects of planning. Based upon the GQM paradigm all these aspects are highly interdependent and performed as a single task. The development of quantitatively analyzable goals is an iterative process. However, we formulate the four planning as-
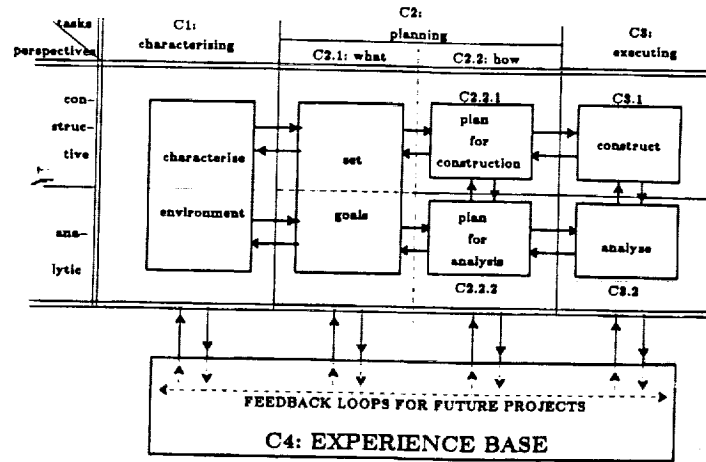
Fig. 1. The improvement-oriented TAME software process model.

pects as four separate components to emphasize the differences between creating plans for development and making those plans analyzable, as well as between stating what it is you want to accomplish and stating how you plan to tailor the processes and metrics to do it.

(C2.1) "What" Planning deals with choosing, assigning priorities, and operationally defining, to the degree possible, the project goals from the constructive and analytic perspectives. The actual goal setting is an instantiation of the front-end of the GQM paradigm (the templates/guidelines for goal definition). The constructive perspective addresses the definition of project goals such as on-time delivery, the appropriate functionality to satisfy the user, and the analysis of the execution processes we are applying. Some of these goals might be stated as improvement goals over the current state-of-the-practice as characterized in component C1. These goals should be prioritized and operationally defined to the extent possible without having chosen the particular construction models, methods and tools yet. The analytic perspective addresses analysis procedures for monitoring and controlling whether the goals are met. This analytic goal perspective should prescribe the necessary learning and feedback paths. It should be operationally defined to the extent allowed by the degree of precision of the constructive goal perspective.

(C2.2) "How" Planning is based upon the results from the "what" planning (providing for the purpose and perspective of a goal definition according to the GQM paradigm front-end) and the characterization of the environment (providing for the environment part of a goal definition according to the GQM paradigm front-end). The "how" planning involves the choice of an appropriately tailored execution model, methods and tools that permit the building of the system in such a way that we can analyze whether we are achieving our stated goals. The particular choice of construction processes, methods and tools

(component C2.2.1) goes hand in hand with fine-tuning the analysis procedures derived during the analytic perspective of the "what" planning (component C2.2.2).

(C2.2.1) Planning for construction includes choosing the appropriate execution model, methods and tools to fulfill the project goals. It should be clear that effective planning for construction depends on well-defined project goals from both the constructive and analytic perspective (component C2.1).

(C2.2.2) Planning for analysis addresses the fine-tuning of the operational definition of the analytic goal perspective (derived as part of component C2.1) towards the specific choices made during planning for construction (C2.2.1). The actual planning for analysis is an instantiation of the back-end of the GQM paradigm; details need to be filled in (e.g., quantifiable questions, metrics) based upon the specific methods and tools chosen.

(C3) Execution must integrate the construction (component C3.1) with the analysis (component C3.2). Analysis (including measurement) cannot be an add-on but must be part of the execution process and drive the construction. The execution plans derived during the planning task are supposed to provide for the required integration of construction and analysis.

(C4) The Experience Base includes the entire body of experience that is actively available to the project. We can characterize this experience according to the following dimensions: a) the degree of precision/detail, and b) the degree to which it is tailored to meet the specific needs of the project (context). The precision/detail dimension involves the level of detail of the experimental design and the level and quality of data collected. On one end of the spectrum we have detailed objective quantitative data that allows us to build mathematically tractable models. On the other end of the spectrum we have interviews and qualitative information that provide guidelines and "lessons learned documents", and permit the better formu-

lation of goals and questions. The level of precision and detail affects our level of confidence in the results of the experiment as well as the cost of the data collection process. Clearly priorities play an important role here. The context dimension involves whether the focus is to learn about the specific project, projects within a specific application domain or general truths about the software process or product (requires the incorporation of formalized experience from prior projects into the experience base). Movement across the context dimension assumes an 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 environments.

(C5) Learning and Feedback are integrated into the TAME process model in various ways. They are based upon the experimental model for learning consisting of a set of steps, starting with focused objectives, which are turned into specific hypotheses, followed by running experiments to validate the hypotheses in the appropriate environment. The model is iterative; as we learn from experimentation, we are better able to state our focused objectives and we change and refine our hypotheses.

This model of learning is incorporated into the GQM paradigm where the focused objectives are expressed as goals, the hypotheses are expressed as questions written to the degree of formalism required, and the experimental environment is the project, a set of projects in the same domain, or a corporation representing a general environment. Clearly the GQM paradigm is also iterative.

The feedback process helps generate the goals to influence one or more of the components in the process model, e.g., the characterization of the environment, or the analysis of the construction processes or products. The level of confidence we have in feeding back the experience to a project or a corporate environment depends upon the precision/detail level of the experience base (component C4) and the generality of the experimental environment in which it was gathered.

The learning and feedback process appears in the model as the integration of all the components and their interactions as they are driven by the improvement and GQM paradigms. The feedback process can be channeled to the various components of the current project and to the corporate experience base for use in future projects.

Most traditional software engineering process models address only a subset of the individual components of this model; in many cases they cover just the constructive aspects of characterization (component C1), "how" planning (component C2.2.1), and execution (component C3.1). More recently developed software engineering process models address the constructive aspect of execution (component C3.1) in more sophisticated ways (e.g., new process models [24],[30], [49], combine various process dimensions such as technical, managerial, contrac-

tual [36], or provide more flexibility as far as the use of methods and tools is concerned, for example via the automated generation of tools [45], [63]), or they add methods and tools for choosing the analytical processes, methods, and tools (component C3.2.2) as well as actually performing analysis (component C3.2) [52], [59]. However, all these process models have in common the lack of completely integrating all their individual components in a systematic way that would permit sound learning and feedback for the purpose of project control and improvement of corporate experience.

III. Automated Support through ISEEs: The TAME System

The goal of an Integrated Software Engineering Environment (ISEE) is to effectively support the improvement-oriented software engineering process model described in Section II-B-3. An ISEE must support all the model components (characterization, planning, execution, and the experience base), all the local interactions between model components, the integration, and formalization of the GQM paradigm, and the necessary transitions between the context and precision/detail dimension boundaries in the experience base. Supporting the transitions along the experience base dimensions is needed in order to allow for sound learning and feedback as outlined in Section II-B-3 (component C5).

The TAME system will automate as many of the components, interactions between components and supporting mechanisms of the TAME process model as possible. The TAME system development activities will concentrate on all but the construction component (component C3.1) with the eventual goal of interfacing with constructive SEEs. In this section we present the requirements and the initial architecture for the TAME system.

A. Requirements

The requirements for the TAME system can be derived from Section II-B-3 in a natural way. These requirements can be divided into external requirements (defined by and of obvious interest to the TAME system user) and internal requirements (defined by the TAME design team and required to support the external requirements properly).

The first five (external) requirements include support for the characterization and planning components of the TAME model by automating an instantiation of the GQM paradigm, for the analysis component by automating data collection, data validation and analysis, and the learning and feedback component by automating interpretation and organizational learning. We will list for each external TAME system requirement the TAME process model components of Section II-B-3 from which it has been derived.

External TAME requirements:

(R1) A mechanism for defining the constructive and analytic aspects of project goals in an operational and quantifiable way (derived from components C1, C2.1, C2.2.2, C3.2).

We use the GQM paradigm and its templates for defin-

ing goals operationally and refining them into quantifiable questions and metrics. The selection of the appropriate GQM model and its tailoring needs to be supported. The user will either select an existing model or generate a new one. A new model can be generated from scratch or by reusing pieces of existing models. The degree to which the selection, generation, and reuse tasks can be supported automatically depends largely on the degree to which the GQM paradigm and its templates can be formalized. The user needs to be supported in defining his/ her specific goals according to the goal definition template. Based on each goal definition, the TAME system will search for a model in the experience base. If no appropriate model exists, the user will be guided in developing one. Based on the tractability of goals into subgoals and questions the TAME system will identify reusable pieces of existing models and compose as much of an initial model as possible. This initial model will be completed with user interaction. For example, if a user wants to develop a model for assessing a system test method used in a particular environment, the system might compose an initial model by reusing pieces from a model assessing a different test method in the same environment, and from a model for assessing the same system test method in a different environment. A complete GQM model includes rules for interpretation of metrics and guidelines for collecting the prescribed data. The TAME system will automatically generate as much of this information as possible.

(R2) The automatic and manual collection of data and the validation of manually collected data (derived from component C3.2).

The collection of all product-related data (e.g., lines of code, complexity) and certain process-related data (e.g., number of compiler runs, number of test runs) will be completely automated. Automation requires an interface with construction-oriented SEEs. The collection of many process-related data (e.g., effort, changes) and subjective data (e.g., experience of personnel, characteristics of methods used) cannot be automated. The schedule according to which measurement tools are run needs to be defined as part of the planning activity. It is possible to collect data whenever they are needed, periodically (e.g., always at a particular time of the day), or whenever changes of products occur (e.g., whenever a new product version is entered into the experience base all the related metrics are recomputed). All manually collected data need to be validated. Validating whether data are within their defined range, whether all the prescribed data are collected, and whether certain integrity rules among data are maintained will be automated. Some of the measurement tools will be developed as part of the TAME system development project, others will be imported. The need for importing measurement tools will require an effective interconnection mechanism (probably an interconnection language) for integrating tools developed in different languages.

(R3) A mechanism for controlling measurement and analysis (derived from component C3.2).

A GQM model is used to specify and control the execution of a particular analysis and feedback session. According to each GQM model, the TAME system must trigger the execution of measurement tools for data collection, the computation of all metrics and distributions prescribed, and the application of statistical procedures. If certain metrics or distributions cannot be computed due to the lack of data or measurement tools, the TAME system must inform the user.

(R4) A mechanism for interpreting analysis results in a context and providing feedback for the improvement of the execution model, methods and tools (derived from components C3.2, C.5).

We use a GQM model to define the rules and context for interpretation of data and for feedback in order to refine and improve execution models, methods and tools. The degree to which interpretation can be supported depends on our understanding of the software process and product, and the degree to which we express this understanding as formal rules. Today, interpretation rules exist only for some of the aspects of interest and are only valid within a particular project environment or organization. However, interpretation guided by GQM models will enable an evolutionary learning process resulting in better rules for interpretation in the future. The interpretation process can be much more effective provided historical experience is available allowing for the generation of historical baselines. In this case we can at least identify whether observations made during the current project deviate from past experience or not.

(R5) A mechanism for learning in an organization (derived from components C4, C5).

The learning process is supported by iterating the sequence of defining focused goals, refining them into hypotheses, and running experiments. These experiments can range from completely controlled experiments to regular project executions. In each case we apply measurement and analysis procedures to project classes of interest. For each of those classes, a historical experience base needs to be established concerning the effectiveness of the candidate execution models, methods and tools. Feedback from ongoing projects of the same class, the corresponding execution models, methods and tools can be refined and improved with respect to context and precision/ detail so that we increase our potential to improve future projects.

The remaining seven (internal) requirements deal with user interface management, report generation, experience base, security and access control, configuration management control, SEE interface and distribution issues. All these issues are important in order to support planning, construction, learning and feedback effectively.

Internal TAME requirements:

(R6) A homogeneous user interface.

We distinguish between the physical and logical user interface. The physical user interface provides a menu or command driven interface between the user and the TAME system. Graphics and window mechansims will be

incorporated whenever useful and possible. The logical user interface reflects the user's view of measurement and analysis. Users will not be allowed to directly access data or run measurement tools. The only way of working with the TAME system is via a GQM model. TAME will enforce this top-down approach to measurement via its logical user interface. The acceptance of this kind of user interface will depend on the effectiveness and ease with which it can be used. Homogeneity is important for both the physical and logical user interface.

(R7) An effective mechanism for presenting data, information, and knowledge.

The presentation of analysis (measurement and interpretation) results via terminal or printer/plotter needs to be supported. Reports need to be generated for different purposes. Project managers will be interested in periodical reports reflecting the current status of their project. High level managers will be interested in reports indicating quality and productivity trends of the organization. The specific interest of each person needs to be defined by one or more GQM models upon which automatic report generation can be based. A laser printer and multicolor plotter would allow the appropriate documentation of tables, histograms, and other kinds of textual and graphical representations.

(R8) The effective storage and retrieval of all relevant data, information, and knowledge in an experience base.

All data, information, and knowledge required to support tailorability and tractability need to be stored in an experience base. Such an experience base needs to store GQM models, engineering products and measurement data. It needs to store data derived from the current project as well as historical data from prior projects. The effectiveness of such an experience base will be improved for the purpose of learning and feedback if, in addition to measurement data, interpretations from various analysis sessions are stored. In the future, the interpretation rules themselves will become integral part of such an experience base. The experience base should be implemented as an abstract data type, accessible through a set of functions and hiding the actual implementation. This latter requirement is especially important due to the fact that current database technology is not suited to properly support software engineering concepts [26]. The implementation of the experience base as an abstract data type allows us to use currently available database technology and substitute more appropriate technology later as it becomes available. The ideal database would be self-adapting to the changing needs of a project environment or an organization. This would require a specification language for software processes and products, and the ability to generate database schemata from specifications written in such a language [46].

(R9) Mechanisms allowing for the implementation of a variety of access control and security strategies.

TAME must control the access of users to the TAME system itself, to various system functions and to the experience base. These are typical functions of a security system. The enforced security strategies depend on the project organization. It is part of planning a project to decide who needs to have access to what functions and pieces of data, information, and knowledge. In addition to these security functions, more sophisticated data access control functions need to be performed. The data access system is expected to "recommend" to a user who is developing a GQM model the kinds of data that might be helpful in answering a particular question and support the process of choosing among similar data based on availability or other criteria.

(R10) Mechanisms allowing for the implementation of a variety of configuration management and control strategies.

In the context of the TAME system we need to manage and control three-dimensional configurations. There is first the traditional product dimension making sure that the various product and document versions are consistent. In addition, each product version needs to be consistent with its related measurement data and the GQM model that guided those measurements. TAME must ensure that a user always knows whether data in the experience base is consistent with the current product version and was collected and interpretated according to a particular model. The actual configuration management and control strategies will result from the project planning activity.

(R11) An interface to a construction-oriented SEE.

An interface between the TAME system (which automates all process model components except for the construction component C3.1 of the TAME process model) and some external SEE (which automates the construction component) is necessary for three reasons: a) to enable the TAME system to collect data (e.g., the number of activations of a compiler, the number of test runs) directly from the actual construction process, b) to enable the TAME system to feed analysis results back into the ongoing construction process, and c) to enable the construction-oriented SEE to store/retrieve products into/from the experience base of the TAME system. Models for appropriate interaction between constructive and analytic processes need to be specified. Interfacing with construction-oriented SEE's poses the problem of efficiently interconnecting systems implemented in different languages and running on different machines (probably with different operating systems).

(R12) A structure suitable for distribution.

TAME will ultimately run on a distributed system consisting of at least one mainframe computer and a number of workstations. The mainframes are required to host the experience base which can be assumed to be very large. The rest of TAME might be replicated on a number of workstations.

B. Architecture

Fig. 2 describes our current view of the TAME architecture in terms of individual architectural components and their control flow interrelationships. The first prototype described in Section IV concentrates on the shaded components of Fig. 2.

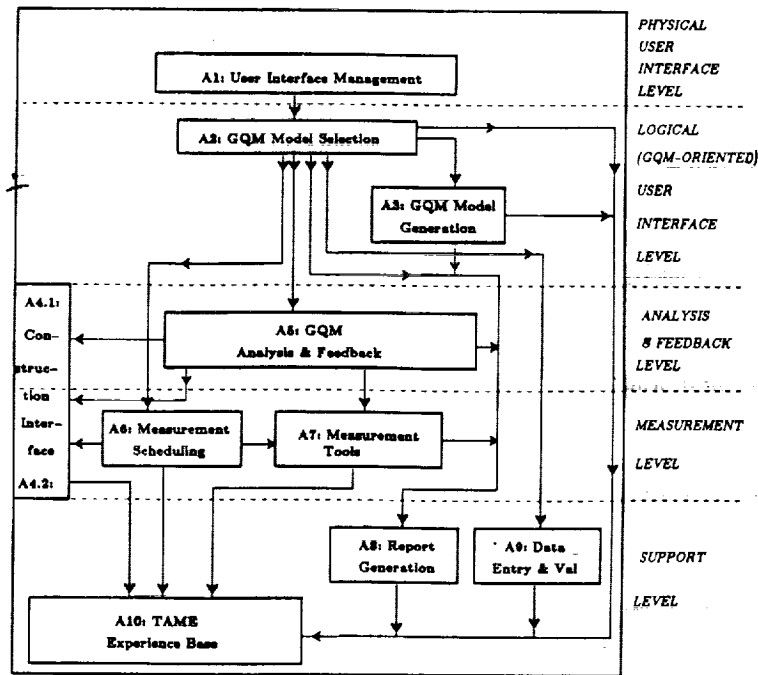We group the TAME components into five logical lev-

Fig. 2. The architectural design of the TAME system.

els, the physical user interface, logical user interface, analysis and feedback, measurement and support level. Each of these five levels consists of one or more architectural components:

• The Physical User Interface Level consists of one component:

(A1) The User Interface Management component implements the physical user interface requirement R6. It provides a choice of menu or command driven access and supports a window-oriented screen layout.

• The Logical (GQM-Oriented) User Interface Level consists of two components:

(A2) The GQM Model Selection component implements the homogeneity requirement of the logical user interface (R6). It guarantees that no access to the analysis and feedback, measurement, or support level is possible without stating the purpose for access in terms of a specific GQM model.

(A3) The GQM Model Generation component implements requirement R1 regarding the operational and quantifiable definition of GQM models either from scratch or by modifying existing models.

• The Analysis and Feedback Level consists of two components:

(A4.1) This first portion of the Construction Interface component implements the feedback interface between the TAME system and construction-oriented SEEs (part b) of requirement R11).

(A5) The GQM Analysis and Feedback component implements requirement R3 regarding execution and control of an analysis and feedback session, interpretation of

the analysis results, and proper feedback. All these activities are done in the context of a GQM model created by A3. The GQM Analysis and Feedback component needs to have access to the specific authorizations of the user in order to know which analysis functions this user can perform. The GQM Analysis and Feedback component also provides analysis functions, for example, telling the user whether certain metrics can be computed based upon the data currently available in the experience base. This analysis feature of the subsystem is used for setting and operationally defining goals, questions, and metrics, as well as actually performing analyses according to those previously established goals, questions, and metrics.

• The Measurement Level consists of three components:

(A4.2) This second portion of the Construction Interface component implements the measurement interface between the TAME system and SEE's (part a) of requirement R11) and the SEE's access to the experience base of the TAME system (part c) of requirement R11).

(A6) The Measurement Scheduling component implements requirement R2 regarding the definition (and execution) of automated data collection strategies. Such strategies for when to collect data via the measurement tools may range from collecting data whenever they are needed for an analysis and feedback session (on-line) to collecting them periodically during low-load times and storing them in the experience base (off-line).

(A7) The Measurement Tools component implements requirement R2 regarding automated data collection. The component needs to be open-ended in order to

5207

allow the inclusion of new and different measurement tools as needed.

• The Support Level consists of three components:

(A8) The Report Generation component implements requirement R7 regarding the production of all kinds of reports.

(A9) The Data Entry and Validation component implements requirement R2 regarding the entering of manually collected data and their validation. Validated data are stored in the experience base component.

(A10) The Experience Base component implements requirement R8 regarding the effective storage and retrieval of all relevant data, information and knowledge. This includes all kinds of products, analytical data (e.g., measurement data, interpretations), and analysis plans (GQM models). This component provides the infrastructure for the operation of all other components of the TAME process model and the necessary interactions among them. The experience base will also provide mechanisms supporting the learning and feedback tasks. These mechanisms include the proper packaging of experience along the context and precision/detail dimensions.

In addition, there exist two orthogonal components which for simplicity reasons are not reflected in Fig. 2:

(A11) The Data Access Control and Security component(s) implement requirement R9. There may exist a number of subcomponents distributed across the logical architectural levels. They will validate user access to the TAME system itself and to various functions at the user interface level. They will also control access to the project experience through both the measurement tools and the experience base.

(A12) The Configuration Management and Control component implements requirement R10. This component can be viewed as part of the interface to the experience base level. Data can only be entered into or retrieved from the experience base under configuration management control.

IV. FIRST TAME PROTOTYE

The first in a series of prototypes is currently being developed for supporting measurement in Ada projects [15]. This first prototype will implement only a subset of the requirements stated in Section III-A because of a) yet unsolved problems that require research, b) solutions that require more formalization, and c) problems with integrating the individual architectural components into a consistent whole. Examples of unsolved problems requiring futher research are the appropriate packaging of the experience along the context and precision/detail dimension and expert system support for interpretation purposes. Examples of solutions requiring more formalization are the GQM templates and the designing of a software engineering experience base. Examples of integration problems are the embedding of feedback loops into the construction process, and the appropriate utilization of data access control and configuration management con-

trol mechanisms. At this time, the prototype exists in pieces that have not been fully integrated together as well as partially implemented pieces.

In this section, we discuss for each of the architectural components of this TAME prototype as many of the following issues as are applicable: a) the particular approach chosen for the first prototype, b) experience with this approach, c) the current and planned status of implementation (automation) of the initial approach in the first TAME system prototype, and d) experiences with using the component:

(A1) The User Interface Management component is supposed to provide the physical user interface for accessing all TAME system functions, with the flexibility of choosing between menu and command driven modes and different window layouts. These issues are reasonably well understood by the SEE community. The first TAME prototype implementation will be menu-oriented and based upon the 'X' window mechanism. A primitive version is currently running. This component is currently not very high on our priority list. We expect to import a more sophisticated user interface management component at some later time or leave it completely to parties interested in productizing our prototype system.

(A2) The GQM Model Selection component is supposed to force the TAME user to parameterize each TAME session by first stating the objective of the session in the form of an already existing GQM model or requesting the creation of a new GQM model. The need for this restriction has been derived from the experience that data is frequently misused if it is accessible without a clear goal. The first prototype implementation does not enforce this requirement strictly. The current character of the first prototype as a research vehicle demands more flexibility. There is no question that this component needs to be implemented before the prototype leaves the research environment.

(A3) The GQM Model Generation component is supposed to allow the creation of specific GQM models either from scratch or by modifying existing ones. We have provided a set of templates and guidelines (Section II-B-2). We have been quite successful in the use of the templates and guidelines for defining goals, questions and metrics. There are a large number of organizations and environments in which the model has been applied to specify what data must be collected to evaluate various aspects of the process and product, e.g., NASA/GSFC, Burroughs, AT&T, IBM, Motorola. The application of the GQM paradigm at Hewlett Packard has shown that the templates can be used successfully without our guidance. Several of these experiences have been written up in the literature [4], [16], [17], [39], [48], [56], [60], [61]. We have been less successful in automating the process so that it ties into the experience base. As long as we know the goals and questions a priori, the appropriate data can be isolated and collected based upon the GQM paradigm. The first TAME prototype implementation is limited to sup-

3-31

port the generation of new models and the modificaton of existing models using an editor enforcing the templates and guidelines. We need to further formalize the templates and guidelines and provide traceability between goals and questions. Formalization of the templates and providing traceability is our most important research issue. In the long run we might consider using artificial intelligence planning techniques.

(A4.1 and A4.2) The Construction Interface component is supposed to support all interactions between a SEE (which supports the construction component of the TAME process model) and the TAME system. The model in Fig. 1 implies that interactions in both directions are required. We have gained experience in manually measuring the construction process by monitoring the execution of a variety of techniques (e.g., code reading [57], testing [20], and CLEANROOM development [61]) in various environments including the SEL [4], [48]. We have also learned how analysis results can be fed back into the ongoing construction process as well as into corporate experience [4], [48]. Architectural component A4.1 is not part of this first TAME prototype. The first prototype implementation of A4.2 is limited to allowing for the integration of (or access to) external product libraries. This minimal interface is needed to have access to the objects for measurement. No interface for the on-line measurement of ongoing construction processes is provided yet.

(A5) The GQM Analysis and Feedback component is supposed to perform analysis according to a specific GQM model. We have gained a lot of experience in evaluating various kinds of experiments and case studies. We have been successful in collecting the appropriate data by tracing GQM models top-down. We have been less successful in providing formal interpretation rules allowing for the bottom-up interpretation of the collected data. One automated approach to providing interpretation and feedback is through expert systems. ARROWSMITH-P provides interpretations of software project data to managers [44]; it has been tested in the SEL/NASA environment. The first prototype TAME implementation triggers the collection of prescribed data (top-down) and presents it to the user for interpretation. The user-provided interpretations will be recorded (via a knowledge acquisition system) in order to accumulate the necessary knowledge that might lead us to identifying interpretation rules in the future.

(A6) The Measurement Scheduling component is supposed to allow the TAME user to define a strategy for actually collecting data by running the measurement tools. Choosing the most appropriate of many possible strategies (requirements Section III-A) might depend on the response times expected from the TAME system or the storage capacity of the experience base. Our experience with this issue is limited because most of our analyses were human scheduled as needed [4], [48]. This component will not be implemented as part of the first prototype. In this prototype, the TAME user will trigger the execution of measurement activities explicitly (which can, of course,

be viewed as a minimal implementation supporting a human scheduling strategy).

(A7) The Measurement Tools component is supposed to allow the collection of all kinds of relevant process and product data. We have been successful in generating tools to gather data automatically and have learned from the application of these tools in different environments. Within NASA, for example, we have used a coverage tool to analyze the impact of test plans on the consistency of acceptance test coverage with operational use coverage [53]. We have used a data bindings tool to analyze the structural consistency of implemented systems to their design [41], and studied the relationship between faults and hierarchical structure as measured by the data bindings tool [60]. We have been able to characterize classes of products based upon their syntactic structure [35]. We have not, however, had much experience in automatically collecting process data. The first prototype TAME implementation consists of measurement tools based on the above three. The first tool captures all kinds of basic Ada source code information such as lines of code and structural complexity metrics [35], the second tool computes Ada data binding metrics, and the third tools captures dynamic information such as test coverage metrics [65]. One lesson learned has been that the development of measurement tools for Ada is very often much more than just a reimplementation of similar tools for other languages. This is due to the very different Ada language concepts. Furthermore, we have recognized the importance of having an intermediate representation level allowing for a language independent representation of software product and process aspects. The advantage of such an approach will be that this intermediate representation needs to be generated only once per product or process. All the measurement tools can run on this intermediate representation. This will not only make the actual measurement process less time-consuming but provide a basis for reusing the actual measurement tools to some extent across different language environments. Only the tool generating the intermediate representation needs to be rebuilt for each new implementation language or TAME host enviroment.

(A8) The Report Generator component is supposed to allow the TAME user to produce a variety of reports. The statistics and business communities have commonly accepted approaches for presenting data and interpretations effectively (e.g., histograms). The first TAME prototype implementation does not provide a separate experience base reporting facility. Responsibility for reporting is attached to each individual prototype component; e.g., the GQM Model Generation component provides reports regarding the models, each measurement tool reports on its own measurement data.

(A9) The Data Entry and Validation component is supposed to allow the TAME user to enter all kinds of manually collected data and validate them. Because of the changing needs for measurement, this component must allow for the definition of new (or modification of existing)

5207

data collection forms as well as related validation (integrity) rules. If possible, the experience base should be capable of adapting to new needs based upon new form definitions. We have had lots of experience in designing forms and validations rules, using them, and learning about the complicated issues of deriving validation rules [4], [48]. The first prototype implementation will allow the TAME user to input off-line collected measurement data and validate them based upon a fixed and predefined set of data collection forms [currently in use in NASA's Software Engineering Laboratory (SEL)]. This component is designed but not yet completely implemented. The practical use of the TAME prototype requires that this component provide the flexibility for defining and accepting new form layouts. One research issue is identifying the easiest way to define data collection forms in terms of a grammar that could be used to generate the corresponding screen layout and experience base structure.

(A10) The Experience Base component allows for effective storage and retrieval of all relevant experience ranging from products and process plans (e.g., analysis plans in the form of GQM models) to measurement data and interpretations. The experience base needs to mirror the project environment. Here we are relying on the experience of several faculty members of the database group at the University of Maryland. It has been recognized that current database technology is not sufficient, for several reasons, to truly mirror the needs of software engineering projects [26]. The first prototype TAME implementation is built on top of a relational database management system. A first database schema [46] modeling products as well as measurement data has been implemented. We are currently adding GQM models to the schema. The experiences with this first prototype show that the amount of experience stored and its degree of formalism (mostly data) is not yet sufficient. We need to better package that data in order to create pieces of information or knowledge. The GQM paradigm provides a specification of what data needs to be packaged. However, without more formal interpretation rules, the details of packaging cannot be formalized. In the long run, we might include expert system technology. We have also recognized the need for a number of built-in GQM models that can either be reused without modification or guide the TAME user during the process of creating new GQM models.

(A11) The Data Access Control and Security component is supposed to guarantee that only authorized users can access the TAME system and that each user can only access a predefined window of the experience base. The first prototype implements this component only as far as user access to the entire system is concerned.

(A12) The Configuration Management and Control component is supposed to guarantee consistency between the objects of measurement (products and processes), the plans for measurement (GQM models), the data collected from the objects according to these plans, and the attached interpretations. This component will not be implemented in the first prototype.

The integration of all these architectural components is incomplete. At this point in time we have integrated the first versions of the experience base, three measurement tools, a limited version of the GQM analysis and feedback component, the GQM generation component, and the user interface management component. Many of the UNIX® tools (e.g., editors, print facilities) have been integrated into the first prototype TAME system to compensate for yet missing components. This subset of the first prototype is running on a network of SUN-3's under UNIX. It is implemented in Ada and C.

This first prototype enables the user to generate GQM models using a structured editor. Existing models can be selected by using a unique model name. Support for selecting models based on goal definitions or for reusing existing models for the purpose of generating new models is offered, but the refinement of goals into questions and metrics relies on human intervention. Analysis and feedback sessions can be run according to existing GQM models. Only minimal support for interpretation is provided (e.g., histograms of data). Measurement data are presented to the user according to the underlying model for his/her interpretation. Results can be documented on a line printer. The initial set of measurement tools allows only the computation of a limited number of Ada-source-code-oriented static and dynamic metrics. Similar tools might be used in the case of Fortran source code [33].

V. SUMMARY AND CONCLUSIONS

We have presented a set of software engineering and measurement principles which we have learned during a dozen years of analyzing software engineering processes and products. These principles have led us to recognize the need for software engineering process models that integrate sound planning and analysis into the construction process.

In order to achieve this integration the software engineering process needs to be tailorable and tractable. We need the ability to tailor the execution process, methods and tools to specific project needs in a way that permits maximum reuse of prior experience. We need to control the process and product because of the flexibility required in performing such a focused development. We also need as much automated support as possible. Thus an integrated software engineering environment needs to support all of these issues.

In the TAME project we have developed an improvement-oriented (integrated) process model. It stresses a) the characterization of the current status of a project environment, b) the planning for improvement integrated into software projects, and c) the execution of the project according to the prescribed project plans. Each of these

---

®UNIX is a registered trademark of AT&T Bell Laboratories.

5207

tasks must be dealt with from a constructive and analytic perspective.

To integrate the constructive and analytic aspects of software development, we have used the GQM paradigm. It provides a mechanism for formalizing the characterization and planning tasks, controlling and improving projects based on quantitative analysis, learning in a deeper and more systematic way about the software process and product, and feeding back the appropriate experience to current and future projects.

The effectiveness of the TAME process model depends heavily on appropriate automated support by an ISEE. The TAME system is an instantiation of the TAME process model into an ISEE; it is aimed at supporting all aspects of characterization, planning, analysis, learning, and feedback according to the TAME process model. In addition, it formalizes the feedback and learning mechanisms by supporting the synthesis of project experience, the formalization of its representation, and its tailoring towards specific project needs. It does this by supporting goal development into measurement via templates and guidelines, providing analysis of the development and maintenance processes, and creating and using experience bases (ranging from databases of historical data to knowledge bases that incorporate experience from prior projects).

We discussed a limited prototype of the TAME system, which has been developed as the first of a series of prototypes that will be built using an iterative enhancement model. The limitations of this prototype fall into two categories, limitations of the technology and the need to better formalize the model so that it can be automated.

The short range (1–3 years) goal for the TAME system is to build the analysis environment. The mid-range goal (3–5 years) is to integrate the system into one or more existing or future development or maintenance environments. The long range goal (5–8 years) is to tailor those environments for specific organizations and projects.

The TAME project is ambitious. It is assumed it will evolve over time and that we will learn a great deal from formalizing the various aspects of the TAME project as well as integrating the various paradigms. Research is needed in many areas before the idealized TAME system can be built. Major areas of study include measurement, databases, artificial intelligence, and systems. Specific activities needed to support TAME include: more formalization of the GQM paradigm, the definition of better models for various quality and productivity aspects, mechanisms for better formalizing the reuse and tailoring of project experience, the interpretation of metrics with respect to goals, interconnection languages, language independent representation of software, access control in general and security in particular, software engineering database definition, configuration management and control, and distributed system architecture. We are interested in the role of further researching the ideas and principles of the TAME project. We will build a series of

evolving prototypes of the system in order to learn and test out ideas.

ACKNOWLEDGMENT

REFERENCES

[1] W. Agresti, "SEL Ada experiment: Status and design experience," in *Proc. Eleventh Annu. Software Engineering Workshop*, NASA Goddard Space Flight Center, Greenbelt, MD, Dec. 1986.

[2] J. Bailey and V. R. Basili, "A meta-model for software development resource expenditures," in *Proc. Fifth Int. Conf. Software Engineering*, San Diego, CA, Mar. 1981, pp. 107-116.

[3] V. R. Basili, "Quantitative evaluation of software engineering methodology," in *Proc. First Pan Pacific Computer Conf.*, Melbourne, Australia, Sept. 1985; also available as Tech. Rep. TR-1519, Dep. Comput. Sci., Univ. Maryland, College Park, July 1985.

[4] V. R. Basili, "Can we measure software technology: Lessons learned from 8 years of trying," in *Proc. Tenth Annu. Software Engineering Workshop*, NASA Goddard Space Flight Center, Greenbelt, MD, Dec. 1985.

[5] ——, "Evaluating software characteristics: Assessement of software measures in the Software Engineering Laboratory," in *Proc. Sixth Annu. Software Engineering Workshop*, NASA Goddard Space Flight Center, Greenbelt, MD, 1981.

[6] V. R. Basili and J. Beane, "Can the Parr curve help with the manpower distribution and resource estimation problems," *J. Syst. Software*, vol. 2, no. 1, pp. 59-69, 1981.

[7] V. R. Basili and K. Freburger, "Programming measurement and estimation in the Software Engineering Laboratory," *J. Syst. Software*, vol. 2, no. 1, pp. 47-57, 1981.

[8] V. R. Basili and D. H. Hutchens, "An empirical study of a syntactic measure family," *IEEE Trans. Software Eng.*, vol. SE-9, no. 11, pp. 664-672, Nov. 1983.

[9] V. R. Basili and E. E. Katz, "Measures of interest in an Ada development," in *Proc. IEEE Comput. Soc. Workshop Software Engineering Technology Transfer*, Miami, FL, Apr. 1983, pp. 22-29.

[10] V. R. Basili, E. E. Katz, N. M. Panlilio-Yap, C. Loggia Ramsey, and S. Chang, "Characterization of an Ada software development," *Computer*, pp. 53-65, Sept. 1985.

[11] V. R. Basili and C. Loggia Ramsey, "ARROWSMITH-P: A prototype expert system for software engineering management," in *Proc. IEEE Symp. Expert Systems in Government*, Oct. 23-25, 1985, pp. 252-264.

[12] V. R. Basili and N. M. Panlilio-Yap, "Finding relationships between effort and other variables in the SEL," in *Proc. IEEE COMPSAC*, Oct. 1985.

[13] V. R. Basili and B. Perricone, "Software errors and complexity: An empirical investigation," *ACM, Commun.*, vol. 27, no. 1, pp. 45-52, Jan. 1984.

[14] V. R. Basili and R. Reiter, Jr., "A controlled experiment quantitatively comparing software development approaches," *IEEE Trans. Software Eng.*, vol. SE-7, no. 3, pp. 299-320, May 1981.

[15] V. R. Basili and H. D. Rombach, "TAME: Tailoring an Ada measurement environment," in *Proc. Joint Ada Conf.*, Arlington, VA, Mar. 16-19, 1987, pp. 318-325.

[16] ——, "Tailoring the software process to project goals and environments," in *Proc. Ninth Int. Conf. Software Engineering*, Monterey, CA, Mar. 30-Apr. 2, 1987, pp. 345-357.

[17] ——, "TAME: Integrating measurement into software environments," Dep. Comput. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1764 (TAME-TR-1-1987), June 1987.

[18] ——, "Software reuse: A framework," in *Proc. Tenth Minnowbrook Workshop Software Reuse*, Blue Mountain Lake, NY, Aug. 1987.

[19] V. R. Basili and R. W. Selby, Jr., "Data collection and analysis in software research and management," in *Proc. Amer. Statist. Ass. and Biomeasure Soc. Joint Statistical Meetings*, Philadelphia, PA, Aug. 13-16, 1984.

[20] ——, "Comparing the effectiveness of software testing strategies," *IEEE Trans. Software Eng.*, vol. SE-13, no. 12, pp. 1278-1296, Dec. 1987.

[21] ——, "Calculation and use of an environment's characteristic software metric set," in *Proc. Eighth Int. Conf. Software Engineering*, London, England, Aug. 1985.

[22] V. R. Basili, R. W. Selby, and D. H. Hutchens, "Experimentation in software engineering," *IEEE Trans. Software Eng.*, vol. SE-12, no. 7, pp. 733-743, July 1986.

[23] V. R. Basili, R. W. Selby, and T.-Y. Phillips, "Metric analysis and data validation across Fortran projects," *IEEE Trans. Software Eng.*, vol. SE-9, no. 6, pp. 652-663, Nov. 1983.

[24] V. R. Basili and A. J. Turner, "Iterative enhancement: A practical technique for software development," *IEEE Trans. Software Eng.*, vol. SE-1, no. 4, pp. 390-396, Dec. 1975.

[25] V. R. Basili and D. M. Weiss, "A methodology for collecting valid software engineering data," *IEEE Trans. Software Eng.*, vol. SE-10, no. 3, pp. 728-738, Nov. 1984.

[26] P. A. Bernstein, "Database system support for software engineering," in *Proc. Ninth Int. Conf. Software Engineering*, Monterey, CA, Mar. 30-Apr. 2, 1987, pp. 166-178.

[27] D. Bjorner, "On the use of formal methods in software development," in *Proc. Ninth Int. Conf. Software Engineering*, Monterey, CA, Mar. 30-Apr. 2, 1987, pp. 17-29.

[28] B. W. Boehm, "Software engineering," *IEEE Trans. Comput.*, vol. C-25, no. 12, pp. 1226-1241, Dec. 1976.

[29] ——, *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[30] ——, "A spiral model of software development and enhancement," *ACM Software Eng. Notes*, vol. 11, no. 4, pp. 22-42, Aug. 1986.

[31] B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative evaluation of software quality," in *Proc. Second Int. Conf. Software Engineering*, 1976, pp. 592-605.

[32] C. Brophy, W. Agresti, and V. R. Basili, "Lessons learned in use of Ada oriented design methods," in *Proc. Joint Ada Conf.*, Arlington, VA, Mar. 16-19, 1987, pp. 231-236.

[33] W. J. Decker and W. A. Taylor, "Fortran static source code analyzer program (SAP)," NASA Goddard Space Flight Center, Greenbelt, MD, Tech. Rep. SEL-82-002, Aug. 1982.

[34] C. W. Doerflinger and V. R. Basili, "Monitoring software development through dynamic variables," *IEEE Trans. Software Eng.*, vol. SE-11, no. 9, pp. 978-985, Sept. 1985.

[35] D. L. Doubleday, "ASAP: An Ada static source code analyzer program," Dep. Comput. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1895, Aug. 1987.

[36] M. Dowson, "ISTAR—An integrated project support environment," in *ACM Sigplan Notices (Proc. Second ACM Software Eng. Symp. Practical Development Support Environments)*, vol. 2, no. 1, Jan. 1987.

[37] M. Dyer, "Cleanroom software development method," IBM Federal Systems Division, Bethesda, MD, Oct. 14, 1982.

[38] J. Gannon, E. E. Katz, and V. R. Basili, "Measures for Ada packages: An initial study," *Commun. ACM*, vol. 29, no. 7, pp. 616-623, July 1986.

[39] R. B. Grady, "Measuring and managing software maintenance," *IEEE Software*, vol. 4, no. 5, pp. 35-45, Sept. 1987.

[40] M. H. Halstead, *Elements of Software Science*. New York: Elsevier North-Holland, 1977.

[41] D. H. Hutchens and V. R. Basili, "System structure analysis: Clustering with data bindings," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 749-757, Aug. 1985.

[42] E. E. Katz and V. R. Basili, "Examining the modularity of Ada programs," in *Proc. Joint Ada Conf.*, Arlington, VA, Mar. 16-19, 1987, pp. 390-396.

[43] E. E. Katz, H. D. Rombach, and V. R. Basili, "Structure and maintainability of Ada programs: Can we measure the differences?" in *Proc. Ninth Minnowbrook Workshop Software Performance Evaluation*, Blue Mountain Lake, NY, Aug. 5-8, 1986.

[44] C. Loggia Ramsey and V. R. Basili, "An evaluation of expert systems for software engineering management," Dep. Comput. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1708, Sept. 1986.

[45] M. Marcus, K. Sattley, S. C. Schaffner, and E. Albert, "DAPSE: A distributed Ada programming support environment," in *Proc. IEEE Second Int. Conf. Ada Applications and Environments*, 1986, pp. 115-125.

[46] L. Mark and H. D. Rombach, "A meta information base for software engineering," Dep. Comput. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1765, July 1987.

[47] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. SE-2, no. 4, pp. 308-320, Dec. 1976.

[48] F. E. McGarry, "Recent SEL studies," in *Proc. Tenth Annu. Software Engineering Workshop*. NASA Goddard Space Flight Center, Greenbelt, MD, Dec. 1985.

[49] L. Osterweil, "Software processes are software too," in *Proc. Ninth Int. Conf. Software Engineering*, Monterey, CA, Mar. 30-Apr. 2, 1987, pp. 2-13.

[50] F. N. Parr, "An alternative to the Rayleigh curve model for software development effort," *IEEE Trans. Software Eng.*, vol. SE-6, no. 5, pp. 291-296, May 1980.

[51] L. Putnam, "A general empirical solution to the macro software sizing and estimating problem," *IEEE Trans. Software Eng.*, vol. SE-4, no. 4, pp. 345-361, Apr. 1978.

[52] C. V. Ramamoorthy, Y. Usuda, W.-T. Tsai, and A. Prakash, "GENESIS: An integrated environment for supporting development and evolution of software," in *Proc. COMPSAC*, 1985.

[53] J. Ramsey and V. R. Basili, "Analyzing the test process using structural coverage," in *Proc. Eighth Int. Conf. Software Engineering*, London, England, Aug. 1985, pp. 306-311.

[54] H. D. Rombach, "Software design metrics for maintenance," in *Proc. Ninth Annu. Software Engineering Workshop*. NASA Goddard Space Flight Center, Greenbelt, MD, Nov. 1984.

[55] ——, "A controlled experiment on the impact of software structure on maintainability," *IEEE Trans. Software Eng.*, vol. SE-13, no. 3, pp. 344-354, Mar. 1987.

[56] H. D. Rombach and 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.

[57] H. D. Rombach, V. R. Basili, and R. W. Selby, Jr., "The role of code reading in the software life cycle," in *Proc. Ninth Minnowbrook Workshop Software Performance Evaluation*, Blue Mountain Lake, NY, August 5-8, 1986.

[58] W. W. Royce, "Managing the development of large software systems: Concepts and techniques," in *Proc. WESCON*, Aug. 1970.

[59] R. W. Selby, Jr., "Incorporating metrics into a software environment," in *Proc. Joint Ada Conf.*, Arlington, VA, Mar. 16-19, 1987, pp. 326-333.

[60] R. W. Selby and V. R. Basili, "Analyzing error-prone system coupling and cohesion," Dep. Comput. Sci., Univ. Maryland, College Park, Tech. Rep., in preparation.

[61] R. W. Selby, Jr., V. R. Basili, and T. Baker, "CLEANROOM software development: An empirical evaluation," *IEEE Trans. Software Eng.*, vol. SE-13, no. 9, pp. 1027-1037, Sept. 1987.

[62] C. E. Walston and C. P. Felix, "A method of programming measurement and estimation," *IBM Syst. J.*, vol. 16, no. 1, pp. 54-73, 1977.

[63] A. I. Wasserman and P. A. Pircher, "Visible connections," *UNIX Rev.*, Oct. 1986.

[64] *Webster's New Collegiate Dictionary*. Springfield, MA: Merriam, 1981.

[65] L. Wu, V. R. Basili, and K. Reed, "A structure coverage tool for Ada software systems," in *Proc. Joint Ada Conf.*, Arlington, VA, Mar. 16-19, 1987, pp. 294-303.

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

Victor R. Basili (M'83-SM'84) is Professor and Chairman of the Department of Computer Science at the University of Maryland, College Park. He was involved in the design and development of several software projects, including the SIMPL family of programming languages. He is currently measuring and evaluating software development in industrial and government settings and has consulted with many agencies and organizations,-including IBM, GE, CSC, GTE, MCC, AT&T, Motorola, HP, NRL, NSWC, and NASA. He is

one of the founders and principals in the Software Engineering Laboratory, a joint venture between NASA Goddard Space Flight Center, the University of Maryland and Computer Sciences Corporation, established in 1976. He has been working on the development of quantitative approaches for software management, engineering, and quality assurance by developing models and metrics for the software development process and product. He has authored over 90 papers. In 1982, he received the Outstanding Paper Award from the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING for his paper on the evaluation of methodologies.

Dr. Basili is currently the Editor-in-Chief of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING and was Program Chairman for several conferences including the 6th International Conference on Software Engineering. He has served on the Editorial Board of the *Journal of Systems and Software*. He is a member of the Board of Governors of the IEEE Computer Society.

H. Dieter Rombach received the B.S. degree (Vordiplom) in mathematics and the M.S. degree (Diplom) in mathematics and computer science from the University of Karlsruhe, West Germany, and the Ph.D. degree (Dr. rer. nat.) in computer science from the University of Kaiserslautern, West Germany.

He is an Assistant Professor of Computer Science at the University of Maryland, College Park. He is also affiliated with the University of Maryland Institute for Advanced Computer Studies (UMIACS) and the Software Engineering Laboratory (SEL), a joint venture between NASA Goddard Space Flight Center, the University of Maryland, and Computer Sciences Corporation. His research interests include software methodologies, measurement of the software process and its products, software engineering environments, and distributed systems.

Dr. Rombach served as Guest Editor for the *IEEE Software* magazine Special Issue on Software Quality Assurance (September 1987). He is a member of the IEEE Computer Society, the Association for Computing Machinery, and the German Computer Society (GI).

5207

VALIDATING THE TAME RESOURCE DATA MODEL*

D. Ross Jeffery (1) & Victor R. Basili (2)

(1) University of New South Wales, Australia
(2) University of Maryland, College Park, MD 20742

Abstract

This paper presents a conceptual model of
software development resource data and validates
the model by reference to the published literature
on necessary resource data for development sup-
port environments. The conceptual model
presented here was developed using a top-down
strategy. A resource data model is a prerequisite
to the development of integrated project support
environments which aim to assist in the processes
of resource estimation, evaluation and control.
The model proposed is a four dimensional view of
resources which can be used for resource estima-
tion, utilization, and review. The model is vali-
dated by reference to three publications on
resource databases, and the implications of the
model arising out of these comparisons is dis-
cussed.

Keywords : software process, methods, tools,
conceptual model, resources, estimation, environ-
ments, software engineering database, validation

INTRODUCTION

To date, the approach taken to the accumulation
of knowledge concerning the software process has
been largely bottom-up. Studies have been carried
out to determine the existence and nature of pro-
ject relationships. These studies, such as [Wolver-
ton 74], [Nelson 67], [Chrysler 78], [Sackman et.al.
68], [Basili, Panlilio-Yap 85], [Basili, Freburger
81], [Basili, Selby, Phillips 83], [Walston, Felix
77]), [Jeffery 87a,87b], and [Jeffery, Lawrence
1979, 1985] have explored the relationships
between project variables, searching for an under-
standing of the software process and product. For
example, relationships between effort and size,
errors and methods, and test strategy and bug
identification, have been found.

This paper has two major aims:

1) To briefly present a top-down characterization
(TDC) structure of software project resource data,
which aims to facilitate :

  1. Further accumulation of knowledge of pro-
  ject resource characteristics and metrics within
  a theoretical structure.

  2. The storage of project resource data in a
  generalized structured way so that estimation,
  evaluation, and control can be facilitated using
  an organized quantitative and qualitative data
  base.

2) To validate this structure against published
resource data models.

The characterization structure of resource data is
a prerequisite to the development of an Integrated
Project Support Environment (IPSE) in which it
is possible to:

  1. Objectively choose appropriate software
  processes.

  2. Estimate the process characteristics such as
  time, cost, and quality

  3. Evaluate the extent to which the resource
  aims are being met during development, and

  4. Improve the software process and product.

The structure presented and validated here is a
part of the TAME (Tailoring A Measurement
Environment) project which seeks to develop an
integrated software project measurement,
analysis, and evaluation environment. This
environment is based in part on the evolutionary
improvement paradigm [discussed in Basili, Rom-
bach 87]. It is also based on the "Goal-Question-
Metric" paradigm outlined in [Basili 85] and
[Basili, Weiss 84].

The aims of this paper are firstly to present the
TDC structure or model for the perception of
software development resources which will assist
in the process of taking those aims of, say, a
development manager and translating them into a
set of questions and metrics which can be used to
measure the software process. It is meant to be
independent of the particular process model used

5207

for development and maintenance. A full description of the model, including its dynamic nature is described in [Jeffery, Basili 87a and 87b]. The paper secondly aims to validate the model by a comparison of the model with the resource data models presented in the literature.

2. THE PROJECT ENVIRONMENT CHARACTERISTICS

Resources are consumed during the software process in order to deliver a software product. The software process has overall characteristics which are super-ordinate to the resources consumed. Therefore, before resource data can be characterized it is necessary that a process characterization profile be established. This characterization includes data on factors such as:

> project type
> organizational development conventions
> project manager preferences
> target computer system
> development computer system
> project schedules or milestones
> project deliverables

In this data the broad project and its environment characteristics are established. For example, is the process using evolutionary development or a waterfall method? Is the project to be developed by in-house staff or external contractors? What organizational constraints are being imposed on the project development time? What management constraints are being imposed, say on staffing levels?

These factors form the environment in which the software process must occur, and will therefore determine, in many ways, the nature of that software process. A simple example of this is the question of the process model - evolutionary or waterfall. This constraint establishes milestones and the pattern of resource use, and therefore partially determines the interpretation of the resource data collected.

3. THE RESOURCE CLASSIFICATION

At the level below the characterization of the project and its environment we are interested in classifying the resources consumed in the generation of the software product. In this section of the paper we present a structure for that classification. This structure covers only the resource aspect of the project and is therefore only concerned with the software process and the resources consumed or used in the process. The model is not concerned with the software product. As stated above, the resource model was first developed and presented in [Jeffery, Basili 87]

The model structure consists of a four dimensional view. This four dimensional view is divided into two segments:

> 1. resource type, and
> 2. resource use

In a software process the two segments being separated are (1) the nature and characteristics of the resource, and (2) the manner in which we look at or consider the consumption of that resource.

3.1 Resource Type

In the first segment we are concerned with classifying the nature of the resource; is it someone's time, or a physical object such as a computer, or a logical object such as a piece of software? We are also interested in describing the properties of those resources such as description, model number, and cost per unit of consumption.

By decomposing the resources into different types different views of the resources can be provided. For example, it may be important for operations personnel to know a breakdown of the hardware resources used on a project according to the different physical machines being used, whereas from a project manager's perspective at a point in time, the specific machine may not be of interest, but the availability of a certain class of machine may be critical. Resource managers will be interested in the types of resources available (for example, people) and the characteristics of those resources for project planning purposes. Thus the categorization provided here is the basis of the resource management environment, in that it is in this segment of the model that the resources are listed and described.

The resources of a software project can be classified as:

> - hardware
> - software
> - human
> - support (supplies, materials, communications facility costs, etc.)

These categories are meant to be mutually exclusive and exhaustive and therefore are able to contain each instance of resource data in one or other of the categories.

Hardware resources encompass all equipment used or potentially able to be used in the environment under consideration. (For example, target and development machines, terminals, workstations).

Software resources encompass all previously existing programs and software systems used or potentially able to be used in the environment under consideration. (For example, compilers, operating systems, utility routines, previously existing application software).

188

5207

Human resources encompass all the people used or potentially able to be used for development, operations, and maintenance in the environment under consideration whether internal or external (subcontractors, consultants, etc)

Support resources encompass all of the additional facilities such as materials, communications, and supplies which are used or potentially able to be used in the environment under consideration.

The values associated with these resources may be stored in both price and volume measures, where volume means, for example, hours of use or availability, or the number of times a resource is needed, and price refers to the $ values associated with that resource. This may be a cost per unit measure or a cost per period of time.

This four-way classification provides an initial resource-type decomposition. The aim in this decomposition is to separate the major resource elements that are used in the software process in order to provide manageability. This initial separation is necessary because of the very different nature of each of these resource types and the consequent difference in attributes and management techniques which are necessary in the estimation, evaluation, and control of each of these resource categories.

Further decomposition within this segment may be desirable and will be dependent on the goals of the responsible persons. The number of different possibilities increase as the decomposition continues within each of the major resource categories. For example, the exact nature of the resource decomposition within the hardware category will vary significantly from one organization to another because of the different hardware utilized and the organizational structure surrounding that hardware utilization. For example, it may be desirable to decompose hardware into target and development hardware if there is a difference, and software into operating systems and languages/editors in order to model say the availability of cross-compilers.

3.2 Resource Use

Over the type segment we need to impose the second segment; the "use" structure. The categorization within this dimension allows the resources consumption to be associated with different perspectives of the software process. For example, it is through this use structure that we are able to distinguish, for example,

between prior-project expectations of consumption and resources actually consumed, or

between resources consumed in each phase of the project, or

between the utilization of a resource and the availability of that resource, or

between an ideal view of resource planning and the resources actually available.

The use structure consists of :

1. INCURRENCE
    1.1 Estimated
    1.2 Actual

2. AVAILABILITY
    2.1 Desirable
    2.2 Accessible
    2.3 Utilized

3. USE DESCRIPTORS
    3.1 Work type
    3.2 Point in Time
    3.3 Resources Utilized

3.2.1 Incurrence

This category allows the resource information to be gathered and used in a manner suitable to the management of the resource. It is necessary, for example, to store data on *estimated* resource usage, resource requirements, and resource availability.

This data is necessarily kept separate from the *actual* resource incurrence or use, which is stored via the actual category.

These two categories then permit process tracking via comparisons between them and extrapolation from the actual data. At the project summary points, explanations and defined data accumulations on estimated and actual resource use provide feedback on the process. This *feedback* should contain reasons for variance between the estimated and actual so that a facility for corporate memory can be established and the necessary data stored to facilitate and explain any updates of the current resource values. It needs to be noted that the model proposed allows for different estimates and actuals at different points in time.

The two classifications are the basis for the structure proposed because they constitute significantly different viewpoints on the process, and again provide mutually exclusive categorization which will facilitate management estimation, evaluation, and control.

This structure requires that process data, as it changes in value during the project, will not be lost but will be stored in an accessible manner so that meaningful analysis of projects can be carried out using a database that provides complete details of the project history.

189

5207

This philosophy specifically addresses the need for a corporate memory concerning past projects. By implementing such a structured project log the basic data for such a memory is available in numeric and text format.

3.2.2 Availability

This category allows storage of a resource use by :

- desirable
- accessible
- utilized

This categorization provides further refinement of the resource data. Through this, and say the incurrence category, it is possible to compare the actual resources utilized with the estimated utilization, and then trace possible reasons for variance through the desirable and accessible dimensions. That is, differences between planned availability and actual availability of a resource will be significant in understanding the software resource utilization that occurred during the process.

Desirable is defined as all the resources that are reasonably expected to be of value on the project.

Accessible is a subset of desirable (when considering the project resources only) and is used to define the resources which are able to be used on the project.

The difference between desirable and accessible is those resources seen as desirable for the project but which were not available for use during the project. This difference may occur, for example, because of budget constraints or inability to recruit staff. The desirable resource list permits an "ideal" planning view. When compared with accessible it allows management to see the compromises that were made in establishing the project, thus facilitating a very explicit basis for risk management within the resource database. The database is thereby able to hold views of not only the resources actually applied to the project but also those resources which were considered to be desirable along with the reasons for their use or non-use. In this way the resource trade-offs are made explicit.

Utilized is a subset of accessible and is defined as the resources which are used in a project.

The difference between accessible and utilized represents those resources available for the project but not used. This difference will arise because of three possible reasons:

1. The resources prove to be inappropriate for the project under consideration, or

2. The resources are appropriate but they are excess to those needed

3. The resources are appropriate, and their use is contingent on an uncertain future event.

The use of these storage categories is somewhat complex and is explained in detail further below in section 3.4.2.

Through this availability category we are able to distinguish between:

(1) the resources which are reasonably expected to be beneficial to the process (desirable),

(2) the resources which exist in the organization and are able to be used if needed (accessible), and

(3) the resources which are used in a project (utilized)

Through this categorization it is then possible to track resource usage and to pinpoint their use or non-use and to ascribe reasons particularly to their non-use as in the case of non-accessibility. As in the INCURRENCE category, the reasons for divergence between desirable, accessible, and utilized are stored in a *feedback* facility.

3.2.3 Use Descriptors

This category provides a description of the consumption of the resource item in terms of three essential characteristics of the consumption that item:

1. *The Nature of the Work* being done by the resource: (e.g. coding, inspecting, or designing) This category can be used in conjunction with other views to distinguish between process activities, such as human resources estimated to be desirable in *design* work, or machine resources actually utilized in *testing* , or elapsed time implications of *inspections.*

2. *Point in Calendar Time* : This category pinpoints the resource item by calendar time. In this way resource items (estimated or actual; desirable, accessible, or utilized) are associated with a specific point in time or period of time. This facilitates tracing of time dependent relationships and the comparison of resource values over time.

3. *Resources Utilized* : This category measures the extent of resource consumption in terms of hours, dollars, units, or whatever is the appropriate measure of use.

The Use Descriptors also provide the link to the work breakdown structure which is commonly embodied in process models. This link is established through the association of a particular piece of work being done at a point in time with the work package described in the work breakdown structure. This point is discussed further below in Section 8, Validating the Model.

190

3-40

5207

3.3 COMBINING THE VIEWS

The structure suggested here can be viewed as a hierarchy for the purpose of explanation. Such a hierarchy is shown in Figure 1.
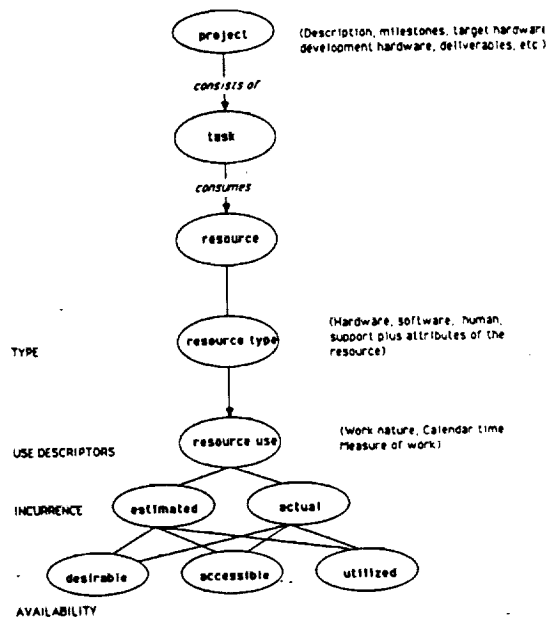

FIGURE 1. THE STRUCTURE OF THE TDC MODEL

In this figure we see that the proposed structure views the software project (which has attributes describing that project) consuming resources. The resources are characterized as having four dimensions of interest (type, use, incurrence, and availability). At the resource type level we describe each resource as being one of hardware, software, human, or support, and having various attributes. The attributes for each of these four types will be different in nature. For example, the human attributes might include name, address, organizational unit, skills, pay rate, unit cost, age, and so forth. The attributes for hardware will be quite different, describing manufacturer, purchase date, memory capacity, network connections, or similar types of characteristics.

At the next level in the diagram we model the use of the resource. In the first instance this involves the type of work that the resource is performing, the point (or span) in calendar time at which the work is being done, and the measure of the amount of work done. This last measure (amount of work) might be expressed in person-time, execution-time, connect-time, or whatever is the relevant measure of work for the resource instance.

The use of the resource is then described as being either estimated or actual, and both of these may be desirable, accessible, or utilized. In this way the following concepts are supported :

1. *Estimated Desirable* : The resources considered "ideal" at various stages of the planning process.

2. *Estimated Accessible* : The resources which are expected to be available for use in the process, given the constraints imposed on the software process (a contingency plan).

3. *Estimated Utilized* : The resources which it is anticipated will be used in the software process.

4. *Actual Desirable* : With hindsight, the resources which proved to be the "ideal" considering the events that occurred in the software process. A part of the learning process.

5. *Actual Accessible* : Again with hindsight, the resources which were actually available and could have been utilized. A part of the learning process.

6. *Actual Utilized* : The resources actually used in the software process.

Categories one through three are used initially for planning purposes. The numeric and text values associated with each of these three categories may be derived from:

    a. individual or group knowledge
    b. a knowledge base
    c. a database of prior projects, and/or
    d. algorithmic models

At the very simplest level, the planning process might establish only numeric values in the estimated utilized category based on individual knowledge alone. In essence, this is the only form of estimation used in many organizations, wherein project schedules and budgets are established by an individual, based on that individuals experience. These estimates represent the expected project and resource characteristics for the duration of the project.

The extensions suggested here allow these estimates to be enlarged in the following dimensions :

    The nature of the estimate
    The source of the estimates
    The timing of the estimates

1. *The nature of the estimate.* The model allows project and resource managers to distinguish between desirable, accessible, and utilized estimates as discussed above. The estimated desirable dimension would be used at a fairly high level in the project planning process to outline the hardware, software, people, and support resources that are considered to be desirable for the project. This may list specific pieces of hardware and software which are desirable at certain points in time. It might also be used to list characteristics of the people (such as skills) that would be ideal on the project. The accessible dimension would

191

3-41

5207

then reflect the expected resources that will actually be available to be used. Again this could be at a fairly high level, indicating the resources available, the differences between these and those desirable, and the reasons why the two categories do not agree; reflecting cost constraints, or risk attitudes which have been adopted as part of the project management profile. The utilized category would normally extend to a lower level in terms of the project plan, detailing estimated resources perhaps down to the work package level and short periods of time.

2. *The source of the estimates.* It was suggested above that there are four major possible sources for these estimates; individuals or groups of people, a knowledge base, a database of prior projects, and algorithmic models of the process. Each of these should be supported in a measurement environment, and each has significant implications with respect to the design of such an environment. The current state of the art appears well equipped to support algorithmic models of some parts of the estimation process (for example, estimates of project effort based on one of the many available estimation packages such as COCOMO [Boehm 81], SLIM [Putnam 81], SPQR [Jones 86]). Similarly the tools available in the database environment allow the storage and retrieval of numeric data on past projects. However the storage and searching of large volumes of text data on prior projects, the use of a knowledge base, and the support of group decision support processes are all the subject of current research (see for example, [Bernstein 87], [Nunamaker, et.al. 86], [Barstow 87], [Valett 87]).

The timing of the estimates. In the structure suggested, all estimates may be made before the commencement of the software process and also at any point in time during the process. However there are certain points in time during the process at which estimates are more likely to be updated. These are:

1. at project milestones

2. at manager initiated points in time at which major divergence between estimate and actual is recognized by the manager

3. at system initiated points in time at which the measurement system recognizes a potentially significant divergence between estimate and actual

The third possibility implies that the measurement system is able to intelligently recognize the existence of a problem with respect to the comparison of actual and estimate. This facility is suggested as needed because one of the major management stumbling blocks is generally not concerned with taking action once a problem is identified, but the identification of the problem in the first place. This identification problem occurs because of the volume of data that needs to be processed in order to recognize a potential problem state. It is the measurement environment

which is expert at processing the data volume. It is the manager who is expert at taking corrective action once the problem is highlighted.

Categories four (actual desirable) and five (actual accessible) of the structure exist to provide a feedback and learning dimension to the project database. These values would be determined after the project is complete. And in the comparison of the estimates made at various stages of the process and these two categories, a process is facilitated in which the organization can learn based on the variance of expectations and actual which have occurred in the past projects. As with the estimates, the categories of desirable and accessible are used in order to allow the comparison of "actual ideal" with "actual available" so that an ex-post view of the management of the process can be captured. The question being asked here is; "How could we have handled resources better?" It is a learning mechanism to generate explicit new knowledge for the knowledge and data bases, and also to improve individual and group knowledge.

Category six (actual utilized) will be the most active category within the structure, carrying all of the values associated with the resources of the project. These values will be updated on a regular basis throughout the software process, and will be the source of the triggering process mentioned in the discussion of updates to the estimates.

The data collected during the project should be able to:

1. increase individual and group knowledge
2. improve the knowledge base
3. add to the prior project database, and/or
4. support the algorithm determination process in the individual organization.

In summary, the model proposed is a four dimensional view of resource data. The four views in the data model are:

1. RESOURCE TYPE: which is a mutually exclusive and exhaustive categorization which captures the nature of the resource.

2. INCURRENCE: which is also mutually exclusive and exhaustive describing actual or estimated resources. It carries an additional feedback element to contain the corporate memory explaining the difference between the category values and differences over time.

3. AVAILABILITY: in which each category is a subset of the the higher category, allowing desirable, accessible, and utilized resources. Again feedback is used to explain the differences between categories and over time.

5207

4. USE DESCRIPTORS: which categorizes specific elements in the nature of the resource use. These are the nature of the work done by the resource, the point in time of the work, and the amount of that work.

3.4 USING THE TDC STRUCTURE

3.4.1 At the project level

Discussion so far has applied the proposed 4D structure to resource classification. It is appropriate to also consider using this structure, or a part of it, for the Project Environment Characteristics outlined in section 2 above. In this way the constraints acting on the software process can be identified as applying:

to a particular type of resource,
either estimated or actual
with a stated availability
at a point in time,
concerning a particular type of work

An overall model of the software project is shown in Figure 2. In this figure the meta-entity project is decomposed into a number of tasks or contracts, each task consuming the meta-entity resource and producing the meta-entity product. In the implementation of this model the meta-entities will require many entities to characterize them.
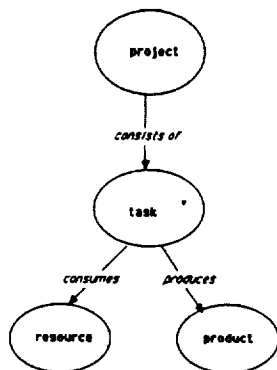

FIGURE 2. AN OVERVIEW OF THE SOFTWARE PROJECT

Thus the project has characteristics, as do the tasks and subtasks, the resources, and the products. Characteristics at all of these levels need to be stored.

Through the storage of the project characteristics, the constraints acting on the product or process determined at any time before or during the project can be tracked for consistency, and any changes noted to facilitate a relationship analysis between the project and the resource occurrence values accumulated during the process.

A simple example of the application of this structure would be where the process organization is changed during the development, say a change toward greater user involvement. This change would be reflected in a difference between the estimated project characteristic and those at the point in time at which the change occurred. This information is then used to explain variances that occur in the process data, such as a changed pattern in staff utilization.

Examples of the data stored at the project level would include:

- the type of project
  e.g.real time, business application
- the project elapsed time
- the total project effort
- the total project cost
- the type of development process
  e.g. evolutionary
- the target computer
- the development computer
- the project deliverables
- the project milestones
- the project risk profile

The application of the TDC model at this level provides a mechanism for storing estimates, accumulating actual values, and facilitating feedback and learning at the level of the project and its development environment.

If we take the project milestones as an example and assume that the milestones apply equally to all resource types, then the model suggests we store:

- *estimated desirable milestones.* This is an "ideal world" view of the project milestones; the dates at which we could deliver if we were not constrained.

- *estimated accessible milestones.* Given the constraints we will be working under, these are the dates at which we could deliver if it were necessary.

- *estimated utilized milestones.* These are the dates at which we expect to deliver, taking into account the dimensions of desirable and accessible.

These three views, in their values and difference, provide a perspective on the risk associated with the project; the smaller the difference between the categories, the higher the risk. More specifically, the difference between estimated desirable and estimated accessible shows the extent to which elapsed time could be changed if the constraints could be modified. For example, if the estimated final desirable milestone were June 30th and the

193

3-43

estimated final accessible milestone was August 30th, the difference of two months measures the estimate of the extent to which the project could be compressed if the restricting constraints could be be removed.

The difference between the estimated accessible and the estimated utilized provides a measure of the available slack in the milestones. This difference is the extent to which the milestones could be compressed, without modifying the project constraints. In the example above, the estimated utilized final milestone might be say November 30th. In this case the difference between accessible and utilized of three months reveals the amount of elapsed time compression that is possible on this project without changing constraints.

In these relationships we see some of the dynamic nature of the project characteristics. This suggests that for the TAME measurement environment, if a change in project characteristics such as the nature of the process occurs, then this should trigger the review of the project milestone and effort values, which will also be reflected at the lower level in the task and resource data values.

In the actual category we need to store the :

- *actual desirable milestones*. As explained above, this category is used for feedback and learning. It carries the values calculated after project completion based on the knowledge gained about the project during its completion. This value is again an "ideal world" value.

- *actual accessible milestones*. This is also a feedback and learning category which says, based on the constraints which did eventuate in the process what milestones could have been achieved?

- *actual utilized milestones*. This category stores the dates of the milestones achieved. Differences between actual and estimated are stored in a feedback facility to provide a mechanism for learning and a mechanism for calculating the actual desirable and accessible at project end.

3.4.2 At the resource level

The description of the use of the TDC structure at the resource level amounts to a process model of resource planning and use in software development. This process can be described as an interacting three-stage process involving the sub-processes of:

1. planning
2. actualization
3. review

The *planning* process establishes and records the resource expectations or estimates before and dur-

ing the software project, and the *actualization* process tracks and records the actual use of resources during the software project. The *review* process compares actuals with estimates for the purposes of modifying the estimates and learning from experience. In this way the *feedback* referred to above provides information for an historic resource database for future planning and estimation. Details of this process model are given in [Jeffery, Basili 87].

Application of the planning and review cycles

In any particular organization, it may be deemed sufficient to use only a part of the planning and review processes outlined here, and therefore only a part of the TDC structure presented in this paper.

For example organizations may not wish to use project reviews, or they may not consider it appropriate to carry out formal contingency planning or risk management. At the simplest level only the estimated utilized and the actual utilized may be used, perhaps providing input to an informal project learning process which occurs at the individual level.

Specifically, it is most likely that in software environments with very little uncertainty (say an implementation of the twentieth slightly different version of a well known system) there may be no need to explicitly consider the desirable or even accessible dimensions of the resource model. If uncertainty is very low, the utilized level of the model may capture all the necessary data. The advantage of the model in this case is that the data excluded is done so in the knowledge that there is no information in those levels not used.

In higher uncertainty environments, the model prompts the estimator to think explicitly of the resource risks and uncertainty of the development process, and to quantify or express that risk as a part of the resource database.

4. VALIDATING THE MODEL

Three significant pieces of work in the literature which provide definitions of the types of data needed to support the measurement of the software process are [Penedo, Stuckle 85], [Tausworthe 79], and [Data & Analysis Center for Software 84, STARS Measurement DID Review].

Penedo and Stuckle (P&S) provide an excellent structure and content of a project database for software engineering environments which can be used here to test whether the model resulting from the top-down methodology employed is able to encapsulate all of the process data suggested by them as needed in a project database. Table 1 lists the entities identified by Penedo and Stuckle and associates the particular model categories which would be used in the model derived here to describe them.

The first aspect which is noticed when mapping the 31 P&S entity types to the TDC model is that the broad structure presented in section 2 above (The Project Environment Characteristics) is an important link between the software process and product. The P&S list contains entities for the project, task, product, and resource categories of Figure 2. In table 1 the P&S entities such as the requirement and risk have been categorized as project characteristics, while entities such as data component, external component, document, interface, product description, product, and software component have been categorized as product instances.

But the focus of this paper is not on the project or the tasks which go together to make up that project. Rather the focus is the resources consumed by those tasks. In this respect we notice that only a subset of the available TDC categories are used in the P&S entities. For example, at the Resource Type level we see instances of all four categories (Hardware, Software, Human, and Support), but at the next level it appears that the P&S model concentrates on actual values. It is difficult to see how the P&S model stores values for estimates, and particularly how the information explaining divergence between estimate and actual can be stored. The same applies to the Availability level of the TDC structure. The P&S model appears to concentrate on the Utilized aspect and does not appear to model the other availability dimensions presented in the TDC structure. This may well be because these dimensions of resource data were considered not to be necessary in the environment of the P&S study.

Table 1. P&S Database Entities in The Model Structure

Penedo & Stuckle Entities	Top Down Model Categories
Accountable Task and Contract	The task and contract are the convergence of process and product and subsets of the project. It is in a contract or task that resources are consumed to produce the product. They are not, therefore, resource entities.
Change Item	This item is generally associated with a product change.
Consumable Purchase	*Support resource, incurrence and availability not specified.
Data Component	Product Entity
Dictionary	*Software resource, or perhaps product entity
Document	Product Entity
Equipment Purchase	*Hardware resource
External Component	*Hardware resource or perhaps Product Entity
Hardware Architecture	*Hardware resource or perhaps product entity
Hardware Component	*Hardware resource or product entity
Interface	Product Entity
Milestone	*Project Entity
Operational Scenario	Product Entity
Person	*Human Resource
Problem Report	*Process as part of feedback or Product entity
Product	Product Entity
Product Description	Product Entity
Requirement	Project Entity
Resource	*Support resource
Risk	*Project Entity
Simulation	Product entity
Software Component	Product Entity
Software Configuration	Product Entity
Software Executable Task	Product Entity
Software Purchase	*Software resource
Test Case	*Software resource and/or product entity
Test Procedure	*Task or project characteristic
Tool	*Software resource
WBS Element	Project Decomposition Entity, may be the same as accountable task and contract

It remains to be seen, of course, whether all of the categories available in the TDC structure are deemed necessary in any particular environment. However, the advantage of such a structure is that exclusion of certain categories of data occurs explicitly rather than implicitly.

The second model suggested as a means of testing the TDC model is that provided by [Tausworthe 79]. In this work the model's entities are not presented in a list form, but are included in text discussion and report forms. For this reason it has been necessary to convert the form to a list of entities. In doing so it is always possible that misconceptions of Tausworthe's ideas may be present. However, even if incomplete, it provides another test of the suitability of the TDC model.

The Tausworthe structure is very much oriented towards a decomposition of the project into tasks and the association of resources with those tasks. Thus the modelling approach used by Tausworthe is somewhat at a tangent to the modelling approach used here since once again our focus is on resources, not the activities which consume those resources. This is not to say, however, that it is not necessary to associate resources with tasks, but that it may be necessary to model resources apart from the tasks that consume them in order to better understand all of the dimensions of resource data.

195

The entities listed here are a partial list derived from the work breakdown structure, the software technical progress report, the software change analysis report, and the software change order of Tausworthe's model. From these sources the following resource data, among others, were identified as necessary to establish a resource database. Only some of the Tausworthe entities have been listed here. This has been done to the extent that is necessary to illustrate the conclusions drawn.

From Table 2 it is clear that the focus of attention in the Tausworthe work is the project and the decomposition of that project into its component parts. Thus we see that the resource data is associated with particular tasks and activities. In viewing the data in this way a structure is provided which is excellent for control purposes, in that it establishes units of accounting which are more easily estimated and controlled. What is not clear from the structure, however, is how questions of desired versus accessible resources can be modelled, nor exactly how actual versus estimated can be compared and conclusions stored for use in later project estimates. It is also difficult to see how the model proposed in the WBS can easily facilitate the analysis of resources consumed on a particular activity type (say inspections), regardless of the project phase in which the inspections were done or the project task in which they were done. Thus questions such as the value to the project of using a particular form of inspection may be difficult to answer because the data model may make this data difficult to isolate.

Perhaps the most detailed resource data collection forms developed so far has been that of the STARS Measurement Data Item Descriptions.

The information which follows in Table 3 was derived from stars Software Development Environment Summary Reports DI-E-SWDESUM, DI-F-RESUM, DI-F-REDET, [06 JULY 1984]. These reports contained information most relevant to the task of validation of the TDC model. The data suggested as necessary by these reports concerned aspects of the project, the process, and the product. In this paper only those aspects concerning the project and the process have been listed. As with the Penedo and the Tausworthe models, the data model implied in the work appears not to have been developed on the basis of a theoretical structure, but rather from a pragmatic evaluation of those data items deemed necessary for project management. In addition, because the data items are
listed in the context of data capture forms, some rearrangement of these items has been carried out in the following data list in order to provide a clearer presentation of these items.

Table 2. Tausworthe Derived Entity List

Tausworthe Entities	Top Down Model Categories
Staff: Staff I.D. Staff Name Staff Phone	Human resource, estimated or actual
Task Activity: Task I.D. Task Activity I.D. Budget $	The dollar value may be a sum of all resources consumed on a task-activity, estimated or actual
Task: Task I.D. Task Name Task Descr Task M'ger Task Budget $, ETC.	The value is a sum of all resources, estimated and/or actual
Software Change Order S/ware ID Change Order # Activity ID Person ID Description Start Date, etc.	The focus is again on the activity. The resources may be any type, estimated or actual.

However, it is clear that the resource data suggested as necessary by Tausworthe are readily modelled in the TDC structure. The importance of the application of the TDC model to the project and task level is highlighted by Tausworthe and also Penedo & Stuckle, so that the association of resource data and project work breakdown structures can be facilitated.

TABLE 3. STARS Measurement Data Items
Descriptions

A. PROJECT NAME

Project Name
Contractor
Contract No.
Start date, Finish Date
Software Level (System, Subsystem, CSCI)
Application Type
Application description
Revision of current project (y/n)
  Revision -version no.
    % of software redeveloped
    Total no. lines of source code
Initial development (y/n)
  if y - Total no. lines source code
    no. of instructions
    no. of data words
System Structure-
  single overlay
  multiple overlay
   (# overlays, avg. size bytes
  independent subsystems
   (# subs, avg.size bytes
  virtual memory system
   (amount of addressable memory, size bytes
Progamming language and % used
Constraints -
  Execution Time, rating
  Main memory size, rating
  Product Complexity, rating
  Database size, rating
  Methodology, rating
  required reliability, rating
  Other, rating
Concurrent Hardware development (y/n)
Operational site development (y/n)
Multiple site development (y/n)
no. of development sites
no. of test sites (if different)
Other Constraints .(text).
cost estimation assumptions made
cost estimation methods used and supporting
  rationale
rationale for discrepencies between current
  estimates and all previous estimates

B. SITE CONFIGURATION INFORMATION

Site ID
Description (development, test)
Computer manufacturer
Model name
Model no.
no. of persons accessing site
no. of input terminals
Terminals in each programmers office (y/n)
Input terminals in central area (y/n)
no. of card readers
no. of printers
no. tape drives
no. disk drives
other peripherals.(specify).
no. documentation sets on hardware/software
  environment available
no. site support personnel
amount of storage in development computer
  main memory real
  main memory virtual
  aux memory

DEVELOPMENT SITE ACCESS

Site I.D.
Access type: % batch
         % interactive
Average job turnaround time
no. hours per day development site available
no. days per week development site available
no. hours per day utilized
no. days per week utilized

TEST SITE ACCESS

Site I.D.
no. hours per day test site available
no. days per week test site available
no. hours per day test site utilized
no. days per week test site utilized

197

3-47

5207

C. PROJECT PHASE INFORMATION
[examples]

requirements

Development system used (y/n)
Documents maintained on the dev. system (y/n)
Methodology (formal spec., functional spec.,
  procedural spec., english spec., none, other)
Tools/Formalisms (requirements analyzer, word
  processor, on-line editor, c.m.t., librarian,
  spec lang, PDL, none, other)
start and finish date
deliverables

design

Development system used (y/n)
Documents developed/maintained on system (y/n)
Methodology (top down, bottom up, hardest
  first, prototyping, iterative enhancement,
  none, other)
Tools/Formalisms (software dev. folders,
  design reviews, walkthru's, flow charts,
  HIPO, etc.)
start and finish date
deliverables

implementation

Development system used (y/n)
Documents maintained on development system (y/n)
Unit testing performed on dev. system (y/n)
Methodology (top down, cpt, prototyping, etc.)
Tools/Formalisms (code reading, pre-compiler,
  dbms, etc)
start and finish date
deliverables

test and integration

Testing performed on development system (y/n)
Documents maintained on system (y/n)
Level of testing performed on dev system
Methodology (spec driven, top down, none, etc)
Tools/Formalisms (......)
start and finish date
deliverables

D. PROJECT PERSONNEL INFORMATION

[these values can be derived from more detailed
 records]

Project Name
Job Classification (supervisor, consultant,
  analyst, programmer, site operator,
  librarian, other)
Avg. no. years application experience
Avg. no. years experience with software
Avg. no. yrs software training
Avg. no. yrs programming language experience
Avg. no. yrs hardware experience
Avg. capability rating

communication

Regular project status meetings (y/n)
How often?
Persons typically in attendance
  (classification, No.)

E. RESOURCE EXPENDITURE ATTRIBUTES

summary level
[these values may be derived]

Project name
total system cost, estimated, actual
total software cost, estimated, actual
total labour cost $, estimated, actual
total software labour cost $, estimated, actual
total labour hours, estimated, actual
total software labour hours, estimated, actual
total staff size, start, finish, estimated,
  actual
total software staff size, start, finish,
  estimated, actual
total computer costs $, estimated, actual
total software computer costs $, estimated,
  actual
total computer hours, estimated, actual
total travel costs $
total material costs $
total miscellaneous costs $

[these may be divided by milestones or activities]

labour costs
[these values may be derived]

labour category id
total hours
no. of people, start, finish
cost $
computer hours
computer costs $

computer costs
[these values may be derived]

no. of computers used
no. of different types of computers
total computer hours

*** for each computer***

computer i.d.
number of hours
total computer costs $
cost of each computer $

task costs
[these values may be derived]

task i.d.
definition
personnel costs
software costs
hardware costs
supplies costs

****for each task****   ****for each task****
****for each labour category****
                               total cost of labour
total hours   total hours of labour
no. of people, start - finish total cost of computer
cost $ total hours of computer
computer hours   total cost of travel
computer cost $ total cost of materials
travel cost $ total cost of miscellaneous

5207

The Table provides data items to describe the project, development and test site configurations and access, project phases, personnel assigned to a project, and resource expenditure summaries. The detail shown here has been selected to highlight the volume of data items which will be necessary in a measurement system.

In terms of the TDC model, the STARS list shows recognition of the need to store resource availability in that the development and test site access data includes an accessible and a utilized dimension. There appears, however, to be no facility for storing the desirable dimension suggested in the TDC model. The STARS list also shows extensive use of the incurrence dimension in section E - Resource Expenditure Attributes, wherein estimated and actual resource use is tracked. The USE DESCRIPTORS of work type, point in time, and resource utilized are also extensively used in the STARS list. It is not possible from the documentation, however, to determine the reasons that the availability dimension was not applied more extensively in the data model (for example accessability of personnel or specific hardware or software items are not modelled). It can be assumed that it was considered to be innappropriate for entities other than site access.

The STARS data list provides considerable support for the theoretical structure provided in the TDC model. It reveals a considered need for the storage of :

1. Project information
2. Resource type information
3. Incurrence information
4. Availability information and
5. Use descriptors

Of considerable significance is the fact that none of the three schemas considered here have suggested data entities or items which cannot be successfully modelled using the TDC structure. It appears that the schemas considered here may be incomplete when compared with the TDC structure, but the reasons for the apparent exclusion of data entities and items are not known, but may be based on purely pragmatic reasons.

5. CONCLUSIONS AND IMPLICATIONS AT THE RESOURCE DATA LEVEL

The model presented here is meant to be general and provide a perspective for project manager and organization in identifying and tracking resources. It should help in better understanding the compromises made in resource allocation. However, it is assumed that any project (or even organization) will work with a subset of this model. For example, one might limit the number of availability views, such as combining desirable and accessible, or track only a subset of the resource categories. The subsetting process provides feedback on what has not been tracked. The actual data collected is driven by the goal/question/metric paradigm based upon the goals set by the project and the organization.

The conclusions to be drawn from this research can be divided into two categories: those concerning the model itself, and those concerning the validation of that model.

In terms of the model itself, the discussion has suggested storage of resource data of a type which has significant storage and access implications; that of numeric and non-numeric project and resource data. It has been assumed in the discussion that the resource database is able to store not only numeric resource values, but also reasons for those values along with the resource environment characteristics.

A system using these suggestions should be able to efficiently search the numeric and non-numeric data in a manner which will eventually enable the system to propose reasons for numeric variances which occur in the database. In this way the system must be able to not only highlight a significant variance, say between an estimated and an actual resource occurrence value, but it should also be able to search the project characteristic database and the numeric and non-numeric resource classification database in order to propose or associate reasons for the variance.

It can be said that the model presented here has four broad implications :

1. It proposes a resource categorization which will allow project database designers to explicitly consider the content of that database against a model of the resource environment. In this way, a particular individual's view of the resource data can be positioned in a context and compared with other external views of the same data. This model should motivate the resource data user to consider the measures that may be beneficial in seeking improvement in the particular process goals.

2. It suggests a project management system's environment which will be able to achieve far more in terms of management support than any known environment available today. It is able to do this because of the extent and dynamic nature of the model of the resource data proposed.

3. It provides a resource categorization which can be used when considering relationships between tasks or contracts and resources. Specifically it provides a focus for the consideration of the resources consumed within a task.

4. It provides assistance when applying the Goal/Question/Metric process paradigm, so that questions which answer the resource purpose of the study are highlighted and the measures appropriate to those questions are suggested.

In terms of the validation of the data model we

5207

C-2

have seen by reference to three published models that the proposed theoretical structure for resource data is able to encompass all that has been suggested as necessary for resource management. Also of significance, is the fact that each of the publications used contains different views of the necessary data and that each one omits certain elements that the other appears to consider of benefit. This is, of course, the norm in comparing different external views in a database design exercise. One advantage of the TDC model is that it is able to act as a data model template, suggesting the data categories which need to be considered when designing a resource data schema. If it is used in this way the data items excluded from the particular resource model instance will have been excluded on the grounds that they are deemed unneccesary in the particular environment, rather than being excluded because the category of data (for example, estimated desirable hardware for testing) was not noticed by the data base designers as necessary.

Thus we can be confident that the theoretical model proposed in the TDC structure can contain all of the project and resource data so far suggested in the literature as necessary in a resource management environment. In addition it appears that there may be project and resource information of use in resource management which has not been included in prior models. The practical need for this additional information has not been justified in this piece of research but is the subject of other current work by the authors.

We have begun to apply the model independent of TAME in a couple of industrial environments and have found it provides a useful framework for planning and tracking resources throughout a project. We have not yet reached the stage where we have been able to evaluate the feedback process, however.

6. REFERENCES

[Barstow 87] D. Barstow, "Artificial Intelligence and Software Engineering," Proc. 9th Intn'l. Conf. on S'ware.Eng. IEEE, Monterey, April, 1987, pp.200-211.

[Basili 85] V.R.Basili, "Quantitative Evaluation of Software Engineering Methodology," Proc. First Pan Pacific Computer Conference, Melbourne, Australia, September, 1985.

[Basili, Freburger 81] V.R.Basili, K.Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," The Journal of Systems and Software, 2, 1981, pp. 47-57.

[Basili, Panlilio-Yap 85] V.R.Basili, N.M.Panlilio-Yap, "Finding Relationships Between Effort and Other Variables in the SEL," Proc. 9th COMPSAC Computer Software & Applications Conference, Chicago, October, 1985, pp. 221-228.

[Basili, Rombach 87] V.R.Basili, H.D.Rombach, "Tailoring the Software Process to Project Goals and Environments," Proc. 9th Intn'l. Conf. on S'ware Eng. Monterey, April, 1987, pp. 345-357.

[Basili, Selby, Phillips 83] V.R.Basili, R.Selby, T.Y.Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," IEEE Trans. on Software Eng. Vol. SE-9 No.6, November, 1983, pp.652-663.

[Basili, Weiss 84] V.R.Basili, D.M.Weiss, "A Methodology for Collecting Valid Software Engineering Data," IEEE Transactions on Software Engineering, SE10,3, November,1984, pp.728-738.

[Bernstein 87] P.A.Bernstein, "Database System Support for Software Engineering," Proc. 9th Intn'l. Conf. on S'ware. Eng., Monterey, April, 1987, pp. 166-178.

[Boehm 81] B.W.Boehm, Software Engineering Economics, Prentice-Hall Englewood Cliffs, New Jersey, 1981.

[Chrysler 78] E.Chrysler, "Some Basic Determinants of Computer Programming Productivity," Comm. of the ACM, 21,6, June, 1978, pp. 472-483.

[Data & Analysis Center for Software 84] STARS Measurement Data Item Descriptions, Data & Analysis Center for Software, RADC/COED, Griffiss AFB, NY. July, 1984.

[Jeffery 87a] D.R.Jeffery, "The Relationship between Team Size. Experience, and Attitudes and Software Development Productivity," Proc. COMPSAC87, Tokyo, October, 1987.

[Jeffery 87b] D.R.Jeffery, "A Software Development Productivity Model for MIS Environments," Jnl. of Systems And Software, June, 1987.

[Jeffery, Basili 87] D.R.Jeffery, V.R.Basili, "Characterizing Resource Data: A Model for Logical Association of Software Data," Technical Report TR-1848, University of Maryland, May 1987, 35pp.

[Jeffery, Lawrence 79] D.R.Jeffery, M.J.Lawrence, "An Inter-Organizational Comparison of Programming Productivity," Proc. 4th Intn'l Conf. on S'ware. Eng. Munich, 1979, pp.369-377.

[Jeffery, Lawrence 85] D.R.Jeffery, M.J.Lawrence, "Managing Programming Productivity," The Journal of Systems & Software, 5,1, February, 1985, pp. 49-58.

[Jones 86] T.C.Jones, SPQR/20 User Guide V1.1, Software Productivity Research Inc. January, 1986.

5207

[Nelson 67] E.A.Nelson, "Management Handbook for the Estimation of Computer Programming Costs," System Development Corporation, Santa Monica, March, 1967.

[Nunamaker, Applegate, Konsynski 86] J.F.Nunamaker, L.M.Applegate, B.R.Konsynski, "Facilitating Group Creativity: Experience with a Group Decision Support System," Proc. 20th Annual Hawaii Intn'l. Conf. on System Sciences, Hawaii, January, 1987, pp.422-430.

[Penedo, Stuckle 85] M.H.Penedo, E.D.Stuckle, "PMDB - A Project Master Database for Software Engineering Environments," Proc. 8th Intn'l. Conf. on S'ware. Eng., London, August, 1985, pp. 150-157.

[Putnam 81] L.H.Putnam, "SLIM A Quantitative Tool for Software Cost and Schedule Estimation," Proc. NBS/IEEE/ACM Software Tool Fair, San Diego, CA, March, 1981, pp. 49-57.

[Sackman, Erikson, Grant 68] H.Sackman, W.J.Erikson, E.E.Grant, "Exploratory Experimental Studies Comparing Online and Offline Programming Performance Comm. of the ACM, 11,1, 1968, pp. 3-11.

[Tausworthe 79] R.C.Tausworthe, Standardized Development of Computer Software: Part II Standards, Prentice-Hall, Englewood Cliffs, New Jersey, 1979.

[Valett 87] J.D.Valett, "The Dynamic Management Information Tool (DYNAMITE): Analysis of the Prototype, Requirements and Operational Scenarios," M.Sc. Thesis University of Maryland, 1987.

[Walston, Felix 77] C.E.Walston, C.P.Felix, "A Method of Programming Measurement and Estimation," IBM Systems Journal, 16,1, 1977, pp.54-73.

[Wolverton 74] R.Wolverton, "The Cost of Developing Large Scale Software," IEEE Transactions on Computers, 23,6, 1974.

201

5207

SECTION 4 – ADA TECHNOLOGY STUDIES

SECTION 4 – ADA TECHNOLOGY STUDIES

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

- "Experiences in the Implementation of a Large Ada Project," S. Godfrey and C. Brophy, Proceedings of the 1988 Washington Ada Symposium, June 1988

- "General Object-Oriented Software Development with Ada: A Life Cycle Approach," E. Seidewitz, Proceedings of the CASE Technology Conference, April 1988

- "Lessons Learned in the Implementation Phase of a Large Ada Project," C. E. Brophy, S. Godfrey, W. W. Agresti, and V. R. Basili, Proceedings of the Washington Ada Technical Conference, March 1988

- "Object-Oriented Programming in Smalltalk and Ada," E. Seidewitz, Proceedings of the 1987 Conference on Object-Oriented Programming Systems, Languages and Applications, October 1987

4-1

EXPERIENCES IN THE IMPLEMENTATION OF A LARGE Ada PROJECT

Sally Godfrey
Code 552
Goddard Space Flight Center
Greenbelt, Md.20771
(301) 286-3600

Carolyn Brophy
Department of Computer Science
University of Maryland
College Park, Md. 20742
(301) 454-8711

BACKGROUND

During the past several years, the Software Engineering Laboratory (SEL) of Goddard Space Flight Center has been conducting an experiment in Ada [6],[8] to determine the cost effectiveness and feasibility of using Ada to develop flight dynamics software and to assess the effect of Ada on the flight dynamics environment. This experiment consists of near parallel developments of a dynamics simulator in both FORTRAN and Ada. A study team consisting of members from the SEL has monitored development progress and has collected data on both projects throughout their development.

Both the Ada and the FORTRAN teams began work in January, 1985, using the same set of requirements and specifications to develop their simulators. The FORTRAN dynamics simulator team completed acceptance testing by June, 1987, after following a development life cycle typical of projects in the flight dynamics environment [5]. The development was carried out on a DEC VAX-11/780 and the completed FORTRAN dynamics simulator consists of about 45,000 source lines of code.

The Ada development began with a period of training [7] in both the Ada language and the methodologies appropriate for Ada [11]. The team was not previously experienced in Ada, although they were more experienced than the FORTRAN team in both the number of years they had programmed (8.6 years compared to 4.8 for the FORTRAN team) and also in the number of languages they knew (7 compared to 3). The Ada team was also experienced in more types of software applications, but only 43% of the Ada team had previous dynamics simulator experience compared to 66% of the FORTRAN team.

Following the training period, the Ada team began a phase of analyzing the requirements and then they began design using an object oriented methodology called GOOD (General Object Oriented Design) which was developed by the team during the training and design phases. More information on GOOD and the lessons learned during the design phase can be found in [2], [4], and [10].

Coding and unit testing began in April, 1986, on a DEC VAX 8600 and continued through June 1987. The Ada project has completed system testing and consists of approximately 135,000 source lines of code[1]. This paper will describe some of the similarities and differences of the two projects and will discuss some of the interesting lessons learned during the code/unit test and integration phases of this project.

INFORMATION COLLECTION

The information presented in this paper was collected by using the following four methods: 1) Collection of SEL forms 2) Interviews 3) Observation of development 4) Code analysis. The SEL forms solicit such information as a detailed breakdown of the hours spent by programmers, managers, and support staff on a project and detailed information on changes and errors which occurred during the development. During the course of the project, over 2000 forms were collected; about 625 of these documented errors and changes.

---

1. A source line of code is defined to be any 80 byte record of code including commentary, blank lines and executable code.

5207

Each member of the Ada team (11 total) was interviewed individually to gain some insight into the experiences he or she had during implementation. Team members were asked questions concerning ease or difficulty of implementing features, unit testing, integration, correcting errors, using tools, etc. Questions concentrated on an individual's particular area of work, but general subjective questions were asked of the entire team. Observation of the development was accomplished by attending reviews and regular implementation meetings held by the team. These regular implementation meetings were actual working meetings in which team members discussed progress, solved implementation problems, clarified interfaces, shared knowledge, and planned implementation strategies. In addition, much information was gained through informal conversations with the team on implementation progress. Information received through code analysis was actually collected two ways. First, the code was examined to tabulate such attributes as number of modules, number of lines of code, number of comments, etc. Second, another Ada team, in the process of Ada training, performed code reading on parts of the dynamics simulator code as a training exercise and they provided their comments on the code.

The remainder of this paper will concentrate on some interesting comparisons between the FORTRAN and the Ada projects and some of the major lessons learned during the implementation phase of the Ada project.

1. FORTRAN/Ada PROJECT COMPARISONS

Several factors need to be considered when trying to directly compare metrics from the FORTRAN project and those from the Ada project. First, the FORTRAN project was considered to be the "real" operational version of the dynamics simulator being developed, and as such, it was necessary for that project to meet the schedules imposed by an impending launch date. The Ada team, on the other hand, was allowed a more relaxed schedule for development which included adequate training time, time to experiment with design methodologies, and finally, time to recode or enhance if "better" methods occurred to the developers. One result of this extra time was the development of a much more sophisticated user-interface for the Ada project.

Second, this general type of dynamics simulator was a very well-known application for the FORTRAN team since similar simulators have been built repeatedly in this environment. Thus, the general design of the FORTRAN simulator was reused from previous designs and was known to be a very satisfactory design for the application. In addition to the design, much of the code was reusable--about 36%. The Ada team developed a new design [1] which they felt was more suitable for Ada and which they felt more accurately represented the actual physical system they were trying to simulate. While this design may be a better physical representation of the problem, it did not have the advantage of previous use to refine and correct any possible problems. No Ada code was available for reuse but several FORTRAN routines were used by the Ada team. These comprised only about 2% of the code.

Keeping in mind these differences in the actual projects, we will discuss some interesting FORTRAN/Ada comparisons.

1.1 Size of Ada project is larger than FORTRAN project.

As mentioned in the background section, a simple count of the number of lines of code, including every line of any type as a line, yields a count of 135,000 source lines of code for the Ada project and a count of 45,500 source lines of code for the FORTRAN project. These figures are really a little misleading, since the Ada line count includes 23,000 lines of blank lines which are inserted for readability. Also, the Ada count includes 49,000 lines of comments compared to 19,500 lines of comments in the FORTRAN count. When the number of executable lines of code are compared, we find that the Ada project has 63,000 lines of executable code compared to 25,500 for the FORTRAN project.

In these particular projects, there were other reasons why the Ada project was larger. As we mentioned earlier, the Ada project was not constrained by schedule pressure and so they developed a system with more functionality--a system with more of the "nice to have, but not required" features. Naturally this increased the size of the system. To some extent, the Ada language itself was a driving factor for the size difference, since it requires more code to write such constructs as package specifications, declarations, etc. In

addition, the Ada team used a style guide [3] that required certain constructs to be spread over several lines of code for readability.

Another interesting way to compare the size of the two projects is to examine the size of the load modules for each one. This also shows the Ada system to be larger-occupying 2300 512-byte blocks, compared to 953 512-byte blocks for the FORTRAN load module.

1.2 Project cost is similar for the two implementations.

One of the problems with trying to compute productivity is that there are many ways to compute it. Usually, in the Software Engineering Laboratory, the calculation is made by taking the total number of source lines of code developed and dividing by the number of hours spent on the project. The number of hours is carefully recorded on forms weekly and includes the hours spent on all phases of the project beginning with requirements analysis and ending with the completion of acceptance testing. In order to compare the FORTRAN and Ada projects, the calculations were made using the number of hours spent on each project from requirements analysis to the completion of system testing since acceptance testing has not yet been completed on the Ada system. As we see in figure 1, using the total number of source lines of code (SLOC) for each project, we get a productivity of 3.8 SLOC/hr. for the

FORTRAN project and a productivity of 6.: SLOC/hr. for the Ada project. Rememberin(that the Ada code included many blank line: of code that were not included in the FORTRAN line count, we recomputed the Ada figure, excluding the blank lines and got a productivity of 5.2 SLOC/hr. When we considered the effort required just to develop new lines of code and not the reusable code, the figures are 2.7 SLOC/hr. for FORTRAN and 6.1 SLOC/hr. for Ada with blanks and 5.0 SLOC/hr. without blanks. This would seem to imply that Ada is more productive, but we must remember that it took many more lines of code to develop the Ada system and that the style guide caused many Ada constructs to be spread over several lines.

Let's look at the figures when we consider only executable lines of code. Using only the number of lines of code which are executable, we got a productivity figure of 2.14 SLOC/hr. for the FORTRAN project and 2.8 SLOC/hr. for the Ada project. When we considered that many of the Ada constructs use more than one line, we looked at the number of executable statements (or semicolons) in the Ada project and recomputed productivity. Similarly for the FORTRAN, we counted statements and their continuations as one executable statement. Now we get a productivity of 1.85 SLOC/hr. for the FORTRAN project and .96 SLOC/hr. for the Ada project. Looking at the number of executable new statements in the FORTRAN yields a figure of 1.2 SLOC/hr. compared to .95 SLOC/hr. for the Ada project. These calculations would make FORTRAN look more productive.

FORTRAN		Ada	
Lines of Code Used for Computation	Productivity	Lines of Code Used for Computation	Productivity
Total lines of code	3.8 SLOC/hr	Total lines of Code	6.17 SLOC/hr
Total lines of code excluding blanks	3.8 SLOC/hr	Total lines of code excluding blanks	5.12 SLOC/hr
Executable lines of code	2.14 SLOC/hr	Executable lines of code	2.8 SLOC/hr
New lines of code	2.7 SLOC/hr	New lines of code	6.08 SLOC/hr
New lines of code excluding blanks	2.7 SLOC/hr	New lines of code excluding blanks	5.03 SLOC/hr
Executable statements	1.85 SLOC/hr	Executable statements	0.96 SLOC/hr
Executable "new" statements	1.2 SLOC/hr	Executable "new" statements	0.95 SLOC/hr

Figure 1: Productivity Comparisons

4-4

5207

Perhaps a better way of viewing the productivity problem is to examine it from the standpoint of cost to produce the product. The total cost of the FORTRAN project from requirements analysis through acceptance testing was about 8.5 man-years of effort. The Ada project cost, using actual figures from requirements analysis through system testing and estimating the acceptance testing cost, is around 12 man-years of effort. When we take into consideration the percentage of reused code in the FORTRAN project and assume all the code generated was new code, it would have taken about 11.5 man-years of effort to develop the FORTRAN system. This makes the cost of developing the two systems roughly the same, especially when we consider that the Ada project was a "first-time" project and that the Ada project had slightly more functionality than the FORTRAN.

1.3 Error types found in both projects show similar profiles.

Detailed information was kept on the types of errors found in both projects and based on 104 forms collected for the FORTRAN project and 174 forms collected for the Ada project, the error types show a similar profile. Figure 2 shows the distribution of error types for each project.

Error Type[a]	FORTRAN[b]	Ada[c]
	%	%
Computational	12	9
Initialization	15	16
Data Value or Structure	24	28
Logic/Control Structure	16	19
Internal Interface	29	22
External Interface	4	6

[a]There may be more than one error reported on a form.

[b]104 forms

[c]174 forms

Figure 2: Error Profile

An example of a computational error might be an error in a mathematical expression. An error like using the wrong variable would have been classified as data value or structure error. Internal interface errors refer to errors in module to module communication, while external interface errors refer to errors in module to external communications.

Perhaps one result here that is suprising is that the team expected to have fewer internal interface errors with Ada, but the percentage is not significantly different from the FORTRAN. When the detailed information on the Ada errors was examined, we learned that many of the errors classified as internal interface errors were caused by a type change of some sort. For example, a variable may have been classified as one type in one portion of the code and a different type in another, or the original type chosen for a variable might not have been suitable. Another common reason that internal interfaces were changed was that a new function was added to the module which required an interface change. Also, in some cases, a developer would find he needed another variable from some other module which he did not originally think he needed.

1.4 The percentage of "very easy to find" errors was less in the Ada project than the FORTRAN project.

Detailed information was captured on the effort required to isolate errors .The error levels were categorized a) very easy or less than one hour b) easy or one hour to one day c) hard or one to three days d) very hard or more than three days. The FORTRAN team found that 81% of their errors were in the "very easy" to isolate category. In comparison, the Ada team found only 59% of their errors in that category. There are several possible explanations for this. First, many of the errors found by the FORTRAN team were types of errors which would have been identified by a more rigorous compiler such as the Ada compiler. Throughout the project, the Ada team felt that the compiler was one of the most useful tools because it was able to pinpoint many errors at the early stage of compilation. Another possible explanation for the difference in effort to locate errors is the difference in experience of the teams with the language. The Ada team was not

experienced in Ada and did not feel they had the same intuition as the FORTRAN team did to aid in isolating errors.

2. MAJOR LESSONS LEARNED DURING IMPLEMENTATION OF THE Ada PROJECT

2.1 A flat structure usually has more advantages than a nested structure. Thus, nesting should be used sparingly.

The object oriented design used by the team [9] seemed to promote a nested structure for information hiding purposes. While the nesting was not explicitly specified in the design, it seemed to be a natural manifestation of the object oriented design--so the parts of an object or a package would be included inside that package instead of being called in from the outside. The team felt that they were implementing nesting conservatively, and indeed, one view of the system shows that it has 124 packages of which 55 are library units. However, the nesting in the system was extensive--many levels deep in some places.

This amount of nesting caused many problems for the Ada developers. First, nesting increased the amount of recompilation necessary during implementation and testing. Many more units had to be recompiled when changes were made to the system since Ada assumes dependencies between nested objects or procedures even when there are none. Since compilation is a lengthy process, this slowed down the development process. Much unneccessary recompilation could have been avoided by the use of more library units.

Second, nesting increased the difficulty of unit testing. In fact, the greater the level of nesting, the more difficult the unit testing was. The lower level units were not in the scope of the test driver, and a debugger was necessary to "see" into these lower level units. For the purposes of unit testing in FORTRAN, a unit is defined as a subprogram. When this same definition was applied to the Ada, unit testing difficulties arose since many of these units could not be tested in isolation. Instead, it was necessary to integrate units which fit logically together, usually integrating up to the package level, before testing was done. Nesting also increased the difficulty

of tracing problems since it is hard t identify the calling module of a neste unit.

2.2 A high degree of nesting was foun to be an impediment for reuse.

Perhaps the major advantage of usin library units instead of nested units i that their use increases the potential o reusability. When nesting is used, the siz of the compilation units, the componen sizes and the file sizes all tend to b larger. Thus when these larger units ar examined for potential reuse, it is muc more likely that only a portion of the larg unit will actually have the code whic performs the needed function for the nei system. Then it becomes necessary to unnesi the code before reuse is possible. Thi: unnesting is very labor intensive.

Another similar Ada project presentl under development in the SEL has examinec this project's code for reuse and has founc that it could use as much as 40% of the original code. However, it was necessary tc unnest all of this code before reuse. This use of library units would have enabled the second project to reuse the code directly.

2.3 "Call-through" units are not an efficient way to implement an object-oriented design.

"Call-throughs" are procedures whose only function is to call another routine. These were used to group appropriate modules exactly as they were represented in the design so that a physical module of code was created for every object in the design. Thus, when objects were nested inside objects, a "call-through" was used to get to the inner object. Implementation of "call-through" units could be accomplished using either nested or library units. This practice resulted in additional code which increased the system size and testing complexity. This unneccessary code could have been eliminated if some of the objects in the design were left as logical objects, rather than coding every object in the design to preserve the exact design structure.

5207

2.4 An abstract data type analysis should be incorporated into the design process to control types.

Since the Ada team was not previously experienced in Ada, it took time to get accustomed to the strong typing of Ada. The tendency was to create too many types. A type would be created with a strict range for a particular portion of the application. Then other areas of the application would need a similar type, but the original one would be too restrictive. So another type was created, along with a corresponding set of operations. Some of the difficulty with this method of typing began to emerge during critical design, where interface problems developed due to typing differences.

Multiple types also increased the difficulty of testing modules. Test drivers needed to be larger to handle multiple types and were often coded as large "case" statements in order to provide a testing capability for each type.

A recommendation for future Ada developments is to incorporate an abstract data type analysis into the design process to control the generation of types. A more general new type would be defined, then many subtypes of that type could be used in various sections of the application. This type analysis would provide the following advantages: 1) operations would be reused, 2) there would be fewer main types to manage, and 3) families of types would be developed that would inherit properties from each other.

SUMMARY

In spite of a lack of experience in Ada at the beginning of the project, the Ada team was able to develop a very suitable dynamics simulator in Ada which meets the requirements originally developed for the FORTRAN development effort. The overall cost of the projects appears to be similar and early indications of reuse potential in the Ada project are very encouraging. Most of the problems encountered by the Ada team are surmountable. Many are either caused by a lack of experience with Ada or an immaturity of the tools. Both of these problems will be resolved in time.

There are still many unanswered questions to be considered on this project-- for example, nothing at all has been mentioned about maintainability, reliabilit or performance. It is still too early t look at these results on this project, but research efforts are continuing on thi project and several other Ada project in th SEL. Hopefully, these efforts will provid even more answers about the use Ada in th future.

REFERENCES

1. Agresti, W., Church, V., Card, D., et al. "Designing with Ada for Satellit Simulation: A Case Study," *Proceeding of 1st Annual Symposium on Ad Applications for NASA Space Station* Houston, Texas, June 1986.

2. Brophy, C. and Godfrey, S.,et. al "Lessons Learned in the Implementatio: Phase of a Large Ada Project, *Proceedings of the Washington Ad. Technical Conference*, March 1988.

3. Goddard Space Flight Center Ada User': Group. *Ada Style Guide (Version 1.1)* Goddard Space Flight Center document SEL-87-002, June 1987.

4. Godfrey, S., and Brophy, C. *Assessin(the Ada Design Process and It: Implications: A Case Study*, Goddar(Space Flight Center document, SEL-87- 004, July 1987.

5. McGarry, F., Page, G., et. al. *Recommended Approach to Softwar(Development*, Goddard Space Flight Center document, SEL-81-205, April 1983.

6. McGarry, F., and Nelson, R. *An Experiment with Ada-The GRO Dynamics Simulator*, Goddard Space Flight Center, April 1985.

7. Murphy, R. and Stark, M. *Ada Traininç Evaluation and Recommendations*, Goddard Spoace Flight Center, October 1985.

8. Nelson, R. "NASA Ada Experiment-- Attitude Dynamics Simulator," *Proceedings of Washington Adɛ Symposium*, March, 1986.

9. Seidewitz, E. and Stark, M. "Towards a General Object Oriented Software Development Methodology," *Proceedings of 1st International Conference on Ada Applications for the Space Station*, June 1986.

10. Seidewitz, E. and Stark, M. *General Object Oriented Software Development*, Goddard Space Flight Center document, SEL-86-002, August 1986.

11. Stark, M. and Seidewitz, E. "Towards a General Object Oriented Ada Lifecycle," *Proceedings of Joint Conference on Ada Tech/Washington Ada Symposium*, March 1986.

5207

GENERAL OBJECT-ORIENTED SOFTWARE DEVELOPMENT WITH ADA: A LIFE CYCLE APPROACH
CASE Technology Conference
April 1988

Ed Seidewitz
Code 554 / Flight Dynamics Analysis Branch

Goddard Space Flight Center
Greenbelt MD 20771
(301) 286-7631

Abstract

The effective use of Ada requires the adoption of modern software-engineering techniques such as object-oriented methodologies. A Goddard Space Flight Center Software Engineering Laboratory Ada pilot project has provided an opportunity for studying object-oriented design in Ada. The project involves the development of a simulation system in Ada in parallel with a similar FORTRAN development. As part of the project, the Ada development team trained and evaluated object-oriented and process-oriented design methodologies for Ada.

In *object-oriented* software engineering, the software developer attempts to model entities in the problem domain and how they interact. Most previous work on object-oriented methods has concentrated on using object-oriented ideas in software design and implementation. However, we have also found that object-oriented concepts can be used advantageously throughout the entire Ada software life-cycle. This paper provides a distillation of our experiences with object-oriented software development. It considers the use of entity-relationship and process/data-flow techniques for an object-oriented specification which leads smoothly into our design and implementation methods, as well as an object-oriented approach to reusability in Ada.

1. Introduction

Increased productivity and reliability from using Ada must come from innovative application of the non-traditional features of the language. However, past experience has shown that traditional development methodologies result in Ada systems that "look like a FORTRAN design" (see, for example, [Basili 85]). *Object-oriented* techniques provide an alternative approach to effective use of Ada. As the name indicates, the primary modules of an object-oriented design are *objects* rather than traditional functional procedures. Whereas a procedure models an action, an object models some *entity* in the problem domain, encapsulating both data about that entity and operations on that data. Ada is especially suited to this type of design because its package facility directly supports the construction of objects.

The Goddard Space Flight Center Software Engineering Laboratory is currently involved in an Ada pilot project to develop a system of about 60,000 lines (20,000 statements) [Nelson 86, McGarry 88]. This project has provided an opportunity to explore object-oriented software development methods for Ada. The pilot system, known as "GRODY", is an attitude dynamics simulator for the Gamma Ray Observatory (GRO) spacecraft and is based on the same requirements as a FORTRAN system being developed in parallel.

The GRODY team was initially trained both in the Ada language and in Ada-oriented design methodologies. The team specifically studied the methodology promoted by Grady Booch [Booch 83] and the PAMELA™ methodology of George Cherry [Cherry 85]. Following this, during a training exercise, the team also began synthesizing a more general approach to object-oriented design. At an early stage of the GRODY development effort, the team produced high-level designs for GRODY using each of these methodologies. Section 2 summarizes the comparison of methodologies made by the GRODY team.

PAMELA is a registered trademark of George W. Cherry.

1

5207

Unfortunately, the system requirements given to our team were highly biased by past FORTRAN designs and implementations of similar systems. Therefore we began by recasting the requirements in a more language-independent way using the "Composite Specification Model" [Agresti 84, Agresti 87]. This method involves the use of state transition and entity-relationship techniques as well as more traditional data flow diagrams. We then designed the system to meet this specification, using object-oriented principles. The resulting design is, we believe, an improvement over the previous FORTRAN designs [Agresti 86]. The system is currently in final system testing.

Previous work by the present authors has concentrated on using object-oriented ideas in software design and implementation. This work resulted in a design method which synthesizes the best methods studied during the GRODY project [Seidewitz 86a, Seidewitz 86b]. However, we have found that object-oriented concepts can be used advantageously throughout the entire Ada software life-cycle [Stark 87]. Section 3 provides a distillation of our experience with GRODY and other Ada projects into an evolving life-cycle methodology.

2. Comparison of Methodologies

This section presents a comparison of design approaches to the GRO dynamics simulator, including the traditional functional approach used for the FORTRAN version, the Booch methodology, PAMELA and the general methodology developed by the team itself. It should be noted that the GRODY team was trained in the Booch and PAMELA methodologies in early 1985. Since then, both methodologies have evolved considerably, in many cases addressing in different ways the very issues that led us to develop our methodology. Nevertheless, as background motivation for the direction taken by the GRODY team, the comparison in this section is in terms of the 1985 versions of the methodologies.

2.1 Functional Design

The design of the FORTRAN version of the simulator is functionally-oriented. This design has a strong heritage in previous simulator and ground support systems. It consists of three major subsystems which interact as shown in figure 1. The "TRUTH MODEL" subsystem includes models of the spacecraft hardware, the external environment and the attitude dynamics; that is, the "real world" as opposed to the spacecraft control system. The SIMULATION CONTROL subsystem alternatively activates the SPACECRAFT CONTROL and TRUTH MODEL subsystems in a cyclic fashion. Each subsystem consists of a single *driver* subroutine which calls on a hierarchy of lower-level subroutines to perform the functions of the subsystem when activated by SIMULATION CONTROL. Data flow between subsystems, as well as system parameterization, is entirely though a set of global COMMON areas.
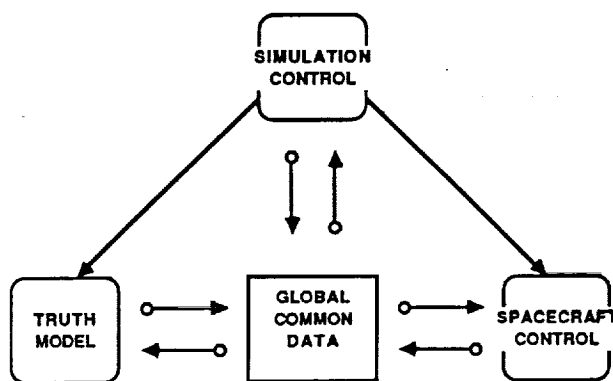
FIGURE 1 *FORTRAN Simulator Functional Design*

The strengths of this functional design lay in its relatively simple structure and direct implementation in FORTRAN. However, its main drawback is the complete lack of encapsulation of global data. The only restrictions on which code may access which global data are enforced by programmer discipline. This can lead, intentionally or not, to illicit corruption of global data by code in one part of the system which is unexpected by another part of the system. Further, most simulation parameters are hard-coded into the global common area, making the user interface for the system hard to modify and impossible to generalize.

2

5207

General Object-Oriented Software Development with Ada

2.2 Booch's Methodology

Grady Booch is, perhaps, the most influential advocate of object-oriented design in the Ada community [Booch 86b, Booch 87]. As learned by the GRODY team, Booch's methodology derives a design from a textual specification or informal design [Booch 83], an approach adopted from Abbott [Abbott 83]. The technique is to underline all the nouns and verbs in the specification. The objects in the design derive from the nouns; object operations derive from the verbs. Obviously, some judgment must be used to disregard irrelevant nouns and verbs and to translate the remaining concepts into design objects. Once the objects have been identified, the design can then be represented diagrammatically using a notation which shows the dependencies between Ada packages and tasks which implement the objects. Figure 2 shows such a diagrammatic top-level design for GRODY.

FIGURE 2 *Object-Oriented Simulator Design (Booch Methodology)*

The Booch design methodology contains all the basic framework of the object-oriented approach. However, application of this methodology to GRODY indicated that it was not readily applicable to sizable systems. The team found the graphical notation clear but not detailed or rigorous enough. Further, Booch gives no explicit method for diagramming a hierarchical decomposition of objects, which is needed for any sizable system. Booch's notation does not, therefore, seem to be a complete design notation.

Note, however, that in more recent work Booch has extended the scope of the notation to address some of these shortcomings [Booch 87].

A second difficulty of Booch's methodology is in the technique for deriving the design from the specification text. This works well when the specification can be written concisely in a few para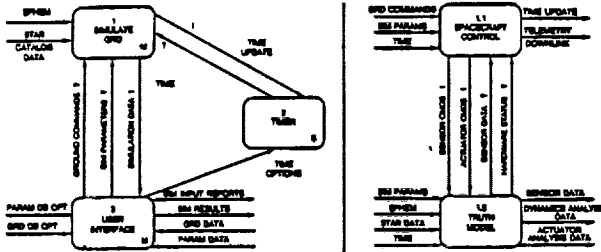graphs. However, when the system requirements are large, as with GRODY, this can be difficult. In addition, any attempt to use such a technique directly on a requirements document such as ours is doomed to failure due to the sheer size and complexity of the document. Realizing such drawbacks, Booch no longer advocates the use of this textual method, which was never actually intended for large systems development [Booch 86b]. Instead, he derives an object-oriented design from a data flow diagram based specification [Booch 86a, Booch 87]. However, from the published examples it is still unclear how to systematically apply this method to realistic systems.

2.3 PAMELA

The second methodology considered by the GRODY team was the Process Abstraction Method for Embedded Large Applications (PAMELA) developed by George Cherry [Cherry 85, Cherry 86]. PAMELA is oriented toward real-time and embedded systems. PAMELA is *process-oriented*, so a PAMELA design consists of a set of interacting *concurrent processes*. A well designed *process* is effectively a concurrent object, thus PAMELA is object-oriented in a general way.

PAMELA uses a powerful graphical notation without many of the drawbacks found in Booch's notation [Cherry 86]. During the PAMELA design processes, the designer successively decomposes processes into concurrent subprocesses until he reaches the level of primitive *single-thread* processes. The GRODY team found that PAMELA provides fairly explicit heuristics for constructing good processes. The designer uses these hints to construct the top-level processes from the system specification. The designer then recursively decomposes each non-primitive processes until only primitive processes remain. The primitive processes can then be coded as Ada tasks with a single thread of control. Non-primitive processes are simply packages of lower level processes and thus contain multiple threads of control. Figure 3 shows the top levels of a PAMELA design for GRODY.

3

update of PAMELA known as "PAMELA 2" which is now explicitly object-oriented [Cherry 88]. In fact, PAMELA now stands for "Pictorial Ada Method for Every Large Application." It is still to early, however, to evaluate the generality of PAMELA 2 as an object-oriented methodology.

2.3 General Object-Oriented Development

As a result of the above experiences, the GRODY team developed its own object-oriented methodology which attempts to capture the best points of the object-oriented approaches studied by the team as well as traditional structured methodologies [Seidewitz 86a, Seidewitz 86b, Stark 87]. It is designed to be quite general, giving the designer the flexibility to explore design alternatives easily. It is also based on principles that guide the designer in constructing good object-oriented designs. This methodology was used to develop the complete detailed design for GRODY.

FIGURE 3 *PAMELA Simulator Design*

PAMELA's heuristics can be very effective when designing a real-time system that is heavily driven by external asynchronous actions. In other cases, however, they require considerable interpretation to be applicable. Although parts of GRODY might conceptually be concurrent (because GRODY simulates actions that happen in parallel in the real world), there is no requirement for concurrency in the simulation of these actions because GRODY does not have to interface with any active external entity (except the user). In addition, since GRODY runs on a sequential machine, the overhead of Ada tasking and rendezvous could greatly degrade the time performance of the system. Thus, one interpretation of PAMELA's principles might leave very large sections of GRODY as primitive single-thread processes, with only a few concurrent objects in the entire design. To proceed further in the decomposition, the designer has to rely more on intuition about what makes a good object and rely less on the methodology.

In fact, at the time that the GRODY team was using PAMELA, it provided no support for the decomposition and design of anything below the level of the primitive process, an Ada task [Cherry 85]. Since then, Cherry has added several concepts to the methodology, including the use of abstract data types [Cherry 86]. Recently he has introduced a major

This general object-oriented development ("GOOD") methodology is based on general principles of abstraction, information hiding and design hierarchy discussed in the next section. These principles are less explicit than Booch's methodology or PAMELA, but they do provide a firm paradigm for generating and evaluating an object-oriented design. Indeed, as mentioned above, the team found the Booch and PAMELA design construction techniques restrictive, often necessitating the designer to rely on intuition for object-oriented design. The GOOD methodology is an attempt to codify this intuition into a basic set of principles that provide guidance while leaving the designer the flexibility to explore various design approaches.

In addition, we have also considered, independently of Booch, the transition from structured analysis [DeMarco 79] to object-oriented design in the context of the GOOD methodology, developing a technique known as *abstraction analysis* [Seidewitz 86a, Seidewitz 86b]. This technique is analogous to transform and transaction analysis used in structured design [Yourdon 78]. However, proceeding into object-oriented design from a structured analysis, by whatever means, requires an "extraction" of problem domain entities from traditional data flow diagrams. From an object-oriented viewpoint, it seems appropriate to instead *begin* a specification effort by identifying the entities in a problem domain and their interrelationships. Study is continuing on including

4

such object-oriented system specification techniques in the GOOD methodology and on applying object-oriented principles throughout the Ada life cycle [Stark 87]. Section 3 will discuss this in more detail.

Figure 4 shows the actual design of the main part of GRODY. The *object diagram* notation [Seidewitz 86b] used in figure 4 shows the dependencies between the various objects which make up a system design, in a manner similar to Booch's diagrams. However, the object diagram notation also explicitly includes the idea of leveled composition of objects, like the PAMELA process graph notation. Moreover, as will be discussed in more detail in section 3, the designer may use object diagrams to express the design from the highest levels all the way down to the procedural level. (This capability has also been added to PAMELA 2 [Cherry 88].)

Since GRODY was derived from the same basic requirements as the FORTRAN design, there are similarities in the designs of the two systems. However, there are also some fundamental differences in the GRODY design that can be traced to the object-oriented methodology. For example, in GRODY the TRUTH MODEL is effectively passive, with the SPACECRAFT CONTROL calling on operations as needed to obtain sensor data and activate actuators. All sensor and command data is passed using these operations. This design approach was encouraged by viewing the TRUTH MODEL as an object with multiple operations rather than as a functional subsystem with a single driver.

The simulation timing of GRODY is also different from the FORTRAN design. The object-oriented methodology led to consideration of a "TIMER" object in GRODY which provides an abstraction of the simulation time. This utility object provides a common time reference for the SPACECRAFT CONTROL and TRUTH MODEL separate from the SIMULATION CONTROL loop. Unlike the FORTRAN design, in GRODY the "cycle times" of the SPACECRAFT CONTROL and TRUTH MODEL are not the same. The GRODY team chose to faithfully model, in the SPACECRAFT CONTROL abstraction, the timing of the actual spacecraft control software, which is not under user control. However, GRODY allows the simulation user to set the cycle time for the TRUTH MODEL over a fairly wide range, to allow the user to trade-off speed and accuracy as desired.
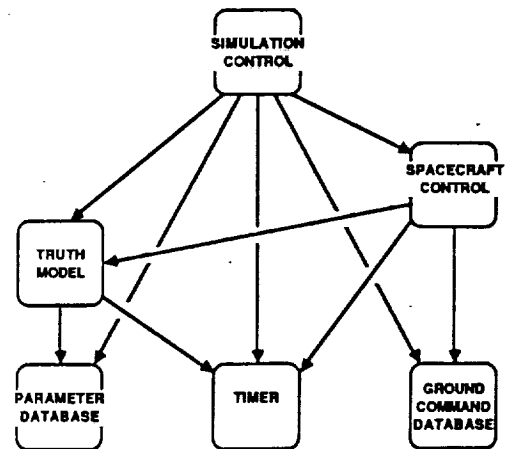

FIGURE 4 *Object-Oriented Simulator Design (GOOD Methodology)*

Finally, the PARAMETER DATABASE and GROUND COMMAND DATABASE objects encapsulate user settable parameters for the simulation. Similar data is contained in COMMON blocks in the FORTRAN design. This encapsulation of "global" data is typical of object-oriented designs. It provides both increased protection of the data encapsulated and increased opportunity for reuse. For example, the simulation parameters in the FORTRAN design are COMMON block parameters which must be hard-coded into the user interface code. (For simplicity the user interface modules have not been included in figure 4.) In the GRODY design, simulation parameters are identified by enumeration constants, which allows the user interface displays to be parameterized by external data files. This should greatly increase the reusability of the user interface.

The differences discussed above could probably have been incorporated into the FORTRAN design. However, it was largely the influence of the object-oriented approach which lead to their consideration for GRODY when they had not been considered in several previous designs of simulators for FORTRAN. Considerations of encapsulation and reusability indicate that the GRODY design may be "better" than the FORTRAN design. This is, of course, the goal of object-oriented methods. However, the true test of the merits of the GRODY design will only come from continuing studies of the comparative maintainability of the FORTRAN and Ada simulators.

5

In terms of the methodology itself, the team found the object diagram notation extremely useful for discussing the design during development. Further, the notation provided complete documentation of the design and was tailored specifically towards Ada. This made the transition to coding very smooth, and allowed the documentation to be readily updated as coding proceeded. By the end of coding, there were no major changes in the design and most changes that did occur were additions rather than alterations.

The object diagram notation evolved considerably during the GRODY project in response to continuing experience with its use. The lack of a specific methodology at the start of the GRODY project was a problem for the team, as was the continuing evolution of the methodology over the duration of the project. Further, the fact that managers were not familiar with the new methodology made the use of object diagrams difficult at reviews. Another problem was that the detail of the object diagrams and the emphasis on keeping the documentation up-to-date required a great deal of effort to maintain a rather large design notebook. The team clearly saw the great need for automated tools to support the methodology in this area. Consideration has also been given to extend the object diagram notation to better cover such topics as generics, abstract data types and large system components.

3. The GOOD Methodology

Section 2 described the background motivation of the GRODY team in developing the GOOD methodology and applying it to the full GRODY design. The experience with the Composite Specification Model and object-oriented design on GRODY, as well as experience on other Ada projects, has led to the continuing evolution of a comprehensive, integrated, object-oriented approach to software development, encompassing all phases of the software life cycle. This section provides an overview of the current GOOD life cycle approach.

3.1 Entities and Relationships

The modules of an object-oriented design are intended to primarily represent problem domain *entities*. From an object-oriented viewpoint, it seems appropriate to begin a software specification effort by identifying the entities in a problem domain and their interrelationships. Entity-relationships and data flow

techniques can then complement each other, the former delineating the static structure problem domain and the latter defining the dynamic function of a system. This is similar to the "contextual" and "functional" views of the Composite Specification Model [Agresti 84, Agresti 87]. A close relation to the specification approach discussed here is described in some detail in [Bailin 88].

An *entity* is some individual item of interest in the problem domain. For example, consider the specification of GRODY. Several problem domain entities immediately come to mind: the spacecraft structure, sensors and thrusters on the spacecraft, the environment, etc. An entity is described in terms of the *relationships* into which it enters other objects. A spacecraft might be in a certain orientation, have certain thrusters, etc. Entities can also have *attributes*, such as spacecraft mass, which are data items describing the intrinsic properties of the entity.

To model the structure of the problem domain requires the identification of *entity types* which are groups of entities with the same types of attributes and relationships. For example, we may define a SPACECRAFT STRUCTURE entity type with SPACECRAFT MASS and DRAG COEFFICIENT attributes. All SPACECRAFT STRUCTURE entities have these attributes, but different individual entities have different specific *values* for the attributes.

A problem domain model must also include a specification of all possible relationships between various types of entities. These relationships may themselves have attributes and enter into other relationships. For example, the ATTITUDE STATE of a spacecraft describes its current orientation relative to inertial space and its current rotational motion. The ATTITUDE STATE is effectively a relationship between the spacecraft, the environment and the effect of any thruster firings used to reorient the spacecraft. This relationship has such attributes as the current spacecraft orientation and the spacecraft angular rotation rates.

The entity-relationship diagram (ERD) is a common graphical tool for entity-oriented specification [Chen 76]. Figure 5 shows an ERD for the GRODY problem domain. The notation for this diagram is based on [Ward 85]. Complex relationships such as ATTITUDE STATE are shown as *associative entities* on ERDs such as figure 5. Associative entities can be identified on an ERD by being connected to a

6

relationship symbol by an arrow. Associative entities are "objectivizations" of relationships which may have attributes and enter into other relationships.
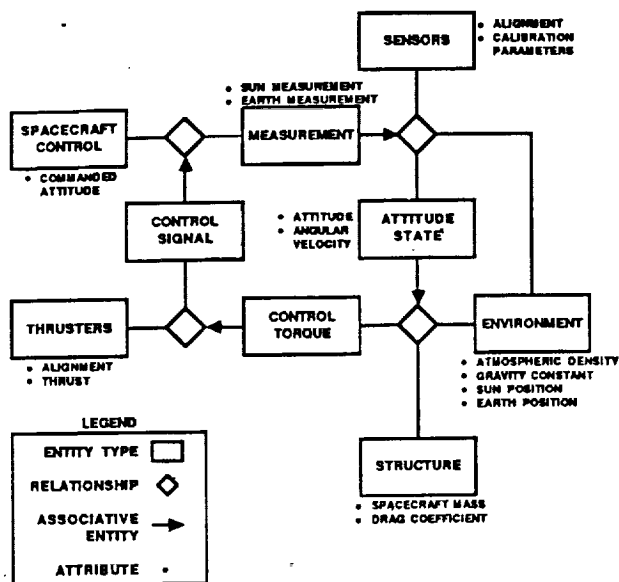

FIGURE 5 *Attitude Dynamics Entity-Relationship Diagram*

Figure 5 shows only a small part of the example problem domain. It would grow as additional entities and relationships are added to describe additional parts of the problem domain. As the specification grows, a complete ERD can quickly become cumbersome. It is possible to "level" ERDS showing complex entities on high-level diagrams which enter into *composite* relationships. These are then broken down in lower-level diagrams. An extended *data dictionary* notation is also useful as a textual representation of entity type and relationship definitions. In addition, the data dictionary provides a common basis for data definition between the static and the dynamic views of the problem domain.

3.2 Processes and Data Flow

ERDs show all *possible* relationships between different types of entities. They do not show the *actual* relationships between specific entities at specific points in time, nor how these actual relationships change over time. Data flow techniques,

however, provide exactly this dynamic view. Traditional data flow diagrams (DFDs) show the flow of data between functional transformations. We will, instead, diagram the flow of data between *processes* which represent the dynamic view of one or more entities in the problem domain. A process is effectively a state machine which accepts input stimuli, reacts to it and produces output stimuli, possibly modifying some internal state data. It has no "operations" as such, only stimuli and responses. These *stimuli* may be either in the form of *data flow* or pure control *signals*.

To construct a dynamics data flow model, one needs to identify those *active entities* which have associated processes. For each relationship in the static entity-relationship model, we choose one of the related entity types to be active. This entity type has an associated process which is charged with maintaining the state of the relationship in response to internal and external stimuli. Note that an entity type may be active relative to one relationship and passive relative to another, and that associative entities may be active or passive.

For example, consider a simplified attitude dynamics simulation system similar to GRODY. The *attitude* of a spacecraft is its orientation relative to inertial space, and an attitude dynamics simulator models the *rotational* motion of the spacecraft in response to external disturbances and the spacecraft control system. Figure 5 describes the problem domain for such a system. The active entities in this case interact in a control loop outlined in figure 6. All the processes shown on figure 6 are associated with active entities on figure 5. A data item flowing on a diagram such as figure 6 may be a passive entity, an attribute or any other composite data item or data element defined in the data dictionary.

The dynamic model must also provide a specification for each individual process. This specification should include a textual description of the object as well as a listing of all inputs and outputs. The process specification also provides a place to include "non-functional" requirements such as timing and accuracy constraints. However, the main point of a process specification is to detail the function of the process. This can be in the form of structured English, a state transition diagram or some other appropriate notation, such as differential equations for the time evolution of the spacecraft attitude.
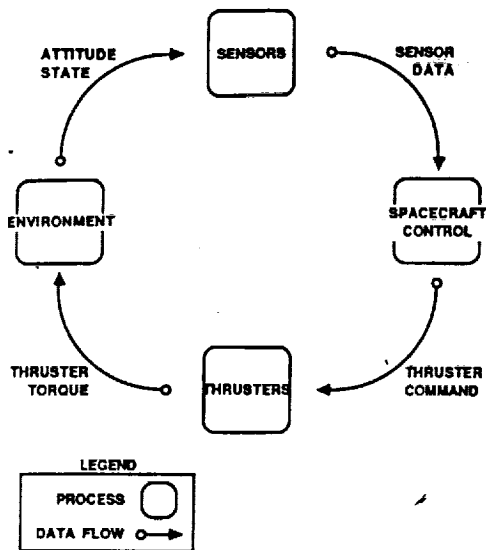
7

5207

FIGURE 6 *Attitude Dynamics Data Flow Diagram*

The function of a process can also be given by a lower-level data flow diagram. Decomposition can continue recursively on all diagrams until all processes have been decomposed into primitive functions and states. This results in a leveling similar to the leveling of traditional DFDs. However, unlike DFDs, each object at each level of a process-data-flow diagram specification has a complete process specification. Each process must also be associated a reasonable problem domain entity independently of its decomposition.

3.3 Object Identification

The intent of an object is to represent a problem domain entity and any associated process. The concept of *abstraction* deals with how an object presents this representation to other objects [Booch 86b, Dijkstra 68]. Ideally, the objects in a design should directly reflect the problem domain entities identified during system specification. However, various design considerations may require splitting or grouping of objects and there will almost always be additional objects in a design to handle "executive" and "utility" functions. Thus there is a spectrum of levels of abstraction of objects in a design, from objects which closely model problem domain entities to objects which really have no reason for existence [Seidewitz 86b]. The following are some points in this scale, from strongest to weakest:

Entity Abstraction - An object represents a useful model of a problem domain entity or class of entities.

Action Abstraction - An object provides a generalized set of operations which all perform similar or related functions (this is similar to the idea of a "utility" object in [Booch 87]).

Subsystem Abstraction - An object groups together a set of objects and operations which are all related to a specific part of a larger system (this is similar to the "subsystem" concept in [Booch 87]).

The stronger the abstraction of an object, the more details are suppressed by the abstract concept. The principle of *information hiding* states that such details should be kept secret from other objects [Booch 87, Parnas 79], so as to better preserve the abstraction modeled by the object.

3.4 Design Hierarchies

The principles of abstraction and information hiding provide the main guides for creating "good" objects. These objects must then be connected together to form an object-oriented design. This design is represented using the graphical object diagram notation [Seidewitz 86b].

The construction of an object-diagram-based design is mediated by consideration of two orthogonal hierarchies in software system designs [Rajlich 85]. The *composition* hierarchy deals with the composition of larger objects from smaller component objects. The *seniority* hierarchy deals with the organization of a set of objects into "layers". Each layer defines a *virtual language extension* which provides services to senior layers [Dijkstra 68]. A major strength of object diagrams is that they can distinctly represent these hierarchies.

The composition hierarchy is directly expressed by *leveling* object diagrams (see figure 7). At its top level, any complete system may be represented by a single object which interacts with *external objects*. Beginning at this system level, each object can then be refined into component objects on a lower-level object diagram, designed to meet the specification for the object. The result is a leveled set of object diagrams which completely describe the structure of a system. At the lowest level, objects are completely decomposed into *primitive objects* such as procedures, tasks and internal state data stores. At higher levels,

8

4-16

object diagram leveling can be used in a manner similar to Booch's "subsystems" [Booch 87].
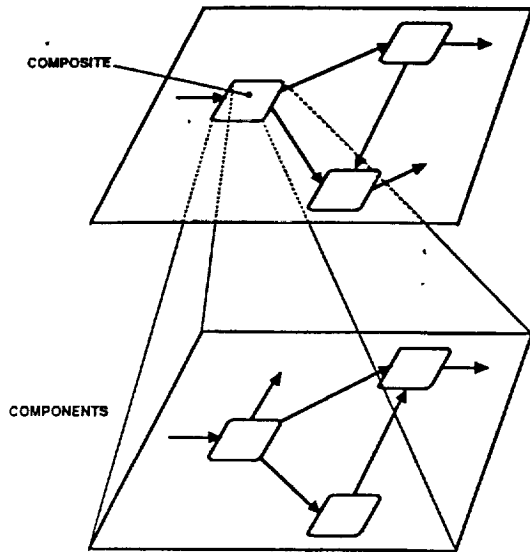

FIGURE 7 *Composition Hierarchy*

The seniority hierarchy is expressed by the topology of connections on a single object diagram (see figure 8). An arrow between objects indicates that one object calls *one or more* of the operations provided by another object. Any layer in a seniority hierarchy can call on any operation in junior layers, but *never* any operation in a senior layer. Thus, all cyclic relationships between objects must be contained within a virtual language layer. Object diagrams are drawn with the seniority hierarchy shown vertically. Each senior object can be designed as if the operations provided by junior layers were "primitive operations" in an extended language. Each virtual language layer will generally contain several objects, each designed according to the principles of abstraction and information hiding.

3.5 System Design

The main advantage of a seniority hierarchy is that it reduces the coupling between objects. This is because all objects in one virtual language layer need to know nothing about senior layers. Further, the centralization of the procedural and data flow control in senior objects can make a system easier to understand and modify.
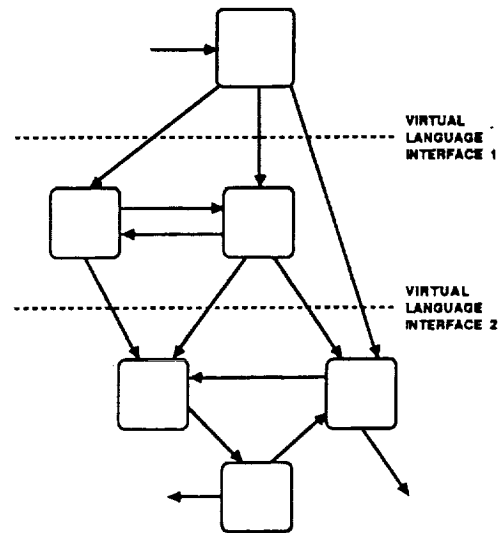

FIGURE 8 *Seniority Hierarchy*

However, this very centralization can cause a messy bottleneck. In such cases, the complexity of senior levels can be traded off against the coupling of junior levels. The important point is that the strength of the seniority hierarchy in a design can be chosen from a *spectrum* of possibilities, with the best design generally lying between the extremes. This gives the designer great power and flexibility in adapting system designs to specific applications.

Figure 9 shows one possible preliminary design for the ATTITUDE SIMULATOR. For simplicity, the sensors and thrusters are represented by a single "SPACECRAFT HARDWARE" object in figure 9. Note that, by convention, the arrow labeled "RUN" is the initial invocation of the entire system. In preliminary design diagrams such as figure 4, it is sometimes convenient to show what data flows along certain control arrows, much in the manner of structure charts [Yourdon 78] or "Buhr charts" [Buhr 84]. These annotations will not appear on the final object diagrams.

In figure 9, the junior level components do not interact directly. All data flow between junior level objects must pass through the senior object, though each object still receives and produces all necessary data (for simplicity not all data flow is shown in figure 9). This design is somewhat like an object-oriented version of the structured designs of Yourdon and Constantine [Yourdon 78].
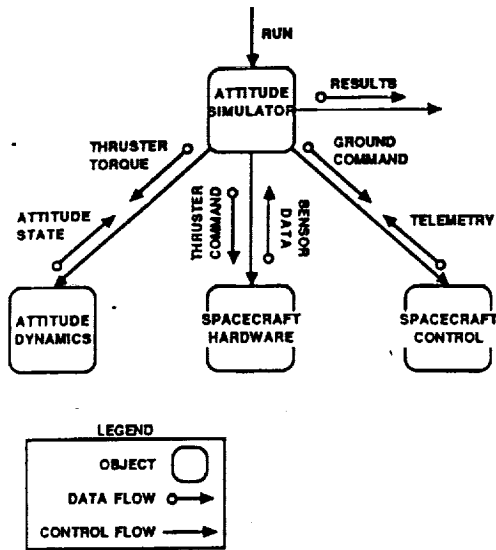
9

4-17

FIGURE 9 *Centralized Design*

We can remove the data flow control from the senior object and let the junior objects pass data directly between themselves, using operations within the virtual language layer (see figure 10). The senior object has been reduced to simply activating various operations in the virtual machine layer, with very little data flow.
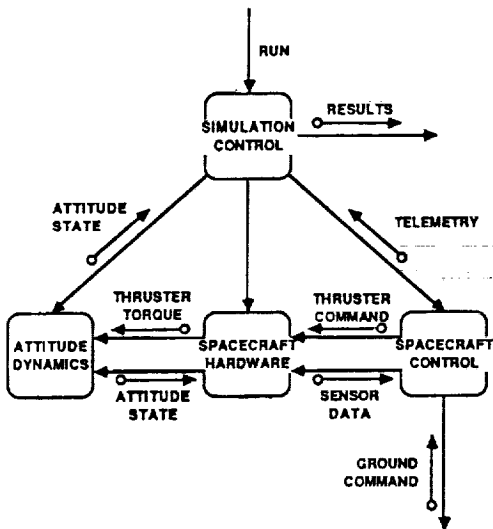

FIGURE 10 *Design with Decentralized Data Flow*

We can even remove the senior object completely by distributing control among the junior level objects (see figure 11). The splitting of the RUN control arrow in figure 11 means that the three objects are activated *simultaneously* and that they run *concurrently*. The seniority hierarchy has collapsed, leaving a *homologous* or non-hierarchical design [Yourdon 78] (no *seniority* hierarchy, that is; the composition hierarchy still remains).

A design which is decentralized like figure 11 at all composition levels is very similar to what would be produced by the PAMELA methodology [Cherry 86]. In fact, it should be possible to apply PAMELA design criteria to the upper levels of an object diagram based design of a highly concurrent system. All concurrent objects would then be composed, at a certain level, of objects representing certain process "idioms" [Cherry 86]. Below this level concurrency would generally no longer be advantageous.
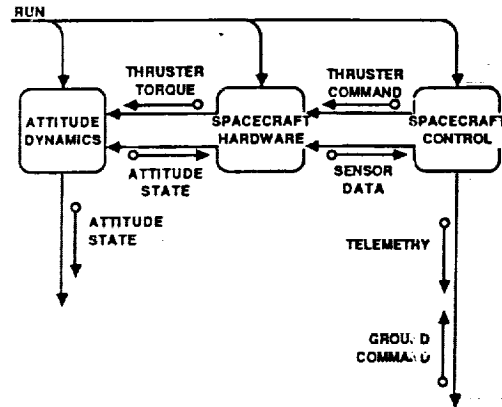

FIGURE 11 *Decentralized Design*

To complete the design, we need to add a virtual language layer of utility objects which preserve the level of abstraction of the problem domain entities. In the case of the ATTITUDE SIMULATOR these objects might include VECTOR, MATRIX, GROUND COMMAND and simulation PARAMETER types. Figure 12 shows how these objects might be added to the simulator design of Figure 10.

10

4-18

5207

Figure 12 gives one complete level of the design of the ATTITUDE SIMULATOR. Note that figure 12 does not include the data flow arrows used in earlier figures. When there are several control paths on a complicated object diagram, it rapidly becomes cumbersome to show data flows. Instead, *object descriptions* for each object on a diagram provide details of the data flow.

An object description includes a list of all operations provided by an object and, for each arrow leaving the object, a list of operations used from another object. We can identify the operations provided and used by each object in terms of the specified data flow and the designed control flow. The object description can be produced by matching data flows to operations. For example, the description for the ATTITUDE DYNAMICS object in figure 12 might be:

Provides:
  procedure Initialize;
  procedure Integrate (For_Duration: in DURATION);
  procedure Apply (Torque: in VECTOR);
  function Current_Attitude return ATTITUDE;
  function Current_Angular_Velocity
   return VECTOR;

Uses:
  5.0 LINEAR ALGEBRA
    Add (Vector)
    Dot
    Multiply (Scalar)
    Multiply (Matrix)

  6.0 PARAMETER DATABASE
    Get

We could next proceed to refine the objects used in figure 12 and recursively construct lower level object diagrams. These lower level designs must meet the functionality of the system specification and provide the operations listed in the object description. The design process continues recursively until the entire system is designed and all objects are completely decomposed.

The GRODY design of figure 4 is basically the same as the example design of figure 12. However, the GRODY team chose to simplify the design by combining the ATTITUDE DYNAMICS and SPACECRAFT HARDWARE objects into a single TRUTH MODEL *subsystem object*, similar to the corresponding subsystem in the FORTRAN design.

Further, in GRODY, the LINEAR ALGEBRA functions are part of a UTILITIES module not shown in figure 4.
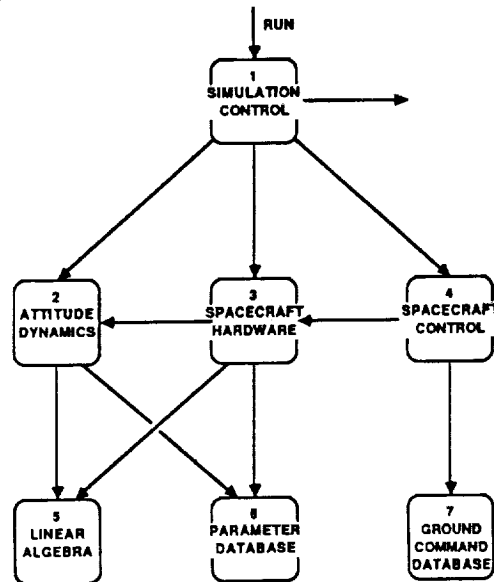

FIGURE 12 *Attitude Dynamics Simulator Design*

3.5 Implementation

The transition from an object diagram to Ada is straightforward. Package specifications are derived from the list of operations provided by an object. For the ATTITUDE DYNAMICS object the package specification is:

package Attitude_Dynamics **is**

  subtype ATTITUDE **is** Linear_Algebra.MATRIX;

  procedure Initialize;
  procedure Integrate
    (For_Duration : **in** DURATION);
  procedure Apply
    (Torque : **in** Linear_Algebra.VECTOR);

  function Current_Attitude
    return ATTITUDE;
  function Current_Angular_Velocity
    return Linear_Algebra.VECTOR;

end Dynamics;

11

4-19

5207

The package specifications derived from the top level object diagram can either be made library units or placed in the declarative part of the top level Ada procedure. For lower level object diagrams the mapping is similar, with component package specifications being nested in the package body of the composite object. States are mapped into package body variables. This direct mapping produces a highly nested program structure. Alternatively, some or all of these packages can be made library units or even reused from an existing library. However, this may require additional packages to contain data types and state variables used by two or more library units. Nevertheless, experience has shown that, to promote reusability and reduce the compilation burden, it is best to avoid nesting of code [Godfrey 87], though it is important to retain leveling in the design.

The process of transforming object diagrams to Ada is followed down all the object diagram levels until we reach the level of implementing individual subprograms. Low-level subprograms can be designed and implemented using traditional functional techniques. They should generally be coded as subunits, rather than being embedded in package bodies.

The clear definition of abstract interfaces in an object-oriented design can also greatly simplify testing. When testing an object, there is a well defined "virtual language" of operations it requires from objects at a junior level of abstraction, some of which may be stubbed-out for initial testing. Further, object-oriented composition encourages incremental integration testing, since the "unit testing" of a composite object really consists of "integration testing" the component objects at a lower level of abstraction.

3.7 Reuse

The concept of *generic* objects provides a powerful tool for reusability. Generic parameters may be used to cut the dependencies of a general object on other specific objects, allowing the general object to be reused in similar but different contexts. Consider, for example, a general numeric integrator with the following package specification:

```
generic

  type REAL is digits <>;
  type STATE_VECTOR is
    array (INTEGER range <>) of REAL;
  with function State_Derivative
    ( T : DURATION;  -- from reference time
      X : STATE_VECTOR )
    return STATE_VECTOR;

package Generic_Integrator is

  procedure Integrate
    ( For_Duration : in DURATION );
  function Current_State
    return STATE_VECTOR;
  procedure Initialize ....;

end Generic_Integrator;
```

This package provides the ability to numerically integrate a vector differential equation with an arbitrary state vector size. The "Integrate" procedure can be implemented as a vector equation, or as a set of individual real-valued functions. To implement it as a single vector equation we will need the operations provided by a LINEAR ALGEBRA object. These operations can be incorporated in two ways. One possibility is to make the operations needed into generic formal parameters. Another is to have the body of the integrator depend directly on LINEAR ALGEBRA.

Each of these methods has advantages and drawbacks. Using generic formal subprograms enhances reusability by making the component self-contained, but if too many are needed the interface becomes complex. Depending on LINEAR ALGEBRA within the GENERIC INTEGRATOR makes a cleaner interface, but couples the generic package to another library unit. The GRODY team has used both methods. Figure 13 shows the composition of GENERIC INTEGRATOR assuming the latter choice.

Figure 14 shows a use of the GENERIC INTEGRATOR in the composition of the ATTITUDE DYNAMICS object. The component object ATTITUDE INTEGRATOR is an instantiation of the GENERIC INTEGRATOR object. The generic object is instantiated in figure 14 with the ATTITUDE EQUATION subprogram as the generic actual parameter. Most of the ATTITUDE DYNAMICS operations are shown in figure 14 as

12

component procedures, represented by rectangles. The "Integrate" operation, however, is directly *inherited* from the ATTITUDE INTEGRATOR object.
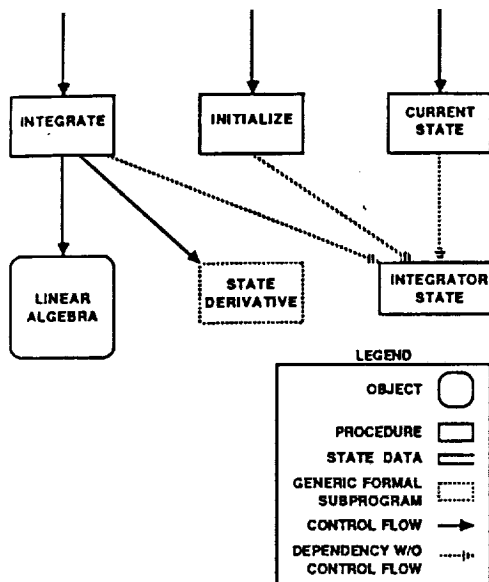

Figure 13 *Generic Integrator Object Composition*

Ada features such as generic packages are useful *tools*, but language features are not sufficient to guarantee high levels of software reuse. What is also needed is an *approach* to specifying and designing reusable components. Using an object-oriented approach is useful not because object-oriented design is essential for reuse, but because the underlying concepts are. These crucial concepts are abstraction, information hiding, levels of virtual languages (often called virtual "machines") and inheritance [Parnas 79, Cox 86].

Smalltalk's subclassing [Goldberg 83] provides an elegant means of supporting inheritance. Ada does not directly support inheritance, but the concept can be simulated by using "call-throughs." A call-through is a subprogram that has little function other than to call on another package's subprogram. To simulate inheritance when implementing the Attitude_Dynamics package the subprogram Integrate would be respecified in the Attitude_Dynamics package, with the subprogram body in Attitude_Dynamics calling on the corresponding operation from Attitude_Integrator.
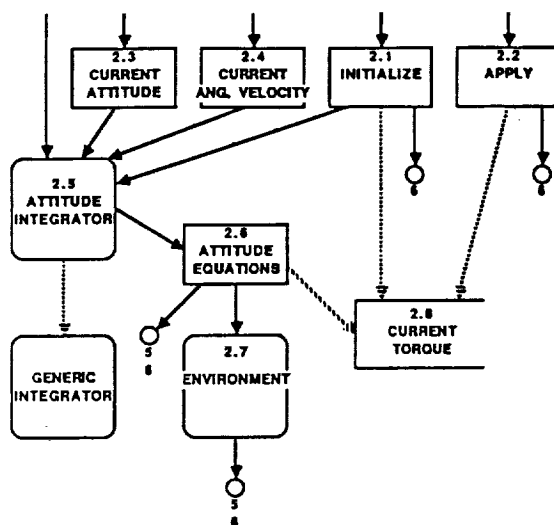

FIGURE 14 *Attitude Dynamics Object Composition*

This technique is clearly less elegant than Smalltalk subclassing, but it also has advantages. First, Ada allows inheritance from more than one object. Second, Smalltalk forces the inheritance of *all* operations and data. An operation can be overridden, but not removed, from a class. The Ada specification of the composite package gives the developer precise control over which operations and data items are visible or accessible. (See [Seidewitz 87] for a more detailed discussion of Ada and the concept of inheritance.)

4. Conclusion

The GRODY project has provided an extremely valuable experience in the application of object-oriented principles to a real system. This experience guided the creation of the GOOD methodology which is now being used on an increasing number of projects inside and outside of the Goddard Space Flight Center. As with any pilot project, some of the major products of GRODY are the lessons learned along the way.

As part of the GRODY project, a detailed assessment has been made of the team's experiences during design [Godfrey 87]. At this time, however, most of the observations must remain qualitative. Nevertheless, it is clear that the GRODY design is significantly different from previous FORTRAN simulator designs [Agresti 86].

13

It also became clear during the GRODY project that the GOOD methodology does not fit comfortably into the traditional life cycle management model. At the very least, the design phase should be extended and design reviews should occur at different points in the life cycle. The preliminary design review should occur later in the design phase and should include detailed object diagrams for the upper levels of the system, perhaps down to the level at which the design becomes more procedural than object-oriented. The critical design review would then include the detailed procedural designs, perhaps using an Ada-based design language. This review might actually take place as a series of incremental reviews of different portions of the design. This later approach is supported by the well-defined modularity of an object-oriented design.

The traditional functional viewpoint provides a comprehensive framework for the entire software life cycle. This viewpoint reflects the action-oriented nature of the machines on which software is run. The object-oriented approach discussed here can also provide a comprehensive view of the life cycle. The object-oriented viewpoint, however, reflects the natural structure of the problem domain rather than the implicit structure of our hardware. Thus, it provides a "higher-level" approach to software development which decreases the distance between problem domain and software solution. By making complex software easier to understand, this simplifies both system development and maintenance.

References

[Abbott 83]
Abbott, R. J. "Program Design by Informal English Description," *Communications of the ACM*, September 1983.

[Agresti 84]
Agresti, William W. "An Approach to Developing Specification Measures," *Proceedings of the 9th Annual Software Engineering Workshop*, GSFC Document SEL-84-004, November 1984.

[Agresti 86]
Agresti, William W., et. al. "Designing with Ada for Satellite Simulation: a Case Study," *Proceedings of the 1st International Conference on Ada Applications for the Space Station*, June 1986.

[Agresti 87]
Agresti, William W. *Guidelines for Applying the Composite Specification Model (CSM)*, GSFC Document SEL-87-003, June 1987.

[Bailin 88]
Bailin, Sidney C. and J. Michael Moore. "An Object-Oriented Specification Method for Ada," to be presented at the *Fifth Washington Ada Symposium*, June 1988.

[Basili 85]
Basili, V. R., et. al. "Characterization of an Ada Software Development," *Computer*, September 1985.

[Booch 83]
Booch, Grady. *Software Engineering with Ada*, Benjamin/Cummings, 1983.

[Booch 86a]
Booch, Grady. "Object-Oriented Software Development," *IEEE Transactions on Software Engineering*, February 1986.

[Booch 86b]
Booch, Grady. *Software Engineering with Ada, 2nd Edition*, Benjamin/Cummings, 1986.

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

[Buhr 84]
Buhr, R. J. A. *System Design with Ada*, Prentice-Hall, 1984.

[Chen 76]
Chen, P. "The Entity-Relationship Model -- Toward a Unified View of Data," *ACM Transactions on Data Base Systems*, March 1976.

[Cherry 85]
Cherry, George W. PAMELA Course Notes, Thought**Tools, 1985.

[Cherry 86]
Cherry, George W. *PAMELA Designer's Handbook*, Thought**Tools, 1986.

[Cherry 88]
Cherry, George W. *PAMELA 2: An Ada-Based Object-Oriented Design Method*, Thought**Tools, February 1988.

14

5207

[Cox 86]
Cox, Brad. *Object-Oriented Programming: an Evolutionary Approach*, Addison-Wesley, 1986.

[Dijkstra 68]
Dijkstra, Edsgar W. "The Structure of the 'THE' Multiprogramming System," *Communications of the ACM*, May 1968.

[Godfrey 87]
Godfrey, Sara, Carolyn Brophy, et. al. *Assessing the Ada Design Process and its Implications: a Case Study*, GSFC Document SEL-87-004, July 1987.·

[Goldberg 83]
Goldberg, Adele and David Robson. *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, 1983.

[McGarry 88]
McGarry, Frank and William Agresti. "Measuring Ada for Software Development in the Software Engineering Laboratory (SEL)," *Proceedings of the 21st Hawaii International Conference on Software Engineering*, January 1988.

[Nelson 86]
Nelson, Robert W. "NASA Ada Experiment -- Attitude Dynamic Simulator," *Proceedings of the Washington Ada Symposium, March 1986.*

[Parnas 72]
Parnas, David L. "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, December, 1972.

[Parnas 79]
Parnas, David L. "Designing Software for Ease of Expansion and Contraction," *IEEE Transactions on Software Engineering*, March 1979.

[Rajlich 85]
Rajlich, Vaclav. "Paradigms for Design and Implementation in Ada," *Communications of the ACM*, July 1985.

[Seidewitz 86a]
Seidewitz, Ed and Mike Stark. "Towards a General Object-Oriented Software Development Methodology," *Proceedings of the 1st International Conference on Ada Applications for the Space Station*, June 1986.

[Seidewitz 86b]
Seidewitz, Ed and Mike Stark. *General Object-Oriented Software Development*, GSFC Document SEL-86-002, August 1986.

[Seidewitz 87]
Seidewitz, Ed. "Object-Oriented Programming in Smalltalk and Ada", *Proceedings of the Conference on Object-Oriented Programming, Languages, Systems and Applications*, October 1987.

[Stark 87]
Stark, Mike and Ed Seidewitz. "Towards a General Object-Oriented Ada Lifecycle," *Proceedings of the Joint Conference on Ada Technology / Washington Ada Symposium*, March 1986.

[Ward 85]
Ward, Paul T. and Stephen J. Mellor, *Structured Development for Real-Time Systems*, Yourdon Press, 1985.

[Yourdon 78]
Yourdon, Edward and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Yourdon Press, 1978.

15

4-23

5207

Lessons Learned in the Implementation Phase of a
Large Ada[TM] Project

Carolyn E. Brophy[1], Sara Godfrey[2],
William W. Agresti[3], and Victor R. Basili[1]

1	Department of Computer Science University of Maryland College Park, MD. 20742	2	Goddard Space Flight Center Code 552.1 Greenbelt, MD. 20771	3	Computer Sciences Corporation System Sciences Division 8728 Colesville Road Silver Spring, MD. 20910

Abstract

We need to understand the effects that introducing Ada has on the software development environment. This paper is about the lessons learned from an ongoing Ada project in the Flight Dynamics division of the NASA Goddard Space Flight Center. It is part of a series of lessons learned documents being written for each development phase.

FORTRAN is the usual development language is this environment. This project is one of the first to use Ada in this environment. The experiment consists of the development of two spacecraft dynamics simulators. One is done in FORTRAN with the usual development techniques, and the other is done with Ada. The Ada simulator is 135,000 lines of code (LOC), and the FORTRAN simulator is 45,000 LOC.

We want to record the problems and successes which occurred during implementation. Topics which will be dealt with include (1) use of nesting vs. library units, (2) code reading, (3) unit testing, and (4) lessons learned using special Ada features.

It is important to remember that these results are derived from one specific environment; we must be very careful when extrapolating to other environments. However, we believe this is a good beginning to a better understanding of Ada use in production environments.

Ada incorporates many software development concepts; it is much more than "just another language". As such, we need to understand the effects of introducing Ada into the software development environment. This paper concentrates on the lessons learned from an ongoing Ada project in the Flight Dynamics Division of the NASA Goddard Space Flight Center (GSFC). The Ada project is sponsored by the GSFC Software Engineering Laboratory (SEL). It is part of a series of lessons learned documents being written for each development phase.

Environment

FORTRAN is the usual development language in this environment. The flight dynamics applications involve mission analysis and spacecraft orbit and attitude determination and control. Many of the software development projects are similar from mission to mission providing, for example, an attitude ground support system or an attitude dynamics simulator. This pattern of developing similar applications is important for domain expertise and for the legacy developed in this environment for code, designs, expectations and intuitions. The similarity between projects allows a high level of reuse of both design and code. Since the problems are basically familiar ones, the development methodologies which involve much iteration do not seem to be necessary. The waterfall development model is basically used here, and seems to work well in this case. Lessons learned from the initial uses of Ada do not include changing this basic methodology.

Project

The project was originally designed as a parallel study with two teams. Each would develop a spacecraft dynamics simulator, one with FORTRAN as the implementation language, and one with Ada as the implementation language. The specifications for each simulator were the same, supporting the upcoming Gamma Ray Observatory (GRO) mission. However, there are many

other differences between the projects which keep the study from being truly "parallel". The FORTRAN version was the production version, thus they had scheduling pressures the Ada team did not have. Without scheduling pressures, the Ada team made enhancements in their version not required by the specifications, which increased time spent on the project. This was also the first time any of these team members had done an Ada project, while the FORTRAN team was quite experienced with the use of FORTRAN. The Ada team required training in the language and development methodologies associated with Ada, while the FORTRAN team did things in the usual way [McGarry, Page et al. 83]. The Ada team also experimented with various design methodologies; this was necessary to find which ones would work better for this development environment. The FORTRAN team was working with a mature and stable environment. In switching to Ada, the legacy of reuse for design, code, intuitions and experience are gone, and will be rebuilt slowly in the new language.

The philosophies of development were quite different between the two projects. The Ada team consistently applied the ideas of data abstraction and information hiding to their design development. The FORTRAN development used structural decomposition methods.

Our goals with this project include:

(1) How is the use of Ada characterized in this environment?

(2) How should the existing development process be modified to best changeover from FORTRAN to Ada?

(3) What problems have been encountered in development? What ways have we found to deal with them?

Current Project Status

Both the FORTRAN and Ada teams started in January, 1985. The Ada team began with training in Ada, while the FORTRAN team immediately began requirements analysis. The FORTRAN team delivered its product (45K) after completing acceptance testing in June, 1987. The Ada team is scheduled to finish system testing its 135K product in February, 1988. Discussions of the product size differences and effort distributions are presented in [McGarry, Agresti 88].

The lessons learned from major phases in the Ada development are being recorded in a series of SEL reports: Ada training [Murphy, Stark 85], design [Godfrey, Brophy 87], and implementation [in preparation]. This paper presents some of the main results from the implementation (code and unit test) lessons learned.

Lessons Learned

1. Nesting vs. Library Units

1.1 The flat structure produced by using library units has advantages over a heavily nested structure.

Nesting has many effects on the resulting product. The primary advantage of nesting is that it enforces the principle of information hiding structurally, because of the Ada visibility rules. Whereas with library units, the only way to avoid violations of information hiding is through self-discipline. In addition, the dot notation tells the package where a module is located.

There are quite a few disadvantages to nesting, however. Nesting makes reuse more difficult. A second dynamics simulator in Ada is now being developed which can reuse up to 40% of the Ada project's code. But in order to reuse it, the nested code has to be unnested, since the new application only needs some of the nested units. This is often a labor intensive operation. Nesting also increases the amount of recompilation required when changes are made, since Ada assumes dependencies between even sibling nested objects/procedures, even when the dependency is not really there. This requires more parts of the system to be recompiled than is necessary when more library units are used. It is also harder to trace problems back through nested levels than it is through levels of library units. There is no easy way to tell where a unit of code was called from, when it is nested. But library units have the "with" clauses to identify the source of a piece of code. For this reason it is now believed that over use of nesting at the expense of using more library units makes maintenance harder. This is contrary to the team's earlier expectations. The team had used nesting successfully before on a 5000 lines of code training project. However, this kind of approach does not scale-up well when developing large projects.

Library units seem to have a lot of advantages. Besides fewer recompilations when changes are made, and easier unit testing, every library unit can easily be made visible to any other library unit merely by use of the "with" clause. In nested units this visibility does not exist, and a debugger becomes essential to see what is happening at the deeper levels that are not within the scope of the test driver. Library units allow smaller components, smaller files, smaller compilation units, and less duplication of code. The system is more maintainable, since it is easier to find the unit desired. Reuse with library units is also easier, since the parts of the system are smaller. Configuration control is also easier with library units since more pieces are separate (i.e., the ratio of changes to code segments modified is closer to 1). The major disadvantage seems to be that a complicated library structure develops, which can lead to errors by the developers. However, if the Ada project were to be done over now, the team would use more library units, and nest less.

5207

Advantages and Disadvantages of
Nesting vs. Library Units

NESTING

Advantages	Disadvantages
* information hiding	* enlarged code
* visibility control	* more recompilations
* type declarations in	* harder to trace problems
one place	through nested
	levels
	* can't easily tell where a
	unit of code called
	from
	* type declarations in one
	place means problems
	for reuse
	* harder maintenance
	* debugger required
	* larger unit sizes
	inhibit code reading
	* harder to reuse part of
	the system

LIBRARY UNITS

Advantages	Disadvantages
* fewer recompilations	* no information hiding
* easier unit testing	* complex library structure
* smaller components	
* smaller files	
* smaller compilation units	
* less code duplication	
* easier maintenance	
* "with" clauses show source	
of other code units used	
* easier reuse	
* easier configuration control	

1.2 The balance between nesting and library units is an important implementation issue, not a design issue.

The issue of whether to use library units or nested units first arises in the design phase. At least this is the case if it is assumed that the design documents reflect this aspect of implementation (i.e., the design documents indicate in some way when nesting is intended vs. when library units should be used). While it is appropriate for the design to show dependencies, these should not dictate implementation, as far as the library unit/nesting question is concerned. The team considered the decisions concerning nesting/library units to be an implementation issue.

The library units in the Ada project went down about 3 to 4 levels, while nesting went down many levels below that. Another view of the system shows the Ada project had 124 packages and 55 library units. During implementation most team members felt an appropriate balance had been reached between nesting levels and number of library units. However, in retrospect, several felt the nesting had been overdone.

4-26

1.3 It appears best to use library units at least down to the subsystem level, and nesting at lower levels where there is minimal interaction among a small number of modules

Experiences with unit testing seem to indicate that library units should at least go down to the subsystem level. This makes testing easier. Below this level the benefits of nesting sometimes become too important to ignore. This is one heuristic which could be used to help determine when the transition from library units to nested units should occur.

An additional way to determine when the transition should occur is to examine the degree of interaction between pieces. For modules which interact heavily, library units are preferred. At the point where the interaction drops off, using nested units is preferable. Sections with nested code are easier to deal with when they are small.

1.4 In mapping design to code, caution should be used in applying too rigorous a set of rules for visibility control.

In an attempt to control visibility, two features appear to have been too rigorously applied. The first feature is nesting. The design of the Ada project seemed to suggest a particular nesting implementation. But this created many objects within objects yielding a high degree of nesting. The second way to control visibility is through the use of many "call-through"s (a procedure whose only function is to call another routine). "Call-through"s were used to group appropriate pieces together exactly as represented in the design. They can be implemented via nesting or library units. Faithfulness to the design structure was maintained this way.

The design had non-primitive objects with specific operations. These objects were implemented as packages. To put the specific operations (subprograms) into the objects (packages) the team used "call-through"s. Thus a physical piece of code was created for every object in the design. "Call-through"s are one of the reasons for the expanded code in the Ada project when compared to the FORTRAN version. It is estimated that out of the 135K LOC making up the Ada system, 22K LOC (specifications and bodies) are because of "call-through"s. While "call-through"s provide a good way to collect things into subsystems, these should be limited to only two or three levels in the future.

If the implementation were to be done over now, many of the existing "call-through"s would be eliminated. Instead of creating actual code to correspond with every object in the design, some objects in the design would remain "logical objects". No actual packages would exist; instead, a logical object would be made up of a collection of lower level objects.

2. Code Reading

Code reading is generally done with unit testing. The developer doing the code reading is not the one who developed the code. Comments are returned to the original developer. After code reading and unit testing, the unit is put under configuration control.

2.1 Code reading helps in training people to use Ada.

Besides helping to find errors, code reading has the benefit of increasing the proficiency of team members in Ada. Individuals can see new ways to handle the algorithms being encoded. Code reading also allows another person besides the original developer to understand a given part of the project. This insight should help understanding and lead to better solutions of problems in the future.

2.2 Code reading helps isolate style and logic errors.

The most common errors found in code reading with Ada were style errors. The style errors involved adding or deleting comments, format changes, and changes to debug code (not left in the final product). Other types of errors found are initialization errors, and problems with incompatibilities between design and code. This can be due to either a design error or a coding error.

Because the Ada compiler exposes many errors not exposable by a FORTRAN compiler, code reading Ada has a different flavor than code reading FORTRAN. For example, the Ada compiler exposes such errors as (1) wrong data types, (2) call sequencing errors, (3) variable errors-- either the variable is declared and never used, or it is used without being declared. So, one seasoned FORTRAN developer working on the Ada project noted that code reading is more interesting in FORTRAN, since there were more interesting errors found in code reading FORTRAN, not found in reading Ada code. In general, logic errors are hard to find in this application domain, but enough logic errors are found to make code reading worthwhile.

Some of the difficulty of code reading with Ada on this project was due to the heavy nesting and the number of "call-through" units. Code reading would have been helped by a flatter implementation. The SEPARATE facility makes it necessary to look in many places at once to follow the code. However, code reading in Ada was easier than in FORTRAN because the code was more English-like, and hence, more readable. Often the reused FORTRAN code is an older variety without the structured constructs available in later versions.

Code reading tended to miss errors that spanned multiple units. This would be expected with any implementation language as well. One example was a problem where records were skipped when they were being output. The debugger actually found the problem.

Despite the implementation language, code reading appears to be important for highly algorithmic routines.

4-27

Groups of routines that are used only to call others may be checked to make sure the design's purity is maintained.

3. Unit Testing

3.1 Unit testing was found to be harder with Ada than with FORTRAN.

The FORTRAN units are already relatively isolated; this makes unit testing easy. Only the global COMMONs need to be added to do the unit tests. On the other hand, the Ada units require a lot of "with'd in" code, and are much more interdependent. Another very different Ada project had perhaps even more interdependence between its modules than the Ada project did. That team also found the interdependence made unit testing very difficult. More interdependence exists between Ada units because there are more relations to express in Ada. There are textual inclusion (nesting), with-ing in (library units), and invocation. FORTRAN only has invocation.

3.2 The introduction of Ada as the implementation language changed the unit testing methods dramatically.

Unit testing with Ada was done very differently. Since one unit depends on many others, it is usually hard to test a unit in isolation, so this was generally not done. The Ada pieces were integrated up to the package level, and then unit testing was done. Then testing was done with groups of units that logically fit together, rather than actual unit testing. The integrated units are tested, choosing only a subset of possible paths at a time. The debugger is used to look at a specific unit, since the test drivers cannot "see" the nested ones. With Ada projects a debugger becomes essential. This is in contrast to the usual development in FORTRAN where no integration occurs at all until after unit testing.

This shows that the biggest difference between the way FORTRAN and Ada projects are done at this point in development is incremental integration. This actually represents a change in the development lifecycle of an Ada product; integration and unit testing are alternately done rather than finishing unit testing before integration.

3.3 The library unit/nesting level issue directly affects the difficulty of unit testing.

The greater the nesting level, the more difficult unit testing is, since the lower level units in the subsystem are not in the scope of the test driver. This is the primary reason a debugger becomes a required testing aid with Ada projects. For this reason, more library units and less nesting would have made testing easier. Library units have to go down to a level in the design that makes testing more feasible. With the Ada project that would have meant taking library units down to a lower level in the design, if the project were to be done over.

Two other ways to deal with the nesting during unit test were tried and were not very successful. One solution pulls an inner package out, and includes the types and "with'd in" modules the outer package used in order to execute the inner one. This is difficult to do for each unit. The other solution is to modify the specifications of the outer package so that nested packages can be "seen" by the test driver. This solution requires lots of recompilation. With more library units, there would be less recompilation, because there would be fewer changes of specifications. Again however, the best way to test was to use the debugger on unaltered code.

3.4 The importance of unit testing seems to be more related to application area than to implementation language.

Whether the implementation is in FORTRAN or Ada, does not seem as important as whether the application has lots of calculations or has lots of data manipulations. Unit testing seemed more valuable with scientific applications; perhaps because calculation errors show up when only a small amount of localized code is executed. But data manipulation errors require more of the system to be operating before it is known if errors are present.

4. Use of Ada's Special Features

4.1 Separation of specifications and bodies is quite beneficial and easy to implement.

The team entered the specifications first, whenever possible, before the rest of the code. This gave a high level view of the system early in the development. Another benefit is that this helped clarify the interfaces early. Separating the specifications and bodies also reduces the amount of recompilation required when changes are made.

4.2 Generics were fairly easy to implement and they reduce the amount of code required.

The only problems encountered were with correct compilation of the generics in some cases, due to compiler bugs in an early version of the compiler, rather than incorrect code. As Ada matures, this will not be a problem at all.

4.3 Using too many types increases coding difficulty.

The strong typing was very difficult to get used to, when one is accustomed to weakly typed languages such as FORTRAN. It was easy to create too many new types as well.

Often a brand new type was created with a strict range appropriate for one portion of the application. Then in other areas where subtypes could have been used, the range on the original type was found to be too restrictive, so another brand new type was created instead to handle the new situation. Then a whole new

4-28

set of operations had to be created as well for the additional new type. Next time the team would recommend creating a more general new type, and using many different subtypes of the original type, rather than creating more new types. In this way operations can be reused and there are far fewer main types to keep track of. Designers need to spend time developing families of types that inherit properties from one another.

The strong typing presented some problems when testing units, though it prevents some kinds of errors, also. It was harder to write test drivers that could deal with all the types in the units being tested. It was also harder to do the I/O, since so many types had to be dealt with.

4.4 Tasking was difficult to code and test, however, this seems due to concurrency in general and not Ada specifically.

Tasks were used in the user interface part of the project. The user was given many options which made the interactions between the tasks of the subsystem very difficult to plan and execute correctly.

It was harder to code tasks from the design than it was to code other types of units. However, this is not really due to Ada, but rather it is the nature of concurrency problems. The language made the use of tasking easier, and encouraged the developers to use tasking more than they would have otherwise. The dynamic relationships of concurrency cannot be represented in the design (termination, rendezvous, multiple threads of control). Correctness was very difficult to assure, as is usual with these kinds of problems, and deadlock was hard to avoid. Functional testing was done, which is the usual type in this environment.

The major problem the developers had was with exceptions. These are no worse with tasking than they are with any other program unit, however.

4.5 Exception handlers have to be coded carefully.

The key problem with exceptions is deciding the best way to handle them. Errors and the sources of errors can be hard to find if the exception handlers are not coded carefully. Suppose a particular procedure will call another unit, expecting some function to be performed, and certain kinds of data to be returned. If an exception is raised and handled in the called unit, and it is non-specific for the problem raising the exception (e.g., "when others") , the caller gets control back without the required function being performed. But the exception was handled and data was returned, so the call looks successful. Yet as soon as the caller tried to use the data from the routine where the exception was raised and handled, it fails. Because of propagation, it can be very difficult to trace back the error to the original source of the problem.

Several members of the team would recommend incorporating the way exceptions are to be handled into the design, rather than leaving this until implementation. Put into the design (1) what exception would be raised, (2) where it will be handled, and (3) what should happen.

Ada Features*

	implementation ease	benefit
tasking	-	+
generics	+	++
strong typing	0	0
exception handling	0	+
nesting	+	-
separate specs/bodies	++	++

* This figure represents a subjective assessment based on team member interviews

Summary
We have learned several important things about four major areas in implementation. There are many advantages to using library units, though nesting can have its usefulness at some point below the subsystem level. Code reading helps train people in Ada, and helps to isolate style and logic errors. Unit testing was substantially changed by using Ada: the first stages of integration often began before unit testing proceeded. Some Ada features are quite powerful and should be carefully used.

It is important to remember that these results are derived from one specific environment. We must be very careful when extrapolating to other environments. There are also many questions still left to be answered. Studies of this project will continue, and other Ada projects are being started. These will help us evaluate the effects on longer term issues such as reuse and maintainability of the Ada projects. We believe this project is a good beginning to a better understanding of Ada use in production environments.

References

[Agresti 85]
Agresti W., "Ada Experiment: Lessons Learned (Training/Requirements Analysis Phase)", Goddard Space Flight Center, Greenbelt, MD 20771, August 1985.

[Godfrey, Brophy 87]
SEL-87-004, "Assessing the Ada Design Process and Its Implications: A Case Study", Godfrey S., and Brophy C., Goddard Space Flight Center, Greenbelt, MD 20771, July 1987.

[McGarry, Agresti 88]
"Measuring Ada for Software Development in the Software Engineering Laboratory", Hawaii International Conference on Systems Science, January, 1988.

[McGarry, Nelson 85]
McGarry F., and Nelson R., "An Experiment with Ada — The GRO Dynamics Simulator Project Plan," Goddard Space Flight Center, Greenbelt, MD 20771, April 1985.

[McGarry, Page et al. 83]
SEL-81-205, "Recommended Approach to Software Development", McGarry F., Page J., Eslinger S., Church V., and Merwarth P., Goddard Space Flight Center, Greenbelt, MD 20771, April 1983.

[Murphy, Stark 85]
SEL-85-002, "Ada Training Evaluation and Recommendations from the Gamma Ray Observatory Ada Development Team", Murphy R., and Stark M., Goddard Space Flight Center, Greenbelt, MD 20771, October 1985.

Biographies

Carolyn E. Brophy is a graduate research assistant at the University of Maryland, College Park. Her research interests are in software engineering, and she is working with the NASA Goddard Software Engineering Laboratory. Ms. Brophy received a B.S. degree from the University of Pittsburgh in biology and pharmacy. She is a member of ACM.

Sara H. Godfrey is with Goddard Space Flight Center in Greenbelt, Maryland, where she has been working with the NASA Goddard Software Engineering Laboratory. She received a B.S. degree from the University of Maryland in mathematics. (picture missing)

William W. Agresti is with Computer Sciences Corporation in Silver Spring, Maryland. His applied research and development projects support the Software Engineering Laboratory at NASA's Goddard Space Flight Center. His research interests are in software process engineering, and he recently completed the tutorial text, New Paradigms for Software Development, for the IEEE Computer Society. From 1973-83 he held various faculty and administrative positions at the University of Michigan-Dearborn. He received the B.S. degree from Case Western Reserve University, the M.S. and Ph.D. from New York University.

Victor R. Basili is Professor and Chairman of the Computer Science Department at the University of Maryland, College Park, Maryland. He was involved in the design and development of several software projects, including the SIMPL family of programming languages. He is currently measuring and evaluating software development in industrial and government settings and has consulted with many agencies and organizations, including IBM, GE, CSC, GTE, MCC, AT&T, Motorola, HP, NRL, NSWC, and NASA.

5207

He is one of the founders and principals in the Software Engineering Laboratory, a joint venture between NASA Goddard Space Flight Center, the University of Maryland and Computer Sciences Corporation, established in 1976. He has been working on the development of quantitative approaches for software management, engineering and quality assurance by developing models and metrics for the software development process and product.

Dr. Basili has authored over 90 papers. In 1982, he received the Outstanding Paper Award from the IEEE Transactions on Software Engineering for his paper on the evaluation of methodologies.

He was Program Chairman for several conferences including the 6th International Conference on Software Engineering. He serves on the editorial boards of the *Journal of Systems and Software* and the IEEE *Transactions on Software Engineering* and is currently Editor-in-Chief of *TSE*. He is a member of the Board of Governors of the IEEE Computer Society.

OBJECT-ORIENTED PROGRAMMING IN SMALLTALK AND ADA

Ed Seidewitz
Code 554 / Flight Dynamics Analysis Branch
Goddard Space Flight Center
Greenbelt MD 20771
(301) 286-7631

Abstract

Though Ada and Modula-2 are not object-oriented languages, an object-oriented viewpoint is crucial for effective use of their module facilities. It is therefore instructive to compare the capabilities of a modular language such as Ada with an archetypal object-oriented language such as Smalltalk. The comparison in this paper is in terms of the basic properties of encapsulation, inheritance and binding, with examples given in both languages. This comparison highlights the strengths and weaknesses of both types of languages from an object-oriented perspective. It also provides a basis for the application of experience from Smalltalk and other object-oriented languages to increasingly widely used modular languages such as Ada and Modula-2.

1. Introduction

Procedural programming techniques concentrate on functions and actions. Object-oriented techniques, by contrast, attempt to clearly model the problem domain. The designers of Simula recognized the attractiveness of this concept for simulation and included specific constructs for object-oriented programming [Dahl 68]. Since then, several programming languages have been designed specifically for -general-purpose object-oriented programming. The archetypal example is, perhaps, Smalltalk because the language is structured so completely around the object concept [Goldberg 83].

Ada* [DOD 83] and Modula-2 [Wirth 83] are not designed to be object-oriented programming languages. However, they do have certain object-oriented features which are descendants of Simula constructs. Further, object-oriented concepts have become extremely popular for design of Ada programs (e.g., see [Booch 83]). This paper compares and contrasts the strict object-oriented capabilities of Smalltalk with the object-oriented features of Ada. The comparison is in terms of the basic object-oriented properties of encapsulation, inheritance and binding. I have attempted to keep the main body of the paper fairly objective, reserving my more judgemental comments for the conclusion.

---

*Ada is a registered trademark of the US Government (Ada Joint Program Office)

5207

2. Encapsulation

An object consists of some private data and a set of operations on that data. The intent of an object is to *encapsulate* the representation of a problem domain entity which changes state over time. *Abstraction* deals with how an object presents this representation to other objects, suppressing nonessential details. The stronger the abstraction of an object, the more details are suppressed by the abstract concept. The principle of *information hiding* states that such details should be kept secret from other objects, so as to better preserve the abstraction modeled by the object. Both Smalltalk and Ada directly support these basic encapsulation concepts for objects. In Smalltalk these features are the central structure of the language while in Ada they are added to a core language of ALGOL/Pascal heritage.

In Smalltalk, objects are always *instances* of a *class* which represents a set of problem domain entities of the same kind. All instances of a class provide the same interface (set of operations) to other objects. A class thus represents a single abstraction. The class definition provides implementations for each of the instance operations (*methods* in Smalltalk) and also defines the form of the internal memory of all instances.

A Smalltalk method is called by sending a *message* to the object, such as:

MyFinances receive: 25.50

The *protocol* of an object is the set of all messages that may be received by the object. A class itself has a protocol which usually includes a few messages to request creation of instances, e.g. "Finances new". Note that protocols are not really a part of the Smalltalk language proper, but are documentation of the abstraction represented by a Smalltalk class.

The basic object-oriented construct in Ada is the *package*. Unlike Smalltalk, objects can be defined directly in Ada without having any class. Further, Ada requires the definition of the interface of an object separately from the implementation of the object. This is done in a package specification. Ada uses a more traditional procedure call syntax for object operations.

Ada is a strongly typed language, so the type of every operation argument and return value must be declared. A package specification provides enough declarative information for compile-time syntax and type checking. Additional operation descriptions, such as in the Smalltalk protocol, can be provided by comments. Other code refers to package operations using a *qualified name*, e.g., "Finances.Receive". The *package body* gives the implementation of the package.

Example 1 -- Finances

Class Finances is a simple class of objects which represent financial accounts of income and debt (all examples are simplified and adapted from [Goldberg 83]). The protocol for this class is:

Finances class protocol

instance creation

initialBalance: amount
Begin a financial account with "amount" as the amount of money on hand.

new
Begin a financial account with 0 as the amount of money on hand.

Finances instance protocol

transactions

receive: amount
Receive an amount of money.

spend: amount
Spend an amount of money.

inquiries

cashOnHand
Answer the total amount of money currently on hand.

totalReceived
Answer the total amount of money received so far.

5207

totalSpent   Answer the total amount of money spent so far.

The implementation of the Finances class must include a method for each of the messages in the protocol. It also defines the names of a set of *instance variables* which represent the internal data of each class instance. The instance variables and the implementations of the methods are hidden from users of instances of the class. In the Smalltalk-80 system, the various parts of a class definition are accessed through an "interactive system browser." The textual description used here is based on the one used in [Goldberg 83]. The definition of class Finances is:

class name	Finances
superclass	Object
instance variable names	income
	debt

class methods

instance creation

initialBalance: amount
^super new setInitialBalance: amount

new
^super new setInitialBalance: 0

instance methods

transactions

receive: amount
income <- income + amount

spend: amount
debt <- debt + amount

inquiries

cashOnHand
^income - debt

totalReceived
^income

totalSpent
^debt

private

setInitialBalance: amount
income <- amount.
debt <- 0

Note that "super new" refers to the system method to create a new instance, "^" indicates returning a value and "<-" indicates assignment. Some examples of use of this class are:

MyFinances <- Finances initialBalance: 500.00.
MyFinances spend: 32.50.
MyFinances spend: foodCost + salesTax.
MyFinances receive: pay.
tax <- taxRate * (MyFinances totalReceived)

The specification for an Ada package Finances corresponding to the above Smalltalk protocol is:

package Finances **is**

  type MONEY **is** FLOAT;

  -- Initialization
  procedure Set (Balance : **in** MONEY);

  -- Transactions
  procedure Receive (Amount : **in** MONEY);
  procedure Spend (Amount : **in** MONEY);

  -- Inquiries
  function Cash_On_Hand **return** MONEY;
  function Total_Received **return** MONEY;
  function Total_Spent **return** MONEY;

end Finances;

The above specification for Finances really does not define a complete object in the Smalltalk sense. This is because a package is a static program module, and cannot be passed around as data. For an object to be passed as data in Ada it must have a *type*. A type is analogous to a Smalltalk class in that it represents a set of objects with the same set of operations and internal data. An object type is called a *private type* in Ada because the representation of the internal data is hidden. The specification for a private type FINANCES is:

```ada
package Finance_Handler is

   type FINANCES is private;
   type MONEY is FLOAT;

   -- Instance creation
   function Initial (Balance : MONEY)
     return FINANCES;

   -- Transactions
   procedure Receive
     ( Account   : in out FINANCES;
       Amount    : in MONEY );
   procedure Spend
     ( Account   : in out FINANCES;
       Amount    : in MONEY);

   -- Inquiries
   function Cash_On_Hand
     ( Account : FINANCES )
     return MONEY;
   function Total_Received
     ( Account : FINANCES )
     return MONEY;
   function Total_Spent
     ( Account : FINANCES )
     return MONEY;

private

   type FINANCES is
     record
       Income   : MONEY := 0.00;
       Debt     : MONEY := 0.00;
     end record;

end Finance_Handler;
```

Private types must be defined within packages. Package Finance_Handler specifies each of the operations on objects of type FINANCES, while the type itself defines the internal data for each object. The *private part* of the package contains the definition of type FINANCES in terms of other Ada type constructs. In this case, objects of type FINANCES are effectively declared to have two instance variables, as in the Smalltalk example. (The private part of a package is logically part of the package implementation, not the specification. It is included in the specification only so that the compiler can tell from the specification alone how much space to allocate for objects of private types.) The package Finance_Handler is

in some ways similar to the *metaclass* of the Smalltalk class Finances. In Smalltalk, a metaclass is the class of a class. Both the metaclass and the handler package provide a framework for the definition of a class, and they also allow for the definition of class variables and class operations.

Since the declaration of instance variables is in the private part of the specification of Finance_Handler, the package body only needs to define implementations for each of the specified operations:

```ada
package body Finance_Handler is

   -- Instance creation
   function Initial (Balance : MONEY)
     return FINANCES is
   begin
     return
         ( Income     => Balance,
           Debt       => 0.00     );
   end Finance_Handler;

   -- Transactions
   procedure Receive
     ( Account   : in out FINANCES;
       Amount    : in MONEY ) is
   begin
     Account.Income := Account.Income
                       + Amount;
   end Receive;

   procedure Spend
     ( Account   : in out FINANCES;
       Amount    : in MONEY ) is
   begin
     Account.Debt := Account.Debt
                     + Amount;
   end Spend;

   -- Inquiries
   function Cash_On_Hand
     ( Account : FINANCES )
     return MONEY is
   begin
     return
         Account.Income - Account.Debt;
   end Cash_On_Hand;
```

```
function Total_Received
  ( Account : FINANCES )
  return MONEY is
begin
  return Account.Income;
end Total_Received;

function Total_Spent
  ( Account : FINANCES )
  return MONEY is
begin
  return Account.Debt;
end Total_Spent;
```

end Finance_Handler;

Each FINANCES operation explicitly includes an Account of type FINANCES as one of its parameters. The instance variables of an Account are then accessed using a qualified notation such as "Account.Income". This access to instance variables is only allowed *within the body* of package Finance_Handler. Some examples of the use of type FINANCES are:

```
declare

  My_Finances
    : Finance_Handler.FINANCES
    := Finance_Handler.Initial
         (Balance => 500.00);

begin

  Finance_Handler.Spend
    ( Account  => My_Finances,
      Amount   => 32.50 );
  Finance_Handler.Spend
    ( Account  => My_Finances,
      Amount   => Food_Cost + Sales_Tax );
  Finance_Handler.Receive
    ( Account  => My_Finances,
      Amount   => Pay );
  Tax := Tax_Rate
           * Finance_Handler.Total_Received
                            (My_Finances);

end;
```

Packages in Ada allow the definition of objects as program modules or the definition of classes as private types. Packages cannot themselves be passed as data, but the instances of private types can. It is also possible in Ada to define

classes of objects which cannot be passed as data. This is done using a *generic package* which serves as a template for instances of the class. For example, the earlier specification for package Finances can be made generic by simply adding the keyword **generic** at the beginning:

```
generic
package Finances is

  ...

end Finances;
```

Other packages can then be declared as *instantiations* of the generic package. For example:

```
declare

  package My_Finances is
    new Finances;

begin

  My_Finances.Receive (Amount => Pay);
  Cash := My_Finances.Cash_On_Hand;

end;
```

I will have more to say later on other important roles of generics in Ada.

3. Inheritance

A class represents a common abstraction of a set of entities, suppressing their differences. At a lower level of abstraction, some entities may differ from others. A *subclass* represents a subset of the entities of a class. A subclass *inherits* general abstract properties from its *superclass*, defining only the specific differences which appear at its lower level of abstraction. This technique of subclass inheritance allows the incremental building of application-specific abstractions from general abstractions.

Smalltalk directly supports the concept of subclassing and inheritance. In Smalltalk every class has a superclass, except for the system class Object which describes the similarities of all objects. Instances of a subclass are the same

as instances of the superclass except for differences explicitly stated in the subclass definition. The allowed differences are the addition of instance variables, the addition of new methods and the overriding of superclass methods. An instance of a subclass will respond to *at least* all of the same messages as instances of its superclass, though not necessarily in exactly the same way.

Ada does not provide direct support for subclassing or inheritance. However, the concept of inheritance can be used profitably within Ada, in some ways more generally than in Smalltalk. When defining a subclass in Ada, it is still necessary to declare *all* operations of that subclass, even those inherited from a superclass. Thus the specification of a subclass package will include all the operations of the superclass and possibly some additional ones. (This also results in a hiding of the use of inheritance reminiscent of the discussion in [Snyder 86].) In the body of the subclass package, inherited operations must be implemented as *call-throughs* to the operations of the superclass.

Example 2 -- Deductible Finances

The class DeductibleFinances is a subclass of the Finances class of Section 2. Instances of DeductibleFinances have the same functions as instances of Finances for receiving and spending money. However, they also keep track of tax deductible expenditures. The definition of DeductibleFinances specifies one new instance variable, four new instance methods and overrides two class methods:

class name	DeductibleFinances
superclass	Finances
instance variable names	deductibleDebt

class methods

instance creation

initialBalance: amount
^(super initialBalance: amount) zeroDeduction

new
^super new zeroDeduction

instance methods

transactions

spendDeductible: amount
self spend: amount deducting: amount.

spend: amount deducting: deductibleAmount
super spend: amount.
deductibleDebt <- deductibleDebt
                         + deductibleAmount

inquiries

totalDeduction
^deductibleDebt

private

zeroDeduction
deductibleDebt <- 0

Note that sending a message to "self" results in a call on one of an object's own methods, while sending a message to "super" results in a call on one of the methods of the superclass Finances.

Now consider an Ada type which defines a subclass of the FINANCES type of Section 2:

```
with Finance_Handler;
package Deductible_Finance_Handler is

  type DEDUCTIBLE_FINANCES is private;
  subtype MONEY is
    Finance_Handler.MONEY;

  -- Instance creation
  function Initial ( Balance : MONEY )
    return DEDUCTIBLE_FINANCES;

  -- Transactions
  procedure Receive
  ( Account : in out DEDUCTIBLE_FINANCES;
    Amount : in MONEY );
  procedure Spend
  ( Account : in out DEDUCTIBLE_FINANCES;
    Amount : in MONEY;
    Deductible_Amount : in MONEY := 0.00 );
  procedure Spend_Deductible
  ( Account : in out DEDUCTIBLE_FINANCES;
    Amount : in MONEY );
```

5207

```ada
-- Inquiries
function Cash_On_Hand
  ( Account : DEDUCTIBLE_FINANCES )
  return MONEY;
function Total_Received
  ( Account : DEDUCTIBLE_FINANCES )
  return MONEY;
function Total_Spent
  ( Account : DEDUCTIBLE_FINANCES )
  return MONEY;
function Total_Deduction
  ( Account : DEDUCTIBLE_FINANCES )
  return MONEY;


private


type DEDUCTIBLE_FINANCES is
  record
    Finances : Finance_Handler.FINANCES;
    Deductible_Debt : MONEY := 0.00;
  end record;

end Finance_Handler;
```

Package Deductible_Finance_Handler has the new operations Spend_Deductible and Total_Deductions, and it has a modified Spend operation. The Spend procedure has a Deductible_Amount parameter with a *default value* of 0.00.

DEDUCTIBLE_FINANCES implements inheritance from FINANCES by using the instance variable Finances of type FINANCES. Inherited operations are then implemented as call-throughs to operations on Finances:

```ada
package body Deductible_Finance_Handler is

  -- Instance creation
  function Initial ( Balance : MONEY )
    return DEDUCTIBLE_FINANCES is
  begin
    return
  ( Finances => Finance_Handler.Initial(Balance),
  Deductible_Debt => 0.00                  );
  end Initial;
```

```ada
-- Transactions
procedure Receive
( Account : in out DEDUCTIBLE_FINANCES;
  Amount : in MONEY ) is
begin
  -- INHERITED --
  Finance_Handler.Receive
    ( Account        => Account.Finances,
      Amount         => Amount           );
end Receive;

procedure Spend
( Account : in out DEDUCTIBLE_FINANCES;
  Amount : in MONEY;
  Deductible_Amount : in MONEY := 0.00 ) is
begin
  Finance_Handler.Spend
    ( Account        => Account.Finances,
      Amount         => Amount           );
  Account.Deductible_Debt
      := Account.Deductible_Debt
                      + Deductible_Amount;
end Spend;

procedure Spend_Deductible
( Account : in out DEDUCTIBLE_FINANCES;
  Amount : in MONEY ) is
begin
  Spend
    ( Account          => Account,
      Amount           => Amount,
      Deductible_Amount => Amount );
end Spend_Deductible;

-- Inquiries
function Cash_On_Hand
  ( Account : DEDUCTIBLE_FINANCES )
  return MONEY is
begin
  -- INHERITED --
  return Finance_Handler.Cash_On_Hand
                    (Account.Finances);
end Cash_On_Hand;

  .
  .
  .

function Total_Deductions
  ( Account : DEDUCTIBLE_FINANCES )
  return MONEY is
begin
  return Account.Deductible_Debt;
end Total_Deductions;

end Deductible_Finance_Handler;
```

Unlike Smalltalk, implementing inheritance in Ada requires an extra level of operation call. Also, in Ada the subclass does not have direct access to the instance variables of the superclass. The superclass package presents the same abstract interface to subclass packages as to any other code. This tightens the encapsulation of the superclass abstraction. It also allows easy extension to *multiple inheritance* where a subclass may inherit operations from more than one superclass. Multiple inheritance simply requires multiple superclass instance variables with inherited operations calling-through to the appropriate superclass operations. In this case the new class is really a *composite* abstraction formed from more general *component* classes.

The main drawback of this approach is that the Ada typing system does not recognize subclassing. In Ada all private types are distinct. Even though the type DEDUCTIBLE_FINANCES is logically a subclass of type FINANCES, the type DEDUCTIBLE_FINANCES is *not* a subtype of type FINANCES. It is not possible, for instance, to pass an instance of type DEDUCTIBLE_FINANCES to a procedure expecting an argument of type FINANCES. The Ada compiler would see this as a type inconsistency. A partial solution to this involves the use of the Ada generic facility, and will be discussed later in Section 4. However, the problem cannot be fully overcome in Ada, and [Meyer 86] clearly shows that true inheritance is more powrful than genericity.

4. Binding

The Smalltalk message passing mechanism operates dynamically. When a message is sent to a Smalltalk object, the method to respond to that message is looked-up at run-time in the object's class (and possibly superclasses). Further, Smalltalk variables are not typed, so they may contain objects of any class. Thus it is generally not possible to determine statically exactly what method in what class will respond to a message. Messages are *dynamically bound* to methods at run-time. If an object cannot respond to a message, there is a run-time error.

The use of dynamic binding gives the programmer great freedom to create general code. Any object can be used in an instance variable or as an argument in a message as long as it can respond to the messages sent to it. Another use of dynamic binding in Smalltalk is with the "pseudo-variable" "self" which is used by an object to send messages to itself. When a message is sent to an object, "self" is set to the object to which the message is sent. The dynamic binding of messages sent to "self" allows a class to call on methods that are really defined in a subclass.

Unlike Smalltalk, Ada is a strongly typed language. This means that all variables and parameters must be declared to be of a single specific type. This allows an Ada compiler to check statically that only values of the correct type are being assigned to variables and used as arguments. The Ada compiler can also always determine exactly what operation from what package (if any) is being invoked by a given call. Operation calls are thus *statically bound* to the proper operation. Undefined operation calls are always discovered at compile-time.

A way around this involves the use of generics. In addition to their role in creating classes of packages, generics also allow a package to be *parameterized* with type and subprogram parameters. This feature can be used to declare a package which can use *any* class with certain needed operations. Generic facilities can also be used to allow a class to defer the implementation of some operations to subclasses.

Example 3 -- Sample Space

The class SampleSpace represents random selection without replacement from a collection of items. It has the following protocol:

SampleSpace class protocol

instance creation

data: aCollection   Create an instance such that aCollection is the sample space.

5207

accessing

next   Answer the next element
                        chosen at random from
                        the sample space,
                        removing it from the
                        space.

next: anInteger   Answer an ordered
                        collection of anInteger
                        number of selections
                        from the sample space.

testing

isEmpty   Answer whether any
                        items remain to be
                        sampled.

size   Answer the number of
                        items remaining to be
                        sampled.

This protocol does *not* specify exactly what
kind of collection must be used for the sample
space. The class definition is:

class name	SampleSpace
superclass	Object
instance variable names	data
	rand

class methods

instance creation

  data: aCollection
  ^super new setData: aCollection.

instance methods

accessing

  next
  | item |
  self isEmpty ifTrue:
[self error 'no values exist in the sample space'].
  item <- data at:
          (rand next * data size) truncated + 1.
  data remove: item.
  ^item

next: anInteger
| aCollection |
aCollection
        <- OrderedCollection new: anInteger.
anInteger timesRepeat:
  [aCollection addLast: self next].
^aCollection

testing

  isEmpty
  ^self size = 0

  size
  ^data size

private

  setData: aCollection
  data <- aCollection.
  rand <- Random new

Note that local variables in methods are listed
between vertical bars at the beginning of the
method. Also, the definition of SampleSapce
uses an instance of the Smalltalk system class
Random to generate random numbers. In the
methods for "next" and "size", SampleSpace
sends the messages "at:", "size" and "remove:" to
the instance variable "data" which holds the
collection of sample space items. This means
that any object which can respond to "at:", "size"
and "remove:" can serve as the collection. This
object could be an instance of a Smalltalk
system class such as Array, or it could be an
instance of a user-defined class. An example of
the use of SampleSpace is shuffling a deck of
cards:

class name	CardDeck
superclass	Object
instance variable names	cards

  shuffle
  | sample |
  sample <- SampleSpace data: cards.
  cards <- sample next: cards size

An Ada generic Sample_Space package needs a
COLLECTION type and At, Size and Remove

operations. A specification for this package is:

```ada
generic

  type COLLECTION_TYPE is private;
  type ELEMENT_TYPE is private;

  with function At
    ( Collection : COLLECTION_TYPE;
      Index      : POSITIVE )
    return ELEMENT_TYPE;
  with function Size
    ( Collection : COLLECTION_TYPE )
    return ELEMENT_TYPE;
  with procedure Remove
    ( Collection : in out COLLECTION_TYPE;
      Element    : in ELEMENT_TYPE );

package Sample_Space is

  Empty : exception;

  type ELEMENT_LIST is
    array (NATURAL range <>)
      of ELEMENT_TYPE;

  -- Initialization
  procedure Set
    ( Data       : in COLLECTION_TYPE );

  -- Accessing
  function Next return ELEMENT_TYPE;
  function Next ( Number : NATURAL )
    return ELEMENT_LIST;

  -- Testing
  function Is_Empty return BOOLEAN;
  function Size return NATURAL;

end Sample_Space;
```

Package Sample_Space uses the generic facility both to parameterize itself and to allow a class of objects (as discussed in Section 2). It would also have been possible to define a generic Sample_Space_Handler package with a SAMPLE_SPACE type. This would have allowed sample spaces to be passed as data, an ability which is not really needed for the present example.

The body of Sample_Space is:

```ada
with Random;
package body Sample_Space is

  -- Instance variable
  Sample_Data : COLLECTION_TYPE;

  -- Initialization
  procedure Set
    ( Data       : COLLECTION_TYPE ) is
  begin
    Sample_Data := Data;
  end Set;

  -- Accessing
  function Next return ELEMENT_TYPE is
    Item : ELEMENT_TYPE;
  begin
    if Is_Empty then
      raise Empty;
    end if;
    Item := At ( Sample_Data, Index =>
        NATURAL((Random.Value*Size)+1) );
    Remove
      ( Collection      => Sample_Data,
        Element         => Item );
    return Item;
  end Next;

  function Next ( Number : NATURAL )
    return ELEMENT_LIST is
    List : ELEMENT_LIST(1 .. Number);
  begin
    for I in 1 .. Number loop
      List(I) := Next;
    end loop;
    return List;
  end Next;

  -- Testing
  function Is_Empty return BOOLEAN is
  begin
    return (Size = 0);
  end Is_Empty;

  function Size return NATURAL is
  begin
    return Size(Sample_Data);
  end Size;

end Sample_Space;
```

The Sample_Space package body assumes the availability of a package Random to generate random numbers. Sample_Space could then be

used to shuffle an instance of private type CARD_DECK:

```
with Sample_Space;
package body Card_Deck_Handler is

    ...

    package Sample is new Sample_Space
    ( COLLECTION_TYPE  => CARD_DECK,
      ELEMENT_TYPE     => CARD_TYPE,
      At               => Card,
      Size             => Deck_Size,
      Remove           => Remove_Card );

    ...

    procedure Shuffle
      ( Cards : in out CARD_DECK ) is
    begin
      Sample.Set (Data => Cards);
      Cards := CARD_DECK
               (Sample.Next(Deck_Size(Cards)));
    end Shuffle;

    ...

end Card_Deck_Handler;
```

Generic package Sample_Space is a template for a general class of sample spaces. Since a COLLECTION_TYPE must be specified when Sample_Space is instantiated, each instance of this class can only handle a *single* type of collection for sampling. Thus an Ada compiler can still perform static type checking for each instantiation of generic packages.

The dynamic binding and lack of typing in Smalltalk allow an instance of a subclass to be used anyplace an instance of its superclass may be used. As mentioned at the end of Section 3 the Ada type system does not allow this because it views all private types as distinct and incompatible. The above generic technique can help with this problem, also. A generic package (or other program unit) which is parameterized by the types and operations it needs will be able to use any type with the necessary operations. Thus if the private type representing some class can be plugged into a generic, then a subclass type can also be plugged into that same generic. However, the generic must be instantiated *separately* for each type. There is no easy way

in Ada have a true *polymorphic* procedure, that is, a *single* procedure with an argument which accepts values of different types.

5. Conclusion

Smalltalk and Ada are based on quite different philosophies. Smalltalk is designed to make it easier to program and to incrementally build and modify systems. Ada, on the other hand, purposefully places certain additional obligations on the programmer so that the final system will be more reliable and more maintainable. The Ada philosophy takes a much more life-cycle-oriented approach, recognizing that most costly phase of software development is maintenance, not coding.

If the languages have such different bases, then why consider using object-oriented ideas for Ada? The answer is that object-oriented concepts really apply to more than just programming. In Ada circles, these concepts are usually applied to design [Booch 83, Seidewitz 86a, Seidewitz 86b]. The object-oriented viewpoint is crucial to designing for effect use of Ada's package facility. Further, the object-oriented approach can be a general way of thinking about software systems which can be applied from system specification through testing. This fits in quite well with the Ada life-cycle philosophy [Booch 86, Stark 87].

Still, Ada has some unfortunate drawbacks for object-oriented programming, especially in its lack of support for inheritance. As an object-oriented programming language Smalltalk is in many ways clearly superior to Ada. However, as a life-cycle software engineering language Ada has great advantages. Static strong typing is crucial to increasing the reliability of software. Even with a good testing methodology, large amounts of code will not be thoroughly tested because it is only executed in rare combinations of situations. But when a system is running continuously for years, any errors that remain in these sections of code will almost certainly occur. This is especially true for the embedded real-time systems which were Ada's original mandate. In Ada, all sections of code are checked by the compiler, and many errors can be caught before the testing phase due to static type checking and static operation binding.

4-42

It is possible to support inheritance and even polymorphism within a statically typed language (as in, for example, Eiffel [Meyer 86, Meyer 87]). Inheritance might be added to Ada without too much change to the design of the language. Incorporation of polymorphism would be much more difficult, and probably require a philosophical change in the Ada language design. However, even with these deficiencies for object-oriented programming, Ada still provides a useful vehicle for applying object-oriented concepts throughout the software development life-cycle.

Much of the above discussion also applies to other modular languages such as Modula-2 (though Modula-2 does not directly support genericity). As these languages become more and more widely used it will be increasingly important to apply to them the experience in object-oriented software development gained from Smalltalk and other object-oriented languages.

References

[Booch 83]
Grady Booch. Software Engineering with Ada, Benjamin/Cummings, 1983.

[Booch 86]
Grady Booch. "Object-Oriented Development," IEEE Transactions on Software Engineering, February 1986.

[Dahl 68]
O-J Dahl. Simula 67 Common Base Language, Norwegian Computing Center, Oslo, Norway, 1968.

[DOD 83]
US Department of Defense, Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983.

[Goldberg 83]
Adele Goldberg and David Robson. Smalltalk-80: The Language and its Implementation, Addison-Wesley, 1983.

[Meyer 86]
Bertrand Meyer. "Genericity versus Inheritance," OOPSLA '86 Conference Proceedings, SIGPLAN Notices, November 1986.

[Meyer 87]
Bertrand Meyer. "Eiffel: Programming for Reusability and Extendability," SIGPLAN Notices, February 1987.

[Seidewitz 86a]
Ed Seidewitz and Mike Stark. "Towards an Object-Oriented Software Development Methodology," Proc. of the 1st Intl. Conf. on Ada Applications for the Spac Station, June 1986.

[Seidewitz 86b]
Ed Seidewitz and Mike Stark. General Object-Oriented Software Development, Goddard Space Flight Center, SEL-86-002, August 1986.

[Snyder 86]
Alan Snyder. "Encapsulation and Inheritance in Object-Oriented Programming Languages," OOPSLA '86 Conference Proceedings, SIGPLAN Notices, November 1986.

[Stark 87]
Mike Stark and Ed Seidewitz. "Towards a General Object-Oriented Ada Life-Cycle," Proc. of the Joint 4th Washington Ada Symposium / Fifth Nat. Conf. on Ada Technology, March 1987.

[Wirth 83]
Niklaus Wirth. Programming in Modula-2, Springer-Verlag, 1983.

5207

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 Engi-
neering 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 Engi-
neering Workshop, August 1976

SEL-77-002, Proceedings From the Second Summer Software En-
gineering 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 Engi-
neering 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: Relation-
ship 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 Pro-
gram Design Language (PDL) in the Goddard Space Flight Cen-
ter (GSFC) Code 580 Software Design Environment,
C. E. Goorevich, A. L. Green, and W. J. Decker, September
1979

9913

SEL-79-005, <u>Proceedings From the Fourth Summer Software En-</u>
<u>gineering Workshop</u>, November 1979

SEL-80-002, <u>Multi-Level Expression Design Language-</u>
<u>Requirement Level (MEDL-R) System Evaluation</u>, W. J. Decker
and C. E. Goorevich, May 1980

SEL-80-003, <u>Multimission Modular Spacecraft Ground Support</u>
<u>Software System (MMS/GSSS) State-of-the-Art Computer Systems/</u>
<u>Compatibility Study</u>, T. Welden, M. McClellan, and
P. Liebertz, May 1980

SEL-80-005, <u>A Study of the Musa Reliability Model</u>,
A. M. Miller, November 1980

SEL-80-006, <u>Proceedings From the Fifth Annual Software Engi-</u>
<u>neering Workshop</u>, November 1980

SEL-80-007, <u>An Appraisal of Selected Cost/Resource Estima-</u>
<u>tion Models for Software Systems</u>, J. F. Cook and
F. E. McGarry, December 1980

SEL-81-008, <u>Cost and Reliability Estimation Models (CAREM)</u>
<u>User's Guide</u>, J. F. Cook and E. Edwards, February 1981

SEL-81-009, <u>Software Engineering Laboratory Programmer Work-</u>
<u>bench Phase 1 Evaluation</u>, W. J. Decker and F. E. McGarry,
March 1981

SEL-81-011, <u>Evaluating Software Development by Analysis of</u>
<u>Change Data</u>, D. M. Weiss, November 1981

SEL-81-012, <u>The Rayleigh Curve As a Model for Effort Distri-</u>
<u>bution Over the Life of Medium Scale Software Systems</u>, G. O.
Picasso, December 1981

SEL-81-013, <u>Proceedings From the Sixth Annual Software Engi-</u>
<u>neering Workshop</u>, December 1981

SEL-81-014, <u>Automated Collection of Software Engineering</u>
<u>Data in the Software Engineering Laboratory (SEL)</u>,
A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-81-101, <u>Guide to Data Collection</u>, V. E. Church,
D. N. Card, F. E. McGarry, et al., August 1982

SEL-81-102, <u>Software Engineering Laboratory (SEL) Data Base</u>
<u>Organization and User's Guide Revision 1</u>, P. Lo and
D. Wyckoff, July 1983

9913

SEL-81-104, <u>The Software Engineering Laboratory</u>, D. N. Card,
F. E. McGarry, G. Page, et al., February 1982

SEL-81-106, <u>Software Engineering Laboratory (SEL) Document
Library (DOCLIB) System Description and User's Guide</u>,
W. Taylor and W. J. Decker, May 1985

SEL-81-107, <u>Software Engineering Laboratory (SEL) Compendium
of Tools</u>, W. J. Decker, W. A. Taylor, and E. J. Smith,
February 1982

SEL-81-110, <u>Evaluation of an Independent Verification and
Validation (IV&V) Methodology for Flight Dynamics</u>, G. Page,
F. E. McGarry, and D. N. Card, June 1985

SEL-81-203, <u>Software Engineering Laboratory (SEL) Data Base
Maintenance System (DBAM) User's Guide and System Descrip-
tion</u>, P. Lo, June 1984

SEL-81-205, <u>Recommended Approach to Software Development</u>,
F. E. McGarry, G. Page, S. Eslinger, et al., April 1983

SEL-82-001, <u>Evaluation of Management Measures of Software
Development</u>, G. Page, D. N. Card, and F. E. McGarry,
September 1982, vols. 1 and 2

SEL-82-003, <u>Software Engineering Laboratory (SEL) Data Base
Reporting Software User's Guide and System Description</u>,
P. Lo, August 1983

SEL-82-004, <u>Collected Software Engineering Papers: Vol-
ume 1</u>, July 1982

SEL-82-007, <u>Proceedings From the Seventh Annual Software
Engineering Workshop</u>, December 1982

SEL-82-008, <u>Evaluating Software Development by Analysis of
Changes: The Data From the Software Engineering Laboratory</u>,
V. R. Basili and D. M. Weiss, December 1982

SEL-82-102, <u>FORTRAN Static Source Code Analyzer Program
(SAP) System Description (Revision 1)</u>, W. A. Taylor and
W. J. Decker, April 1985

SEL-82-105, <u>Glossary of Software Engineering Laboratory
Terms</u>, T. A. Babst, F. E. McGarry, and M. G. Rohleder,
October 1983

9913

SEL-82-606, <u>Annotated Bibliography of Software Engineering Laboratory Literature</u>, S. Steinberg, November 1988

SEL-83-001, <u>An Approach to Software Cost Estimation</u>, F. E. McGarry, G. Page, D. N. Card, et al., February 1984

SEL-83-002, <u>Measures and Metrics for Software Development</u>, D. N. Card, F. E. McGarry, G. Page, et al., March 1984

SEL-83-003, <u>Collected Software Engineering Papers: Volume II</u>, November 1983

SEL-83-006, <u>Monitoring Software Development Through Dynamic Variables</u>, C. W. Doerflinger, November 1983

SEL-83-007, <u>Proceedings From the Eighth Annual Software Engineering Workshop</u>, November 1983

SEL-84-001, <u>Manager's Handbook for Software Development</u>, W. W. Agresti, F. E. McGarry, D. N. Card, et al., April 1984

SEL-84-002, <u>Configuration Management and Control: Policies and Procedures</u>, Q. L. Jordan and E. Edwards, December 1984

SEL-84-003, <u>Investigation of Specification Measures for the Software Engineering Laboratory (SEL)</u>, W. W. Agresti, V. E. Church, and F. E. McGarry, December 1984

SEL-84-004, <u>Proceedings From the Ninth Annual Software Engineering Workshop</u>, November 1984

SEL-85-001, <u>A Comparison of Software Verification Techniques</u>, D. N. Card, R. W. Selby, Jr., F. E. McGarry, et al., April 1985

SEL-85-002, <u>Ada Training Evaluation and Recommendations From the Gamma Ray Observatory Ada Development Team</u>, R. Murphy and M. Stark, October 1985

SEL-85-003, <u>Collected Software Engineering Papers: Volume III</u>, November 1985

SEL-85-004, <u>Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics</u>, R. W. Selby, Jr., May 1985

SEL-85-005, <u>Software Verification and Testing</u>, D. N. Card, C. Antle, and E. Edwards, December 1985

SEL-85-006, <u>Proceedings From the Tenth Annual Software Engineering Workshop</u>, December 1985

B-4

9913

SEL-86-001, <u>Programmer's Handbook for Flight Dynamics Software Development</u>, R. Wood and E. Edwards, March 1986

SEL-86-002, <u>General Object-Oriented Software Development</u>, E. Seidewitz and M. Stark, August 1986

SEL-86-003, <u>Flight Dynamics System Software Development Environment Tutorial</u>, J. Buell and P. Myers, July 1986

SEL-86-004, <u>Collected Software Engineering Papers: Volume IV</u>, November 1986

SEL-86-005, <u>Measuring Software Design</u>, D. N. Card, October 1986

SEL-86-006, <u>Proceedings From the Eleventh Annual Software Engineering Workshop</u>, December 1986

SEL-87-001, <u>Product Assurance Policies and Procedures for Flight Dynamics Software Development</u>, S. Perry et al., March 1987

SEL-87-002, <u>Ada Style Guide (Version 1.1)</u>, E. Seidewitz et al., May 1987

SEL-87-003, <u>Guidelines for Applying the Composite Specification Model (CSM)</u>, W. W. Agresti, June 1987

SEL-87-004, <u>Assessing the Ada Design Process and Its Implications: A Case Study</u>, S. Godfrey, C. Brophy, et al., July 1987

SEL-87-005, <u>Flight Dynamics Analysis System (FDAS) Build 3 User's Guide</u>, S. Chang et al., October 1987

SEL-87-006, <u>Flight Dynamics Analysis System (FDAS) Build 3 System Description</u>, S. Chang, October 1987

SEL-87-007, <u>Application Software Under the Flight Dynamics Analysis System (FDAS) Build 3</u>, S. Chang et al., October 1987

SEL-87-008, <u>Data Collection Procedures for the Rehosted SEL Database</u>, G. Heller, October 1987

SEL-87-009, <u>Collected Software Engineering Papers: Volume V</u>, S. DeLong, November 1987

SEL-87-010, <u>Proceedings From the Twelfth Annual Software Engineering Workshop</u>, December 1987

9913

SEL-88-001, <u>System Testing of a Production Ada Project: The GRODY Study</u>, J. Seigle and Y. Shi, November 1988

SEL-88-002, <u>Collected Software Engineering Papers: Volume VI</u>, November 1988

SEL-RELATED LITERATURE

Agresti, W. W., <u>Definition of Specification Measures for the Software Engineering Laboratory</u>, Computer Sciences Corporation, CSC/TM-84/6085, June 1984

[4]Agresti, W. W., V. E. Church, D. N. Card, and P. L. Lo, "Designing With Ada for Satellite Simulation: A Case Study," <u>Proceedings of the First International Symposium on Ada for the NASA Space Station</u>, June 1986

[2]Agresti, W. W., F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," <u>Program Transformation and Programming Environments</u>. New York: Springer-Verlag, 1984

[1]Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," <u>Proceedings of the Fifth International Conference on Software Engineering</u>. New York: IEEE Computer Society Press, 1981

[1]Basili, V. R., "Models and Metrics for Software Management and Engineering," <u>ASME Advances in Computer Technology</u>, January 1980, vol. 1

Basili, V. R., <u>Tutorial on Models and Metrics for Software Management and Engineering</u>. New York: IEEE Computer Society Press, 1980 (also designated SEL-80-008)

[3]Basili, V. R., "Quantitative Evaluation of Software Methodology," <u>Proceedings of the First Pan-Pacific Computer Conference</u>, September 1985

[1]Basili, V. R., and J. Beane, "Can the Parr Curve Help With Manpower Distribution and Resource Estimation Problems?," <u>Journal of Systems and Software</u>, February 1981, vol. 2, no. 1

[1]Basili, V. R., and K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," <u>Journal of Systems and Software</u>, 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," <u>Proceedings of the International Computer Software and Applications Conference</u>, October 1985

9913

[4]Basili, V. R., and D. Patnaik, <u>A Study on Fault Prediction and Reliability Assessment in the SEL Environment</u>, University of Maryland, Technical Report TR-1699, August 1986

[2]Basili, V. R., and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," <u>Communications of the ACM</u>, January 1984, vol. 27, no. 1

[1]Basili, V. R., and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," <u>Proceedings of the ACM SIGMETRICS Symposium/Workshop: Quality Metrics</u>, March 1981

[3]Basili, V. R., and C. L. Ramsey, "ARROWSMITH-P--A Prototype Expert System for Software Engineering Management," <u>Proceedings of the IEEE/MITRE Expert Systems in Government Symposium</u>, October 1985

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," <u>Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity, and Cost</u>. 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," <u>Proceedings of the Joint Ada Conference</u>, 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," <u>IEEE Transactions on Software Engineering</u>, June 1988

[2]Basili, V. R., R. W. Selby, and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," <u>IEEE Transactions on Software Engineering</u>, November 1983

[3]Basili, V. R., and R. W. Selby, Jr., "Calculation and Use of an Environments's Characteristic Software Metric Set," <u>Proceedings of the Eighth International Conference on Software Engineering</u>. New York: IEEE Computer Society Press, 1985

9913

Basili, V. R., and R. W. Selby, Jr., <u>Comparing the Effective-ness of Software Testing Strategies</u>, University of Maryland, Technical Report TR-1501, May 1985

[4]Basili, V. R., R. W. Selby, Jr., and D. H. Hutchens, "Experimentation in Software Engineering," <u>IEEE Transactions on Software Engineering</u>, July 1986

[5]Basili, V. and R. Selby, "Comparing the Effectiveness of Software Testing Strategies," <u>IEEE Transactions on Software Engineering</u> (in press)

[2]Basili, V. R., and D. M. Weiss, <u>A Methodology for Collecting Valid Software Engineering Data</u>, 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," <u>IEEE Transactions on Software Engineering</u>, November 1984

[1]Basili, V. R., and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives," <u>Proceedings of the Fifteenth Annual Conference on Computer Personnel Research</u>, August 1977

Basili, V. R., and M. V. Zelkowitz, "Designing a Software Measurement Experiment," <u>Proceedings of the Software Life Cycle Management Workshop</u>, September 1977

[1]Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," <u>Proceedings of the Second Software Life Cycle Management Workshop</u>, August 1978

[1]Basili, V. R., and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," <u>Computers and Structures</u>, August 1978, vol. 10

Basili, V. R., and M. V. Zelkowitz, "Analyzing Medium Scale Software Development," <u>Proceedings of the Third International Conference on Software Engineering</u>. New York: IEEE Computer Society Press, 1978

[5]Brophy, C., W. Agresti, and V. Basili, "Lessons Learned in Use of Ada-Oriented Design Methods," <u>Proceedings of the Joint Ada Conference</u>, 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," <u>Proceedings of the Washington Ada Technical Conference</u>, March 1988

9913

[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

[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

[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

9913

[6]Jeffery, D. R., and V. R. Basili, "Validating the TAME Resource Data Model," <u>Proceedings of the Tenth International Conference on Software Engineering</u>, 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," <u>Proceedings of the 22nd Annual Hawaii International Conference on System Sciences</u>, January 1989

[5]McGarry, F. and W. Agresti, "Measuring Ada for Software Development in the Software Engineering Laboratory (SEL)," <u>Proceedings of the 21st Annual Hawaii International Conference on System Sciences</u>, January 1988

[3]McGarry, F. E., J. Valett, and D. Hall, "Measuring the Impact of Computer Resource Quality on the Software Development Process and Product," <u>Proceedings of the Hawaiian International Conference on System Sciences</u>, January 1985

[3]Page, G., F. E. McGarry, and D. N. Card, "A Practical Experience With Independent Verification and Validation," <u>Proceedings of the Eighth International Computer Software and Applications Conference</u>, 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," <u>Proceedings of the Eighth International Conference on Software Engineering</u>. New York: IEEE Computer Society Press, 1985

[5]Rombach, H. D., "A Controlled Experiment on the Impact of Software Structure on Maintainability," <u>IEEE Transactions on Software Engineering</u>, March 1987

[6]Rombach, H. D., and V. R. Basili, "Quantitative Assessment of Maintenance: An Industrial Case Study," <u>Proceedings from the Conference on Software Maintenance</u>, September 1987

[6]Rombach, H. D., and L. Mark, "Software Process and Product Specifications: A Basis for Generating Customized SE Information Bases," <u>Proceedings of the 22nd Annual Hawaii International Conference on System Sciences</u>, January 1989

9913

[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

Stark, M., and E. Seidewitz, "Towards a General Object-Oriented Ada Lifecycle," _Proceedings of the Joint Ada Conference_, March 1987

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

9913

[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.

9913