SOFTWARE ENGINEERING LABORATORY SERIES

SEL-91-005

COLLECTED SOFTWARE ENGINEERING PAPERS: VOLUME IX

NOVEMBER 1991

(NASA-TM-105508) COLLECTED SOFTWARE ENGINEERING PAPERS, VOLUME 9 (NASA) 112 p CSCL 09B

N92-18126

Unclas

G3/61 0068918

NASA

National Aeronautics and Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

SOFTWARE ENGINEERING LABORATORY SERIES SEL-91-005

COLLECTED SOFTWARE ENGINEERING PAPERS: VOLUME IX

NOVEMBER 1991

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 1976 and has three primary organizational members:

NASA/GSFC, Systems Development Branch

University of Maryland, Department of Computer Science

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.

Single copies of this document can be obtained by writing to

Systems Development Branch
Code 552
Goddard Space Flight Center
Greenbelt, Maryland 20771

[illegible] ii INTENTIONALLY BLANK

PRECEDING PAGE BLANK NOT FILMED

Table of Contents

[illegible] iv [illegible]

10000174

PRECEDING PAGE BLANK NOT FILMED

SECTION 1—INTRODUCTION

This document is a collection of selected technical papers produced by participants in the Software Engineering Laboratory (SEL) from November 1990 through October 1991. 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 ninth 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 eight papers contained here are grouped into three major categories:

- Software Models Studies
- Software Measurement Studies
- Ada Technology Studies

The first category presents studies on reuse models, including a software reuse model applied to maintenance and a model for an organization to support software reuse. The second category includes experimental research methods and software measurement techniques. The third category presents object-oriented approaches using Ada and object-oriented features proposed for Ada.

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

SECTION 2—SOFTWARE MODELS STUDIES

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

- "Software Reuse: A Key to the Maintenance Problem," H. D. Rombach, *Butterworth Journal of Information and Software Technology*, January/ February 1991
- *Support for Comprehensive Reuse*, V. R. Basili and H. D. Rombach, University of Maryland, Technical Report TR-2606, February 1991
- *A Reference Architecture for the Component Factory*, V. R. Basili, G. Caldiera, and G. Cantone, University of Maryland, Technical Report TR-2607, March 1991
- *A Pattern Recognition Approach for Software Engineering Data Analysis*, L. C. Briand, V. R. Basili, and W. M. Thomas, University of Maryland, Technical Report TR-2672, May 1991

Software reuse: a key to the maintenance problem

H D Rombach

Software maintenance is defined as the performance of all activities required to keep a software system operational and responsive after delivery. This includes adaptations to changing requirements and corrections of faults. Today, most practical maintenance environments have to spend a larger portion of their maintenance budget on fault corrections than they should have to, leaving not enough resources to respond to changing requirements properly. As a result, maintenance is widely viewed as an undesirable add-on to development. Improvement needs to be aimed at reducing the overall maintenance portion related to fault corrections and increasing the productivity and quality of the remaining maintenance activities. Maintenance is inherently reuse-oriented. It seems natural to explore how maintenance could benefit from recent advances in the area of software reuse. The article points out a number of crucial maintenance problems, presents a comprehensive framework that has been proposed to address similar problems in the area of reuse, and suggests how software maintenance may benefit from results produced in the reuse community by adopting a reuse-oriented framework.

software development, software maintenance, software reuse, software costs, process models

Software maintenance is defined as the performance of all activities required to keep a software system operational and responsive after it is accepted and placed into production[1]. Maintenance activities can be categorized as being perfective (i.e., triggered by a change of requirements), adaptive (triggered by a change of the operational environment), or corrective (i.e., triggered by the detection of a software failure* or fault)[2]. In an ideal maintenance world, where failure-free systems are maintained according to sound maintenance processes, no corrective maintenance is needed. Perfective and adaptive maintenance would be aimed at maintaining or increasing the economic value of an existing software system in the light of changing requirements and environment characteristics.

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

*The terms error fault failure are used according to the IEEE definition[3]: Humans commit errors; errors get manifested as faults in a software product; faults in turn may cause a deviation of the system's actual behaviour from its specified or intended behaviour. The term defect refers to any of the above.

In the real world, fault-prone and inappropriately documented systems are maintained according to *ad hoc* and error-prone maintenance processes. As a result, most maintenance environments have to spend a larger portion of their maintenance budget on corrective maintenance than they should have to, leaving not enough resources to respond to important perfective or adaptive requests properly. In many environments, the maintenance portion of the overall software budget had to be increased to the point that much needed new developments had to be delayed[4]. All the above has contributed to the perception that maintenance is an undesirable, non-creative, non-rewarding add-on to software development.

The maintenance improvement goal should not primarily be to minimize the overall amount of maintenance; instead the goal should be to minimize the number of necessary corrective maintenance actions and to maximize the effectiveness of each remaining (predominantly perfective and adaptive) maintenance activity.

As early as 1976 maintenance or maintenance-related problems had been reported. Belady and Lehman described the problems with the evolution of large software systems[5]. Mills suggested that large numbers of faults and or long-lived faults (i.e., faults that remain in the system a long time before being detected) spell trouble for maintenance[6]. Today, it is common software engineering knowledge that modifications of fault-prone systems tend to be error-prone themselves (i.e., they are likely to create even more faults) and corrections of faults become harder and more costly the longer they remain in an evolving system[4]. There is also overwhelming evidence that inappropriate documentation is the cause of many maintenance requests[7]. Why is it then that low-quality systems are still accepted into maintenance and maintenance still performed according to *ad hoc* processes? Maybe understanding of the quantity of these maintenance problems and their underlying causes is only beginning, no simple solutions to these problems exist, and the existing negative maintenance perception does not help attract good professionals into maintenance to facilitate change.

One of the central causes for many maintenance problems is the lack of good maintenance product and process models and understanding of their dependencies. A maintenance product model describes the product characteristics needed to perform effective maintenance.

A maintenance process model describes the individual steps necessary to satisfy an individual maintenance request. Different process models may require different product models and vice versa. Different types of maintenance requests, environment characteristics, and/or budgetary constraints may suggest different process and product models. Basili has discussed a number of alternate maintenance process models in detail[8]. No single, generally applicable, solution exists because all maintenance environments are different[9]. Each environment has to build its own maintenance models that reflect the weaknesses and strengths of its specific maintenance approach and improve from there. Many environments are still not willing to accept this challenge, not realising that the alternative is continued chaos in the short run and, maybe, loss of competitiveness in the long run. In a recent study initial models of the maintenance environment at NASA's Software Engineering Laboratory (SEL) have been built and several weaknesses and strengths have been identified through the use of measurement[10]. As important as improved technical maintenance approaches are good professionals who can apply these approaches effectively. To compete with development for good professionals, comparable reward structures and career opportunities are needed for maintenance.

Viewing maintenance as a reuse-oriented activity is natural and promising. It is natural because maintenance and reuse have many characteristics in common and share many fundamental problems. Both can be viewed as creating something 'new' from something 'old'. The 'old' and 'new' tend to refer to the same system in the case of maintenance, to (components of) different systems in the case of reuse. In both cases one of the central questions is: What do I need to know of the 'old' to create the 'new' effectively. A comprehensive framework that addresses this question for reuse has been developed[11]. The adoption of a reuse-oriented maintenance view promises not only to solve some of the notorious maintenance problems, but also to integrate development and maintenance more naturally. After all, why is the development of a new system Y whose requirements are formulated as 'same requirements as a previous system X, except for requirement R1' being treated as necessarily different from a maintenance task for system X triggered by a request 'change requirement R1'?

This article points out a number of crucial maintenance problems (second section), presents a comprehensive framework that has been proposed to address similar problems in the area of reuse (third section), and suggests how software maintenance may benefit from results produced in the reuse community by adopting a reuse-oriented view (fourth section).

SOFTWARE MAINTENANCE PROBLEMS

In this section, serious software maintenance problems are identified. They are derived from a characterization of the state-of-the-practice in software maintenance covering maintained products, maintenance processes, and maintenance personnel. The characterization is based on the author's experience from studying industrial maintenance environments (e.g., NASA's Software Engineering Laboratory (SEL)[10], Burroughs Corporation[7]) and the experiences of others[12-14]. Examples from a maintenance study in NASA's SEL will be used throughout this section (in italic font) for illustration purposes[10].

Software systems passed into production and maintenance are typically packaged based on a development perspective. No widely accepted product models have been developed from a maintenance perspective. As a result developers do not understand what maintainers really need and maintainers do not have sound criteria to certify a delivered system as being fit for maintenance or not. Typical complaints by maintainers are that inappropriate or redundant information is passed to them (e.g., *PDL is really useless because it provides the same information at the same level of abstraction as does well-commented source code*), important information is missing (e.g., *design rationale is missing, traceability between requirements and design or code components is not clear, debug code assumes intimate familiarity with the source code*), the form of information is not tailored to maintenance needs (e.g., *global information is encoded redundantly in multiple Fortran common blocks resulting in frequent inconsistencies*), and the quality of the system is not adequate (e.g., *too many development faults, inconsistent configurations*). All these problems require unnecessary resources either to improve the delivered information or to recreate it (i.e., reverse engineering). In general, not all problems with the originally delivered system can be resolved during maintenance. Remaining problems are the source for unproductive maintenance and further faults. Having it right at the time of delivery is the only promising approach (e.g., *about 25% of all maintenance effort in the SEL is related to fault correction and could be saved*). To achieve this goal, maintenance-oriented product models need to be devised that establish sound criteria for accepting software into maintenance.

Maintenance activities can be categorized as being perfective (i.e., triggered by a change of requirements), adaptive (triggered by a change of the operational environment), or corrective (i.e., triggered by the detection of a software failure or fault)[5]. Under the idealistic assumption that high-quality software systems are delivered into production and maintenance, maintenance should be dominated by perfective and adaptive activities. For example, software systems controlling the production line of a car manufacturing plant need to be adapted to changes in the production process. It should be viewed as an advantage of software over hardware to be (at least potentially) easier to change. High maintenance costs are not necessarily a bad sign as long as they are the result of a large number of perfective maintenance requests and not their inefficient completion. It may simply reflect the speed at which car manufacturing technology advances. Advantage should be taken of the fact that software is (at least in theory) more easily adaptable. Bob Glass once said provocatively: 'Software maintenance is not a problem, but a solution'[15]. The author prefers the following slightly modified version: 'EFFECTIVE software maintenance is not a problem, but a solution'.

So the ideal maintenance scenario would consist of (almost) no corrective maintenance and any number of perfective and adaptive maintenance changes — each change performed at 'minimal' cost. Reality is different. Many environments have a larger corrective maintenance portion than necessary (e.g., in the SEL, *more than 50% of the completed maintenance requests were found to be corrections; however, they were less complex than other changes consuming only little more than 25% of the overall maintenance resources*). The large number of corrective maintenance requests stemming from low system quality or inappropriate maintenance processes typically does not allow resolving many of the 'real' maintenance requests. Keeping the system functional takes precedence over adaptation to changing needs. Even worse, much needed new software systems cannot be developed because the necessary resources are tied up in maintenance[4]. In addition, less sophisticated practices, methods, and tools are used for corrective maintenance than for fault correction during development. The result is a degradation of maintenance as a whole. Maintenance is perceived as an undesirable add-on to development that does not have merits of its own.

Relative to the overall life-cycle cost associated with large-scale software products, maintenance is reported to consume significantly more than 50% of all life-cycle resources[4,16,17]. In light of the above discussion of the role of maintenance, the author believes the current emphasis on reducing the overall cost of maintenance is wrong. Of course attempts should be made to reduce the cost of corrective maintenance to the degree possible by not allowing low-quality systems into maintenance in the first place. However, it may not be right to attempt to reduce the overall amount of perfective and adaptive maintenance; instead, it should be attempted to improve the ability to satisfy each single request more effectively.

A general software maintenance process model consists of all (or a subset) of the following seven steps:

- detect need for modifications
- understand need for modifications
- isolate necessary modifications
- design modifications
- implement modifications
- test modifications
- release modifications

Each of these tasks requires knowledge regarding a number of issues related to the maintained product. Detecting the need for a modification requires an understanding of the product's functional and behavioural characteristics in comparison to the needs of its external environment. Understanding the need for modification requires an understanding of the potential implications of a detected discrepancy between expected functionality and actual functionality to make a decision whether it can be tolerated or should lead to a maintenance request. Isolating the necessary modifications requires a good understanding of the relationship between requirements and design and implementation to pinpoint the components that need to be changed. At this point the cost of a change can be estimated. Designing the modification requires a detailed understanding of the actual implementation to devise all changes without creating unintended side-effects that may result in follow-up problems. Implementing the modifications requires an understanding of the coding standards to maintain implementation consistency; beyond that only local understanding of the algorithms is needed. Testing the modified system can be done effectively, if previous tests are known and can be re-run with or without modifications depending on the modifications. The needs for integration, system, and acceptance tests depends on the structure and use of the product in question. Releasing the modifications requires an understanding of the current use of the system to be able to update the running version with the modified version with minimal interference. Practical software maintenance process models tend to be much more *ad hoc* and are performed without easy access to the appropriate information. In addition, the maintenance technology level (i.e., methods for design or code reading, automated tools, computer time) is much lower than during development (e.g., *in the SEL, maintenance shares computer resources with groups in charge of development and operations. Maintenance has the lowest access priority.*). The result of all the above is unproductive and error-prone maintenance.

A system's maintenance personnel are typically not its original developers, more junior than development personnel, lacking ownership of the product they are maintaining, and missing the same career opportunities that developers have. When a product's maintainers are different from its original developers, the maintainers entirely depend on the information (system plus documentation) passed to them at the time of delivery. Many organizations use maintenance to train their junior personnel on-the-job. This means that maintainers on average are less experienced with both the application domain and solution domain of the products to be maintained. In summary, less experienced personnel are expected to maintain badly documented software products. And there is no real incentive for maintainers to push for improvements because, after all, they are only maintaining someone else's product. A typical quote is 'I only fix, I don't create software'. On the contrary, there is an incentive to leave maintenance as quickly as possible because of lack of maintenance career opportunities. A typical quote to this effect is 'I cannot move up in my company as a maintainer'.

SOFTWARE REUSE MODEL

Recently, software reuse has re-emerged as a promising approach to improve the quality and productivity of software development. The encouraging difference to

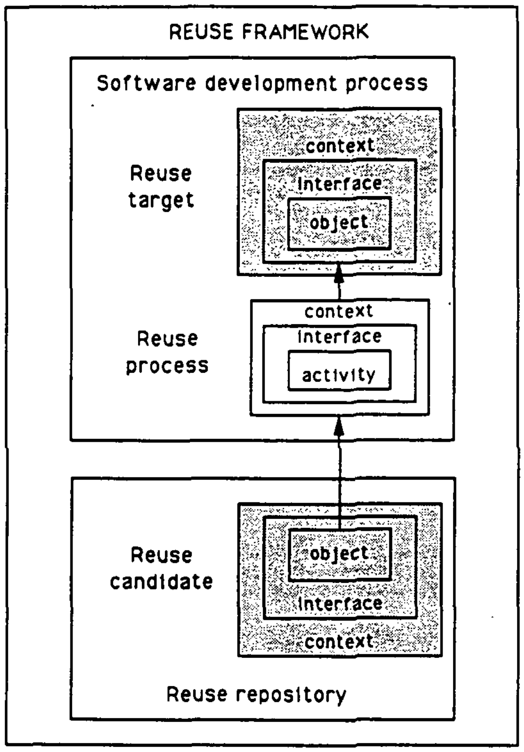


Figure 1. Reuse framework

previous emergences is that people seem to have learned a few lessons:

- it is not sufficient to focus on code reuse
- how to package reuse candidates in a reuse repository appropriately depends on their intended use in future projects and the actual reuse process itself
- reuse is enabled by some development process and both reusable components and employed reuse process need to be tailored to and integrated into that development process model

A framework and supporting characterization scheme has been developed to increase understanding, analysis, motivation, and improvement of reuse issues reflecting the above lessons[11].

This framework (see Figure 1) distinguishes between the reuse target (or reuse specification), the reuse candidate, and the reuse process.

Both the reuse target (or reuse specification) as well as each reuse candidate available in some repository are characterized in terms of the object itself, its interface with other objects, and the context knowledge required to understand it.

The object itself is characterized in terms of name of the object (e.g., component name 's_buffer.ada'), function performed by the object (e.g., string buffer), type of the object (e.g., source code), granularity of the object (e.g., package), and representation (e.g., Ada).

The interface is characterized in terms of input/output (e.g., less than six formal input and output parameters) and other dependencies (e.g., the assumption that the user knows Ada).

The context is characterized in terms of the application domain for which the object was originally developed or is intended to be reused (e.g., business software for banking), the solution domain in which the object was originally developed or is intended to be reused (e.g., waterfall life-cycle model), and the quality exhibited by the object (e.g., less than 1 fault per 1KLOC detected during development, documented according to certain standards).

The reuse process is assumed to include the following five basic steps:

- understanding of a given reuse target (or specification)
- identification of reuse candidates (based on matching certain key characteristics of the reuse specification with the set of available reuse candidates)
- evaluation of the reuse potential of each candidate (by predicting the cost related to bridging the discrepancies between each candidate and the given reuse specification) and selection of the best suited object if any
- modification of the selected object (by making the object satisfy the given reuse specification)
- integration of the modified object into the ongoing development

Each reuse activity is characterized in terms of the activity itself, its interface with the enabling development process, and the necessary context.

The activity itself is characterized in terms of name (e.g., unique activity name), function (e.g., a functional description of the activity), type (e.g., modification), and mechanisms (e.g., parameterized in the case of modification).

The activity interface is characterized in terms of input/output (e.g., input criteria and output criteria for performing the activity) and dependencies (e.g., phase of modification).

The activity context is characterized in terms of experience transfer (e.g., candidate packages are provided through a repository) and reuse quality (e.g., high reliability).

The following partial example is intended to demonstrate the usefulness of the above scheme to describe a specific reuse scenario involving Ada packages. The suggested five-step reuse process is followed:

- Understanding. The existence of a reuse specification corresponding to the example categories used above is assumed. The objective is to develop a string buffer package in Ada.
- Identification. All object characteristics of the reuse specification except 'name' are used as search criteria to identify candidates in the reuse repository. Pretend that three candidates were found, all of which satisfy

all search criteria except for 'function'. The first is an integer buffer, the second a string list, and the third a generic buffer.
- Evaluation. Next, evaluate the reuse potential of each identified candidate. The aim is to be able to predict the amount of resources needed to transform each candidate into a package that satisfies the given reuse specification entirely. Pretend that all three candidates satisfy all the interface and context characteristics of the reuse specification except 'input/output' and 'quality'. All three candidates have a higher than desired input/output interface and provide no information about faults detected during development. It depends on the mechanisms used for modification which of the three candidates is viewed closest with respect to 'function'. If the use of Ada's generic instantiation mechanism is assumed, the third candidate (the generic buffer) seems closest; if manual modification is assumed, the first candidate (the integer buffer) seems closest. Regarding the reduction of the input/output interface, again it needs to be known whether manual or automated modification will be used. In addition, quantitative information about the success of prior similar attempts and the cost involved would be helpful. With respect to development quality (i.e., number of faults detected during development), no information exists at all. That means that an uncalculable risk may have to be accepted. If, under these circumstances, there is willingness to reuse any of the candidates, the third candidate (generic Ada package) may be the most likely selection.
- Modification. The generic instantiation mechanism of Ada allows low-cost and reliable modification to create the desired Ada string buffer package.
- Integration. The integration of the newly created Ada string buffer package needs to follow the procedures established within the enabling development process: compilation, binding, integration testing, etc.

It is obvious that the presented reuse characterization does not improve reuse *per se*. However, it enables reasoning about the implications of reusing a particular software component to satisfy a given reuse specification using a given reuse process by pointing out important issues. To the degree future reuse specifications are known, the scheme can help establish criteria for components accepted into a reuse repository. There is no point in populating reuse repositories with components that violate characteristics that are non-negotiable in a particular environment. For example, if it is anticipated that all systems built in the future require a minimal reliability level, it would make no sense to pollute a repository with less reliable components.

Reuse is complicated by the fact that the personnel attempting to reuse existing candidates are typically different from the ones who originally created them. That makes the reuser totally dependent on the explicit information packaged together with the reuse candidates. Reuse can be facilitated in different ways:

- Packaging reuse candidates together with the appropriate information.
- Re-creating lost information post mortem — either at the time of entering the candidate into a repository or at every time it is attempted to be reused.
- Have the creators of reuse candidates be involved in reusing them. Most industrial reuse success stories are mainly due to stable personnel, meaning that reusers of software objects are either their original creators or at least have easy access to the original creators.

The scope of reuse has been expanded in that consideration is given not only to experience in the form of products, but also in the form of processes and other reusable knowledge[11]. The above reuse scheme has proved to be useful in defining packaging schemes for such comprehensive software engineering experiences bases. They have also been useful in defining entry criteria for including objects into an experience base or transforming objects within an experience base to increase its reuse potential[18]. The classification scheme is also being used in a project aimed at developing a 'code factory'[19].

REUSE-ORIENTED MAINTENANCE

It is obvious that software maintenance is a particularly intense form of reuse[8]. In the study of the NASA SEL maintenance environment it was observed that the relationship between modified and newly developed modules was about 50:1. No components were added during corrective and adaptive maintenance. And even perfective maintenance had a surprisingly low percentage of newly developed components — less than 5% — in the presence of rather time-consuming changes. That, of course, implies that changes of reused components must have been substantial. Now what kind of reuse takes place during maintenance will be explored in more detail.

Traditionally, reuse during maintenance is limited to reusing parts of the system at hand. This may range from reusing just source code to reusing the entire system (i.e., including all documentation). Basili has discussed in much detail example reuse process models that correspond to these two extremes: the quick-fix process model and the iterative-enhancement model[8]. According to the former, the response to a maintenance request consists primarily of changing the code. Other forms of documentation are updated if time permits. According to the latter, another evolutionary cycle through the entire development process is performed starting with changing the requirements (if needed) all the way down to changing source code. Each approach may be appropriate under certain circumstances. The quick-fix model may be appropriate to perform a fault correction for a system that will be retired soon and is not safety-critical by a maintainer who is intimately familiar with the system. Another justification for the quick-fix model can be a tight budget, provided the implications on future maintenance are understood. The iterative-enhancement model may be appropriate to perform an additive requirements change (i.e., one that does not require a general overhaul of the existing system architecture) to a system that needs to stay in production for some time.

However, the limitation to reuse only parts of the existing system can often be viewed as an unnecessary constraint, even in the case of complicated corrective maintenance requests. For example, a corrective maintenance request can be imagined that suggests a major change of the overall system architecture (e.g., to improve system performance significantly) or a perfective maintenance request that suggests the addition of several new components. In either case, it may be worth while to explore the possibility of reusing components from previous versions of the same system or different systems within the same application domain. It is unlikely that this expanded form of reuse can work without access to a reuse repository, appropriate packaging of interface and context information together with those systems, etc. Basili refers to this form of maintenance process model as the full-reuse model[8].

The full-reuse maintenance model is totally compatible with what has been suggested for reuse in general. This is reflected in the similarities between the general process models suggested for maintenance and reuse in the second and third sections, respectively. Both maintenance and reuse are aimed at creating new objects from old objects. Why should there be a technical difference between the development of a new string buffer whose requirements are formulated as 'same requirements as integer buffer, except that the base element type should be string' and the performance of a perfective maintenance request asking for the 'modification of an existing Ada integer buffer package to handle strings instead of integers'? The differences between reuse-oriented maintenance and development may be diminishing. The main remaining difference is that development of a system should be performed on a fixed budget, maintenance on a variable budget depending on the number of perfective and adaptive maintenance tasks — or should it be named 'mini-development'?

However, it would be wrong to impose any one of the discussed reuse process models. On the contrary, the reuse-oriented framework — as introduced in the third section — supports the identification of the most effective form of reuse (e.g., quick-fix, iterative-enhancement, full-reuse, or any other form of reuse) for any type of maintenance request, environment characteristics, and budgetary constraints. That includes — according to the proposed reuse model — the identification of reuse candidates suited to satisfy the maintenance request, the assessment of each candidate and selection of the best one, modification of the selected candidate if necessary, and its integration into the existing system. Each maintenance request is transformed into a change specification. If, for example, a maintenance request results in a requirements specification, there is a choice of either reusing the existing system architecture and modifying it accordingly or checking the repository for an architectural design that may be better suited. In this case, the characterization of the desired architectural design needs to be matched against the characteristics of any architectural design available in the repository. Of course, each candidate has to be assessed as to how likely it is that the cost of modification will not exceed the cost of changing the existing architectural design. Hopefully, enough information is packaged together with each candidate to assess the implications of reusing it.

There is also economical pressure to integrate development and maintenance under one umbrella concept of reuse. As the demand for new software systems is increasing faster than the ability to develop them[4], it seems to be appropriate to consider requirements for new systems and major maintenance requests to change existing systems at a par. Furthermore, as software development technology moves towards more extensive reuse, it is no longer tolerable to have *ad hoc* maintenance processes decrease the quality — and thereby the future reuse potential — of existing systems.

RELATED RESEARCH AT UNIVERSITY OF MARYLAND

Reuse-related work at the University of Maryland is currently concentrating on reuse and maintenance both from a theoretical and practical perspective. Theoretically, the presented reuse model and related characterization schemes have to be further refined and necessary reuse support mechanisms defined[11,18]; an experience base scheme is being designed to hold all types of experience reused in a software development organization such as processes, products, and other knowledge; and formal languages are being developed to represent individual process, product, and quality models (joint work with Victor Basili and others in the TAME project[20]). Practically, real-world maintenance scenarios are being analysed with the objective to improve the ease with which existing systems can be reused (e.g., in the Software Engineering Laboratory at NASA/Goddard Space Flight Center[10,21,22]); and prototype implementations of the TAME experience base concept (work going on in Victor Basili's research group), an experimental process modelling language (the author's research group), and a source code-oriented reuse environment (work by Victor Basili and Gianluigi Caldiera in the CARE project[19]) are being developed.

ACKNOWLEDGEMENT

Research for this study was in part funded by NASA grant NSG-5123 to the University of Maryland.

REFERENCES

1 **Federal Information Processing Standards** *Guideline on software maintenance* US Dept of Commerce/National Institute of Standards and Technology, FIPS PUB 106 (June 1984)

2 **IEEE** 'IEEE standard glossary of software engineering terminology' *Rep. IEEE-Std-729-1983* IEEE (1983)

3 **Swanson, E B** The dimensions of software maintenance' in

Proc. 2nd IEEE Int. Conf. Software Engineering (October 1976) pp 492–497

4 **Boehm, B W and Papaccio, P N** 'Understanding and controlling software costs' *IEEE Trans. Soft. Eng.* Vol 14 No 10 (October 1988) pp 1462–1477

5 **Belady, L and Lehman, M** 'A model of large program development' *IBM Syst. J.* Vol 15 No 3 (1976) pp 225–252

6 **Mills, H D** 'Software development' *IEEE Trans. Soft. Eng.* Vol 13 No 12 (December 1976) pp 265–273

7 **Rombach, H D and Basili, V R** 'Quantitative assessment of maintenance: an industrial case study' in *Proc. IEEE Conf. Software Maintenance* Austin, TX, USA (September 1987) pp 134–143

8 **Basili, V R** 'Viewing maintenance as reuse-oriented software development' *IEEE Software* Vol 7 No 1 (January 1990) pp 19–25

9 **Basili, V R and Rombach, H D** 'Tailoring the software process to project goals and environments' in *Proc. 9th IEEE Int. Conf. Software Engineering* Monterey, CA, USA (March 1987) pp 345–357

10 **Rombach, H D, Ulery, B T and Valett, J** 'Measurement based improvement of maintenance in the SEL' in *Proc. 14th Annual Software Engineering Workshop* NASA/Goddard Space Flight Center, Greenbelt, MD, USA (November 1989)

11 **Basili, V R and Rombach, H D** 'Towards a comprehensive framework for reuse: model-based reuse characterization schemes' *Technical report* University of Maryland, USA (April 1990), to be published in *Soft. Eng. J.* (July 1991)

12 *Proc. IEEE Conf. Software Maintenance* IEEE (1985, 1987, 1988, and 1989)

13 **Schneidewind, N F (ed)** 'Special section on software maintenance' in *Proc. IEEE* Vol 77 No 4 (April 1989) pp 581–624

14 **Schneidewind, N F (ed)** 'Special section on software maintenance' *IEEE Trans. Soft. Eng.* Vol 13 No 3 (March 1987) pp 301–361

15 **Glass, R** 'Software maintenance is a solution, not a problem' in *Proc. IEEE Conf. Software Maintenance* Miami, FL, USA (October 1989) pp 224–225

16 **Boehm, B W** 'Software engineering' *IEEE Trans. Computers* Vol 25 No 12 (December 1976) pp 1226–1241

17 **Lientz, B P, Swanson, E B and Tompkins, G E** 'Characteristics of application software maintenance' *Commun. ACM* Vol 21 No 6 (June 1978) pp 466–471

18 **Basili, V R and Rombach, H D** 'Towards a comprehensive framework for reuse: a reuse-enabling software evolution environment' *Technical report* Dept of Computer Science and UMIACS, University of Maryland, USA (December 1988)

19 **Caldiera, G and Basili, V R** 'Reengineering existing software for reusability' *Technical report (UMIACS-TR-90-30, CS-TR-2419)* Dept of Computer Science, University of Maryland, USA (February 1990)

20 **Basili, V R and Rombach, H D** 'The TAME project: towards improvement-oriented software environments' *IEEE Trans. Soft. Eng.* Vol 14 No 6 (June 1988) pp 758–773

21 **Rombach, H D and Ulery, B T** 'Improving software maintenance through measurement' in *Proc. IEEE* Vol 77 No 4 (April 1989) pp 581–595

22 **Rombach, H D and Ulery, B T** 'Establishing a measurement based maintenance improvement program: lessons learned in the SEL' in *Proc. IEEE Conf. Software Maintenance* Miami Beach, FL, USA (October 1989) pp 50–57

UMIACS-TR-91-23 February 1991
CS-TR-2606

Support for Comprehensive Reuse[*][†]

V.R. Basili and H.D. Rombach

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

Abstract

Reuse of products, processes and other knowledge will be the key to enable the software industry to achieve the dramatic improvement in productivity and quality required to satisfy the anticipated growing demands. Although experience shows that certain kinds of reuse can be successful, general success has been elusive. A software life-cycle technology which allows comprehensive reuse of all kinds of software-related experience could provide the means to achieving the desired order-of-magnitude improvements. In this paper, we introduce a comprehensive framework of models, model-based characterization schemes, and support mechanisms for better understanding, evaluating, planning, and supporting all aspects of reuse.

[*]A revised version of this TR will be published in the SOFTWARE ENGINEERING JOURNAL, British Computer Society, July 1991.

[†]Research for this study was supported in part by NASA grant NSG-5123, ONR grant NOOO14-87-K-0307 and Airmics grant 19K-CN983-C to the University of Maryland.

TABLE OF CONTENTS:

1. INTRODUCTION

The existing gap between demand and our ability to produce high quality software cost–effectively calls for an improved software development technology. A reuse oriented development technology can significantly contribute to higher quality and productivity. Quality should improve by reusing all forms of proven experience including products, processes as well as quality and productivity models. Productivity should increase by using existing experience rather than creating everything from scratch.

Reusing existing experience is a key ingredient to progress in any discipline. Without reuse everything must be re–learned and re–created; progress in an economical fashion is unlikely. Reuse is less institutionalized in software engineering than in any other engineering discipline. Nevertheless, there exist successful cases of reuse, i.e. product reuse. The potential payoff from reuse can be quite high in software engineering since it is inexpensive to store and reproduce software engineering experience compared to other disciplines.

The goal of research in the area of reuse is to develop and support systematic approaches for effectively reusing existing experience to maximize quality and productivity. A number of different reuse approaches have appeared in the literature (e.g., [10, 12, 14, 17, 18, 19, 20, 26, 27, 29]).

This paper presents a comprehensive framework for reuse consisting of a reuse model, characterization schemes based upon this model, the improvement oriented TAME environment model describing the integration of reuse into the enabling software development processes, mechanisms needed to support comprehensive reuse in the context of the TAME environment model, and (partial) prototype implementations of the TAME environment model. From a number of important assumptions regarding the nature of software development and reuse we derive four essential requirements for any useful reuse model and related characterization scheme (Section 2). We illustrate that existing models and characterization schemes only partially satisfy these essential requirements (Section 3). We introduce a new reuse model which is comprehensive in the sense that it satisfies all four reuse requirements, and use it to derive a reuse characterization scheme (Section 4). Finally, we point out the mechanisms needed to support effective reuse according to this model (Section 5). Throughout the paper we use examples of reusing *generic Ada packages*, *design inspections*, and *cost models* to illustrate our approach.

2. SCOPE OF COMPREHENSIVE REUSE

The reuse framework presented in this paper is based on a number of assumptions regarding software development in general and reuse in particular. These assumptions are based on more than fifteen years of analyzing software processes and products [2, 5, 7, 8, 9, 23]. From these assumptions we derive four essential requirements for any useful reuse model and related characterization scheme.

2.1. Software Development Assumptions

According to a common software development project model depicted in Figure 1, the goal of software development is to produce project deliverables (i.e., project output) that satisfy project needs (i.e., project input) [30]. This goal is achieved according to some development process model which coordinates the interaction between available personnel, practices, methods and tools.

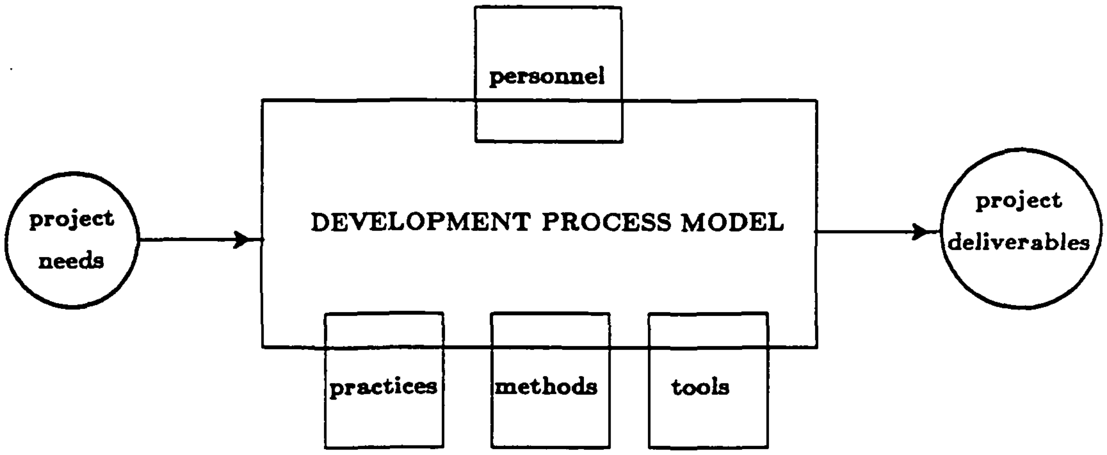


Figure 1: Software Development Project Model

With regard to software development we make the following assumptions:

- **Software development needs to be viewed as an 'experimental' discipline**: An evolutionary model is needed which enables organizations to learn from each development and incrementally improve their ability to engineer quality software products. Such a model requires the ability to define project goals; select and tailor the appropriate process models, practices, methods and techniques; and capture the experiences gained from each project in reusable form. Measurement is essential.
- **A single software development approach cannot be assumed for all software development projects**: Different project needs and other project characteristics may suggest and justify different approaches. The potential differences may range from different development process models themselves to different practices, methods and tools supporting these development process models to different personnel.
- **Existing software development approaches need to be tailorable to project needs and characteristics**: In order to reuse existing development process models, practices, methods and tools across projects with different needs and characteristics, they need to be

tailorable.

2.2. Software Reuse Assumptions

Reuse oriented software development assumes that, given the project–specific needs '$\overline{x}$' for an object 'x', we consider reusing some already existing object 'x_k' instead of creating 'x' from scratch. Reuse involves identifying a set of reuse candidates 'x_1', ..., 'x_n' from an experience base, evaluating their potential for satisfying '$\overline{x}$', selecting the best–suited candidate 'x_k', and – if required – modifying the selected candidate 'x_k' into 'x'. Similar issues have been discussed in [16]. In the case of reuse oriented development, '$\overline{x}$' is not only the specification for the needed object 'x', but also the specification for all the mentioned reuse activities.

As we learn from each project which kinds of experience are reusable and why, we can establish better criteria for what should and what shouldn't be made available in the experience base. The term experience base suggests that anticipate storage of all kinds of software related experience, not just products. The experience base can be improved from inside as well as outside. From inside, we can record experience from ongoing projects which satisfies current reuse criteria for future reuse, and we can re–package existing experience through various mechanisms in order to better satisfy our current reuse criteria. From outside, we can infuse experience which exists out–side the organization into the experience base. It is important to note that the remainder of this paper deals only with the reuse of experience available in an experience base and the improvement of such an experience base from inside (shaded portion of Figure 2).

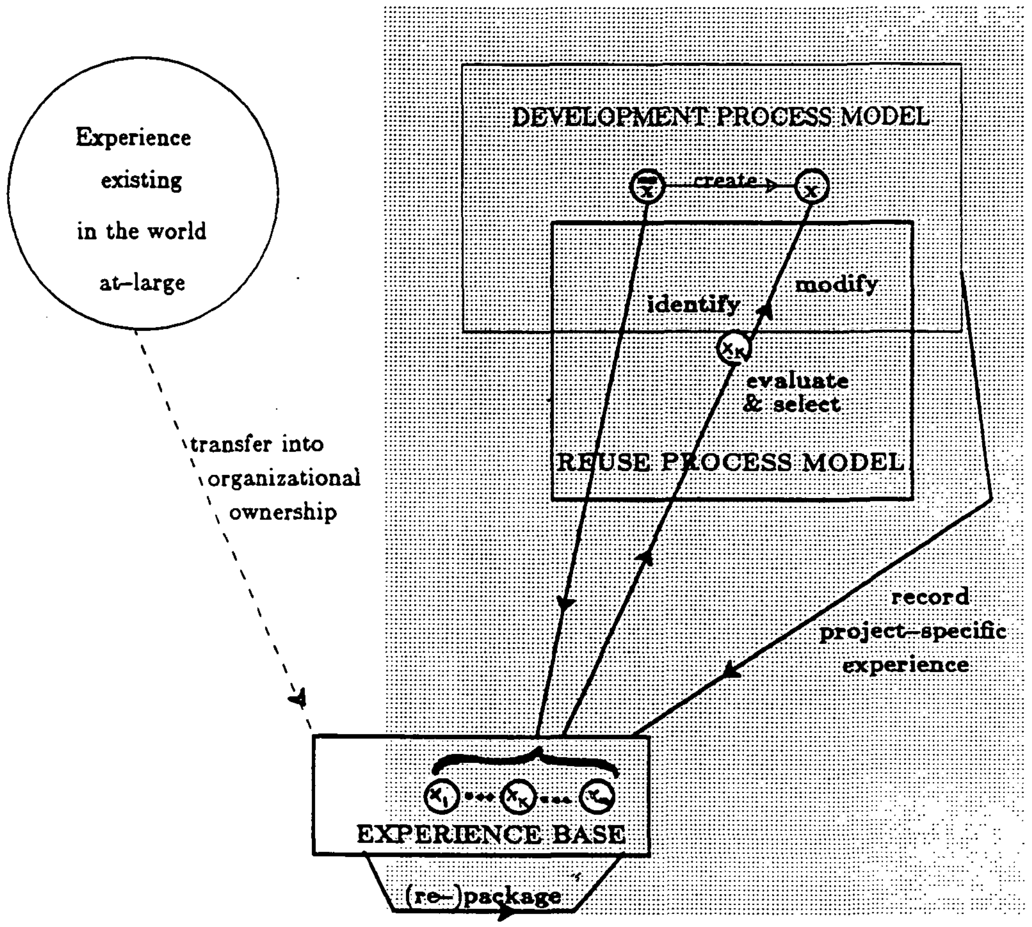

Experience existing in the world at–large
transfer into organizational ownership
DEVELOPMENT PROCESS MODEL
create
identify
modify
evaluate & select
REUSE PROCESS MODEL
record project–specific experience
EXPERIENCE BASE
(re–)package

Figure 2: Reuse Oriented Software Development Model

With regard to software reuse we make the following assumptions:

- **All experience can be reused**: Traditionally, the emphasis has been on reusing concrete objects of type 'source code'. This limitation reflects the traditional view that software equals code. It ignores the importance of reusing all kinds of software–related experience including products, processes, and other knowledge. The term 'product' refers to either a concrete document or artifact created during a software project, or a product model describing a class of concrete documents or artifacts with common characteristics. The term 'process' refers to either to a concrete activity or action – performed by a human being or a machine – aimed at creating some software product, or a process model describing a class of activities or actions with common characteristics. The phrase 'other knowledge' refers to anything useful for software development, including quality and productivity models or models of the application being implemented.

The reuse of 'generic Ada packages' represents an example of product reuse. Generic Ada packages represent templates for instantiating specific package objects according to a parameter mechanisms. The reuse of 'design inspections' represents an example of process reuse. Design inspections are off-line fault detection and isolation methods applied during the module design phase. They can be based on different techniques for reading (e.g., ad hoc, sequential, control flow oriented, stepwise abstraction oriented). The reuse of 'cost models' represents an example of knowledge reuse. Cost models are used in the estimation, evaluation and control of project cost. They predict cost (e.g., in the form of staff-months) based on a number of characteristic project parameters (e.g., estimated product size in KLoC, product complexity, methodology level).

- **Reuse typically requires some modification of the object being reused**: Under the assumption that software developments may be different in some way, modification of experience from prior projects must be anticipated. The degree of modification depends on how many, and to what degree, existing object characteristics differ from the needed ones. The time of modification depends on when the reuse needs for a project or class of projects are known. Modification can take place as part of actual reuse (i.e., the 'modify' within the reuse process model of Figure 2) and/or prior to actual reuse (i.e., as part of the re-packaging activity in Figure 2).

To reuse an Ada package 'list of integers' to organize a 'list of reals' we need to modify it. We can either modify the existing package by hand, or we can use a generic package 'list' which can be instantiated via a parameter mechanism for any base type.

To reuse a design inspection method across projects characterized by significantly different fault profiles, the underlying reading technique may need to be tailored to the respective fault profiles. If 'interface faults' replace 'control flow faults' as the most common fault type, we can either select a different reading technique all together (e.g., step-wise abstraction instead of control-flow oriented) or we can establish specific guidelines for identifying interface faults.

To reuse a cost model across projects characterized by different application domains, we may have to change the number and type of characteristic project parameters used for estimating cost as well as their impact on cost. If 'commercial software' is developed instead of 'real-time software', we may have to consider re-defining 'estimated product size' to be measured in terms of 'function points' instead of 'lines of code' or re-computing the impact of the existing parameters on cost. Using a cost model effectively implies a constant updating of our understanding of *the relationship between project parameters and cost.*

- **Analysis is necessary to determine when and if reuse is appropriate**: The decision to reuse existing experience as well as how and when to reuse it needs to be based on an analysis of the payoff. Reuse payoff is not always easy to evaluate [1]. We need to understand (i) the reuse needs, (ii) how well the available reuse candidates are qualified to meet these needs, and (iii) the mechanisms available to perform the necessary modification.

  Assume the existence of a set of Ada generics which represent application-specific components of a satellite control system. The objective may be to reuse such components to build a new satellite control system of a similar type, but with higher precision. Whether the existing generics are suitable depends on a variety of characteristics: Their correctness and reliability, their performance in prior instances of reuse, their ease of integration into a new system, the potential for achieving the higher degree of precision through instantiation, the degree of change needed, and the existence of reuse mechanisms that support this change process. Candidate Ada generics may theoretically be well suited for reuse; however, without knowing the answers to these questions, they may not be reused due to lack of confidence that reuse will pay off.

  Assume the existence of a design inspection method based on ad-hoc reading which has been used successfully on past satellite control software developments within a standard waterfall model. The objective may be to reuse the method in the context of the Cleanroom development method [22, 25]. In this case, the method needs to be applied in the context of a different life-cycle model, different design approach, and different design representations. Whether and how the existing method can be reused depends on our ability to tailor the reading technique to the stepwise refinement oriented design technique used in Cleanroom, and the required intensity of reading due to the omission of developer testing. This results in the definition of the stepwise abstraction oriented reading technique [11].

  Assume the existence of a cost model that has been validated for the development of satellite control software based on a waterfall life-cycle model, functional decomposition oriented design techniques, and functional and structural testing. The objective may be to reuse the model in the context of Cleanroom development. Whether the cost model can be reused at all, how it needs to be calibrated, or whether a completely different model may be more appropriate depends on whether the model contains the appropriate variables needed for the prediction of cost change or whether they simply need to be re-calibrated. This question can only be answered through thorough analysis of a number of Cleanroom projects.

- **Reuse must be integrated into the specific software development**: Reuse is intended to make software development more effective. In order to achieve this objective we need to tailor reuse practices, methods and tools towards the respective development process.

  We have to decide when and how to identify, modify and integrate existing Ada packages. If we assume identification of Ada generics by name, and modification by the generic parameter mechanism, we require a repository consisting of Ada generics together with a description of the instantiation parameters. If we assume identification by specification, and modification of the *generic's code by hand, we require a suitable specification of each generic, a definition of semantic closeness* of specifications so we can find suitable reuse candidates, and the appropriate source code documentation to allow for ease of modification. In the case of identification by specification we may consider identifying reuse candidates at high-level design (i.e., when the component specifications for the new product exist) or even when defining the requirements.*

We have to decide on how often, when, and how design inspections should be integrated into the development process. If we assume a waterfall-based development life-cycle, we need to determine how many design inspections need to be performed and when (e.g., once for all modules at the end of module design, once for all modules of a subsystem, or once for each module). We need to state which documents are required as input to the design inspection, what results are to be produced, what actions are to be taken, and when, in case the results are insufficient, and who is supposed to participate.

We have to decide when to initially estimate cost and when to update the initial estimate. If we assume a waterfall-based development life-cycle, we may estimate cost initially based on estimated product and process parameters (e.g., estimated product size). After each milestone, the estimated cost can be compared with the actual cost. Possible deviations are used to correct the estimate for the remainder of the project.

2.3. Software Reuse Model Requirements

The above software reuse assumptions suggest that 'reuse' is a complex concept. We need to build models and characterization schemes that allow us to define and understand, compare and evaluate, and plan the reuse needs, the reuse candidates, the reuse process itself, and the potential for effective reuse. Based upon the above assumptions, such models and characterization schemes need to satisfy the following four requirements:

- **Applicable to all types of reuse objects**: We want to be able to include products, processes and all other kinds of knowledge such as quality and productivity models.
- **Capable of modeling reuse candidates and reuse needs**: We want to be able to capture the reuse candidates as well as the reuse needs in the current project. This will enable us to (i) judge the suitability of a given reuse candidate based on the distance between the characteristics of the reuse needs and the reuse candidate, and (ii) establish criteria for useful reuse candidates based on anticipated reuse needs.
- **Capable of modeling the reuse process itself**: We want to be able to (i) judge the ease of

* Definitions of semantic closeness can be derived from existing work [24].

bridging the gap between different characteristics of reuse candidates and reuse needs, and (ii) derive additional criteria for useful reuse candidates based on characteristics of the reuse process itself.

- **Defined and rationalised so they can be easily tailored to specific project needs and characteristics**: We want to be able to adjust a given reuse model and characterization scheme to changing project needs and characteristics in a systematic way. This requires not only the ability to change the scheme, but also some kind of rationale that ties the given reuse characterization scheme back to its underlying model and assumptions. Such a rationale enables us to identify the impact of different environments and modify the scheme in a systematic way.

3. EXISTING REUSE MODELS

A number of research groups have developed (implicit) models and characterization schemes for reuse (e.g., [12, 14, 17, 26, 27]). The schemes can be distinguished as *special purpose schemes* and *meta schemes*.

The large majority of published characterization schemes have been developed for a special purpose. They consist of a fixed number of characterization dimensions. There intention is to characterize software products as they exist. Typical dimensions for characterizing source code objects in a repository are 'function', 'size', or 'type of problem'. Example schemes include the schemes published in [14, 17], the ACM Computing Reviews Scheme, AFIPS's Taxonomy of Computer Science and Engineering, schemes for functional collections (e.g., GAMS, SHARE, SSP, SPSS, IMSL) and schemes for commercial software catalogs (e.g., ICP, IDS, IBM Software Catalog, Apple Book). It is obvious that special purpose schemes are not designed to satisfy the reuse modeling requirements of section 2.3.

A few characterization schemes can be instantiated for different purposes. They explicitly acknowledge the need for different schemes (or the expansion of existing ones) due to different or changing needs of an organization. They, therefore, allow the instantiation of any imaginable scheme. An excellent example is Ruben Prieto–Diaz's facet–based meta–characterization scheme [18, 21]. Theoretically, meta schemes are flexible enough to allow the capturing of any reuse aspect. However, based on known examples of actual uses of meta schemes, such broadness has not been utilized. Instead, most examples focus on product reuse, are limited to the reuse candidates, and ignore the reuse process entirely. Meta schemes were not designed to satisfy the reuse modeling requirements of section 2.3.

To illustrate the capabilities of existing schemes, we give the following instance of an example meta scheme[*] :

- **name**: What is the product's name? (e.g., buffer.ada, queue.ada, list.pascal)
- **function**: What is the functional specification or purpose of the product? (e.g., integer_queue, <element>_buffer, sensor control system)
- **type**: What type of product is it? (e.g., requirements document, design document, code document)
- **granularity**: What is the product's scope? (e.g., system level, subsystem level, component level, module – package, procedure, function – level)
- **representation**: How is the product represented? (e.g., informal set of guidelines, schematized templates, languages such as Ada)
- **input/output**: What are the external input/output dependencies of the product needed to completely define/extract it as a self–contained entity? (e.g., global data referenced by a code unit, formal and actual input/output parameters of a procedure, instantiation parameters of a generic Ada package)
- **application domain**: What application classes was the product developed for? (e.g. ground support software for satellites, business software for banking, payroll software)

- The scheme is applicable to all reuse product candidates. For example, a generic Ada package 'buffer.ada' may be characterized as having identifier 'buffer.ada', offering the function '<element>_buffer', being usable as a 'product' of type 'code document' at the 'package module level', and being represented in 'Ada'. The self–contained definition of the package requires knowledge regarding the instantiation parameters as well as its visibility of externally

[*] Characterization dimensions are marked with '•'; example categories for each dimension are listed in parentheses.

defined objects (e.g., explicit access through WITH clauses, implicit access according to nesting structure). In addition, effective use of the object may require some basic knowledge of the language Ada and assume thorough documentation of the object itself. It may have been developed within the application domain 'ground support software', according to a 'waterfall life–cycle' and 'functional decomposition design', and exhibiting high quality in terms of 'reliability'. In order to characterize reuse candidates of type process or knowledge, new categories need to be generated.

- Such a scheme has typically been used to characterize reuse candidates only. However, in order to evaluate the reuse potential of a reuse candidate in a given reuse scenario, one needs to understand the distance between its characteristics and the stated or anticipated reuse needs. In the case of the Ada package example, the required function may be different, the quality requirements with respect to reliability may be higher, or the design method used in the current project may be different from the one according to which the package has been created originally. Without understanding the distance to be bridged between reuse requirements and reuse candidates it is hard to (a) predict the cost involved in reusing a particular object, and (b) establish criteria for populating a reuse repository that supports cost–effective reuse.

- The scheme provides no information for characterizing the reuse process. To really predict the cost of reuse we do not only have to understand the distance to be bridged between reuse candidates and reuse needs, but also the intended process to bridge it (i.e., the reuse process). For example, it can be expected that it is easier to bridge the distance with respect to function by using a parameterized instantiation mechanism rather than modifying the existing package by hand.

- There is no explicit rationale for the eight dimensions of the example scheme. That makes it hard to reason about its appropriateness as well as modify it in any systematic way. There is no guidance in tailoring the example scheme to new needs with respect to what is to changed (e.g., only some categories, dimensions, or the entire implicitly underlying model) or how it is to be changed. For example, it is not clear what needs to be changed in order to make the scheme applicable to reuse candidates of type process or knowledge.

In summary, existing schemes - special purpose as well as meta schemes - only partially satisfy the requirements laid out above. The most crucial shortcoming is the lack of rationales which makes it hard to tailor such schemes to changing needs and environment characteristics. This observation suggests the need for new, broader reuse models and characterization schemes. In the next section, we suggest a comprehensive reuse model and characterization schemes which satisfy all four requirements.

4. A COMPREHENSIVE REUSE MODEL

In this section we define a comprehensive reuse model and characterization schemes which satisfy the requirements stated in section 2.3. We start with a very general reuse model, refine it step by step until it generates reuse characterization dimensions at the level of detail needed to understand, evaluate, motivate or improve reuse. This modeling approach allows us to deal with the complexity of the modeling task itself, and document an explicit rationale for the resulting model.

4.1. Reuse Model

The comprehensive reuse model used in this section is consistent with the view of reuse represented in section 2.2. Reuse comprises the transformation of existing reuse candidates into needed objects which satisfy established reuse needs. The transformation is referred to as reuse process. Specifications of the needed objects are an essential part of the reuse needs which guide any reuse process.

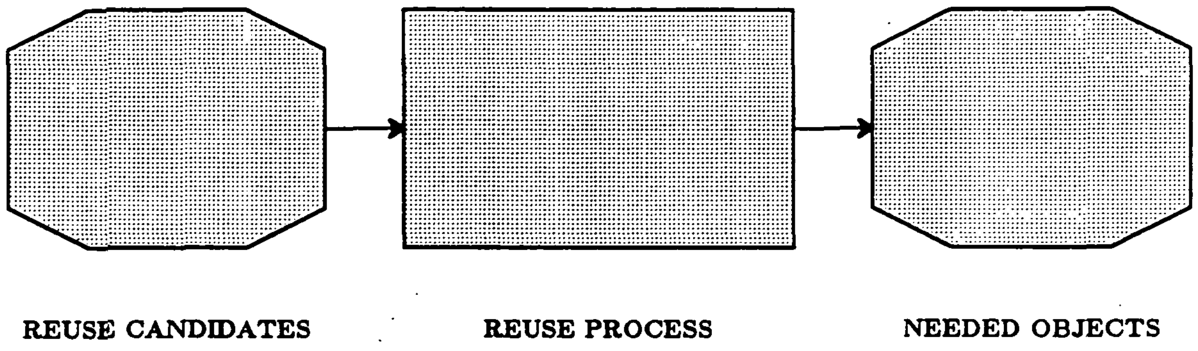
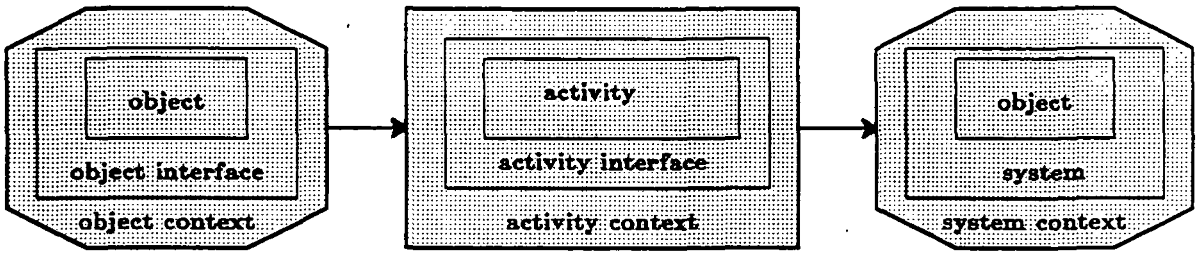


Figure 3: Abstract Reuse Model (Refinement level 0)

The reuse candidates represent experience from the same project, prior projects, or other sources, that have been evaluated as being of potential reuse value, and have been made available in some form of experience base. The reuse needs specify objects needed in the current project. In the case of successful reuse, these needed objects would be the potentially modified versions of reuse candidates. Both the reuse candidate and reuse needs may refer to any type of experience accumulated in the context of software projects ranging from products to processes to knowledge. The reuse process transforms reuse candidates into objects which satisfy given reuse needs.

In order to better understand reuse related issues we refine each component of the reuse model further. The result of this first refinement step is depicted in Figure 4.



REUSE CANDIDATES REUSE PROCESS NEEDED OBJECTS

Figure 4: Our Reuse Model (Refinement level 1)

Each ***reuse candidate*** is a specific ***object*** considered for reuse. The object has various attributes that describe and bound it. Most objects are physically part of a system, i.e. they interact with other objects to create some greater object. If we want to reuse an object we must understand its interaction with other objects in the system in order to extract it as a unit, i.e. ***object interface***. Objects were created in some environment which leaves its characteristics on the object, even though those characteristics may not be visible. We call this the ***object context***.

Given ***reuse needs*** may be satisfied by a set of reuse candidates. Therefore, we may have to consider different attributes. The ***system*** in which the transformed object is integrated and the ***system context*** in which the system is developed must also be classified.

The ***reuse process*** is aimed at extracting a reuse candidate from a repository based on the characteristics of the known reuse needs, and making it ready for reuse in the system and context in which it will be reused. We must describe the various ***reuse activities*** and classify them. The reuse activities need to be integrated into the reuse–enabling software development process. The means of integration constitute the ***activity interface***. Reuse requires the transfer of experience across project boundaries. The organizational support provided for this experience transfer is referred to as ***activity context***.

Based upon the goals for the specific project, as well as the organization, we must assess (i) the required qualities of the reused object as stated by the reuse needs, (ii) the quality of the reuse process, especially its integration into the enabling software evolution process, and (iii) the quality of the existing reuse candidates.

4.2. Model–Based Reuse Characterization Scheme

Each component of the First Model Refinement (Figure 4) is further refined as depicted in Figures 5(a–c). It needs to be noted that these refinements are based on our current understanding of reuse and may, therefore, change in the future.

4.2.1. Reuse Candidates

In order to characterize the object itself, we have chosen to provide the following six dimensions and supplementing categories: the object's name (e.g., buffer.ada), its function (e.g., integer_buffer), its possible use (e.g., product), its type (e.g., requirements document), its granularity (e.g., module), and its representation (e.g., Ada language). The object interface consists of such things as what are the explicit inputs/outputs needed to define and extract the object as a self–contained unit (e.g., instantiation parameters in the case of a generic Ada package), and what are additionally required assumptions and dependencies (e.g., user's knowledge of Ada). Whereas the object and object interface dimensions provide us with a snapshot of the object at hand, the object context dimension provides us with historical information such as the application classes the object was developed for (e.g., ground support software for satellites), the environment the object was developed in (e.g., waterfall life–cycle model), and its validated or anticipated quality (e.g., reliability).

The resulting model refinement is depicted in Figure 5a.

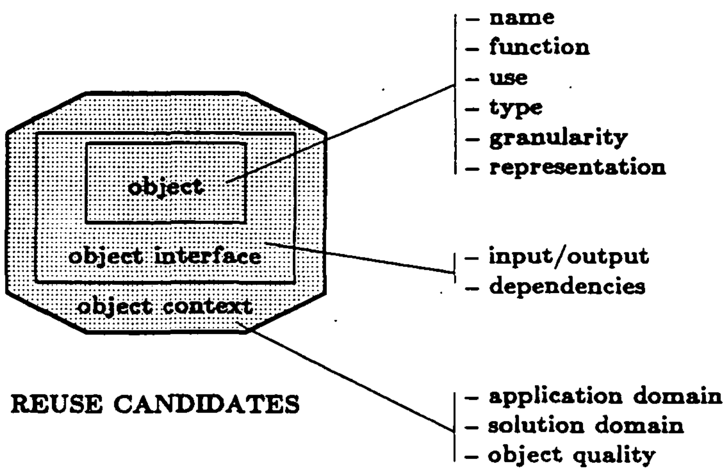


Figure 5a: Reuse Model (Reuse Candidates / Refinement level 2)

Each reuse candidate is characterized in terms of

- **name**: What is the object's name? (e.g., buffer.ada, sel_inspection, sel_cost_model)
- **function**: What is the functional specification or purpose of the object? (e.g., integer_queue, <element>_buffer, sensor control system, certify appropriateness of design documents, predict project cost)
- **use**: How can the object be used? (e.g., product, process, knowledge)
- **type**: What type of object is it? (e.g., requirements document, code document, inspection method, coding method, specification tool, graphic tool, process model, cost model)
- **granularity**: What is the object's scope? (e.g., system level, subsystem level, component level, module – package, procedure, function – level, entire life cycle, design stage, coding stage)
- **representation**: How is the object represented? (e.g., data, informal set of guidelines, schematized templates, formal mathematical model, languages such as Ada, automated tools)
- **input/output**: What are the external input/output dependencies of the object needed to completely define/extract it as a self–contained entity? (e.g., global data referenced by a code unit, formal and actual input/output parameters of a procedure, instantiation parameters of a generic Ada package, specification and design documents needed to perform a design inspection, defect data produced by a design inspection, variables of a cost model)
- **dependencies**: What are additional assumptions and dependencies needed to understand the object? (e.g., assumption on user's qualification such as knowledge of Ada or qualification to read, specification document to understand a code unit, readability of design document, homogeneity of problem classes and environments underlying a cost model)
- **application domain**: What application classes was the object developed for? (e.g. ground support software for satellites, business software for banking, payroll software)
- **solution domain**: What environment classes was the object developed in? (e.g., waterfall life–cycle model, spiral life–cycle model, iterative enhancement life–cycle model, functional decomposition design method, standard set of methods)
- **object quality**: What qualities does the object exhibit? (e.g., level of reliability, correctness, user–friendliness, defect detection rate, predictability)

A subset of this scheme has been used in Section 3. In contrast to Section 3, we now have (i) a rationale for these dimensions (see Figure 5a) and (ii) understand that they cover only part (i.e., the reuse candidate) of the comprehensive reuse model depicted in Figure 4.

4.2.2. Needed Objects

In order to characterize the needed objects (or reuse needs), we have chosen the same eleven dimensions and supporting categories as for the reuse candidates. The resulting model refinement is depicted in Figure 5b:

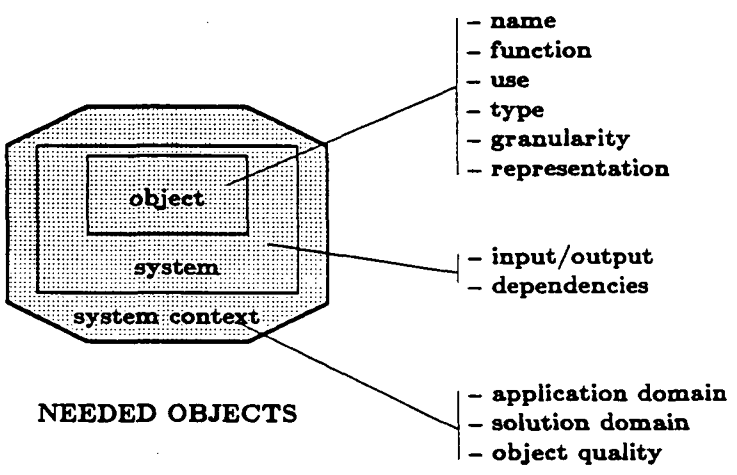


Figure 5b: Reuse Model (Reuse Needs / Refinement level 2)

However, an object may change its characteristics during the actual process of reuse. Therefore, its characterizations before and after reuse can be expected to be different. For example, a reuse candidate may be a compiler (type) product (use), and may have been developed according to a waterfall life-cycle approach (solution domain). The needed object is a compiler (type) process (use) integrated into a project based on iterative enhancement (solution domain).

This means that despite the similarity between the refined models of reuse candidates and needed objects, there exists a significant difference in emphasis: In the former case the emphasis is on the potentially reusable objects themselves; in the latter case, the emphasis is on the system in which these object(s) are (or are expected to be) reused. This explains the use of different dimension names: 'system' and 'system context' instead of 'object interface' and 'object context'.

The distance between the characteristics of a reuse candidate and the needed object give an indication of the gap to be bridged in the event of reuse.

4.2.3. Reuse Process

The reuse process consists of several activities. In the remainder of this paper, we will use a model consisting of four basic activities: identification, evaluation, modification, and integration. In order to characterize each reuse activity we may be interested in its name (e.g., modify.p1), its function (e.g., modify an identified reuse candidate to entirely satisfy given reuse needs), its type (e.g., identification, evaluation, modification), and the mechanism used to perform its function (e.g., modification via parameterization). The interface of each activity may consist of such things as the explicit input/output interfaces between the activity and the enabling software evolution environment (e.g., in the case of modification: performed during the coding phase, assumes the existence of a specification), and other assumptions regarding the evolution environment that need to be satisfied (e.g., existence of certain configuration control policies). The activity context may include information about how reuse candidates are transferred to satisfy given reuse needs (experience transfer), and the quality of each reuse activity (e.g., reliability, productivity).

This refinement of the reuse process is depicted in Figure 5c.

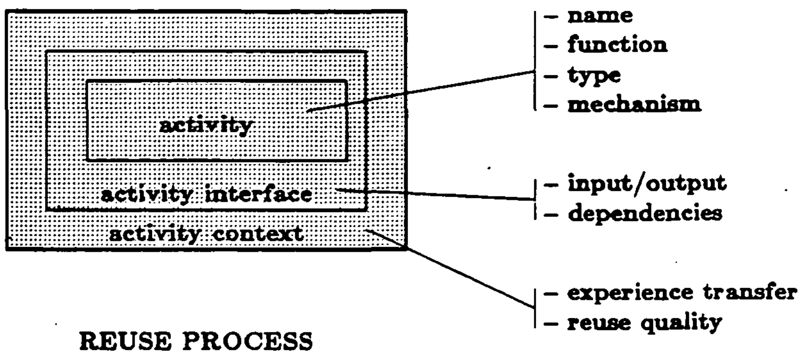


Figure 5c: Reuse Model (Reuse Process / Refinement level 2)

In more detail, the dimensions and example categories for characterizing the reuse process are:

- **REUSE PROCESS**: For each reuse activity characterize:
  + **Activity**:
    - **name**: What is the name of the activity? (e.g., identify.generics, evaluate.generics, modify.generics, integrate.generics)
    - **function**: What is the function performed by the activity? (e.g., select candidate objects $\{x_i\}$ which satisfy certain characteristics of the reuse needs 'X'; evaluate the potential of the selected candidate objects of satisfying the given system and system context dimensions of the reuse needs 'X' and pick the most suited candidate 'x_k'; modify 'x_k' to entirely satisfy 'X'; integrate object 'x' into the current development project)
    - **type**: What is the type of the activity? (e.g., identification, evaluation, modification, integration)
    - **mechanism**: How is the activity performed? (in the case of identification: e.g., by name, by function, by type and function; in the case of evaluation: e.g., by subjective judgement, by evaluation of historical baseline measurement data; in the case of modification: e.g., verbatim, parameterized, template–based, unconstrained; in the case of integration: e.g., according to the system configuration plan, according to the project/process plan)
  + **Activity Interface**:
    - **input/output**: What are explicit input and output interfaces between the reuse activity and the enabling software evolution environment? (in the case of identification: e.g., description of reuse needs / set of reuse candidates; in the case of modification: e.g., one selected reuse candidate, specification for the object to be reused / object to be reused)
    - **dependencies**: What are other implicit assumptions and dependencies on data and information regarding the software evolution environment? (e.g., time at which reuse activity is performed – relative to the enabling development process: e.g., during design or coding stages; additional information needed to perform the reuse activity effectively: e.g., package specification to instantiate a generic package, knowledge of system configuration plan, configuration management procedures, or project plan)
  + **Activity Context**:
    - **experience transfer**: What are the support mechanisms for transferring experience across

projects? (e.g., human, experience base, automated)

- **reuse quality**: What is the quality of each reuse activity? (e.g., high reliability, high predictability of modification cost, correctness, average performance)

4.3. Example Applications of the Comprehensive Reuse Model

We demonstrate the applicability of our model-based reuse scheme by characterizing the three hypothetical reuse scenarios which have been used informally throughout this paper: Ada generics, design inspections, and cost models. The resulting characterizations are summarized in tables 1, 2, and 3:

	Reuse Examples		
Dimensions	Ada generic	design inspection	cost model
name	buffer.ada	sel_inspection.waterfall	sel_cost_model.fortran
function	<element>_buffer	certify appropriateness of design documents	predict project cost
use	product	process	knowledge
type	code document,	inspection method	cost model
granularity	package	design stage	entire life cycle
representation	Ada/ generic package	informal set of guidelines	formal mathematical model
input/output	formal and actual instantiation params (type and number)	specification and design document needed, defect data produced	estimated product size in KLOC, complexity rating, methodology level, cost in staff_hours
dependencies	assumes Ada knowledge	assumes a readable design, qualified reader	assumes a relatively homogeneous class of problems and environments
application domain	ground support sw for satellites	ground support sw for satellites	ground support sw for satellites
solution domain	waterfall (Ada) life-cycle model, functional de-composition design method	waterfall (Ada) life-cycle model, standard set of methods	waterfall (Ada) life-cycle model standard set of methods
object quality	high reliability (e.g., < 0.1 defects per KLoC for a given set of acceptance tests)	average defect detection rate (e.g., > 0.5 defects detected per staff_hour)	average predictability (e.g., < 10% prediction error)

Table 1: Characterizations of Reuse Candidates

	Reuse Examples		
Dimensions	Ada generics	design inspection	cost model
name	string_buffer.ada	sel_inspection.cleanroom	sel_cost_model.ada
function	string_buffer	certify appropriateness of design documents	predict project cost
use	product	process	knowledge
type	code document,	inspection method	cost model
granularity	package	design stage	entire life cycle
representation	Ada	informal set of guidelines	formal mathematical model
input/output	formal and actual instantiation params (type and number)	specification and design document needed, defect data produced	estimated product size in KLOC, complexity rating, methodology level, cost in staff_hours
dependencies	assumes Ada knowledge	assumes a readable design, qualified reader	assumes a relatively homogeneous class of problems and environments
application domain	ground support sw for satellites	ground support sw for satellites	ground support sw for satellites
solution domain	waterfall (Ada) life-cycle model, object oriented design method	Cleanroom (Fortran) development model, stepwise refinement oriented design, statistical testing	waterfall (Ada) life-cycle model, revised set of methods
object quality	high reliability (e.g., < 0.1 defects per KLoC for a given set of acceptance tests), high performance (e.g., max. response times for a set of tests)	high defect detection rate (e.g., > 1.0 defects detected per staff_hour) wrt. interface faults	high predictability (e.g., < 5% prediction error)

Table 2: Characterisations of Needed Objects

	Reuse Examples		
Dimensions	Ada generics	design inspection	cost model
name	modify.generics	modify.inspections	modify.cost_models
function	modify to satisfy target specification	modify to satisfy target specification	modify to satisfy target specification
type	modification	modification	modification
mechanism	parameterized (generic mechanism)	unconstrained	template-based
input/output	buffer.ada, reuse specification/ string_buffer.ada	sel_inspection.waterfall, reuse specification/ sel_inspection.cleanroom	sel_cost_model.fortran, reuse specification/ sel_cost_model.ada
dependencies	performed during coding stage, package specification needed, knowledge of system configuration plan	performed during planning stage, knowledge of project plan	performed during planning stage, knowledge of historical Ada project profiles
experience transfer	automated	human and experience base	experience base
reuse quality	correctness	correctness	correctness

Table 3: Characterisations of Reuse Processes

5. SUPPORT MECHANISMS FOR COMPREHENSIVE REUSE

According to the reuse oriented software development model depicted in Figure 2, effective reuse needs to take place in an environment that supports continuous improvement, i.e., recording of experience across all projects, appropriate packaging and storing of recorded experience, and reusing existing experience whenever feasible. In the TAME project at the University of Maryland, such an environment model has been proposed and (partial) prototype environments are currently being built according to this model. In the remainder of this section, we introduce the reuse oriented TAME environment model, discuss a number of mechanisms for effective reuse, and introduce several prototype environments being built according to the TAME model.

5.1. The Reuse Oriented TAME Environment Model

The important components of the reuse oriented TAME environment model are depicted in Figure 6: the project organization which performs individual development projects, the experience base which stores and actively modifies development experience from all projects, and the mechanisms for learning and reuse. The shaded areas in Figure 6 indicate how the reuse model of Figure 3 intersects with the TAME environment model.

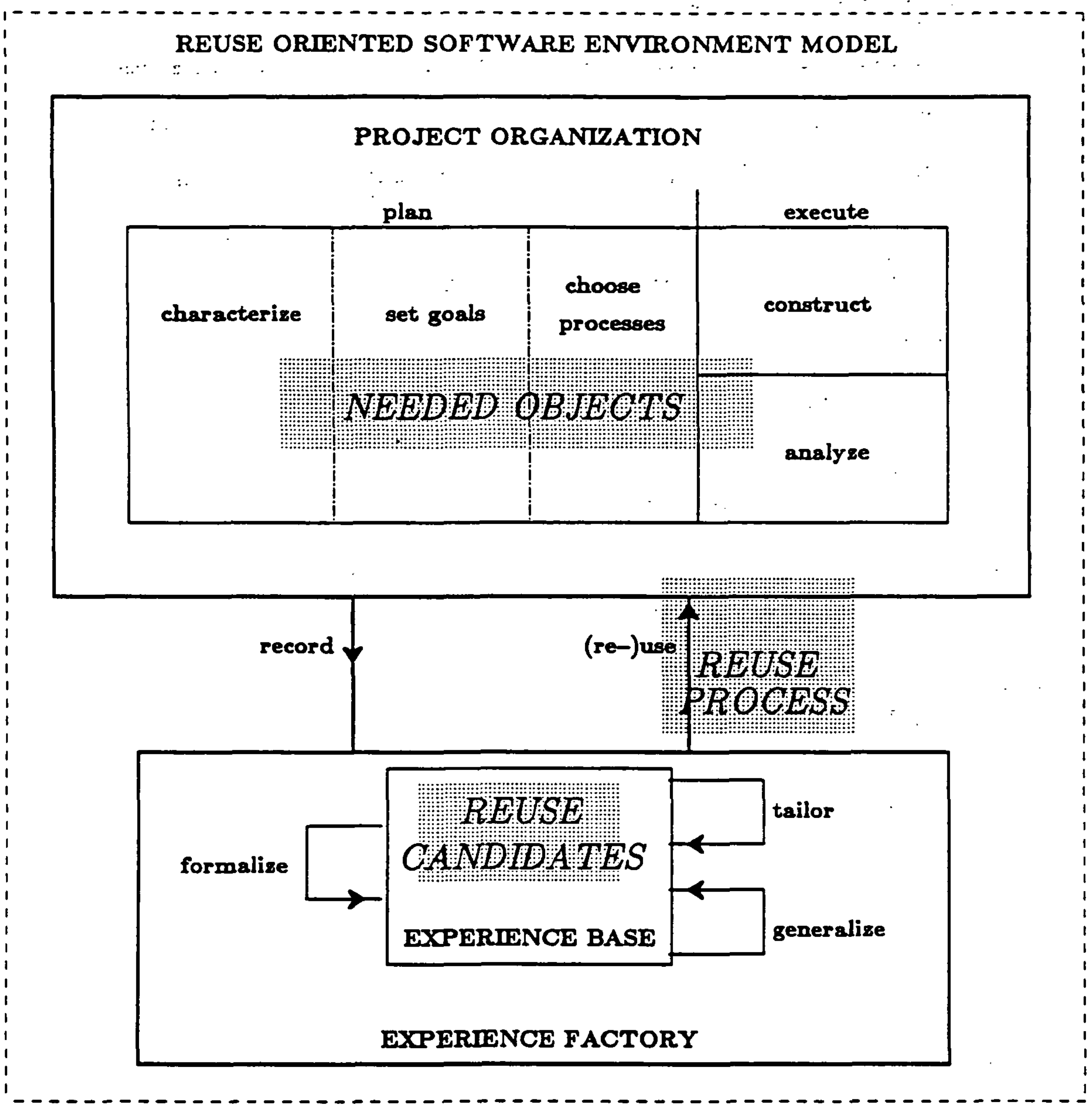


Figure 6: Reuse Oriented Software Environment Model

Within the project organization each development project is performed according to the quality improvement paradigm [3, 9]. The quality improvement paradigm consists of the following steps:

1. **Plan:** Characterize the current project environment so that the appropriate past experience can be made available to the current project. Set up the goals for the project and refine them into quantifiable questions and metrics for successful project performance and improvement over previous project performances (e.g., based upon the goal/question/metric paradigm [9, 13]). Choose the appropriate software development process model for this project with the supporting methods and tools – for both construction and analysis.
2. **Execute:** Construct the products according to the chosen development process model, methods and tools. Collect the prescribed data, validate and analyze it to provide feedback in real–time for corrective action on the current project.
3. **Package:** Analyze the data in a post–mortem fashion to evaluate the current practices, determine problems, record findings and make recommendations for improvement for future projects. Package the experiences in the form of updated and refined models and other forms of structured knowledge gained from this and previous projects, and save it in an experience base so it can be available to future projects.

The experience base contains reuse candidates of different types, granularity and representation. Example entries in the case of the examples described in section 4.3 include objects of type 'code document', granularity 'package' and representation 'Ada'; objects of type 'inspection method', granularity 'design stage' and representation 'schematized template'; and objects of type 'cost model', granularity 'entire life cycle' and representation 'formal mathematical model'.

During each step of a development project performed according to the quality improvement paradigm reuse needs are identified and matches made against reuse candidates available in the experience base. During the characterization step, characteristics of the current project environment can be used to identify appropriate past experience in the experience base, e.g. based on project characteristics the appropriate instantiation of a cost model can be generated. During the planning step, project goals can be used to identify existing similar goal/question/metric models or process/product/quality models in the experience base, e.g., based on project goals a goal/question/metric model can be chosen for evaluating a design inspection method. During the execution step, *product specifications can be used to identify existing components from prior projects*, such as Ada generics. During the feedback step, the analysis goals generated during planning are used as the basis of analysis by fitting baselines to compare against the current data. As part of the feedback step a decision is made as to which experiences are worth recording. The degree of guidance that can be provided for entering reuse candidates into the experience base depends upon the accumulated knowledge of expected reuse requests for future projects.

The experience base is part of an active organizational entity, referred to a the Experience Factory [4], that supports project developments by analyzing and synthesizing all kinds of experience, acting as a repository for such experience, and supplying that experience to various projects on demand. In the context of the reuse oriented software environment model, the Experience Factory not only stores experience in a variety of repositories, but performs the constant modification of experience to increase its reuse potential. Example modifications address the formalization of experience (e.g., building a cost model empirically based upon the data available), tailoring of experience to fit the needs of a specific project (e.g., instantiating an Ada package from a generic package), and the generalizing of experience to be applicable across project classes (e.g., developing a generic package from a specific package). It plays the role of an organizational 'server' aimed at satisfying project specific reuse requests effectively [4]. The constant collection of measurement data regarding reuse needs and the reuse processes themselves enables the judgements needed to populate the experience base effectively and select the best suited reuse candidates. The use of the quality improvement paradigm within the project organization enables the integration of measurement-based analysis and construction.

5.2. Mechanisms to Support Effective Reuse in the TAME Environment Model

Improvement in the reuse oriented TAME environment model of Figure 6 is based on the feedback of experience captured from prior projects into ongoing and future software developments. The mechanisms needed to support effective feedback are listed in Figure 7.

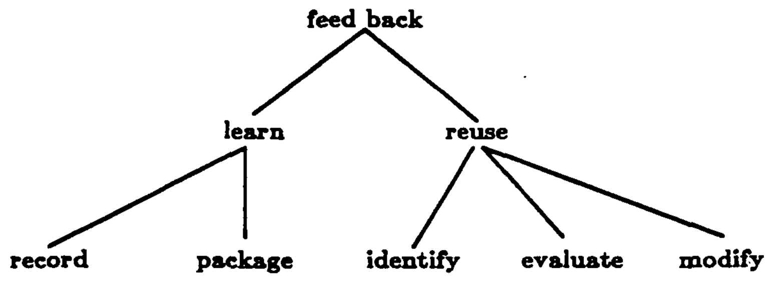


Figure 7: Mechanisms needed to Support Effective Feedback of Experience

Feedback requires learning and reuse. Although learning and reuse are possible in any environment, we are interested in addressing and supporting them explicitly and systematically. Systematic learning requires support for the recording of experience in some experience base and its packaging in order to increase its reuse potential for anticipated reuse needs in future developments. Systematic reuse requires support for the identification of candidate experience, its evaluation, and modification.

Reuse and learning are possible in any environment. However, we want learning and reuse to be explicitly planned, not implicit or coincidental. In the reuse oriented software development environment, learning and reuse are explicitly modeled and become desired characteristics of software development. They are specific processes performed in conjunction with the Experience Factory.

5.2.1. Recording of Experience

The objective of recording experience is to create a repository of well specified and organized experience. This requires a precise characterization of the reuse candidates to be recorded, the design and implementation of a comprehensive experience base, and effective mechanisms for collecting, qualifying, storing and retrieving experience. The characterization of reuse candidates is derived from characterizations of known reuse needs and reuse processes. The characterization of reuse candidates describes what information needs to be stored in addition to the objects themselves in order to make them reusable, and how it should be packaged. The experience base replaces the project database of traditional environment models by the more comprehensive concept of an experience base which is intended to capture the entire body of experience recorded during the planning and execution steps of all software projects within an organization.

Examples of recording experience include the storing of Ada generics, design inspection methods, and cost models. Based on our reuse model, Table 1 describes the information needed in conjunction with each of these object types in order to make them likely reuse candidates to satisfy the hypothetical reuse needs using the hypothetical reuse processes described in Tables 2 and 3, respectively. For example, in the case of Ada generics, we may require each object to be augmented with information on the number of instantiation parameters, the application and solution domain, and the expected or demonstrated reliability. If we can quantify such information (e.g., Ada generics developed within ground support software projects, Ada generics with less than 5 instantiation parameters are acceptable), we can use it to exclude inappropriate objects from being recorded in the first place.

5.2.2. Packaging of Experience

The objective of packaging experience is to increase its reuse potential. This requires a precise characterization of the new reuse needs or processes, and effective mechanisms for generalizing, generalizing and formalizing experience. Packaging may take place at the time of first recording experience into the experience base or at any later time when new reuse needs reuse needs become known or our understanding of the interrelationship between reuse candidates, reuse needs and reuse processes changes.

The objective of generalizing existing experience prior to its reuse is to make a candidate reuse object useful in a larger set of potential target applications. The objective of tailoring existing experience prior to its potential reuse is to fine–tune a candidate reuse object to fit a specific task or exhibit special attributes, such as size or performance. The objective of formalizing existing experience prior to its actual reuse is to increase the reuse potential of reuse candidates by encoding them in more precise, better understood ways. These activities require a well-documented cataloged and categorized set of reuse candidates, mechanisms that support the modification process, and an understanding of the potential reuse needs. Generalization and tailoring are specifically concerned with changing the application and solution domain characteristics of reuse candidates: from project specific to domain specific to project specific and vice versa. Objectives and characteristics are different from project to project, and even more so from environment to environment. We cannot reuse past experience without modifying it to the needs of the current project. The stability of the environment in which reuse takes place, as well as the origination of the experience, determine the amount of tailoring required. Formalization activities are concerned with movement across the boundaries of the representation dimension within the experience base: from informal to schematized and then to formal.

Examples of tailoring experience include the instantiation of a set of specific Ada packages from a generic package available in an object oriented experience base, the fine–tuning of a cost model to the specific characteristics of a class of projects, and the adjustment of a design inspection method to focus on the class of defects common to the application. Examples of generalizing experience include the creation of a generic Ada package from a set of specific Ada packages, the creation of a general cost model from a set of domain specific cost models, and the definition of an application and solution domain specific design inspection method based on the experience with design inspections in a number of specific projects. Examples of formalization include the writing of functional specifications for generic Ada packages, providing automated support for checking adherence to entry and exit criteria of a design inspection method, and building a cost model empirically based upon the data available in an experience base.

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

5.2.3. Identification of Candidate Experience

The objective of identifying candidate experience is to find a set of candidates with the potential to satisfy project specific reuse needs. This requires a precise characterization of the reuse needs, some organizational scheme for the reuse candidates available in the experience base, and an effective mechanism for matching characteristics of the project specific reuse needs against the experience base.

Let's assume, for example, that we need an Ada package which implements a 'string_buffer' with high 'reliability and performance' characteristics. This need may have been established during the project planning phase based on domain analysis, or during the design or coding stages. We identify candidate objects based on some subset of the object related characteristics stated in Table 2: string_buffer.ada, string_buffer, product, code document, package, Ada [28]. The more characteristics we use for identification, the smaller the resulting set of candidate objects will be. For example, if we include the name itself, we will either find exactly one object or none. Identification may take place during any project stage. We will assume that the set of successfully identified reuse candidates contains 'buffer.ada', the object characterized in Table 1.

5.2.4. Evaluation of Experience

The objective of evaluating experience is to characterize the degree of discrepancies between a given set of reuse needs (see Table 2) and some identified reuse candidate (Table 1), and (ii) predict the cost of bridging the gap between reuse candidates and reuse needs. The first type of evaluation goal can be achieved by capturing detailed information about reuse candidates and reuse needs according to the dimensions of the presented characterization scheme. The second goal requires the inclusion of data characterizing the reuse process itself and past experience about similar reuse activities. Effective evaluation requires precise characterization of reuse needs, reuse processes and reuse candidates; knowledge about their relationships, and effective mechanisms for measurement.

The knowledge regarding the interrelationship between reuse needs, processes and candidates is the result of the proposed evolutionary learning which takes place within the reuse oriented TAME environment model. The mechanisms used for effective measurement are based on the goal/question/metric paradigm [9, 11, 13]. It provides templates for guiding the selection of appropriate metrics based on a precise definition of the evaluation goal. Guidance exists at the level of identifying certain types of metrics (e.g., to quantify the object of interest, to quantify the perspective of interest, to quantify the quality aspect of interest). Using the goal/question/metric paradigm in conjunction with reuse characterizations like the ones depicted in Tables 1, 2, and 3, provides very detailed guidance as to what exact metrics need to be used. For example, evaluation of the Ada generic example suggests metrics to characterize discrepancies between the reuse needs and all available reuse candidates in terms of (i) function, use, type, granularity, and representation on a nominal scale defined by the respective categories, (ii) input/output interface on an ordinal scale 'number of instantiation params', (iii) application and solution domains on nominal scales, and (iv) qualities such as performance based on benchmark tests.

For example, we want to evaluate the reuse potential of the object 'buffer.ada' identified in the previous subsection. We need to evaluate whether and to what degree 'buffer.ada' (as well as any other identified candidate) needs to be modified and estimate the cost of such modification compared to the cost required for creating the desired object 'string_buffer' from scratch. Three characteristics of the chosen reuse candidate deviate from the expected ones: it is more general than needed (see function dimension), it has been developed according to a different design approach (see solution domain dimension), and it does not contain any information about its performance behavior (see object quality dimension). The functional discrepancy requires instantiating object 'buffer.ada' for data type 'string'. The cost of this modification is extremely low due to the fact that the generic instantiation mechanism in Ada can be used for modification (see Table 3). The remaining two discrepancies cannot be evaluated based on the information available through the characterizations in section 4.3. On the one hand, ignoring the solution domain discrepancy may result in problems during the integration phase. On the other hand, it may be hard to predict the cost of transforming 'buffer.ada' to adhere to object oriented principles. Without additional information about either the integration of non-object oriented packages or the cost of modification, we only have the choice between two risks. Predicting the cost of changes necessary to satisfy the stated object performance requirements is impossible because we have no information about the candidate's performance behavior. It is noteworthy that very often practical reuse seems to fail because of lack of appropriate information to evaluate the reuse implications a-priori, rather than because of technical infeasibility [15].

The characterization of both reuse candidates and needs and the reuse process allow us to understand some of the implications and risks associated with discrepancies between identified reuse candidates and target reuse needs. Problems arise when we have either insufficient information about the existence of a discrepancy (e.g., object performance quality in our example), or no understanding of the implications of an identified discrepancy (e.g., solution domain in our example). In order to avoid the first type of problem, one may either constrain the identification process further by including characteristics other than just the object related ones, or not have any objects without 'performance' data in the reuse repository. If we had included 'desired solution domain' and 'object performance' as additional criteria in our identification process, we may not have selected object 'buffer.ada' at all. If every object in our repository would have performance data attached to it, we at least would be able to establish the fact that there exists a discrepancy. In order to avoid the second type of problem, we need have some (semi-) automated modification mechanism, or at least historical data about the cost involved in similar past situations. It is clear that in our example any functional discrepancy within the scope of the instantiation parameters is easy to bridge due to the availability of a completely automated modification mechanism (i.e., generic instantiation in Ada). Any functional discrepancy that cannot be bridged through this mechanisms poses a larger and possibly unpredictable risk. Whether it is more costly to re-design 'buffer.ada' in order to adhere to object oriented design principles or to re-develop it from scratch is not obvious without past experience. A mechanism for modeling all kinds of experience is given in [6].

5.2.5. Modification of Experience

The objective of modifying experience is to bridge the gap between a selected reuse candidates and given reuse needs. This requires a precise characterization of the reuse needs, and effective mechanisms for modification. Technically, modification mechanisms are very similar to the tailoring (and generalization) mechanism introduced for packaging experience. Tailoring here is different in that during modification the target is described by concrete, project specific reuse needs, whereas during packaging the target is typically imprecise in that it reflects anticipated reuse needs in a class of future projects. We refer to tailoring (and generalizing) as 'off-line' (during packaging) or 'on-line' (during modification) depending on whether it takes place before or as part of a concrete instance of reuse.

Examples of modifying experience – similar to the examples given earlier for tailoring – include the instantiation of a set of specific Ada packages from a generic package available in an object oriented experience base, the fine-tuning of a cost model to the specific characteristics of a class of projects, and the adjustment of a design inspection method to focus on the class of defects common to the application.

5.3. TAME Environment Prototypes

In the TAME (Tailoring A Measurement Environment) project, we investigate fundamental issues related to the reuse– (or improvement–) oriented software environment model of Figure 6 and build a series of (partial) research prototype versions [8, 9, 15].

Current research topics include the formalization of the goal/question/metric paradigm for effective software measurement and evaluation; the development of formalisms for representing software engineering experience such as quality models, lessons learned, process models, product models; the development of models for packaging experience in the experience base; and the development of effective mechanisms to support learning and reuse within the experience factory (e.g., qualification, formalization, tailoring, generalization, synthesis). In addition, various slices of an evolving TAME environment are being prototyped in order to study the definition and integration of different concepts.

Aspects of the TAME research prototypes, currently being developed at the University of Maryland, can be classified best by the different classes of experience they attempt to generate, maintain and reuse:

- Support for identifying objects by browsing through projects, goals and processes based on a facet–based characterization mechanism.
- Support for the generalization, tailoring, and integration of a variety experience types based on an object oriented experience base model.
- Support for the definition of environment specific cost and resource allocation models and their tailoring, generalization and formalization based on project experience.
- Support for the definition of test techniques in terms of entry and exit criteria that provides a method for selecting the appropriate technique for each project phase based on environment characteristics, data models, and project goals.
- Support for the definition of process models and their formalization, generalization and tailoring based on project experience.

• Support for an experience factory architecture that supports the evolution of the organization.

6. CONCLUSIONS

We have introduced a comprehensive reuse framework consisting of reuse models, model-based characterization schemes, the TAME environment model supporting the integration of reuse into software development, and ongoing research and development efforts toward a TAME environment prototype.

The presented reuse model and related model-based characterization schemes have advantages over existing models and schemes in that they (a) allow us to capture the reuse of any type of experience, (b) address reuse candidates and reuse needs as well as the reuse process itself, and (c) provide a rationale for the chosen characterizing dimensions. We have demonstrated the advantages of such a comprehensive reuse model and related schemes by applying them to the characterization of example reuse scenarios. Especially their usefulness for defining and motivating the support mechanisms for comprehensive reuse and learning were stressed.

Finally, we introduced the TAME environment model which supports the integration of reuse into software developments. Several partial instantiations of the TAME environment model, currently being developed at the University of Maryland, have been mentioned. In order to make reuse a reality, more research is required towards understanding and conceptualizing activities and aspects related to reuse, learning and experience factory technology.

7. ACKNOWLEDGEMENTS

We thank all our colleagues and graduate students who contributed to this paper, especially all members of the TAME, CARE and LASER projects. We also thank the Guest Editors, Nazim H. Madhavji and Wilhelm Schaefer, and the anonymous referees for their excellent suggestions for

improving this paper.

8. REFERENCES

[1] B. H. Barnes and T. B. Bollinger, "Making Reuse Cost-Effective", IEEE Software Magazine, January 1991, pp. 13-24.

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

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

[4] V. R. Basili, "Software Development: A Paradigm for the Future", Proc. 13th Annual International Computer Software & Applications Conference, Orlando, FL, September 20-22, 1989.

[5] V. R. Basili, "Viewing Maintenance as Reuse Oriented Software Development", IEEE Software Magazine, January 1990, pp. 19-25.

[6] V. R. Basili, G. Caldiera, and G. Cantone, "A Reference Architecture for the Component Factory", Technical Report TR-3333, Dept. of Computer Science, University of Maryland, College Park, MD 20742, March 1991.

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

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

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

[10] V. R. Basili and H. D. Rombach, "Towards a Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment (part I)/ Model-Based Reuse Characterization Schemes (part II)", Technical Reports, Dept. of Computer Science (CS-TR-2158/CS-TR-2446) and UMIACS (UMIACS-TR-88-92/UMIACS-TR-90-47), University of Maryland, College Park, MD 20742, December 1988/April 1990.

[11] V. R. Basili and R. W. Selby, "Comparing the Effectiveness of Software Testing Strategies", IEEE Transactions on Software Engineering, vol.SE-13, no.12, December 1987, pp.1278-1296.

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

[13] V. R. Basili and 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.

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

[15] G. Caldiera and V. R. Basili, "Reengineering Existing Software for Reusability", Technical Report (UMIACS-TR-90-30, CS-TR-2419), Dept. of Computer Science, University of Maryland, College Park, MD 20742, February 1990.

[16] S. Cardenas and M. V. Zelkowitz, "Evaluation Criteria for Functional Specifications", Proc. of the 12th IEEE International Conference on Software Engineering, Nice, France, March 26–30, 1990, pp. 26–33.

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

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

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

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

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

[22] A. Kouchakdjian, V. R. Basili, and S. Green, "The Evolution of the Cleanroom Process in the Software Engineering Laboratory", IEEE Software Magazine (to appear 1990).

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

[24] A. Mili, W. Xiao-Yang, and Y. Qing, "Specification Methodology: An Integrated Relational Approach", Software - Practice and Experience, vol. 16, no. 11, November 1986, pp. 1003–1030.

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

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

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

[28] P. A. Straub and E. J. Ostertag, "EDF: A Formalism for describing and Reusing Software Experience", Proceedings of the International Symposium on Software Reliability Engineering, Austin, Texas, May 1991.

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

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

UMIACS–TR–91–24
CS–TR–2607

March 1991

A Reference Architecture for the Component Factory*

Victor R. Basili and Gianluigi Caldiera
Institute for Advanced Computer Studies and
Department of Computer Science
University of Maryland
College Park, MD 20742, U.S.A.

Giovanni Cantone
Universita' di Napoli
Naples, Italy

Abstract

Software reuse can be achieved through an organization that focuses on utilization of life cycle products from previous developments. The component factory is both an example of the more general concepts of experience and domain factory and an organizational unit worth being considered independently. The critical features of such an organization are flexibility and continuous improvement. In order to achieve these features we can represent the architecture of the factory at different levels of abstraction and define a reference architecture from which specific architectures can be derived by instantiation. A reference architecture is an implementation and organization independent representation of the component factory and its environment. The paper outlines this reference architecture, discusses the instantiation process and presents some examples of specific architectures comparing them in the framework of the reference model.

*Research for this study was supported in part by NASA (Grant NSG-5123), by ONR (Grant NOOO14-87-K-0307) and by Italian CNR (Grant 89.00052.69).

1. INTRODUCTION

The issue of productivity and quality is becoming critical for the software industry at its current level of maturity. Software projects are requested to do more with less resources: deliver the required systems faster, reduce turn-around time in maintenance, increase performance reliability and security of systems. All of this implies radical changes to the way software is produced today. A straightforward solution to the problem of increasing quality and productivity can be synthesized in three goals: improve the effectiveness of the process, reduce the amount of rework, and reuse life cycle products.

The production of software using reusable components is a significant step forward for all three of those goals. The idea is to use pre-existing well designed, tested and documented elements as building blocks of programs, amplifying the programming capabilities, reducing the amount of work needed on new programs and systems, and achieving a better overall control over the production process and the quality of its products.

The possibility of assembling programs and systems from modular software units "classified by precision, robustness, time-space performance, size limits, and binding time of parameters" [McIlroy 1969] has been suggested from the beginnings of software engineering. However it has never acquired real momentum in industrial environments and software projects, despite the large amount of informal reuse already there. Reuse is a very simple concept: it means use the same thing more than once. But as far as software is concerned, it is difficult to define what is an object by itself, in isolation from its context [Freeman 1983]. We have programs, parts of programs, specifications, requirements, architectures, test cases and plans, all related to each other. Reuse of each software object implies the concurrent reuse of the objects associated with it, with a fair amount of informal information traveling with the objects. This means we need to reuse more than code. Software objects and their relationships incorporate a large amount of experience from past development activities: it is the reuse of this experience that needs to be fully incorporated into the production process of software and that makes it possible to reuse software objects [Basili and Rombach 1991].

Problems in achieving higher levels of reuse are the inability to package experience in a readily available way, to recognize which experience is appropriate for reuse, and to integrate reuse activities into the software development process. Reuse is assumed to take place totally within the context of the project development. This is difficult because the project focus is the delivery of the system; packaging of reusable experience can be, at best, a secondary focus of the project. Besides, project personnel are not always in the best position to recognize which pieces of experience are appropriate for other projects. Finally, existing process models are not defined to support and to take advantage of reuse, much less to create reusable experience. They tend to be rigidly deterministic where, clearly, multiple process models are necessary for reusing experience and creating packaged experience for reuse.

In order to practice reuse in an effective way, an organization is needed whose main focus is to make reuse easy and effective. This implies a significant cultural change in the software industry, from a project-based frame of mind centered on the ideas and experience of project designers, to a corporate-wide one, where a portion of those ideas and experience becomes a permanent corporate asset, independent from the people who originate it. This cultural change will probably happen slowly, and a way to facilitate it is to provide an organization that is flexible enough to accept this evolution. Two characteristics stand out in this context

- *Flexibility*: the organization must be able to change its configuration without a negative impact on its performance, incrementally gaining control over the main factors affecting production.

- *Continuous improvement* (the Japanese "kaizen"): the organization must be able to learn from its own experience and to evolve towards higher levels of quality building competencies and reusing them.

This paper will present some ideas on how to design the production of software using reusable components. In particular we will show the effectiveness of a representation of the organization with different levels of abstraction in order to achieve the desired flexibility. This will lead us to the concept of reference architecture that will provide a representation of the organization as a collection of interacting parts, each one independent of the way the other ones perform their task. In this framework, methods and tools can be changed inside each one of those independent "development islands" without the need for a change in the other ones. We will outline a possible reference architecture and illustrate it with some examples.

After having automated other organizations' business and production, the software industry is facing today the problem of a substantial automation of its activities. Without mechanically translating concepts that are pertinent to very different production environments, we can say that flexible manufacturing systems combine many desirable features: modular architecture, integration of heterogeneous methods and tools, configuration and reconfiguration capabilities, wide automation under human control. Flexible manufacturing comes to age in the software industry through the definition of integrated software engineering environments if they are based on the concepts of flexibility and continuous improvement we have mentioned earlier.

The next two sections of this paper will present an organizational framework designed to make reuse happen in the most effective way. Sections 4 and 5 will discuss the representation that we propose for this framework, that uses different levels of abstraction in order to obtain a flexible and evolutionary organizational design. Section 6 will outline our methodology for deriving a particular environment from the general one, and section 7 will illustrate this derivation with some theoretical and actual examples.

2. A REUSE-ORIENTED ORGANIZATION

In order to address the problems of reuse in a comprehensive way, Basili has proposed an organizational framework that separates the project specific activities from the reuse packaging activities, with process models for supporting each of the activities [Basili 1989]. The framework defines two separate organizations: a project organization and an experience factory.

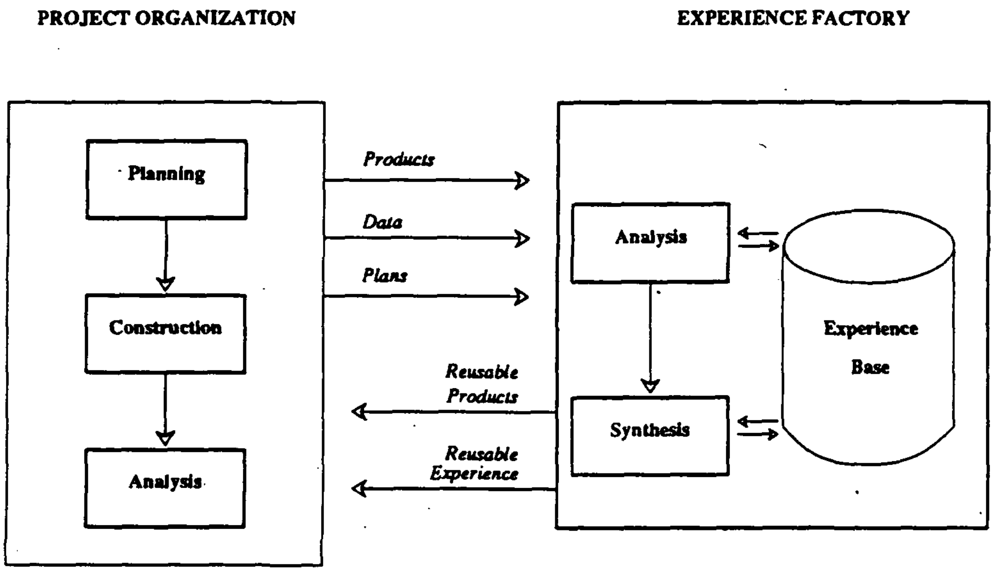
One organization is project-oriented. Its goal is to deliver the systems required by the customer. It is called the *project organization*. The other organization, called *experience factory*, has the role of monitoring and analyzing project developments, developing and packaging experience for reuse in the form of knowledge, processes, tools and products, and supplying it to the project organization upon request.

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

The experience factory, conceptually represented in Figure 1, is a logical and/or physical organization that supports project developments by analyzing and synthesizing all kinds of experience, acting as a repository for such experience, and supplying that experience to various projects on demand. There are a variety of forms for packaged experience. There are, for instance,

- reusable products of the life cycle (i.e. the Ada package that creates and updates B-trees);
- equations defining the relationship between variables (e.g. $Effort = a * Size^{b}$);
- charts of data (e.g. Pareto chart of classes of software defects);
- management curves (e.g. product size growth over time with confidence levels);
- specific lessons learned associated with project types, phases and activities (e.g. in code inspections reading by step-wise abstraction is most effective for finding interface faults);
- models and algorithms specifying processes, methods and techniques (e.g. an SADT diagram defining Design Inspections with the reading technique as a variable dependent upon the focus and the reader perspective.

Figure 1
The Experience Factory



10000174

From this brief discussion we see that the organization of the experience factory can be divided into several sub-organizations, each one dedicated to processing a particular kind of experience.

A first level of subdivision of the experience factory can be based on the application domain [Neighbors 1989]: for each different domain we have a different *domain factory*, conceptually represented in Figure 2a, whose purpose is to define the process for producing applications within the domain, to implement the environment needed to support that process, to monitor and improve that environment and process. Examples of domains are Satellite Ground Support Software, Information Management Software for Insurance Companies, C^3 Systems, etc. The experience manipulated by the domain factory are the definition of an application domain and the experience relative to engineering within that specific domain [Caldiera 1991].

A further subdivision of the experience factory, that will be the object of the discussion in this paper, is the development and packaging of software components. This function is performed by an organization we call the *component factory*, conceptually represented in Figure 2b, which supplies software components to projects upon demand, and creates and maintains a repository of those components for future use. The experience manipulated by the component factory is the programming and application experience as it is embodied in requirements, specifications, designs, programs and associated documentation. *The software component produced and manipulated by the component factory is a collection of artifacts that provide the project organization with everything needed to integrate it in an application system and to provide life cycle support for this system.*

The separation of project organization and (experience, domain or component) factory is not as simple as it appears in our diagrams. Having one organization that designs and integrates only, and another one that develops and packages only, is an ideal picture that can present itself in many different variations. In many cases, for instance, some development will be performed in the project organization according to its needs. Therefore, the flows of data and products across the boundary are different from the ones we have shown in the figures of this section. One of the aims of this paper is to deal with this complexity, providing a rigorous framework for the representation of problems and solutions.

We will discuss these issue focusing on the concept of component factory, both as an example of the more general concept of experience factory and as an organizational unit worth being considered independently. On one hand, the component factory will be the framework used to define and discuss various organizational structures via our concept of reference architecture, which is applicable as well to the more general frameworks of domain and experience factory. On the other hand, the discussion will give us better insight into the role of the component factory as a milestone on the roadway to an industrialization of software development.

Figure 2a

The Domain Factory

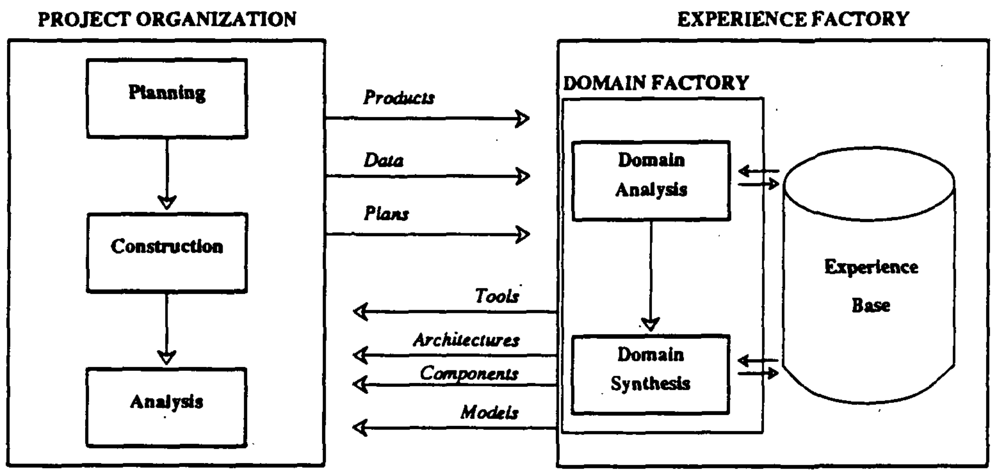
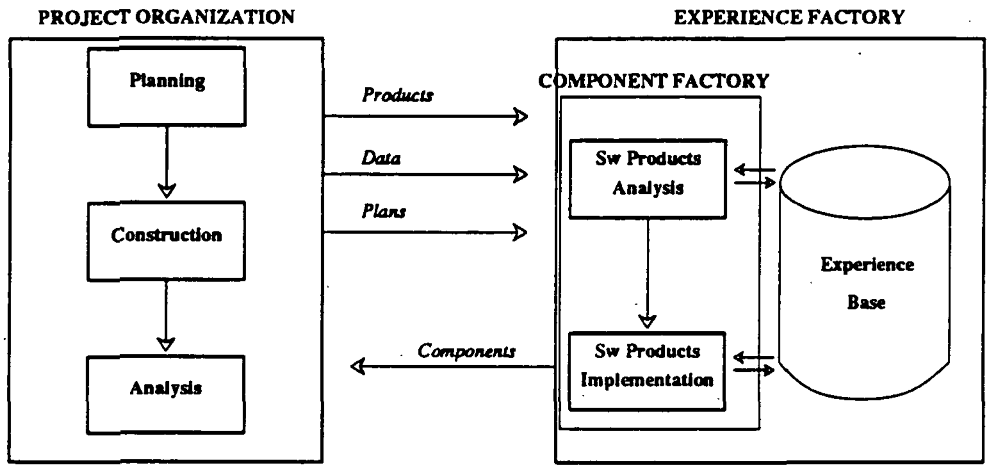


Figure 2b
The Component Factory



3. THE COMPONENT FACTORY

The concept of component factory is an extension and a redefinition of the concept of software factory, as it has evolved from the original meaning of integrated environment to the one of flexible software manufacturing [Cusumano 1989]. The major difference is that, while the software factory is thought of as an independent unit producing code and using an integrated production environment, the component factory handles every kind of code-related information and experience. The component factory is defined as a part of the experience factory, and therefore it is recognized that its potential benefits can be fully exploited only within this framework.

As noted earlier, a software component is any product in the software life cycle. We have

- Code components: objects implemented in some compilable, interpretable or executable language. This includes programs, subprograms, program fragments, macros, simple classes and objects, etc.

- Designs: objects representing function, structure and interfaces of software components (and collections) written in a language that can be formal, semi-formal, graphic, or natural. This includes structured design specifications, interface specifications, functional specifications, logical schemata, etc. In some cases designs are code components themselves.

- Collections of Code Component or Designs: objects obtained by aggregation of several functionally homogeneous code components or designs. This includes the libraries (mathematical, statistical, graphic, etc.), the collections of packages of Ada-like languages, the composite classes of object-oriented languages, the architectures of structured design techniques, etc.

- Documents: textual objects written in natural language with figures, tables and formulas to communicate information in some organized way. Hypertext objects can be, in many cases, considered in this class. This includes requirements documents, standards and policy documents, recommendations, lessons learned documents, reports from specific studies and analyses, etc.

The software component produced and maintained in the component factory is the *reusable software component* (RSC): it is a composite object made of a software component packaged with everything that is necessary to reuse and to maintain it in the future. This means very different things in different contexts, but it should include at least the code of the component, its functional specification, its context (i.e., borrowing the term from the Ada language, the list of the software components that are in some way associated with it), a full set of test cases, a classification according to a certain taxonomy and a reuser's manual [Caldiera and Basili 1990].

The project organization uses reusable software components to integrate them into the programs and the system that have been previously designed.

The capability of the component factory to make reuse happen in an efficient and reliable way is a critical element for the successful application of the reuse technology. Therefore the catalog of available components must be rich in order to reduce the chances of development from scratch, and look up must be easy.

There are three major groups of activities associated with the production of software through reusable software components

- *Use reusable software components*

  When the project organization needs a component described by a certain specification, the catalog of available components is searched:

  - if a ready-to-integrate component that matches the specification is found, the project organization uses it;
  - if a component that needs some adaptation in order to match the specification is found, the project organization uses it after the needed modifications have been applied to the component; organization;
  - if either no component is found that matches the specification or the needed adaptation is too large to be considered a simple modification of a pre-existing component, the component is ordered.

- *Develop and maintain reusable software components*

  A reusable software component enters into the production process for one of two reasons:

  - because it has been recognized as useful from a preliminary analysis of the application domains the project organizations deal with [Arango 1989];
  - because it was needed, it has been deemed "reusable", and wasn't available;

  Once the need for some component has been recognized or the component has been ordered there are three ways for the component to enter into the production process:

  - by direct development either from scratch or from pre-existing generic elementary processes [Joo 1990];
  - by direct procurement from an external source (an external repository, a vendor, etc.) [Tracz 1987];
  - by extraction and adaptation from existing programs and systems [Caldiera and Basili 1990].

In whatever way the reusable component has entered into the production process, it is

- adapted for further reuse by enhancement or generalization of its functions and structure, by fusion with other similar components, or by tailoring to expected needs;
- maintained to satisfy the evolving needs of the applications as identified by domain analysis;
- maintained to guarantee its correctness.

- *Collect and package experience from activities*

  The activities of project organizations and component factory are recorded and processed in order to be preserved in a reusable form. This means they are packaged in units that are

  - understandable by everybody interested in using them;
  - either independent or explicitly declared and packaged with their dependencies;
  - retrievable using some kind of search procedure.

  The experience is formalized and translated into a *model*. Different kinds of models are produced in order to represent the knowledge the project organizations and the component factory have of products, processes, resources, quality factors, etc.:

  - product models representing the observable characteristics of the software components through measures;
  - process models representing the observable characteristics of the production process, its phases and states, and the measures that allow its control;
  - resource and cost models representing the amount of resources allocated to, or estimated for, a set of activities and their distribution
  - lessons-learned models representing the knowledge acquired from former projects and experiments.

This first summary analysis shows that the ability to anticipate future needs is critical to the efficient implementation of a reuse oriented paradigm, and to the work of the component factory in order to satisfy the requests coming from the project organization as soon as possible. Also critical is the ability to learn from past activities and to improve the service while providing it. Therefore, crucial to the component factory are creation and improvement of the models based on a methodology to systematize the learning and reuse process, and to make it more efficient. We will now briefly discuss our methodological approach to creation and improvement of models of experience.

The methodology to develop formal models of experience is provided by the *Goal/Question/Metric (GQM) approach* [Basili and Weiss 1984]. The GQM approach defines a model on three levels:

• Goal level:	a goal is defined that characterizes a certain organizational intent or problem
• Question level:	a set of questions is used to characterize in an operational way how a specific goal is going to be dealt with
• Metric level:	a set of metrics is associated with every question in order to answer it in a quantitative way

Each GQM model has, therefore, a three-level hierarchical structure (Figure 3), but several goals can use the same question and the underlying set of metrics, and several questions can share some metrics.

The formal models developed using the GQM approach are characterized by a combination of

• Object:	A model of the object that is represented by the GQM model, e.g. a process, a product, or any other kind of experience.
• Focus:	A model of the particular characteristic or property of the object that the GQM model takes into account.
• Viewpoint:	The perspective of the person or organization unit needing the information represented by the model.
• Purpose:	The purpose of the GQM model, e.g. characterization, evaluation, prediction, motivation, improvement.
• Environment:	The characteristics of the context where the analysis is performed.

The resulting GQM models are defined and constantly improved through use, learning from past experience, and translating this knowledge into changes to the model. This adds a new dimension to the characterization of a GQM model: its multiple evolving versions. The methodology that systematizes the learning and improving processes required to generate these versions is provided by the *Improvement Paradigm* [Basili 1984] and is outlined in the following steps (Figure 4b):

Figure 3

The Structure of a GQM Model

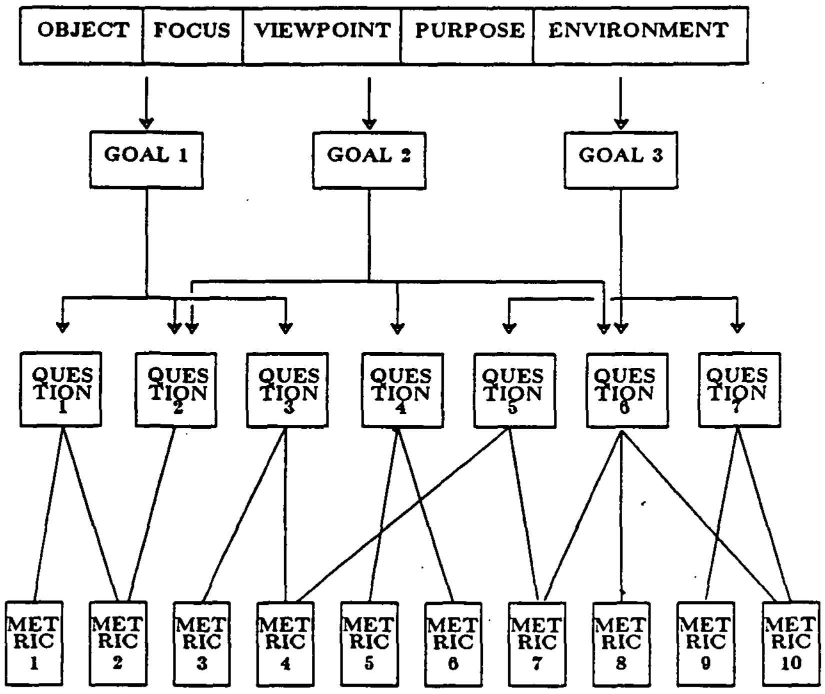


Figure 4a

The Deming Cycle

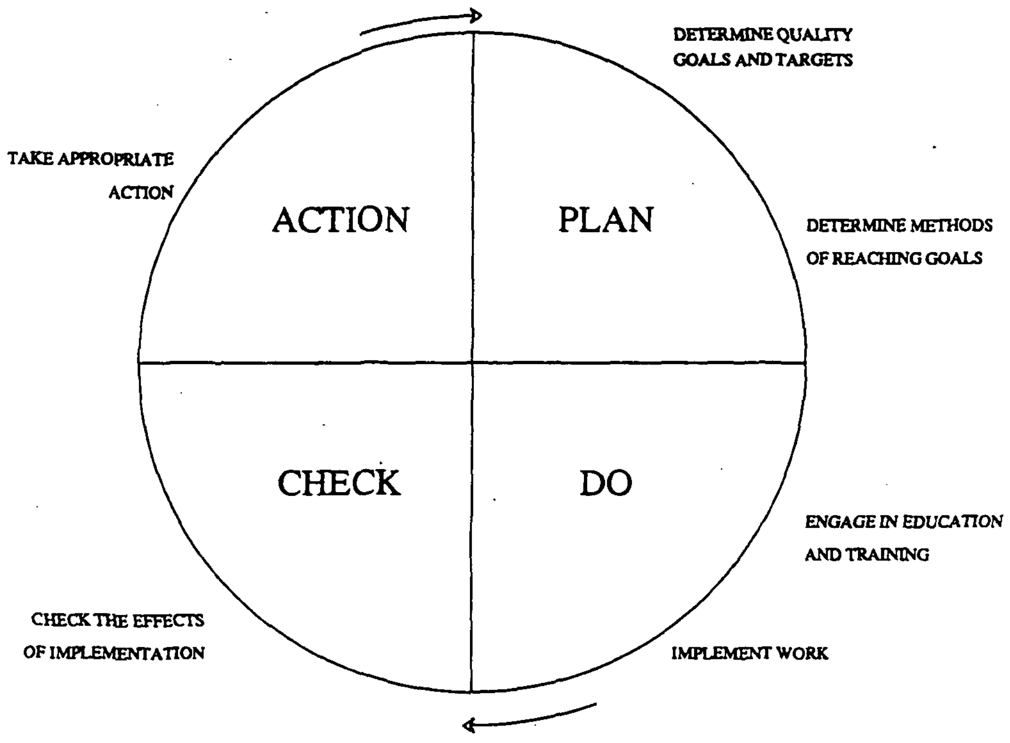
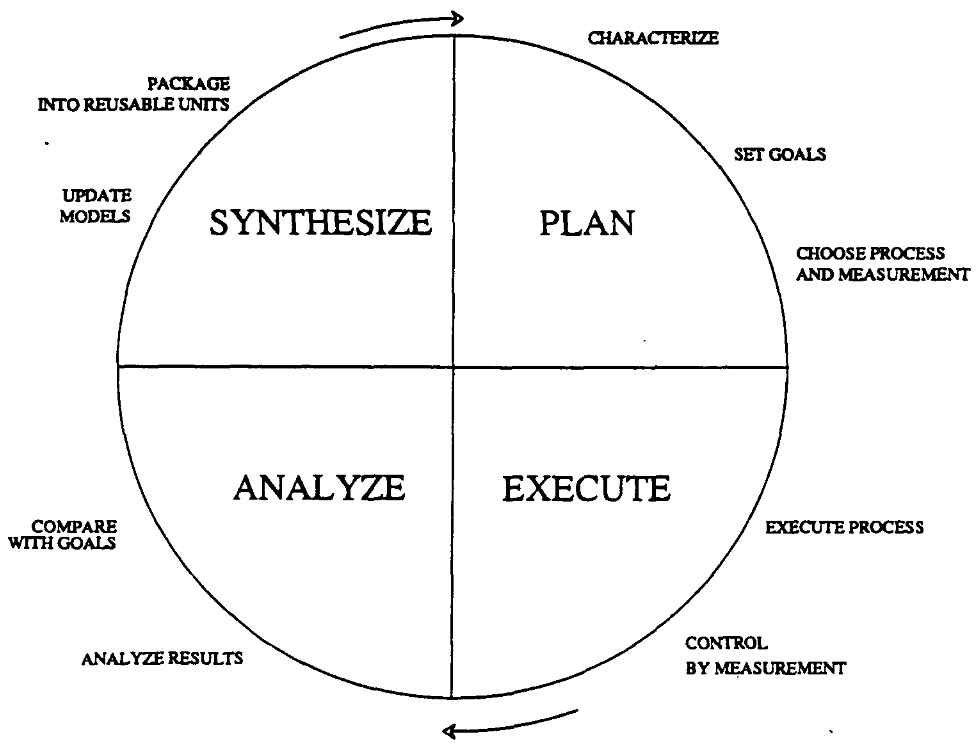

DETERMINE QUALITY GOALS AND TARGETS
DETERMINE METHODS OF REACHING GOALS
ENGAGE IN EDUCATION AND TRAINING
IMPLEMENT WORK
CHECK THE EFFECTS OF IMPLEMENTATION
TAKE APPROPRIATE ACTION
ACTION
PLAN
CHECK
DO

Figure 4b
The Improvement Paradigm



1. *Plan*

1.1 Characterize the activity and the environment in order to identify and isolate the relevant experience
1.2 Set the goals and refine them into a measurable form (this is done using the GQM approach)
1.3 Choose the execution process model, the supporting methods and tools, and the associated control measurement

2. *Execute* the process, control it, using the chosen measurement, and provide real-time feedback to the project organization

3. *Analyze* the results and compare them with the goals defined in the planning phase

4. *Synthesize*

4.1 Consolidate the results into updates to the formal models and to their relationships
4.2 Package the updated models into reusable units and store them for future reuse

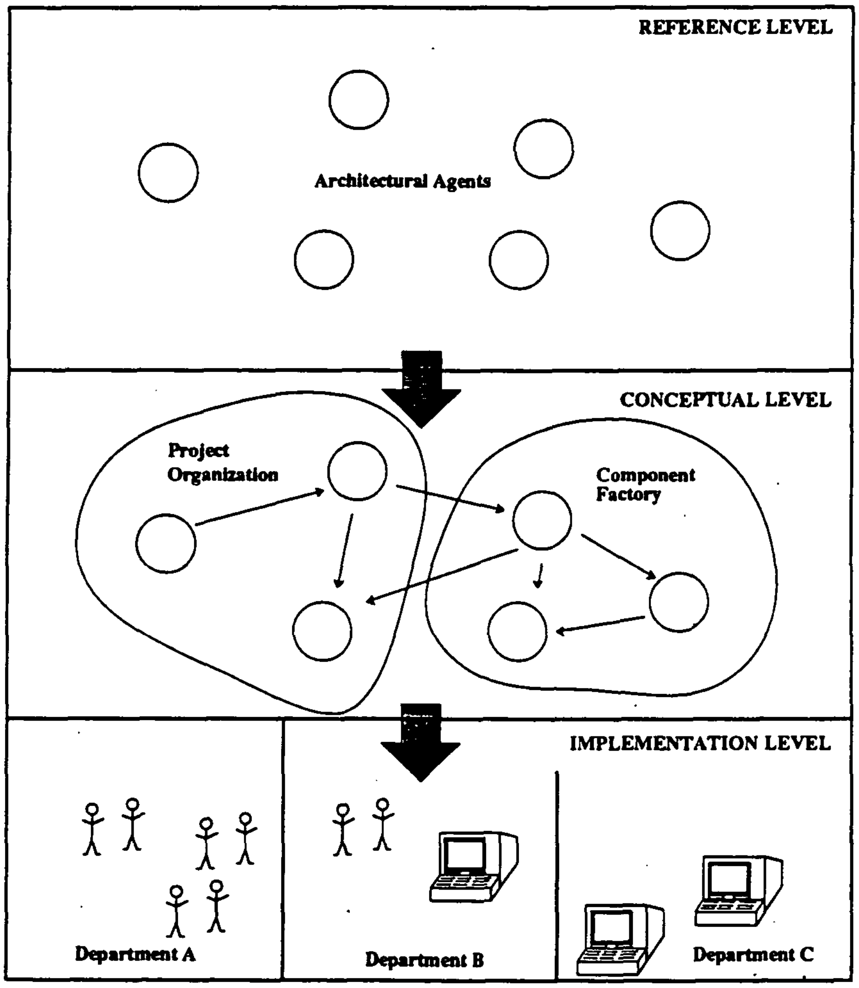
The improvement paradigm is based upon the notion that improving the software process and product requires the continual accumulation of evaluated experiences in a form that can be effectively understood and modified into a repository of integrated models that can be accessed and modified to meet the current needs. It is an evolution of the famous Shewart-Deming Cycle [Deming 1986] Plan/Do/Check/Act (Figure 4a) in an environment in which formalization and institutionalization of experience are critical factors to the performance of the whole process.

4. LEVELS OF REPRESENTATION OF A COMPONENT FACTORY

The experience factory and its specialization, the component factory, are necessarily very general organizational elements. Every environment has its characteristics and pursues certain goals by certain means different from every other one. Therefore we need different levels of abstraction in the description of the architecture of a component factory in order to introduce at the right level the specificity of each environment. The allocation of a function to an organizational unit is a first distinction, the actual implementation of some functions, for instance through automated tools is another distinction. However these choices are only variations of the paradigm of the component factory that can be captured using different levels of abstraction in representing the framework of the factory.

In this section we will discuss briefly the levels of abstraction that we want to use in order to represent the *architecture of a component factory* (Figure 5):

Figure 5
The Levels of Abstraction



10000174

- *Reference level*

  At this first and more abstract level we represent the building blocks and the rules to connect them, that can be used to represent an architecture. This is not the description of a component factory but a modeling language for it. The basic building blocks are called *architectural agents* and represent the active elements performing tasks within the component factory or interacting with it. They exchange among each other software objects and messages.

  Example:
  In the last section we have discussed the activities associated with the production of software through reusable components and decided that a reusable component may enter into the production process by direct development, by procurement from an external source, or by extraction and adaptation from pre-existing software. Each one of these possibilities becomes a function that can be assigned to an active element of the reference level:

  - an agent decides which reusable components are needed
  - an agent develops reusable components
  - an agent provides external off-the-shelf reusable components
  - an agent extracts and re-engineers reusable components
  - an agent manages the internal repository of reusable components

  These agents have potential connections with other agents: for instance, the agent that develops reusable components is able to receive specifications from another agent, and the agent that decides which reusable components are needed is able to provide these specifications, but the actual connection is not specified at the reference level.

- *Conceptual level*

  At this level we represent the interface of the architectural agents and the flows of data and control among them, and specify who communicates with whom, what is done in the component factory and what in the project organization. The boundary of the component factory, i.e. the line that separates it from the project organization, is defined at this level based on needs and characteristics of an organization, and can change with them.

  Example:
  In a possible conceptual architecture an agent that designs systems and orders components, located in the project organization, is connected with an agent that develops reusable components, located in the component factory. Specifications and designs are exchanged between these two agents.

In another conceptual architecture the agent that designs systems and orders components, still located in the project organization, communicates with an agent that coordinates the activities of component development and adaptation in the component factory. It is this agent that communicates with an agent that develops reusable components exchanging specifications and designs with it.

- *Implementation level*

  At this level we define the actual implementation, both technical and organizational, of the agents and of their connections specified at conceptual level. We assign to them process and product models, synchronization and communication rules, appropriate performers (people or computers) and specify other implementation details. The mapping of the agents over the departments of the organization is included in the specifications provided at this level of abstraction.

  Example:
  The organization is partitioned into the following departments
  - System Analysis and Deployment
  - System Development
  - Quality Assurance and Control
  - Software Engineering Laboratory

  In a possible organizational choice, the functions of component design are allocated to System Development, and the functions of system integration are divided between System Analysis and Deployment and Quality Assurance and Control. The Software Engineering Laboratory analyzes the production process, provides the models to control the activities and improves those models through experience. The Component Factory includes System Development and Software Engineering Laboratory. The process model for design is the iterative enhancement model: it starts from a group of kernel functions and delivers it to System Analysis and Deployment, and then it expands this group of functions in order to cover the whole set of requirements in successive makes of the system. The development is done in Ada using a language-sensitive structured editor and an interactive debugger.

Each one of these levels of abstraction can be divided into sub-levels by different operations applied to the agents or to their connections:

- functional decomposition: an agent is decomposed in a top-down fashion into more specific agents;

- connection decomposition or specialization: a connection between agents is decomposed into different pipelines, or the object types are refined, or synchronization rules are defined in greater detail;

The drawing of the boundary separating project organization and component factory at the conceptual level is motivated by the need for defining a formal interface between the two organizations, more formal than the one between the agents that lay inside each one. We can, for instance, imagine the boundary as a validation and verification checkpoint for the products that go across it. More generally, we can look at the couple project organization/component factory as at a user/server pair that can be used for quality measurement and improvement.

The next section will discuss in some detail the reference level and then we will deal with the problem of the instantiation of a generic architecture into a specific one.

5. THE REFERENCE ARCHITECTURE

While the reference level describes a generic architecture, the conceptual and implementation levels describe a specific one at different levels of detail. In order to obtain a specific architecture we can either design it from scratch or obtain it as an instantiation of a generic architecture. The problem with the architecture designed from scratch is that it doesn't give enough assurances it will be able to evolve in order to match the needs of the organization or to follow its evolution: designing from scratch is often focused on the needs and goals of the present organization and does not take into account the evolution of this organization.

The instantiation of a generic architecture is instead a technique that leads to more flexible architectures, because the adaptability is already in the abstraction. The specific architecture can be obtained from the abstract one through instantiation. It can be modified by altering the instantiation without changing the generic architecture, which is explicitly parametric and reusable. Besides, a generic architecture is a common denominator for comparing and evaluating different specific instantiations. It allows us to choose the architecture that is best suited for a particular organization.

The reference level provides this generic architecture. A *reference architecture for the component factory* is a description of the component factory in terms of its parts, structure and purpose, defining which parts might cooperate and to what purpose [Biemans 1986]. The interacting parts of the reference architecture are kept free from unnecessary linkages in order to keep the concerns separate and to leave this to the instantiation process. The genericness of the reference model is represented by the many possible ways of connecting those parts at the conceptual level for a specific architecture, and the many ways to map them into the implementation level.

The elements of the reference architecture, the architectural agents, are the components of the factory production process. We don't make any assumption about the way they are implemented: they can be a person, a group of people, a computer based system. The reference architecture specifies only their tasks and the necessary communication paths, leaving to particular instantiations their implementation and the specification of their nature. We can look at the agents as islands in the component factory, whose nature and implementation can be changed according to the needs and the improvement goals of the organization, independent of other agents. This allows us to implement the tasks of an agent today with a group of people, and tomorrow with a computer-based system, introducing changes that do not impact the whole organization.

From our discussion in section 3 of the activities associated with the production of software through reusable components, we can derive a set of functional requirements for the architectural agents operating in the component factory or interact with it:

- Receive the specifications of a system and produce its design taking into account the information about existing reusable software components made available by the component factory. These functions specify the role of the *Designer* agent.

- Build the specified system according to the design using the reusable components made available by the component factory, and perform the system test in order to verify the conformance of the developed system with the requirements. These functions specify the role of the *Integrator* agent.

- Choose how to fulfill a given request for a reusable software component, based on the specification of the component, and on the information about existing reusable software components already available in the component factory. The choice between development from scratch and adaptation of an existing component is probably done according to a model of cost and time effectiveness, i.e. whether it is worth modifying an existing component or the modification would be so substantial that it is more convenient to develop a new component. These functions specify the role of the *Shopfloor Coordinator* agent.

- Develop a reusable software component according to a given specification. The development can be done from scratch or assembling pre-existing components and elementary processes. It includes the design of the component, its implementation and verification. These functions specify the role of the *Developer* agent.

- Modify a reusable software component that is "close enough" to the one described by a given specification. This can be done generalizing the existing component, tailoring it to meet a specific need, combining it with other components, etc. The modification activities include the analysis of the impact of a certain adaptation, its implementation and verification. These functions specify the role of the *Adapter* agent.

- Produce and update reusable components based on domain knowledge, extract reusable components from existing code, and generalize already existing reusable components into other reusable components. The main difference between these functions and the development ones is that these are asynchronous with respect the production process in the project organization. These functions specify the role of the *Component Manipulator* agent.

- Develop formal models analyzing the experience developed in the component factory and in its interfaces with the project organizations. The experience that is collected and processed has different levels of formalization, according to the incremental perspective provided by the improvement paradigm. It starts with very simple models that are improved in a continuous way in order to fit better in the actual environment of the component factory. These functions specify the role of the *Experience Modeler* agent.

- Manage the collection of objects and information that is used by the component factory to store experience and products derived from its activities (experience base). In particular, this function includes the management of the repository of reusable software components, which is a subset of the experience base. Every agent, while performing its tasks, accesses the experience base either to use its contents or to record a log of its activity. The access control, the manipulation of objects and the search strategy to answer a request are the main experience base management functions. These functions specify the role of the *Experience Base Manager* agent.

- Supply commercial or public domain off-the-shelf reusable components that satisfy the specifications developed by the organization. This function specifies the role of an *External Repository Manager* agent

This list of agents represents a complete set of architectural agents that covers all the activities of the component factory and of the project organization. However, we could have defined the agents differently. For example, the reference level can be decomposed into several sub-levels, and the set of agents in the reference architecture presented here is one of these levels. Agents can be composed into larger ones or decomposed into more specific ones in a way that is very similar to the functional decomposition used in structured analysis. For instance, the Designer agent can be merged with the Shopfloor Coordinator obtaining an agent that specifies not only the components that are needed but also the way of obtaining them. Or, the Designer agent can be decomposed into a System Designer agent that performs the preliminary design of the system, and the Software Designer agent that performs the critical design.

It is also possible to expand the scope of the analysis, for instance by introducing the collection and analysis of requirements into the picture, and by specifying the agent that performs these functions in the reference architecture.

Besides a set of agents, the reference architecture contains a set of *architectural rules* that specify how the architectural agents can be configured and connected in the specific architectures derived from a reference architecture.

One set of rules deals with the presence and replication of the agents. The agents in the component factory can be unique or replicated: in the former case only one instance of the agent can be active, in the latter many instances of the agent are possibly active at the same time. For example: there might be two Designer agents that share the service provided by one Adapter agent.

Another set of rules deals with the connections between agents. The agents communicate with each other exchanging objects and experience at different levels of formalization, and cooperating towards the completion of certain tasks. This communication is realized through communication *ports*. A port is specified when we know what kind of objects can travel through that port. For instance: the Component Manipulator agent has a port through which it receives and returns reusable software components. There are data ports and control ports: the port through which the Designer agent receives the requirements for the system is a data port; the port through which it receives the process model to design the system is a control port. A port bundles several *channels*: each one is an elementary access point specifying the kinds of objects traveling through it and the direction. For instance: the port through which the Designer communicates, say, with the Shopfloor Coordinator has two channels: an output channel to send component specifications and an input channel to receive the requested reusable software components.

Ports can be mandatory, i.e. always present on the agent in one or more instances, or optional, i.e. possibly absent in certain implementations of the agent. For instance: the Developer agent has always a port through which it receives the specifications of the components it must develop and a port through which it returns the components it develops, these are mandatory ports. On the other hand, the Developer might or might not have a port to receive external off-the-shelf reusable components to be used in the development of the requested components. From the point of view of the reference architecture we don't make this choice, leaving it to the specific architecture.

We can represent our reference architecture using an Ada-like language in which the agents are task types that encompass port types in the way Ada task types encompass entries. This allows us to use the distinction between specification and body of the task type to defer the implementation of the agent to the specific architecture, and also to use Ada generics to represent certain abstractions that are specified at the conceptual level. In this representation, the architectural rules are declarative statements, incorporated in the definition of the agent they are applied to.

Without getting into the details of the representation, but to provide an example, we present a sample specification for the Developer agent

```
Generic

--  These variables are used as options in the ports that
--  are optional: their instantiation to "true" will imply
--  that the port is present.
        search_components is boolean;
        external_acquisition is boolean;
        recommended_components is boolean;

Task type Developer is

--  This is the port through which the agent receives the
--  specs and returns the components that have been
--  developed
Data port component_development
        (specs : in  component_specification,
        component : out  reusable_software_component)
end component_development;

--  This is the port for access to the internal components
--  repository: it is present if this access is permitted
Data port internal_components_acquisition
        options (search_components)
        (specs: out component_specification,
        components: in list_of_reusable_software_components)
        end internal_components_acquisition;

--  This is the port for access to external components
--  repositories: it is present if access is permitted:
--  there might be many instances of this port
--  corresponding to different repositories
Data port type external_components_acquisition
        options (external_acquisition)
        (specs: out component_specification,
        components: in list_of_reusable_software_components)
        end external_components_acquisition;

--  This port is used to receive components from another
--  agent (probably the Designer or the Shopfloor
--  Coordinator) and use them in the development of a new
--  component
Data port components_reception
        options (recommended_components)
        (components: in list_of_reusable_software_components)
        end components_reception;

--  The next two ports are used to interface with the
-- experience base
Data port activity_report
        (current_process_data: out process_data,
        current_product_data: out product_data,
        current_resources_data: out resource_data)
        end activity_report;

Control port models
        (current_process_model: in process_model,
        current_product_model: in product_model,
        current_resources_model: in resource_model)
        end models;

End Developer;
```

This *specification language* for the reference architecture is complemented by a *configuration language* whose purpose is to represent the choices made in order to instantiate the reference architecture into a specific conceptual architecture: presence and number of agents, presence and number of ports, connection of the ports.

In order to give an idea of the way this configuration language works, let's see a possible way of configuring a Developer agent. We do this by creating new tasks where

- it is specified what is the type of the agent that is being specified;
- the generic variables used as options in ports are instantiated in order to specify the presence of a port;
- instances of replicated ports are represented by declaring their port type;
- connections between agents are specified by a **connected with** statement in the specification of a port, declaring with which port of which agent it is connected.

The resulting task is conceptually represented by:

```
Task Developer_1 is new
        Developer (search_components := true; external_acquisition := true;
        recommended_components := false)

Data port component_development
    connected with Shopfloor_Coordinator_1.component_acquisition;

Data port internal_components acquisition
```

```
        connected with Experience_Base_Manager_1.component_supply;

    Data port repository_AAA is external_components_acquisition
        connected with External_Repository_AAA_Manager.component_supply;

    Data port repository_BBB is external_components_acquisition
        connected with External_Repository_BBB_Manager.component_supply;

    Data port activity_report
        connected with Experience_Base_Manager_1.reports;

    Control port models
        connected with Experience_Base_Manager_1.models_out;

    End Developer_1;
```

The implementation of the agents is specified by another language, the *implementation language*, that assigns to each task type a task body where **port** statements are associated with receive statements (like **accept** in Ada) having the same parameters.

6. INSTANTIATION OF THE REFERENCE ARCHITECTURE

The reference architecture has been defined as a collection of architectural agents and rules supported by an experience base managed by an agent. There are some degrees of freedom in this collection that are eliminated by choices made when the architecture is instantiated at the different levels of abstraction:

A. Choices at the conceptual level:

a. Boundary between component factory and project organization. In making this choice one tries to optimize reuse, on one hand, by incorporating more functions into the component factory, and to optimize customer service, on the other hand, by concentration of the appropriate activities in the project organization.

b. Presence of agents, number of agents of each type, fusion of several agents. This choice is about communication complexity: a large number of agents increases the complexity and the overhead due to communication, a small number of agents produces bottlenecks that would affect the whole organization.

c. Presence of ports and number of ports. As in case b., this choice deals with the communication complexity: a large number of ports increases the complexity of the activities of a single agent but reduces the impact of possible bottlenecks.

d. Interconnection of ports between different agents. This choice is about distribution of control: concentrating the control in a small group of agents makes planning easier, but serializes many activities that could be otherwise performed concurrently.

B. Choices at the implementation level:

a. Distribution of the agents over organizational units. This choice deals with the optimization of the already existing organization units and the smooth evolution to factory concepts. It takes into account the available resources and the historical roles of those units.

b. Implementation of the functions of the agents (the task bodies). In choosing algorithms, procedures, methods and tools one tries to achieve an organizational and technical profile that is correct, efficient and best suited to the overall mission by dealing with the available resources and technology.

Therefore, in order to design a specific component factory, we need to instantiate the reference architecture by an instantiation process based on the levels of abstraction introduced earlier (Figure 6). This instantiation process is embedded in the methodological framework of the improvement paradigm now applied to the specific architecture. The four steps of the paradigm, introduced previously in a different context, become in this context:

1. *Plan component factory*: the desired instantiation is designed based on the characteristics of the organization and on the goals to be achieved:

1.1 Characterize the activities and the environment of the current organization: production process, products, formal and informal models currently in use, software tools, standards, etc.;

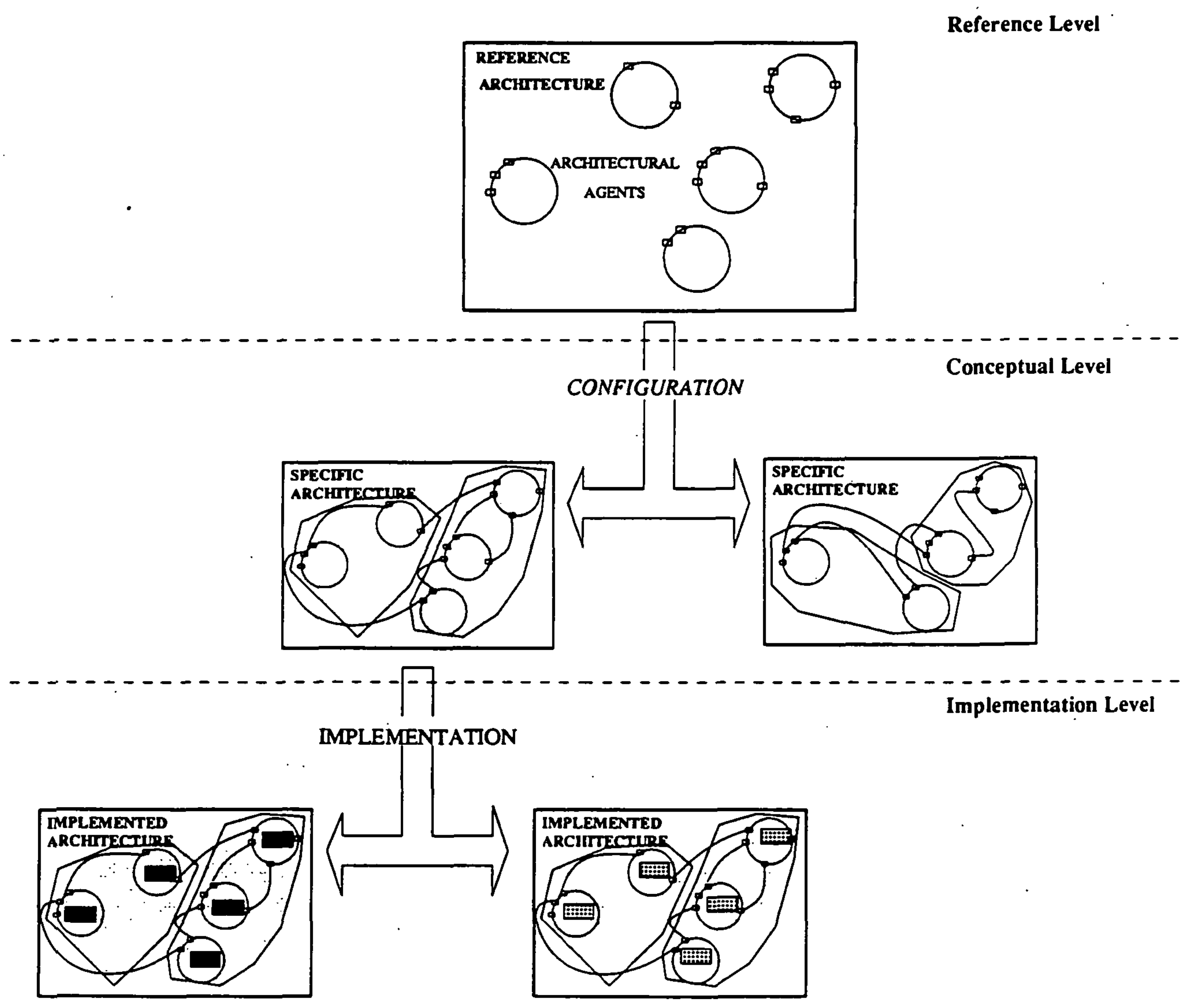
1.2 Set goals and priorities for the introduction of the component factory and for its separation from the project organization: productivity, customer satisfaction, product maintenance, environment stability. The goals can be refined into questions and metrics that will be used to control the production of the component factory.

1.3 Instantiate the reference architecture into a particular component factory architecture and define the associated measurement environment:

Instantiation process

A. *Configuration of the architecture*
(reference level → conceptual level)

Figure 6
Instantiations of the Reference Architecture



A.1 Definition of the activities of the organization and mapping of those activities into specific architectural agents

A.2 Identification of the boundary of a specific factory by specifying which agent is in the project organization and which one is in the component factory.

A.3 Definition of the conceptual representation of the specific component factory by specifying the agents and connecting their ports using the configuration language.

B. *Implementation of the architecture*
(conceptual level → implementation level)

B.1 Specification of the mapping between the agents and their functions and the departments of the organization, and of the responsibilities for production control.

B.2 Definition of the implementation representation of the specific component factory by mapping agents and functions over specific units (e.g. people, automated or semi-automated tools), and specifying algorithms, protocols and process models.

2. *Produce components* for the project organizations and load products and information into the experience base:

2.1 Execute the production process using the particular architecture that has been defined;

2.2 Control the process while executing by using the measurement environment that has been defined.

3. *Analyze the results*, after a pre-established period of time, assessing the level of achievement of the goals that were behind the introduction of the component factory.

4. *Synthesize the results*

4.1 Consolidate the results of the analysis into plans for new products, models, measures, etc., or for updates for the existing ones

4.2 Package the new and updated products, models, measures into reusable units and store them for future reuse;

4.3 Modify the instantiation of the particular architecture and the measurement environment associated with it.

The crucial point of the process is the possibility, offered by the reference architecture, of modifying the particular architecture without modifying the interfaces between its building blocks. The modular structure allows configuration and reconfiguration of the processes as required by an efficient and realistic implementation of an optimizing paradigm. The evolution of the conceptual level and sub-levels is more difficult because it has impact on the implementation level, but the explicit definition of the interface types, which is part of the reference architecture, offers a certain freedom in the evolution, even at the conceptual level. Changes in the automation and organizational choices have definitely a lower impact, if they are applied to the implementation level leaving unchanged the conceptual level.

7. EXAMPLES OF COMPONENT FACTORY ARCHITECTURES

7.1 CLUSTERED AND DETACHED ARCHITECTURES

In order to illustrate the concepts of reference architecture and instantiation we can present two different conceptual architectures for the component factory.

The two architectures differ for the different role they assign to the Designer agent:

- in the first architecture the Designer coordinates all software development activities from the side of the project organization, we call it "clustered" architecture;

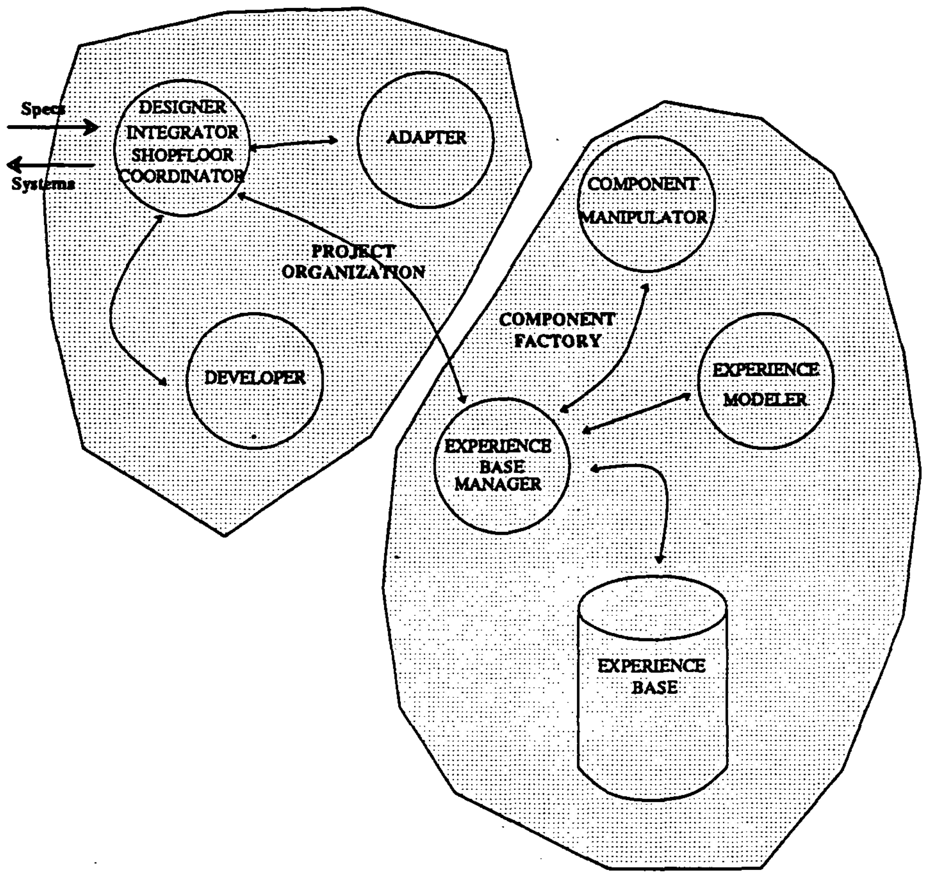
- in the second architecture the development activities are concentrated in the component factory under the control of the Shopfloor Coordinator agent, we call it "detached" architecture.

In a *clustered component factory architecture* (Figure 7a) every development takes place in the project organization and the role of the component factory is to perform the activities of processing and providing existing reusable software components.
The agents are assigned in the following way

Project Organization	**Component Factory**
· Designer/Integrator/ Shopfloor Coordinator	· Component Manipulator
· Developer	· Experience Modeler
· Adapter	· Experience Base Manager

Figure 7a

The Clustered Architecture



In a *detached component factory architecture* (Figure 7b)no development takes place in the project organization but only design and integration. The project organization develops its design of the system based on the information existing in the experience base and requests from the component factory all necessary developments. Then it integrates the components received from the component factory according to the design. The agents are assigned in the following way

Project Organization	**Component Factory**
· Designer	· Shopfloor Coordinator
· Integrator	· Developer
	· Adapter
	· Component Manipulator
	· Experience Modeler
	· Experience Base Manager

The activities of the agents that form the kernel of the component factory (Component Manipulator, Experience Modeler, Experience Base Manger) don't change in the two instantiations.However, the role of the component factory in the detached architecture is much more relevant because it encompasses activities that are both synchronous and asynchronous with the project organization.

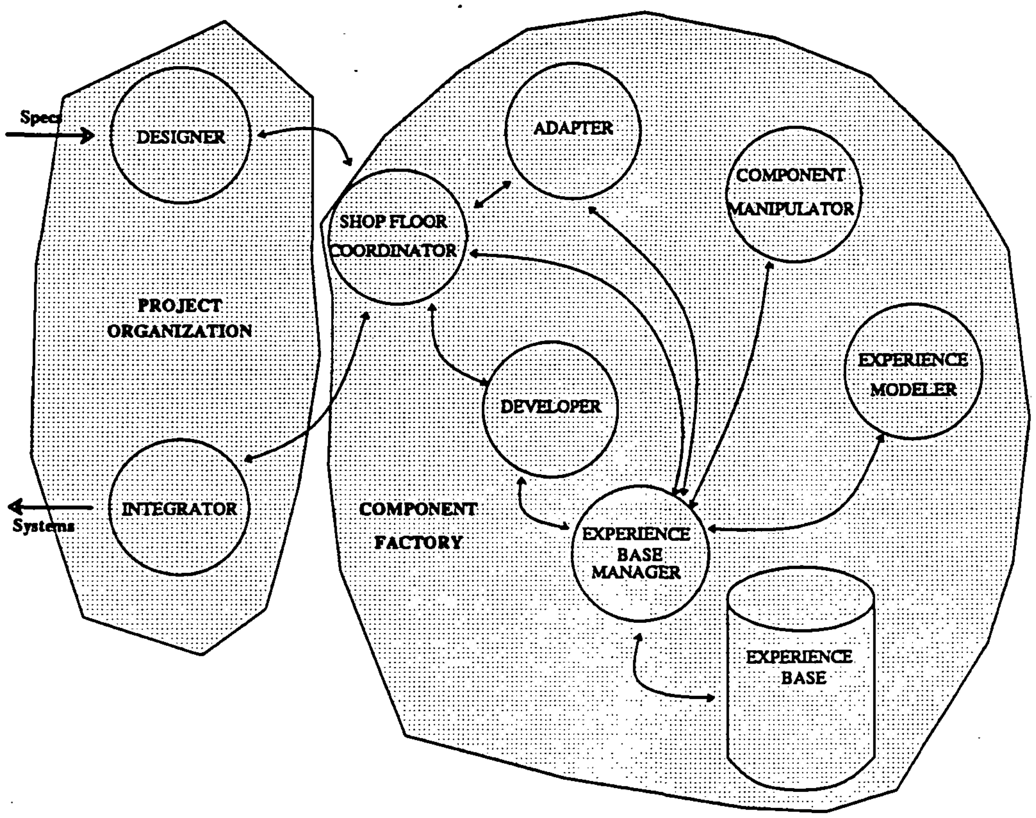
An evaluation of the two instantiations can be performed using the GQM approach mentioned earlier. In order to compare the two architectures we can develop, for instance, the GQM models having as

• Object :	the clustered and the detached architecture;
• Focus:	characteristics such as performance, functionality and evolutionary nature;
• Viewpoint:	the technical management of the organization
• Purpose:	the evaluation of the architecture
• Environment:	the specific organization

From these goals it is possible to derive questions and metrics that allow us to collect data to perform the comparison. Without getting into the details of a particular application, we can make the following general remarks, based on the characteristics that we have listed as focus of the evaluation:

Figure 7b

The Detached Architecture



10000174

- Performance: the level of productivity and serviceability of the project organization/component factory system.

    - In the clustered architecture, the project organization develops the components that are not available, therefore, if it has enough resources, it performs probably faster because there is less communication overhead and more pressure for their delivery. On the other hand the components developed in the framework of a project are more context dependent and this puts more load on the component factory and in particular on the Component Manipulator.

    - In the detached architecture there is more emphasis on developing general purpose components in order to serve more efficiently several project organizations: planning is easier and the optimization of resources is more effective. On the other hand there are more chances for bottlenecks and for periods of inactivity due to a lack of requests from the projects, that would affect the overall performance of the organization.

- Functionality: the conformance to the operating characteristics of an organization producing software using reusable components

    - In the clustered architecture all functions are implemented but the most critical ones are concentrated on the Designer/Integrator/Shopfloor Coordinator. This means that errors and operating failures of this agent can affect the functionality of the whole organization.

    - In the detached architecture the high modularity of functions reduces the impact of errors and failures of one agent but increases the possibility of communication errors.

- Evolutionary nature:

    - The clustered architecture is much closer to the way software is currently implemented and therefore its impact on the organization would be less drastic.

    - The detached architecture provides the component factory with enormous possibilities for adaptation and configuration making continuous improvement easier and less expensive.

The detached architecture is probably better suited for environments where the practice of reuse is somewhat formalized and mature. An organization that is just starting should probably instantiate its component factory using the clustered architecture and then, when it reaches a sufficient level of maturity and improvement with this architecture, start implementing the detached architecture to continue the improvement. The improvement paradigm, as applied to the component factory in the last section, provides a methodology for a step-by-step approach to this implementation. In this way the organization takes advantage of the flexibility and evolutionary nature of this approach, that are among the primary benefits of reasoning in terms of instantiations of a reference architecture.

7.2 THE TOSHIBA SOFTWARE FACTORY

A further illustration of the concepts of reference architecture and instantiation comes from the analysis of a real case study.

One of the most significant accomplishments in the attempt to make software development into an industrial process is represented by the experience of Toshiba Corp. in establishing, in 1977, the Fuchu Software Factory to produce application programs for industrial process control systems [Matsumoto 1987]. In 1985 the factory employed 2,300 people and shipped software at a monthly rate of 7.2 million EASL[1] per month.

The organizational structure of Fuchu Software Factory is designed to achieve a high level of reusability (Figure 8). Projects design, implement and test the application systems reusing parts that are found in a Reusable Software Items Database. Just to give an idea of the size of an application system developed by a project, we have an average size of 4 million EASL but there are projects that go up to 21 millions EASL. The parts are made available by a Parts Manufacturing Department based on the requirements specified by a Software Parts Steering Committee made of Project people and of Parts manufacturing People. Statistics on alteration and utilization of parts are processed and maintained by the Parts Manufacturing Department.

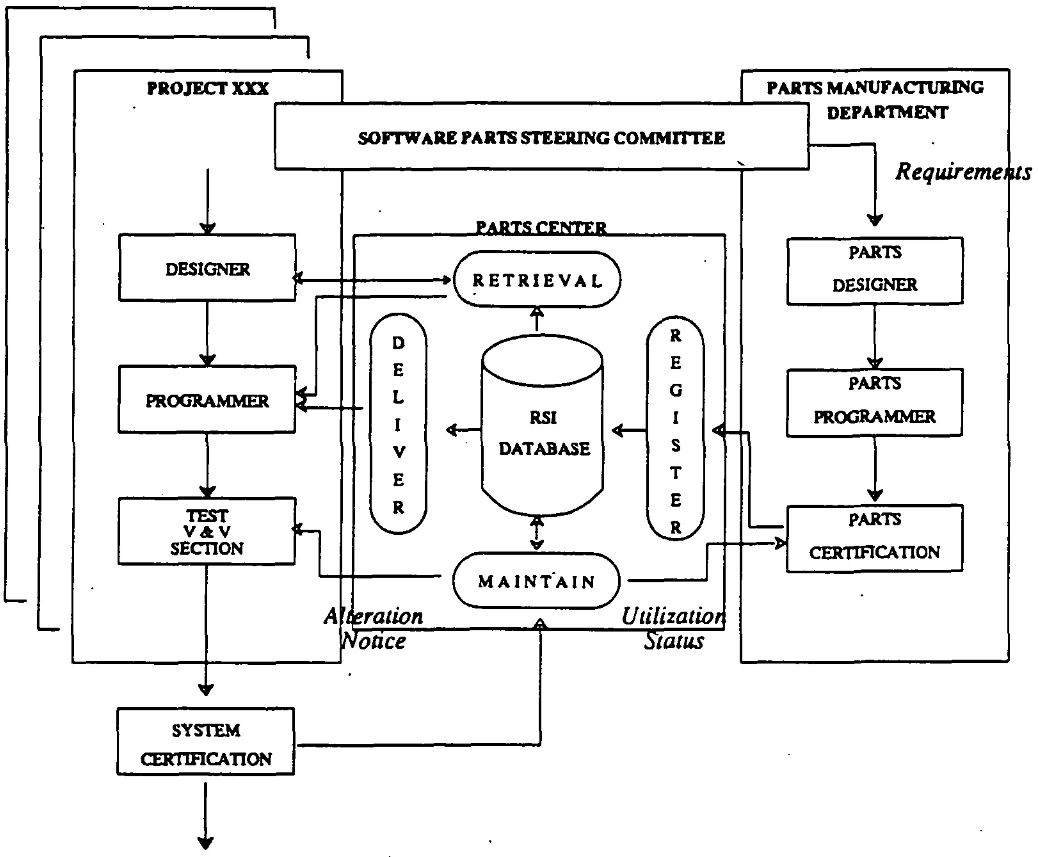
The conceptual architecture of the Fuchu Software Factory (Figure 9) presents the replication of some agents:

- Project Organization
  - Shopfloor Coordinator: this agent performs the functions of the Software Parts Steering Committee. It is very much project oriented but some of its functions, such as planning for reuse can be identified with some functions of the Experience Modeler in the reference architecture therefore we can position it at the border between project organization and component factory.
  - Designer 1: this agent designs the application system and the components that have been deemed project specific by the Coordinator.
  - Developer 1: this agent develops the software components specified by Designer 1.

[1] EASL: Equivalent Assembler Source Line of code

Figure 8

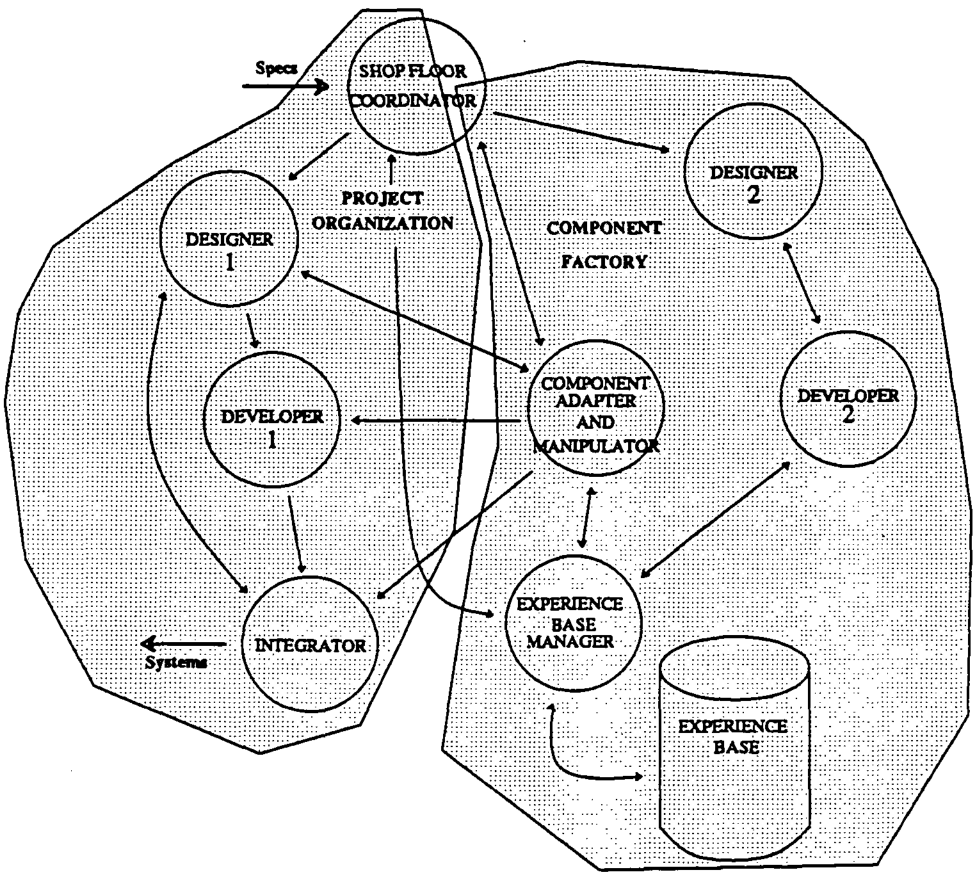
The Toshiba Fuchu Software Factory



10000174

Figure 9

The Toshiba Software Factory Architecture



10000174

- Integrator: this agent assembles the system using the components received from Developer1 and from the component factory according to the design provided by Designer 1, and verifies its conformance with the requirements.

- Component Factory

  - Designer 2: this agent designs the components that have been deemed reusable by the Coordinator.

  - Developer 2: this agent develops the software components specified by Designer 2.

  - Component Adapter and Manipulator: this agent can be identified with the Parts Manufacturing Department that adapts and supplies parts under request and, based on the statistics on the utilization of the software parts contained in the database, modifies and improves those parts.

  - Experience Base Manager: this agent is in charge of the management of the Parts Center and, in particular, of the access to the Reusable Software Items Database.

A function that is not explicitly implemented in this architecture is the experience modeling function even though the factory uses state-of-the-art techniques to manage its projects. The absence of formal experience modeling is probably one of the causes for which, according to the data reported by Matsumoto, the major factor affecting productivity is the reuse of code (52.1%) while improvement of processes, techniques and tools have a less significant impact.

8. CONCLUSIONS

Flexible automation of software development combined with reuse of life cycle products seems the most promising program to solve many of the quality and productivity problems of the software industry. It is very likely that in the coming years we will see a wide and deep change in this industry, similar to the one that took place in manufacturing through the introduction of CIM (Computer Integrated Manufacturing).

The abstraction levels and the reference architecture presented in this paper are aimed at providing a framework to make both automation and reuse happen. The major benefits of this approach are

- a better understanding of reuse and of the requirements that need to be satisfied in order to implement it in a cost-effective way;

- the possibility of using a quantitative approach, based on models and metrics, in order to analyze the tradeoffs in design associated with the component factory use;

- the formalization of the analysis of the software development process and organization with a consequent enhancement of the possibilities for automation;

- the definition and the use of an evolutionary model for the improvement of a reuse-oriented production process.

One of the major problems the software industry is facing today, when using automated production support tools, like the CASE tools (application generators, analysis and design tools, configuration management systems, debuggers, etc.), are rigidity and lack of integration. These problems affect dramatically also the chances to reuse life cycle products across different projects.

Tools modeled over the reference architecture would represent a significant step towards the solution of those problems because the interfaces would be specified and standardized. An organization would have the possibility of using different set of methodologies and tools in different contexts without dramatic changes to the parts of the organization that are not affected by the specific choice. Besides, different alternatives can be analyzed and benchmarked based on the input provided by the experience base on the performance of methods and tools in similar situations.

This aspect of simulation based on historical data and formal models, is one of the most important benefits of the proposed approach and is one of the focuses of our research. The development of a complete specification for a component factory and its execution in a simulation environment using historical data as well as the study of the connection between application architecture and factory architecture will be the main goals of our future work in this field.

8. REFERENCES

[Arango 1989]
G. Arango, "Domain Analysis: From Art to Engineering Discipline," *Proceedings of the Fifth International Workshop On Software Specification and Design* (*Software Engineering Notes*, Vol. 14, No. 3), May 1989, pp. 152 - 159.

[Basili 1984]
V.R.Basili, "Quantitative Evaluation of Software Methodology", *Computer Science Technical Report Series*, University of Maryland, College Park, MD, July 1985, CS-TR-1519.

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

[Basili and Rombach 1991]
V.R.Basili, H.D.Rombach, "Support for Comprehensive Reuse", Software Engineering Journal, July 1991, (also, *Computer Science Technical Report Series*, University of Maryland, College Park, MD, February 1991, CS-TR-2606 and UMIACS-TR-91-23).

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

[Biemans 1986]
F.Biemans, "Reference Model of Production Control Systems", in *Proceedings of IECON 86*, Milwaukee, September 29 - October 3, 1986.

[Caldiera and Basili 1991]
G.Caldiera, V.R.Basili, "Identifying and Qualifying Reusable Software Components", *IEEE Computer*, Vol.24, No.2, Feb.1991, pp.61-70.

[Caldiera 1991]
G.Caldiera, "Domain Factory and Software Reusability", Proceedings of the Software Engineering Symposium S.E.SY. 1991, Milano, Italy, May 1991.

[Cusumano 1989]
M.A.Cusumano, "The Software Factory: A Historical Interpretation", *IEEE Software*, March 1989, pp.23-30.

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

[Freeman 1983]
P.Freeman, "Reusable Software Engineering Concepts and Research Directions", *ITT Proceedings of the Workshop on Reusability in Programming*, 1983, pp.129-137.

[Joo 1990]
Bok-Gyu Joo, "Adaptation and Composition of Program Components", PhD Thesis, Department of Computer Science, University of Maryland, College Park, MD, January 1990.

[Matsumoto 1986]
Y.Matsumoto, "Management of Industrial Software Production", *IEEE Computer*, Vol.17, No.2, February 1984, p.59-72.

[Matsumoto 1987]

Y.Matsumoto, "A Software Factory: An Overall Approach to Software Production", in P.Freeman (Ed.), *Tutorial: Software Reusability*, Computer Society Press, Washington, DC, 1987, pp.155-178.

[McIlroy 1969]

M. McIlroy, "Mass Produced Software Components", Software Engineering Concepts and Techniques, *Proceedings of the NATO Conference on Software Engineering*, 1969.

[Neighbors 1989]

J.M.Neighbors, "Draco: A Method for Engineering Reusable Software Systems", in T.J.Biggerstaff and A.J.Perlis (Eds.), *Software Reusability - Volume 1: Concepts and Models*, ACM Press, New York, NY, 1989, pp.295-319.

[Tracz 1987]

W. Tracz, "Ada Reusability Efforts: A Survey of the State of the Practice," *Proceedings of the Joint Ada Conference, Fifth National Conference on Ada Technology and Washington Ada Symposium*, U.S. Army Communications-Electronics Command, Fort Monmouth, New Jersey, pp.35-44.

UMIACS-TR-91-69 May 1991
CS-TR-2672

A Pattern Recognition Approach for Software Engineering Data Analysis*

L.C. Briand, V.R. Basili and W.M. Thomas

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

Abstract

In order to understand, evaluate, predict, and control the software development process with regard to such perspectives as productivity, quality, and reusability, one needs to collect meaningful data and analyze them in an effective way. However, software engineering data have several inherent problems associated with them and the classical statistical analysis techniques do not address these problems very well. In this paper, we define a specific pattern recognition approach for analyzing software engineering data, called Optimized Set Reduction (OSR), that overcomes many of the problems associated with statistical techniques. OSR provides mechanisms for building models for prediction that provide accuracy estimates, risk management evaluation and quality assesssment. The construction of the models can be automated and evolve with new data over time to provide an evolutionary learning approach (the Improvement Paradigm) to software modeling and measurement. Experimental results are provided to demonstrate the effectiveness of the approach for the particular application of cost estimation modeling.

*Research for this study was supported in part by NASA grant NSG 5123, Coopers & Lybrand (IAP member) and Vitro Corporation (IAP member).

1 Introduction

Managing a large scale software development requires the use of quantitative models to provide insight and support control based upon historical data from similar projects. This implies the need for a quantitative approach.

- to build models of the software process, product, and other forms of experience (e.g., effort schedule, reliability, ...) based upon common characteristics for the purpose of prediction.
- to recognize and quantify the influential factors (e.g., personnel capability, storage constraints, ...) on various issues of interest (e.g., productivity improvement, effort estimation, ...) for the purpose of understanding and monitoringthe development.
- to evaluate software products and processes from different perspectives (e.g., productivity, fault rate) by comparing them with projects with similar characteristics.
- to identify strengths and weaknesses in the current environment.
- to understand what we can and cannot predict and control so we can monitor it more carefully.

In Section 2 of this paper, we discuss the needs and the constraints in building effective models for the software development environment. In Section 3, we review the failings of the classical statistical approaches with regard to software engineering data. We offer a new approach for analyzing software engineering data in Section 4, called Optimized Set Reduction (OSR), that overcomes many of the problems associated with statistical techniques. The approach is based upon pattern recognition techniques tailored to the software engineering field and offers advantages that overcome many of the problems associated with statistical techniques.

Besides overcoming some of the drawbacks of statistical analysis for software engineering model building, OSR provides mechanisms for building models for prediction that provide estimates of the accuracy of the prediction, models for risk management evaluation for the risk areas of interest, and quality assessment relative to the pertinent quality models.

The construction of the models can be automated and evolve with new data over time to provide an evolutionary learning approach to software modeling and measurement in order to support software development management.

In Section 5, experimental results are provided to demonstrate the effectiveness of the approach for the particular application of cost estimation modeling. The data set used are the COCOMO [BOE81] and Kemerer [KEM87] data sets. Although the particular example of cost estimation is chosen, the approach is also being used for other forms of model building, e.g., the building of models for maintenance.

Section 6 discusses some general issues related to multivariate data analysis in the context of OSR. A paradigm that defines the learning aspects of software development and management (called the Improvement Paradigm) is described in Section 7 and it is shown how OSR performs with the learning and model refinement issues in the IP framework.

2 Requirements for an Effective Data Analysis Procedure

Based upon the constraints associated with the data and the analysis procedures, we generate a set of requirements for model building approaches. In the text that follows, we will refer the variable to be assessed as the "Dependent Variable" (DV)(e.g. productivity, fault rate) and the variable explaining the phenomenon as "Independent Variables" (IV) (e.g. personnel skills, data base size) [BOE81]. We will see later on that all the issues related to prediction, evaluation and risk management may be formalized under this form. The various IVs will form the dimensions of an Euclidian space called the "sample space" in the following text.

2.1 Constraints related to Software Engineering Data

Model building in Software Engineering is faced with the following difficulties:

- C_1: There is no theory proven to be effective in any environment that would give a formal relationship among measured metrics in the development process. Therefore the capabilities of classical statistical approaches seem very limited and statistical simulation (i.e. Monte Carlo approach) appear improbable.
- C_2: The best we can do is make assumptions about the probability density distributions, with respect to the dependent and independent Variables of interest, with very little evidence to support our assumptions.
- C_3: The sample size is usually small relative to the requirements of the classical statistical techniques, the quality of the data collected, and the number of significant independent variables (IV). This is due to the nature of the studied objects in software engineering and it is difficult to avoid (e.g. software system, module, change, defect ...).
- C_4: "Software engineering modelers" have to deal with missing, interdependent and non-relevant independent variables: This is due to a lack of understanding of the software development process.
- C_5: Both data defined on a continuous (i.e. ratio, interval) and a discrete (i.e. nominal, ordinal) range have to be handled. Collecting data in a production environment is a difficult task and discrete data collection is sometimes performed to facilitate the measurement process. Also, the nature of some of the data may be discrete.

2.2 Requirements to alleviate these constraints

Matching the constraints, we can define requirements for effective data analysis procedures by the following list:

- R_1 [matches C_1, C_2]: The data analysis procedure should avoid assumptions about the relationships between the Variables regarding the probability density distribution on the IV and DV ranges.

- R_2 [C_3, C_4]: A mechanism is needed to evaluate accuracy for each performed estimation. The variations of accuracy lie in a large range depending on the object to be assessed. For example, you may want to assess a software project from the point of view of productivity.

  C_3 The amount of available relevant data may differ according to the characteristics of the project to be assessed (i.e. location of the project in the sample space). For example, you may have more data with respect to data processing business applications than with respect to real time systems. In small samples, this phenomenon can have significant consequences.

  C_4 The performed data collection may be more suitable to certain kinds of objects than others. For example, measuring objectively time constraints for real time systems may be difficult and therefore may introduce uncertainty in assessement.

- R_3 [C_4]: The data analysis procedure must be as robust as possible to missing, non-relevant, interdependent IVs and outliers. Then, some procedures must be available in order to detect and alleviate the effects related to these kind of disturbances in the data set.

- R_4 [C_5]: The data analysis procedure must be able to handle easily both discrete and continuous metrics without biasing the results obtained.

3 Statistical Background in the Field

Most of the studies in software engineering have been based on multiple regression and related techniques (e.g. F tests, stepwise selection). These procedures do not seem to match most of the previously defined requirements for the following reasons:

- The adjusted coefficient of correlation is a parameter globally calculated over all the data points. It is the ratio of the variance explained by the calculated correlation versus the variance still unexplained. This does not provide accuracy of individual estimations (see requirement R_2). Calculating confidence intervals does not seem possible with an acceptable accuracy. Their calculations are based on an approximation of the distribution (i.e. normal) and variance of the residuals in the studied population. Too few data are usually available to make a reasonable estimation of the variance. Moreover, we should not expect a constant variance all over the regression space. The disturbance due to some missing independent variables in the regression equation may have a variable intensity in different parts of the regression space. Considering that in the field of Software Engineering we already make approximations on the mathematical shape of the regression function, with weak theories to support our assumptions, this variability may be significant.

- This technique is very sensitive to perturbations related to **missing, interdependent** and **non-relevant** metrics (i.e. independent variables) [DIL84]. Procedures to deal with these problems are complex and errorprone (e.g stepwise selection, F partial tests, detection of outliers). These problems make difficult any quantitative analysis of the development environment and affect the accuracy of predictions.

- **Continuous** metrics are at times difficult to compare because the calculated regression coefficients are dependent on their corresponding measurement units. The classical way to solve this problem is to work with Beta coefficients (i.e. standardized regression coefficients). The dependent variable and the independent variables are divided by their standard deviation before calculating the regression equation. In other words, IVs and DVs are normalized to their variability. This way, the used regression parameters become unitless numbers and are independent of the magnitude of units. However, the magnitude of the Beta coefficients are dependent on the variability of the independent variables in the particular sample that provide the data. This is another cause of innacuracy in the performed estimations.
- **Discrete** metrics are very difficult to handle because the notion of distance between categories would be required to build Multiple Regression based models. Then, some strong assumptions are necessary to integrate these data in a regression calculation (i.e. assumptions about the relative distances between the defined categories). The usual way to handle categorical data is to create a regression parameter for each category of each categorical metric (i.e. dummy variable). For example, The value of each parameter is either 1 or 0 according to the categories to which the object to be assessed belongs. Two major drawbacks may be identified:
  - The number of regression parameters increases rapidly with the number of categorical metrics.
  - The discrete values (e.g. $0, 1$)assigned to these parameters are arbitrary and then make the calculated regression coefficient meaningless for analyzing the respective influence of the various regression parameters. No beta-coefficient can be calculated.

There have been some significant attempts to develop alternative techniques to regression based procedures to analyze software engineering data. The use of statistical classification techniques has been used to make predictions with respect to errorprone modules in large scale software systems [SEL88]. These experiments showed encouraging and interesting results. However, if we consider the usual data availability in Software Engineering, the used data set was very large (5000 data points). The objective was simply to classify objects among two categories (i.e. non errorprone and errorprone modules) and not to come up with a comprehensive data analysis procedure. These experiments opened a door to a huge research field to be investigated.

4 A Pattern Recognition Approach for Analyzing Data

Based upon pattern recognition principles [TOU74], we propose a data analysis procedure that is intended to fulfill, to a certain extent, the previously described requirements for effective data analysis. This procedure and the main principles supporting are described in this section.

4.1 Description of the Main Underlying Technique

The technique has as its goal the recognition of patterns in a data set. These patterns are used as a basis for understanding and assessing the development process, product and environment.

4.1.1 The Basic Concepts and Terminology

- A **learning sample** consists of N vectors $(DV_i, IV_{1,i}, ..., IV_{n,i})$, $i \in (1, ..., N)$, containing one dependent and n independent variables. These vectors are defined in an Euclidian space called the "sample space". These vectors, which we will call **pattern vectors**, represent measurements taken in the environment.

- A **measurement vector** is defined as the set of independent variable values representing a particular object whose dependent variable value is to be predicted. That is, it is a pattern vector without the dependent variable.

- To be able to make predictions on the **dependent variable**, its range has to be sub-divided or grouped into what we will call **DV classes**. These classes correspond to natural situations that can be encountered in the measurement environment, with respect to the dependent variable, e.g. productivity. If the Dependent Variable is either "ratio", "interval" or "ordinal", the dependent variable range is sub-divided into intervals, if the Dependent Variable is "nominal", categories may be grouped into a smaller set of classes. They are called "states of nature" in decision theory and "pattern classes" in the pattern recognition field [TOU74]. We have chosen the name DV classes in order to make the connection with a classical statistical approach for multivariate analysis.

- To be able to use the **independent variables** as a basis for predicting the dependent variable, they, like the Dependent variables, must be mapped into **IV classes** by subdividing or grouping.

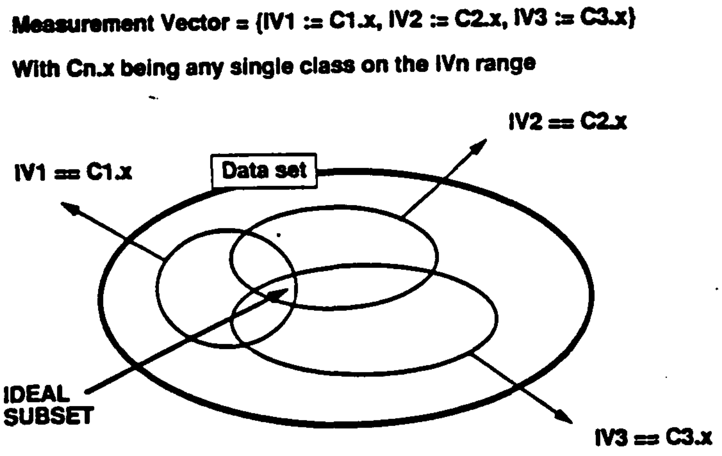
- A **pattern** is defined as a non-uniform distribution of probabilities across the DV classes.The further a distribution is from uniformity, the more the pattern is considered as significant (measurable metric developed below).

4.1.2 A Particular Pattern Recognition Process

The problem of predicting the dependent variable for a particular project can be stated as follows: Given a particular measurement vector (MV), determine the probability that the actual dependent variable value lies in each of the DV classes. The shape of the probability density function on the DV class range associated with MV is unknown. The goal and the basic principle of this process is to find a subset of pattern vectors in the data set, whose values for the independent variable are similar to the values for the independent variable of MV and show a significant pattern among the DV classes.

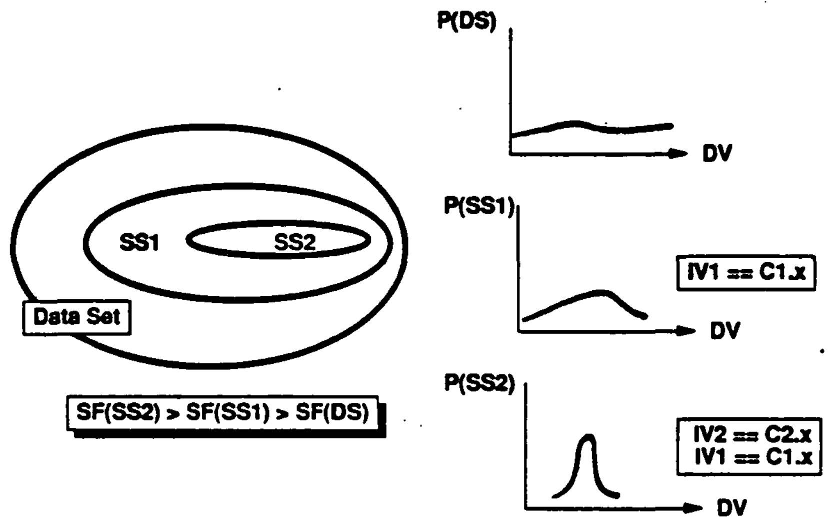
Taking this approach in the ideal, we can assume that given a learning sample, LS, and a measurement vector, MV, we could select an ideal subset of all the pattern vectors in LS having the exact same IV instances as MV (Figure 1). However, since we are usually working with small samples and numerous independent variables the ideal subset is typically too small to be useful, so this ideal approach is not applicable.

Figure 1 - Ideal approach


Measurement Vector = {IV1 := C1.x, IV2 := C2.x, IV3 := C3.x}
With Cn.x being any single class on the IVn range
IV2 == C2.x
IV1 == C1.x
Data set
IDEAL SUBSET
IV3 == C3.x

Nevertheless, we need to find a subset, SS, of LS that contains pattern vectors similar to MV with respect to some IVs and that yields significant patterns. SS must be large enough to be usable (notion defined below) and must contain sufficiently homogeneous pattern vectors to yield significant patterns. To extract SS from LS, we have to select a subset of IVs among those available, that will be used to select the pattern vectors that will form SS. The pattern vectors matching the MV instances with respect to the selected IVs will be extracted from LS. This IV selection will be performed in a stepwise manner and based upon a **selection function SF** (Figure 2). In other words, each pattern resulting from a potential IV selection will be evaluated using SF which provides information on the degree of significance of the pattern.

Figure 2 - Pattern recognition process



Two reasons justify a stepwise process:

- We wish to stop the subsetting whenever the resulting subset SS is too small.
- If we increase the pattern significance progressively, set reduction after set reduction, we increase the probability of not getting a good pattern purely by chance due to the small size of the sample on which the pattern is based. A constant trend is more convincing than a single pattern based on a small sample.

In the Software Engineering Laboratory at the University of Maryland, a set of experiments have led us to develop the following pattern recognition process (called Optimized Set Reduction) applied for any MV:

- Step 1: DV classes are formed either by dividing the DV range into intervals or by grouping the defined DV categories. For optimal results, a similar number of pattern vectors should be in each class. The mechanism for creating classes is further described below.
- Step 2: IV classes are formed in a similar way.
- Step 3: The learning sample is successively decomposed into subsets. At each step, an IV is selected (according to a **selection function** described below) and the objects having the same instance for the IV as the object to be assessed are extracted to

form the reduced subset. This is done recursively on the reduced subsets. We call the performed process an **Optimized Set Reduction.**

- Step 4: When a predefined condition is reached, the reduction stops. This condition will be referred to as the **termination criteria** and will be discussed below. The subsets resulting from this criteria are called the **terminal subsets.**
- Step 5: The pattern vectors in the terminal subset(s) are then used to calculate **probability** that the actual dependent variable value lies in each of the DV classes. Several alternatives may be considered for calculating these probabilities and two of them are described below.

The resulting probabilities (that form the obtained pattern) may be used either for **DV predictions**, **risk management** or **quality evaluation** in a way that will be described in Section 4.3.

Despite an apparent simplicity, this approach opens up a set of research questions associated with each of the steps, that need to be further investigated. The details of each of the steps, as well as the open questions are discussed here:

- Which IV **selection function** should be used as a reduction mechanism (Step 3)?

  The best we have found so far is Entropy (F). The measure of entropy generalized for m classes from information theory can be used as the impurity evaluation function [BRE84, SEL88]:

  $$F = \sum_{i=1}^{m} -P(C_i/x)\log_m P(C_i/x)$$

  where $P(C_i/x)$ is the conditional probability of x of belonging to

  the DV class C_i, $i \in (1, ..., m)$.

  We assume that the lower the entropy the more likely we are to have found a significant pattern. This assumption has been supported by the performed experiments: a good correlation between Entropy and MRE (i.e. Magnitude of Relative Error) has been observed. The selected IV is the one, that minimizes the defined selection function.

- How is the **termination criteria** determined (Step 4)?

  The termination criteria needs to be tuned to the environment, i.e. the available data set. Logically, if measuring the significance of a pattern by calculating its entropy is reasonable, then the entropy should be strongly correlated to the observed prediction accuracy (i.e. Magnitude of Relative Error for continuous DVs and Misclassification Rate for discrete DVs). Therefore, an estimation of the prediction MRE/MR is possible by looking at the decomposition entropy. There are two bounds on the calculation. If there were no termination criteria, the reduction could decompose to a subset of a single pattern vector, yielding the meaningless minimum entropy of zero. On the other hand, if we stop the reduction too soon, we have not sufficiently decomposed the data set to provide an the most accurate characterization of the object to be assessed. Thus we are interested in achieving an accurate approximation of the selection function based upon some minimum number of pattern vectors in the terminal subsets. To find this minimum number, we must experiment with the learning sample by examining the correlation between the MRE/MR and the selection function (e.g., entropy). If this correlation becomes too weak, then the acceptable minimal number of pattern vectors in a subset should be increased. The goal is to find a compromise between a good correlation and a sufficient number of decompositions to provide a reasonable accuracy for predicting the value of the DV. This determines the number of pattern vectors used as our termination criteria (Figure 3).

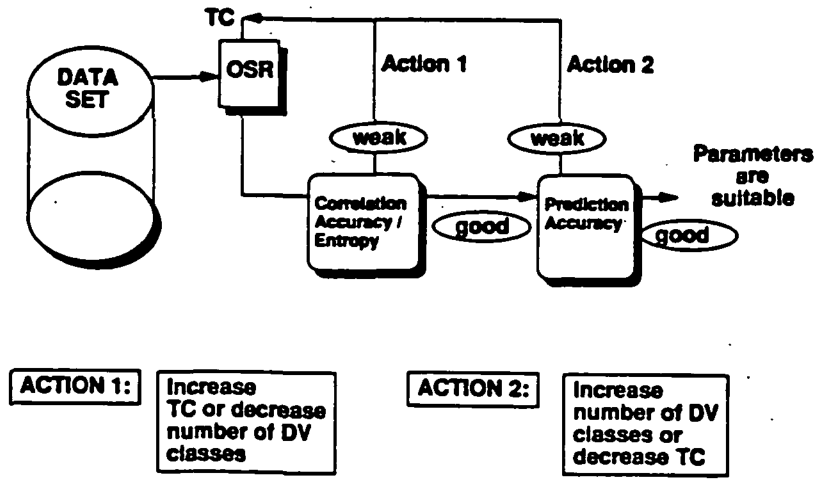
- How to create suitable DV and IV classes (Steps 1 and 2)?

  Whenever the variable is either continuous or ordinal, the range is divided in a fixed number of classes. This number may be fixed through a trial and refinement procedure (Figure 3).

  At least two concurrent factors have to be taken into account:

  - The amount of data available: Increasing the number IV classes decreases the average number of reductions during the OSR process, because the average set reduction rate (ratio # of pattern vectors before red./ # of pattern vectors after red.) decreases. This tends to decrease the DV prediction accuracy. Whenever the number of DV classes increases, the ratio # pattern vectors / # of DV classes decreases and the calculation of the conditional probabilities is less accurate. Thus the strength of the correlation *Accuracy/Entropy* is also very dependent on the number of defined DV classes. So decreasing the number of DV classes is also a way of improving the correlation. One has to determine, based on the learning sample, which of the two solutions: decreasing the number of DV classes or increasing the termination criteria, is affecting the DV prediction accuracy the most.

Figure 3 - Tuning the OSR parameters


TC
DATA SET
OSR
Action 1
Action 2
weak
weak
Parameters are suitable
Correlation Accuracy / Entropy
good
Prediction Accuracy
good
ACTION 1:
Increase TC or decrease number of DV classes
ACTION 2:
Increase number of DV classes or decrease TC

- The granularity of the classes: Decreasing the number of DV or IV classes may make predictions more inaccurate because you may be missing real distinctions in the values of the data. Granularity is clearly dependent upon the ability to collect the data accurately. The required accuracy of the DV prediction is a major criterion for DV granularity.

Whenever the variable is nominal, categories may be grouped to form IV or DV classes. With respect to DVs, the grouping will depend on the classification to be performed and also the size of the data set , as above. For IVs, grouping may be subjective and depends upon the goals of the analysis. For example, assume one wishes to predict productivity and one of the available IVs is "Programming language used". If the possible instances for the variable are "COBOL, FORTRAN, ADA, C++," one could create a class "high level language" containing ADA, C++ (because they allow better rates of code reuse and easier testing procedures) and a second class "Low level languages" containing COBOL, FORTRAN. If the amount of data makes it possible, four classes with the four programming languages may be used.

- How does one estimate the **conditional probabilities** that the object to be assessed falls into the various DV classes (Step 5)?

  A simple rule would be to calculate the probabilities as the ratios of pattern vectors falling into the various categories versus the total number of pattern vectors. This is the only solution for discrete DVs because there is no notion of distance in the range of values. A more refined approach for continuous DVs might be to sum the distances between the subset pattern vectors and the class mean for each of the DV classes. Call this TD_n, where n represents the class index. Note that TD_n is inversely proportional to the concentration of pattern vectors around the class mean for class n. Then calculate:

$$P(C_n/x) \approx \frac{1 - (\frac{TD_n}{\sum_{i=1}^{m} TD_i})}{m-1}$$

  where m is the number of DV classes.

  This formula assumes that the probability is inversely related to the total distance (TD_n) of the pattern vectors to the class mean. This approach refines the probability calculation since it takes into account the distances between the subset of pattern vectors and the class means, no just their membership in a specific class. We can further refine this probability calculation by defining TD as the sum of the exponentials of the distances in order to decrease the weight of the pattern vectors furthest from some particular class mean.

The OSR process could be improved by investigating the following issues:

- Could non-parametric statistical classification techniques (e.g. K-nearest neighbors) [COV67] be used in order to improve the accuracy of the results ?

These techniques could be used, for example, in the terminal subsets to help refine the results in cases where the entropy is high. The K-nearest neighbors technique yielded interesting results in experiments on data sets other than the one used as illustration in this paper. This approach needs to be further investigated in order to come up with effective empirical procedures.

- Could some decomposition heuristics be used to improve the pattern recognition process effectiveness?

  The simplest way of building an optimized set is to select in a stepwise manner the IVs yielding the best decomposition entropy until the Termination Criteria is reached as described above. Although this is certainly the fastest way of generating an optimized set, it is not the most efficient in terms of pattern recognition. One can imagined more sophisticated heuristics for example to get better entropies and therefore more accurate estimations. The user could select, for the S first stages of reduction, the A IVs yielding the best decomposition entropies. Then, in order to make an estimation of the studied dependent variable, the user would have to consider S^A terminal subsets (each of them yielding an independent result). These results can be weighted and averaged. The weights can be calculated using, for each terminal subset, the respective entropy and/or number of pattern vectors. Moreover, the variance of the obtained results is another insight into the reliability of the prediction.

4.2 Optimized Set Reduction and Decision Trees

The reader may notice similarities with decision tree techniques (e.g. the use of entropy as a selection function). However, the approach differs in several critical points, the two most significant discussed below. Moreover, unlike decision trees whose the goal is classification [BRE84], the goal of OSR is the estimation of the conditional probabilities on DV ranges associated with specific pattern vectors. These probabilities are intended to be used for prediction, risk management and quality evaluation as described in Section 4.3.

Decision trees and OSR use different decomposition strategies. For example, Selby and Porter identified two different selection functions (linear and logarithmic) [SEL88]. Since the two functions showed similar behaviors and only the logarithmic function is generalizable to N classes where $N > 2$ (note that decision trees usually deal with binary decisions), we have been using the logarithmic function, i.e., the information theory entropy function. Even with this function, some heuristics still need to be developed in order to improve the IV selection and the pattern recognition process. In the decision tree approach, one set of pattern vectors is decomposed by selecting the IV yielding the best average weighted entropy across the various formed subsets. This decomposition is repeated in each branch of the formed tree until the predefined termination criterion is met. The Optimized Set Reduction approach tends to be more effective in recognizing significant patterns. The following example illustrates a typical case where the difference is noticable:

Let us assume that Complexity ($CPLX$) and Personnel capability ($PERS$) are among several IVs available for the decomposing the available data set, S, to predict the dependent variable, productivity. S can be decomposed into two pairs of subsets $SS1, SS2$ and

$SS3, SS4$ using $CPLX$ and $PERS$, respectively. Assume that $SS1$ and $SS2$ contain respectively low and high complexity projects. In a similar way, assume that $SS3$ and $SS4$ contain projects with low and high experience personnel, respectively.

Let the average entropies for each of the two decompositions be $E1$ and $E2$ where $E1 < E2$. The IV $CPLX$ is thus selected for the first decomposition step of the decision tree. Now suppose the *entropy*($SS1$) > *entropy*($SS3$) because for a low complexity system, $PERS$ is the most influential parameter. However, suppose *entropy*($SS2$) < *entropy*($SS4$) because $CPLX$ is still the most significant IV for complex systems. If we want to assess an object (e.g. project) $o1$ that falls in either $SS1$ or $SS4$ (according to the selected decomposition), the result will be different according to which fo the two decomposition techniques is used. For the decision tree and OSR techniques, the subsets used for estimating productivity will be $SS1$ and $SS3$, respectively. To assess an object $o2$ that falls in either $SS2$ or $SS4$, the result with both techniques will be the same: $SS2$ will be used.

This example shows that the OSR technique is more effective since it always selects the decomposition that yields the best entropy for the object to be assessed and is therefore more likely to achieve better entropy values with fewer levels of decomposition. This property is crucial if you consider that the number of possible decompositions is very limited for small samples.

In conclusion, the decision tree technique may be seen as an optimized partition of the data set. The OSR approach is a set reduction process for recognizing the most significant (with respect to its pattern) subset for the specific object being assessed. This makes the OSR technique more calculation intensive, but all automatable.

Another important difference between the OSR and decision tree technique is the definition of the decomposition termination criterion. In the decision tree approach, the termination criteria is some minimal class conditional probability. If one of the two classes of the DV range is above this probability, it is selected as the one to which the object to be classified belongs. This selection appears to be arbitrary. In the OSR approach, the termination criteria is based upon the need to evaluate the accuracy of the prediction, i.e., it is determined by the correlation between accuracy, e.g., MRE and the selection function, e.g., entropy.

4.3 Prediction, evaluation and risk management

The three processes, prediction, quality evaluation and risk assessment, are all based on the recognized patterns and follow a similar quantitative approach even though they apparently deal with three different purposes.

- Prediction:

  In this case, one is interested in estimating only one Dependent Variable based on the set of available Independent Variables. The dependent variable is a measurable object characteristic that is not known or accurately estimatable at the time it is needed. For example, one may wish to predict the error rate expected for a particular project development process in order to determine whether to apply a particular code inspection intensive development process. So, one tries to estimate the error rate based on other characteristics (IVs) that may be measured, evaluated subjectively with a reasonable accuracy, or estimated through other models.

If the dependent variable is defined on a continuous range (i.e. the notion of distance between two values on the range is meaningful), the following approach may be used: by dividing the DV range into m successive intervals (C_i classes: $i \in (1...m)$) and calculating $P(C_i/x)$ for each of them, we have in fact approximated the actual density function $P(DependentVariable/x)$ by assuming it to be uniform in each class C_i.

Therefore, the following expected μ_i value can be calculated on C_i:

$$\mu_i = E[Productivity/C_i, x] = \frac{lower\_boundary\_C_i + upper\_boundary\_C_i}{2}$$

In other words, the actual density function is approximated by a histogram, where each column represents the conditional probability of a particular pattern vector x that lies in a particular DV class C_i. No assumption has been made with respect to the form of this probability density function.

The expected value on the total DV range can be approximated as follows:

$$E[Prod/x] = \mu \approx \sum_{i=1}^{m} P(C_i/x) \times \mu_i$$

This expected value can be used as an estimate of the Dependent Variable. The average error interval that can be expected may be estimated quite accurately by using the correlation of accuracy to entropy. This correlation will be confirmed by the experiments described in Section 5.

If the Dependent Variable is defined on a discrete range, then prediction becomes a classification problem:

Given a set of probabilities that a particular pattern vector x belongs to each DV class C_i, the decision maker must decide to which class to assign x. The class with the highest probability may not always be chosen. Rather, we may choose the class based upon the loss associated with incorrect classifications. This is the Bayesian approach. A risk (or loss) matrix L has to be defined by the decision maker where Lij represents the loss of having chosen the strategy appropriate for C_j when the DV class (or state of nature) is actually C_i. A Bayesian classifier [TOU74] will try to minimize the conditional average risk or loss $Rj(x)$ (j = 1 ... m) considering the m defined DV classes.

$$Rj(x) = \sum_{i=1}^{m} LijP(C_i/x)$$

$P(C_i/x)$ represents the probability that pattern vector x comes from the pattern class C_i. The bayesian classifier assigns a pattern vector x to the class j with the lowest R value.

- Risk management:

  Software development organizations are interested in assessing the risk associated with management and technical decisions in order to guide and improve the development processes. Referencing [CHA89], the risk associated with an action (e.g. software development) may be described through three dimensions:

  - $D1$: The various possible outcomes
  - $D2$: The potential loss associated with them
  - $D3$: The chance of occurrence for each outcome

  One encounters multiple kinds of interdependent risks during software development (e.g. technical, schedule, cost) [CHA89] and this make risk management and models complex. Also, the notion of risk is by definition subjective because the associated loss strongly depends upon one's point of view. Charette in [CHA89] writes:" One individual may view a situation in one context, and another may view the exact same situation from a completely different one". According to his/her goals and responsibilities, one will define the risk in different ways, in the form of various models.

  If we try to make the link between the above description of risk and OSR, the following straightforward associations may be established:

  - Outcomes (i.e. dimension $D1$) and DV classes.
  - Potential loss (i.e. dimension $D2$) and distance on the DV range between the DV class mean and the planned DV value.
  - Chance of occurrence (i.e. dimension $D3$) and conditional probability for each DV class.

  In order to analyze risk during software development, we propose the following approach based upon OSR:

  First, based on the three previously described risk dimensions, we calculate the expected difference (distance on the range) between planned and actual values for each DV representing a potential risk (e.g. schedule, effort, ...). Let us call these distances "DV expected deviations". From a decision maker's perspective, the potential loss resulting from his/her decisions is intrinsically a function of several DV expected deviations that may be seen as a specific and subjective risk model. Therefore, a "loss function" is used as a risk analysis model and may be defined as a function that combines several DV expected deviations, parameters (e.g. reflecting management constraints) and constants (e.g. weights). The calculation details are illustrated in the example below.

  Decision making will result in various sets of IV instances and therefore will have an impact on the numerical results of an OSR. Through this mechanism, management choices will have an impact directly on the calculated DV plan/actual deviations and indirectly (i.e. through the selected loss function) on the calculated potential loss.

  Consider the following example with the two continuous DVs, productivity and fault rate. A budget and schedule have been imposed on the project manager by upper management. Therefore a specified productivity P_r will be required to reach the management goals. From the point of view of the project manager, the risk of failure may be represented as a simple function calculating the Productivity Expected Deviation (PED):

$$PED = \sum_{i=1}^{m} P(C_i/x) \times (P_r - \mu_i)$$

where μ_i is the mean of C_i

According to the result of this estimation, the project manager will be able assess the difficulty of the job and make a decision with respect to the development process in order to alleviate the staff loading and make a suitable trade-off between quality and productivity. Some analysis can be performed by the manager to see how the risk evolves according to controllable project parameters (i.e. some of the Independent variables). If the project manager wants to make a risk/effort trade-off, in order to improve competitiveness on a commercial proposal for example, he/she can calculate how the risk evolves according to the productivity required. Based on these observations, a suitable risk/effort tradeoff can be selected to maximize chances of success.

One's perspective of risk may be more complex than the previously defined function, PED. For example, assume that a contractor wishes to define risk of financial loss if the system is delivered late and/or there are effort overruns. One can define the Schedule Expected Deviation (SED) as the expected delay, i.e., the difference between the planned and predicted schedule and the Effort Expected Deviation (EED) as the expected effort overrun, i.e., the difference between the planned and predicted effort expenditures. Then

- $SED = \frac{Estimated\_Size}{PED \times Avg\_Team\_Size}$
- $EED = \frac{Estimated\_Size}{PED}$

where $Estimated\_Size$ is either a parameter, like Avg_Team_Size (i.e.

provided as an input by the manager), or another Dependent Variable (i.e. result of an OSR using some Function Point-like metrics, for example, as IVs). So the financial loss function can be defined as a function of both variables SED and EED.

Now suppose that the cost of delay on a particular contract is exponential to the delay itself. This exponential assumption is based upon predictions with respect to the delay of other projects dependent upon the completion of this project and the resulting compensations to be given to the customer. Thus, the SED needs to be weighted by some Cost per Delay Unit that is an exponential function of SED, call this CDU. Also suppose that CEU is the average Cost per Effort Unit, i.e., the average cost per staff hour for the specific project development team.

Then we can define

$$Financial\_loss = SED \times CDU + EED \times CEU$$

- Quality Evaluation:

  In any quality model, you need a baseline in order to be able to make sensible comparisons. For example, let us assume that the quality perspectives of interest (i.e. quality drivers) are ***productivity*** and *fault_rate* since management views quality as reliable and inexpensive software.

  Assume that using some project features as IVs, the OSR approach yields clear patterns (i.e. low entropy) with respect to productivity in the available data set. These patterns represent the expected productivity distributions in the current development environment for the project under study. The relationship between the actual productivity for the project under study and the expected value of the predicted patterns provides the basis for quality evaluation, from the productivity perspective.

  For example, suppose the actual productivity for the project under study falls far below the expected value of the predicted patterns. This implies that the quality of the project with respect to productivity is low. Using the pattern as a basis of comparison, we may ask where the difference comes from.

  Several causes may be investigated: incomplete or inadequate data collection, some possible new features or variables affecting the development process, or, of course, the process quality (e.g. process conformance) is quite low.

  In order to quantify quality, from the perspective of productivity, according to some quality model, a quality value could be defined as a function the distance between the actual productivity and predicted value(s) of productivity based on the recognized pattern(s).

  This distance may be defined as:

$$Prod\_deviation = AP - \sum_{i=1}^{m} P(C_i/x) \times \mu_i$$

  with AP the actual measured productivity.

  If we try to include in the quality model both the *Fault_rate* and *Productivity* quality drivers and we assume an approach similar to the *Prod_deviation* evaluation for calculating a *Fault_deviation*, then a global quality evaluation may be formalized by the following quality model.

  Let us define NFD as ***Fault_deviation*** (i.e. fault rate deviation) normalized by the fault rate standard deviation in the available data set and NPD as the equivalent variable for ***Prod_deviation***. Based upon these unitless deviations, we define the following quality model:

  - If $NFD < 0$, $NPD > 0$, the larger $|NFD \times NPD|$ is, the better the quality.
  - If $NFD > 0$, $NPD < 0$, the larger $|NFD \times NPD|$ is, the worse the quality.
  - If both NFD and NPD are negative, the larger $\frac{NFD}{NPD}$ is, the better the quality.
  - If both NFD and NPD are positive, the smaller $\frac{NFD}{NPD}$ is, the worse the quality.

- If both NFD and NPD have the same sign and $\frac{NFD'}{NPD}$ has a value close to 1, then quality may be assessed as average or nominal.

This particular quality model takes into account two dependent variables and illustrates that a quality model may be a subjective function of several distances on the rerspective dependent variable ranges. This model might be modified, according to the user perspective of quality, to change the weighting of the various DVs or factors, e.g., doubling the effect of fault rate in the evaluation of quality.

4.4 Development Environment Analysis

The user may wish to analyze the collected historical data in order to get some intuition about the development environment (e.g. what significant features affect productivity?). This could help an organization improve its development process(es) and management techniques by tailoring them to their environment.

To obtain experimental results, one may perform an OSR on each of the pattern vectors in the available data set, using all other pattern vectors as the learning sample. Based on the N Optimized Set Reductions (N being the number of pattern vectors in the test sample), the occurrences of each IV in the N terminal subsets are counted and weighted.

Several possibilities are available to weight the counts :

- according to the ranks where they appear in the terminal subsets
- according to the number of data points on which they are selected.
- according to the entropy variations they create

This count should give an insight into the significance ranking (according to the dependent variable) of the available IVs in the development environment where the data has been collected. Obviously, these counts can be performed on subsets of the data set and compared (e.g. real time versus business applications).

If an IV is known from experience as very influential but it does not appear very often in the terminal subsets, several reasons are possible :

- the IV represents a factor that is very influential with respect to the dependent variable but is quite constant in the development environment. This may due to inappropriate interval boundaries on the range of the independent variable that do not sufficiently differentiate the objects of study in the learning sample. This ability to differentiate may change over time with the evolution of technology.
- the IV is not selected because it is highly interdependent with other available IVs selected during the set reductions (this issue is explained more in detail in Section 6).
- an innacurate data collection

5 Experimental Results

In this section we demonstrate the effectiveness of the approach by applying the OSR modeling process to the problem of cost estimation and showing that OSR is able to recognize meaningful patterns on the available data sets. Although we will only be dealing with the prediction capability, this experiment can be viewed as demonstrating the effectiveness of the risk assessment and quality evaluation capabilities since the three processes have been shown in Section 4.3 to all be based on the conditional probabilities estimated on the dependent variable range. Therefore the accuracy of the three formulas are all dependent on the significance of the recognized patterns.

We will also determine the most influential factors affecting productivity, regardless of the development environment. We will do this by assessing the significance of each factor in predicting productivity. We will then analyze the results to determine if they are reasonable.

The largest part of our data set comes from the COCOMO database published in 1981 (63 data points/pattern vectors) [BOE81]. A second source of data is provided by Kemerer (15 data points), collected in a COCOMO format and published in 1987 [KEM87].

The first data set is a mix of business, system, control, high level interface and scientific applications. A significant percentage of these projects has been developed in FORTRAN (38%) and a very small number in Cobol (8%). The second set of projects reflects a variety of data processing applications most of them developed in Cobol (87%). The following sections describe an experimental evaluation of the OSR technique as applied to effort estimation based upon these two data sets. Results obtained with the OSR approach are compared to results from other standard approaches in order to provide some basis for comparison.

In the following sections, we use three different data analysis procedures to make cost predictions for the fifteen Kemerer's projects based on their COCOMO cost drivers:

- the OSR approach.
- a calibrated intermediate COCOMO model.
- a pure regression approach.

In what follows, we will use the term data set to refer to the COCOMO and Kemmerer data sets, the test sample refers to the Kemerer data set which is the sample on which we are going to assess the OSR capabilities, and the learning sample for each optimized set reduction is the data set minus the project we are trying to assess.

Thus, 15 optimized set reductions will be performed, one for each of the test sample pattern vectors. Each time, the pattern vector to be assessed will be removed from the whole data set to form the learning sample (77 projects) in order to get realistic results. This would be similar to a situation where an organization has collected data about 77 projects and wants to assess a new project.

However, the reader must consider that various subsets of the 78 projects have been developed in various environments, at different points in time and collected by different people according to different procedures, in different organizational structures. The difficulties in tailoring the COCOMO cost drivers to various environments causes a loss of consistency in the data collection regardless of the analysis technique. Moreover, it is important to notice that the project productivities lie over a very large range (i.e. from 20 to 2491 LOC/MM). The 78 pattern vector data set is small enough to assess the capability of the approach to deal with small samples. The number of independent variables used (15) compared to the available data set and the nature of these IVs (i.e. ordinal, nominal) make any pure regression analysis based approach difficult to apply.

In order to evaluate the performance of the OSR technique, we will first reference the results obtained by Kemerer in using some of the main models available:

- SLIM: MRE=772%
- Intermediate COCOMO: MRE=583%
- FP: MRE=103%
- ESTIMACS: MRE=85%

According to the author, one of the reasons for which the last two models yielded substantially better results is that they are built on business data processing application projects. Since the data used to develop the FP and ESTIMACS models were either not available or not provided in a COCOMO format, we cannot include them in our data set even though they may be more suitable as a learning sample for the fifteen projects of the test sample.

5.1 Understanding and assessing the cost of development using OSR

As the Dependent Variable, we use project productivity (i.e. size/effort). The size metric used is the "Adjusted Delivered Source Instruction" as defined in COCOMO, and the effort unit is staff-months. The Independent variables are the COCOMO cost drivers. The ranges for all but one (SCED) of the IVs have been decomposed into two intervals (i.e. the boundary being located either just below or above nominal depending on the IV) and the DV range into five intervals containing an equivalent number of pattern vectors, to the extent possible.

Using the termination approach defined in Section 4.1, the termination criterion was set to 8 projects, after being tuned based upon the learning sample. No more sophisticated decomposition heuristic was used.

Table 1 gives the results for each of the fifteen data points of the test sample. The five columns contain the project number, the actual productivity, the predicted productivity, the actual effort, the predicted effort and the entropy yielded by the OSR processes.

Table 1

PROJECT	Actual prod	Predicted prod	Actual effort	Predicted effort	ENTROPY
1	884	299	287	846	0.63
2	491	935	82	44	0.24
3	580	674	1107	668	0.45
4	2467	643	87	333	0.06
5	1338	952	336	473	0.27
6	595	1196	84	42	0.47
7	1853	1016	23	42	0.47
8	1535	1006	130	199	0.52
9	2491	431	116	670	0.56
10	542	1028	72	38	0.06
11	983	1028	258	247	0.06
12	557	1025	231	125	0 06
13	1028	1035	157	155	0.06
14	667	1070	247	154	0.27
15	861	964	70	62	0.06

Despite an encouraging average prediction accuracy which is discussed below, the three data points with highest productivity (project 4, 7 and 9 in Table 1) yield large effort overestimation. These three projects have a productivity far above the other 75 projects of the learning sample and these productivity values might be the result of several problems:

- the size evaluation (i.e. adjusted KDSI) has not been performed according to the rules described in [BOE81]. This lead to a size overestimation.
- something occurred that was not captured by the 15 COCOMO cost drivers.

More information on these particular projects would be necessary in order to make a more thorough analysis. However, in order to keep these three projects from introducing noise in our analysis, we will also analyze the results obtained by removing them from the learning sample.

Tables 2.1 and 2.2 summarize the previous results by giving, for three entropy intervals, the average Magnitude of Relative Error of the effort estimation in the respective intervals for each modeling technique, (columns $MRE-OSR$, $-CC$ for calibrated COCOMO and $-SR$ for stepwise regression), and the percent of the test sample falling in that interval, (column $\%TS$). Table 2.1 takes into account the projects 4, 7, 9 and Table 2.2 ignores them. This provides some insight into the correlation between the accuracy of the effort estimation and the entropy.

Table 2.1

SUBSET	MRE-OSR	MRE-CC	MRE-SR	% TS
F <= 0.06	65%	34%	78%	40%
0.06 < F<= 0.47	50%	168%	80%	40%
0.47< F<= 0.63	246%	622%	261%	20%

Table 2.2

SUBSET	MRE-OSR	MRE-CC	MRE-SR	% TS
F <= 0.06	21%	33%	47%	42%
0.06 < F<= 0.47	43%	81%	60%	42%
0.47< F<= 0.63	124%	76%	105%	16%

The reader should notice from Table 2.2 that as the entropy gets lower, the accuracy (i.e. MRE) gets higher. Therefore, whenever one makes an estimate, the entropy of the pattern on which the estimate is based is calculable and provides an assessment of the accuracy of the estimate. For example, if the obtained entropy is around 0.06, you know that the expected accuracy should be around 21%, according to the results obtained in Table 2.2. In Table 2.1, the results seem disturbed by the three high productivity projects. They create large standard deviations with respect to the accuracies in each entropy interval. Table 2.3 gives the standard deviations calculated in each interval. The first column takes into account the three high productivities projects. The second one ignores them.

Table 2.3

SUBSET	SD-15	SD-12
F <= 0.06	102	20
0.06 < F<= 0.47	9	4
0.47< F<= 0.63	323	58

According to the second column of Table 2.3, the standard deviations of the average accuracy in the two lower entropy intervals are relatively small showing that the prediction accuracy may be estimated with a reasonable precision. If the entropy is below 0.06, the accuracy has a high probability of being under 41% (average MRE plus the standard deviation in this specific entropy category). For example, if the distribution of the residuals around the average MRE is assumed Normal, then this probability value lies around 85%. Since the highest entropy interval contains only two projects, it is difficult to draw any conclusion from its high standard deviation. However, a high standard deviation of the MRE values should be expected in high entropy categories since patterns are not very significant and therefore yield inaccurate predictions.

A problem that may affect the analysis is that the estimation accuracy may be disturbed significantly when the actual productivities of projects are close to the extreme boundaries of the productivity ranges. This is because the density of projects in these parts of the range may be much lower than projects found in the middle of interval ranges. Only intuition based upon understanding the particular situation can help the manager detect an unusual, extremely low/high productivity so the effort estimate may be increased/decreased. Obviously, something that has never or rarely occurred is difficult to predict.

In summary, based on Table 2.2, the results are very encouraging, especially if we consider that we are using only fifteen new projects and with different profiles. That is, despite the constraints, impressive results ($averageMRE = 32\%$) was achieved in 84% of the cases (projects 4, 7, 9 are ignored). We have clearly satisfied requirement R_2 from Section 2.2. Another important point is that no assumptions have been made of the kind discussed in requirement R_1.

Thus, someone with only 15 projects specific to his/her environment may use the COCOMO data set to form a learning sample on which to construct an OSR model and get some useful predictions.

In order to determine the most influential factors affecting productivity, counts of occurrences of the various independent variables have been performed according to the description given in Section 4.4. Table 3 has three different columns. The first gives the raw counts of occurrences, i.e., the number of occurrences of each IV used to create some decomposition for the given test sample. The second gives the same counts of occurrences weighted according to the level of decomposition where the independent variables appears in the OSR processes (e.g. if an IV appears at the level two of a decomposition then its occurrence count increases by $1/level = 0.5$). The third column shows the total variation of the productivity mean due to each cost driver in the fifteen OSRs performed. This table gives several insights into the influence of each of the fifteen cost drivers on the productivity of the fifteen business applications studied. The three columns give three different perspectives of the impact of each cost driver on the predictive ability of the model.

Table 3

C. DRIV	OCC	W. OCC	MEAN VAR.
RELY	2	1.33	720
DATA	3	2.33	594
CPLX	1	0.25	270
TIME	1	0.33	280
STOR	11	10.5	2770
VIRT	4	2.66	65
TURN	4	1.83	85
ACAP	7	3.5	706
AEXP	1	0.33	350
PCAP	7	2.66	2075
LEXP	0	0	0
VEXP	1	0.5	268
MODP	1	0.5	270
TOOL	0	0	0
SCED	0	0	0

When a factor yields a low number of occurrences, several causes may be considered:

- The factor is quite constant in the learning sample and therefore will not help in differentiating projects. It should be noted that when a factor is constant in an environment, there is a loss of information about the effects of that variable.
- The factor has not much influence on the Dependent variable studied.
- There is an interdependence between the IV and some more influential IV.

Table 4 provides an assessment of the influence of the cost drivers on productivity relative to only one of the three columns from Table 3, the unwieghted occurrences. Table 4 has 2 columns: the first one gives non-weighted counts of occurrences and the second one shows the cost drivers matching them. Two distinct cost driver subsets may be observed that yield very different counts of occurrences (≤ 4, ≥ 7). Considering the test sample size, we may only say that there is clearly a set of three very influential cost drivers for which the data collection must be as accurate and consistent as possible, (ACAP, PCAP, STOR).

Table 4

# OCC	Cost Drivers
0	Tool, Sced, Lexp
1	Cplx, Time, Aexp, Vexp, Modp
2	Rely
3	Data
4	Virt, Turn
7	Acap, Pcap
11	Stor

Storage constraints appear to be very significant. This makes sense for systems focusing on data processing and dealing with very large amounts of data. The data set shows that most of these fifteen projects fall in the categories "high" and "very high" with respect to the cost driver DATA which makes the previous statement reasonable. The cost driver

STOR appeared at the top level of decomposition in ten of the fifteen OSRs performed. The average entropy for the ten projects is much lower (i.e. 0.17) than the average entropy for the five projects where STOR was not used (i.e. 0.52). This furthers the argument that STOR is very significant for the prediction ability of the OSR model for this particular test sample. STOR was not used in any case where storage constraints were rated above nominal. As a consequence, high storage constraints make productivity difficult to predict for this particular data set.

Of course, as shown in Software Engineering Economics [BOE81], the staff capabilities (Analyst, Programmers) play a crucial role in achieving optimal productivity. When STOR, ACAP and PCAP were all included in the decompositions, the entropies were the best obtained (i.e. 0.06). Therefore, ACAP and PCAP seem to significantly improve the average entropy of the recognized patterns (i.e. 0.06 instead of 0.17). Moreover, all the projects where ACAP and PCAP were used had high capability teams. This shows that productivity predictability is more accurrate at higher ranges of capabilities.

Table 4 shows that the reliability and complexity factors have a weaker influence on productivity than expected (according to the results published in [BOE81]). However, the fact that most of the fifteen projects of the test sample fall in the "nominal" category and there are no extreme complexities or reliabilities present may explain the lack of significance of these parameters for this particular test sample.

Therefore, based on the above examples, it should be noted that the significance of the IVs (i.e. cost drivers) in predicting productivity seems to be very dependent on the localization of the test sample pattern vectors on the IV ranges themselves. STOR does not seem to help predict productivity when the storage constraints are high in the range (i.e. above nominal). In this case, the variability (and the entropy) in the produced patterns is larger and therefore STOR is not selected in the decompositions. The same phenomenon may be observed for low ranges of PCAP and ACAP. With respect to CPLX and RELY, results are more difficult to interpret because of the lack of variation in the test sample. However, their low significance in the test sample might be explained as a consequence of the nominal reliability and complexity of the projects included in the test sample. Considering that most of these projects are business data processing applications, this result seems to make sense.

Neither "Virtual machine volatility" (VIRT) nor "Virtual machine experience" (VEXP) show much influence here on productivity. There is nothing in the data that explains this result and therefore we may conclude that these factors are not significant. The factors MODP (i.e. modern programming practices) and TOOL (i.e. use of software tools) show almost no significance. This result contradicts the results shown in COCOMO. No interdependence with other factors was detected in the data set. This latter result may be due to a weak variability in the level of technology involved in the system developments or inconsistent data collection between the two data sets used. For example, the definitions of the MODP categories are quite fuzzy and leave room for interpretation and inconsistencies.

AEXP also has minimal impact. This may be understandable in this context where the application domain of these 15 projects does not involve high-technology or mathematically intensive problems. LEXP shows no influence on productivity. Considering the low complexity of the programming language COBOL, the learning process for this language may be estimated as very fast for any experienced programmer. SCED (i.e. schedule constraints) shows no impact on productivity and therefore confirms the conclusions of the COCOMO model.

According to the counts of occurrences in Table 4, the three most influential cost drivers (i.e. ACAP, PCAP, STOR) belong to the category of the seven most influential cost drivers in the COCOMO's cost driver ranking, despite the different nature of the considered fifteen projects.

5.2 Effort Estimation Using a Regression Tailored COCOMO

To allow for an evaluation of value of the OSR technique for the prediction of productivity and effort, a comparison with more conventional techniques is provided. In Section 5.1, we referenced an evaluation by Kemerer [KEM87], where the average MRE was 583%. However, the model overpredicted on every project, so it seems that COCOMO may be calibrated too high for this environment. It seems unfair to directly compare uncalibrated results to the results obtained through the OSR technique. In this section we describe an experiment in tailoring the intermediate COCOMO model to the data supplied by Kemmerer using regression based techniques. The calibration technique chosen was the one that most closely resembled the previously described OSR experiment. A new model was developed from project data from both COCOMO and Kemerer, using the intermediate COCOMO project information. The base data set for the experiment was the data points of the COCOMO model, along with all but one of the data points from Kemerer's data set. The mode of the removed project was determined, and a model was developed from all of the projects of that mode from the base data set, with the local data points being weighted three times the COCOMO data points. A regression was performed on the equation $ln(E/\pi) = A + b ln(S)$ (where π is the product of the effort multipliers) to determine the best values for the constant term a (where $a = e^A$) and the scale factor b. The values of the effort multipliers were not adjusted in any way. The model for the prediction of effort then becomes $E = \pi a S^b$.

This model was then evaluated by comparing its prediction to the actual effort for the remaining project. This experiment was done for each of the 15 projects in the Kemerer data set. Table 5 provides a summary of the experiment:

Table 5

Project	Predicted effort	MRE
1	230.9	.196
2	54.5	.339
3	1641.6	.483
4	120.2	.383
5	849.5	1.526
6	187.5	1.232
7	163.7	6.055
8	303.7	1.33
9	2105.8	17.153
10	68.2	.053
11	294.7	.139
12	134.9	.415
13	264.1	.682
14	383.0	.551
15	41.7	.403

The results are not so good. The average MRE overall is 206%; however, if project 9 (an apparent outlier) is removed from the data set, MRE is reduced to 99%. The results for projects of the semi-detached mode were better than either of the other modes. This was expected, as there are many more projects in this mode than in either of the other modes. For the semi-detached mode, the average MRE is 44%, with 3 of the 9 projects having an MRE of < 20%. For the other modes, there may not be enough data points to accurately adjust the model to local environment. For the 2 projects categorized as organic mode, the average MRE was 104%, and for the 4 embedded mode projects, the average MRE was 623%. Removing project 9 from the data set lowers the embedded mode average MRE to 259%. The model still tends to overpredict, with overestimations on 2/3 of the projects, which furthers the notion that the COCOMO cost multipliers are not representative of the local projects. Further experimentation indicated that the best prediction could be obtained by ignoring the mode and all the effort multipliers, and utilizing only the local effort and size data in the development of the model. The fact that the mode was not useful when tailoring the model is either due to the small number of data points in the organic and embedded modes, or an indication that mode is not significant in this environment. This indicates that the COCOMO effort multipliers are either not significant in this environment, or that they do not have the appropriate values for this environment. Unfortunately, since the data set is small, and the original values for the COCOMO effort multipliers were developed somewhat heuristically, we can not accurately adjust the values to be useful for prediction in this environment.

5.3 Effort Estimation Using a Pure Regression Approach

We performed a standard regression process having the following characteristics:

The dependent variable of the regression was productivity, as in the OSR experiment previously described. A linear functionnal form was used for the regression equation. The fifteen COCOMO cost drivers were the potential parameters included in the equation. Then a stepwise selection of the regression parameters was run based on the F partial values of the various regression parameters [DIL84].

The results for all the fifteen projects are the following:

- $R^2 = 0.36$
- $MRE = 115\%$

If projects $4, 7, 9$ are removed from the test sample (see previous sections), then the results are improved:

- $R^2 = 0.54$
- $MRE = 62\%$

Detailed results are provided in Table 6.

Table 6

Project	Predicted Productivity	Predicted Effort	MRE
1	553.5	458.2	.596
2	628.9	64.4	.219
3	849.6	529.7	.522
4	740.7	289.5	2.331
5	578.1	778.2	1.314
6	1049.1	47.7	.433
7	659.9	65.2	1.809
8	620.2	322.5	1.475
9	370.2	780.7	5.73
10	489 .0	79.8	.108
11	384.0	662.0	1.559
12	1304.9	98.6	.573
13	1159.9	139.1	.114
14	1292.2	127.5	.483
15	939.6	64.1	.083

5.4 A Comparison of the Three Techniques

Comparing the results of the OSR and regression based techniques leads to several observations. First, for this data set, the OSR technique provides a significantly better prediction than either a tailored COCOMO or a stepwise regression. For 10 of the 15 projects, the prediction of the OSR model was more accurate than that of both regression models. If outliers are not removed, the two regression based models had an average MRE of 206% and 115% respectively, while the OSR model had an average MRE of 94%. If the projects 4, 7 and 9 that showed extremely high productivities are not considered, the MRE for the regression models becomes61% and 62% respectively, while the OSR model is 47%. Moreover, the OSR results were obtained without using the notion of "development mode", making the estimation process easier. The results for OSR are much better that the regression techniques in the two lower entropy categories (32% vs. 57% and 54% respectively for the calibrated COCOMO and the stepwise regression models). This result should have been expected since poor entropy implies that no significant pattern has been found and so the poor entropy projects bias the OSR results negatively. Consequently, in the highest entropy category, regression based techniques can perform better.

A second benefit of the OSR technique its ability to provide an indication of the expected accuracy of the prediction, as demonstrated by the clear correlation of MRE to entropy. Projects with characteristics that have previously shown widely varying productivities (i.e. no clear patterns) are flagged with high entropies, allowing the manager to recognize that the prediction may be suspect. The regression based models provide no such indication of the accuracy for an individual prediction.

For exmaple, the tailored COCOMO model provided relatively accurate predictions for projects of the semi-detached mode (an average MRE of 32%), except for a poor prediction for project 8, with an MRE of 133%. The prediction for project 8 using the OSR model was more accurate (MRE of 53%); however the prediction was flagged with a high entropy, indicating an unstable prediction. The regression based models provide no indication of the potential of the innacurate prediction, while the OSR technique indicates that no significant pattern has been recognized in the available data set. The worse prediction obtained by applying the OSR technique was for project 1 (if you eliminate projects $4, 7, 9$ whose the bad results are likely to be due to extreme productivities and missing IVs). In agreement with the expectation, this poor prediction is visibly flagged with the worse entropy (i.e. 0.63) among the fifteen projects of the test sample.

Despite this important advantages of the OSR technique with respect to prediction, the regression based approaches may be very useful because their prediction abilities appear better when OSR yields bad entropies (see Table 2.2), i.e., when no significant pattern has been recognized. Therefore, an overall prediction process could be defined as a combination of all the modeling techniques previously cited.

6 Data Analysis Disturbances

The decision maker will have to take into consideration the following issues in order to perform a sensible analysis. These problems will have to be better understood and investigated. Some procedures will have to be defined, from the perspective of the OSR approach, in order to alleviate their effects on the data analysis. This section of the paper does not give definitive solutions but rather poses the issues in clear terms and shows how an OSR approach might deal with the described problems.

6.1 Interdependence among explanatory variables

There are two different kinds of interdependence between two variables IV_1 and IV_2, i.e. positive and negative correlation. The first case is where the two variables have a similar influence on a Dependent Variable defined on a continuous range (e.g. DV decreases when either IV_1 or IV_2 increases). For example, let's suppose the dependent variable is productivity and for the given applications, code complexity is strongly correlated with the size of the database (i.e. number of entities and relationships for a relational model) . Both have a tendency to decrease productivity. The variables do not lose their explanatory power and will be effective for prediction using an "optimized set reduction" approach. However, the count of occurrences will be disturbed and will not represent accurately the independent influences of either "code complexity" or "database size" on productivity. There is a causality relationship between the two independent variables. A large "database size" will systematically create a high "code complexity". Thus, the numbers of occurrences for both the independent variables cannot be considered distinctively. However, the two variables may be combined in a higher level IV called for example "System Complexity".

In the second case, the two variables have an opposite influence on the dependent variable. For example, the DV being productivity and due to an effective project management organization, experienced programmers are assigned on complex programs. This situation is much more harmful in the sense that the two IVs lose their explanatory power (i.e. the productivity will not seem to decrease with complexity). They are not likely to be selected in the decompositions of the OSRs despite their strong influence on productivity. The accuracy of the results should not be significantly affected. However, the actual impact of these factors on productivity becomes difficult to assess. With respect to the previous example, using the ratio Complexity/Experience as an IV could be a solution to investigate in order to minimize the impact of interdependence.

If a missing IV is interdependent with available IV in the data set, then the previously described problems occur and become even more difficult to detect.

From an OSR perspective, an easy way of measuring the interdependence between two positively correlated IVs IV_p and IV_q is to calculate the variation of occurrences O_p if IV_q is withdrawn from the used IV set.

Let us define:

- IVS_1 (IV set 1) $= IVS_2 - IV_q$
- O_{np} being the counted occurrences with IVS_n for $IV_p, n \in (1,2)$

The Level of Interdependence (LI) between A_p and A_q can be define as follows :

$$LI = \frac{|O_{1p} - O_{2p}|}{O_{1p}}$$

where the higher the value of LI, the greater the effect of interdependence on the analysis of the influence of the factors.

The effects of interdependence are not avoidable independent of the modeling technique. The only solution consists in taking a larger sample, preferably in a way that decreases interdependence between IVs. However, being able to detect such phenomena in the data set in a simple and effective way, is required in order to partially fulfill the requirement R_3 (Section 2.2).

This issue is still to be further investigated in order to come up with a well defined procedure based on OSR, correlation matrices, etc. and refined based upon lessons learned from experiments.

6.2 Outliers and Missing Independent Variables

The detection of outliers based on multiple regression is quite subjective and difficult to perform. Outliers are defined as: "Observations that have a disproportionate influence on the calculated model" [DIL84]. Some techniques are available to evaluate the influence of a data point (or a group of data points) on the regression coefficients, coefficient of determination, or residuals. However, no objective rule or heuristic exists to determine the level when a (group of) data point has a "disproportionate influence". Also one outlier may mask the effect of another and the detection process may become even less effective. Then, you have to examine the influential effects of subsets of data points. It may be difficult to apply (in N-dimension space with N greater than 2) because there is no simple way (e.g., visually) of selecting the right group of data points in order to avoid the "masking effect".

Outlier detection in the context of an "Optimized Set Reduction" becomes much easier and more objective and therefore partially fulfills R_3. By performing OSRs, you obtain patterns on the DV range. If some data points are far from the main distribution forming the pattern, then they may be considered as outliers matching the following definition : "Outliers correspond to objects that do not lie in the expected DV intervals because the data collection has not captured the phenomenon responsible for the different behavior of these objects with respect to the DV". According to this definition, outliers are strongly related to the "missing independent variable" issue. Therefore, analyzing the outliers may lead to redefining the data collected.

In Section 5, three among the 78 projects present in the data set had extremely high productivities (i.e. projects 4, 7, 9 in the Kemerer data set). Accordingly, these three extremely high productivities were not explained by the fifteen defined cost drivers and are typical examples of outliers.

7 Learning using an OSR approach

Understanding, evaluating, predicting, and controlling software development requires the ability to build quantitative models of various software processes, products, and other forms of experience, e.g. resource estimation and allocation, defect prediction, reusability. Model development is further complicated by the fact that organizations change and our knowledge evolves over time. Therefore it is necessary that the models evolve over time in an effective way, e.g., valid, cost-effective.

Based upon our experience in trying to evaluate and improve software quality in several organizations [BAS85, BW81, RB87, SB88] a quality-oriented, evolutionary life cycle model has been developed, called the **Improvement Paradigm** (IP) [BAS89, BR88]. The IP is measurement based and requires the development of models that can learn and evolve with an organization.

The IP is defined as a set of six activities:

- $S1$: Characterize the current project and its environment.
- $S2$: Set up goals and refine them into quantifiable questions and metrics for successful project performance and improvement over previous project performances.
- $S3$: Choose the appropriate software project execution model for this project and supporting methods and tools.

- $S4$: Execute the chosen processes and construct the products, collect the prescribed data, validate it, and analyze the data to provide feedback in real-time for corrective action on the current project.
- $S5$: Analyze the data to evaluate the current practices, determine problems, record the findings and make recommendations for improvement for future projects. This is an off-line process which involves the structuring of experience so that it can be reused in the future.
- $S6$: Package the experience in the form of updated and refined models and other forms of structured knowledge gained from this and prior projects and save it in an experience base so it is available for future projects.

In the context of the Improvement Paradigm, two distinct kinds of improvement goals can be seen. First, the manager wants to improve the models that are being used to describe the processes of the environment. Additionally, it is desireable to improve the processes themselves to better meet organizational goals. Unfortunately, these improvement goals conflict. As the processes change, the models that describe these processes may lose there validity. In an environment featuring a great deal of change and experimentation with new methods, techniques and tools, it is essential that the models be refined and assessed continuously.

The OSR approach can be used to address both improvement goals. Analysis of the patterns found in projects with higher or lower productivities can help to focus areas of improvements in the development processes. For example, it may be noticed that projects with tight timing constraints and teams with little prior experience with the target machine typically have low productivities. The manager may choose to ensure that on projects with such timing constraints, the team must have significant target machine experience. Similarly, if a particular project profile yields a high entropy value (indicating widely varying productivities), the manager may make changes to the development process to obtain a more predictable project.

As the processes evolve, the OSR automatically incorporates the new experience into the models. Future predictions will be based on both the new and the old data in a manner transparent to the user. However, some independent variables may become less significant as the technology evolves (e.g. complexity, storage constraints). This may be easily observed by taking several test samples covering the time range and comparing the counts of occurences. If a trend is observed, then one of the two following conclusions may be drawn:

- Ideally, the more one collects data, the better are the chances of improving the process. However, if a metric is expensive to collect and it appears insignificant in the OSR experiments, the team in charge of the data collection might stop collecting the corresponding factors in order to lower the measurement cost.
- The intervals on an IV range, as currently defined, have become unsuitable over time. For instance, let us say the new systems have become more reliable. As a consequence, most of them will fall in the category "very high" and the factor RELY will lose its

explanatory power over time. The way data are collected and/or the way the IV range is divided must be changed in order to improve the prediction process.

8 Conclusions

The Optimized Set Reduction (OSR)has been developed in order to address the specific data analysis issues within the software development process, as defined by the Improvement Paradigm. The procedure has the following positive characteristics that allow prediction, risk assessment and quality evaluation:

- It makes no assumptions with respect to probability density functions on the dependent and independent variable ranges. It does not attempt to fit data to predefined distributions, rather it uses the data to approximate the actual distribution (i.e. patterns). Also, no particular mathematical relationship between the DV and IVs needs to be assumed. Thus OSR seems to fulfill R_1.
- It allows an estimation of accuracy for each analysis so we can answer the question : is the estimation usable? This fulfills R_2.
- It is robust to non-relevant and missing metrics and allows a more objective way of dealing with outliers. First, the OSR process is intended to select the combination of IVs yielding the best patterns and therefore selects automatically the most significant group of IVs. Significance has been evaluated by calculating the entropy of the distributions on the DV range. Non-relevant IVs will not be chosen. Second, OSR detects outlier more easily because it analyzes distributions on the DV range as opposed to influencing analysis of pattern vectors in the multi-dimension sample space. Third, OSR provides an easier way of detecting missing significant IVs by two different means, the recognition of bad entropies and the detection of outliers. The issue of interdependence between IVs requires more investigation. R_3 is therefore partially fulfilled.
- It handles discrete and continuous IVs in a natural way and therefore meets R_4.
- It provides an automated refinement of the model as new data is incorporated into the data set.
- The process for selecting IVs, among those available in the data set, can be automated. Thus, the prediction process may be effectively automated and supported by a tool.
- It provides more objective baselines for comparisons and therefore allows more sensible evaluations (e.g. product and process quality). Then learning procedures, based on more objective quantitative evaluation, may be more accurate and effective.
- The use of the approach supports common sense and intuition as opposed to complex mathematical assumptions. Therefore, project managers are more able to plan, predict, control and make corrective actions supported by intuition.

Finally, the results of the preliminary experiments have been encouraging. They strengthen the idea that predictions may be accurate enough to be usable, despite the inherent constraints of software measurement in a production environment (see 2.1).

A prototype tool supporting the OSR approach is being developed at the University of Maryland as a part of the TAME project.

Future research directions for this work include:

- refine the OSR process by addressing the issues presented in Section 4.1.2.
- analyze other data sets that highlight the issues related to discrete DVs.
- address the unsolved issues related to data interdependence.

9 Acknowledgements

We thank G. Caldiera, L. Kanal, C. Kemerer, B. Pugh and M. Zelkowitz for their excellent suggestions for improving both the structure and the content of this paper.

References

[SB88] R.W. Selby and V. Basili, "Analyzing Error-prone System Coupling and Cohesion", *TR-88-46, Institute for advanced Computer Studies*, University of Maryland, College Park, MD, June, 1988.

[RB87] H.D. Rombach and V. Basili "A Quantitative assessement of Software Maintenance: An Industrial Case Study', *Conference on Software Maintenance*, Austin, TX, September, 1987.

[BW81] V. Basili and D.M. Weiss "Evaluation of a Software Requirement Document by Analysis of Change Data", *Proceedings of the Fifth International Conference on S.E.*, San Diego, CA, March 1981, pp. 314-323.

[BAS85] V. Basili, "Can we Measure Software Technology: Lessons Learned from 8 Years of Trying", *Proceedings of the Tenth Annual Software Engineering Workshop, NASA Goddard Space Flight Center*, Greenbelt, MD, December, 1985.

[BAS89] V. Basili, "Software Development: A Paradigm for the Future (Keynote Address)", *Proceedings COMPSAC '89*, Orlando, FL, September, 1989.

[BR88] V. Basili and H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments" *IEEE Trans. Software Engineering* 14 (6), June, 1988

[BOE81] B. Boehm, *"Software Engineering Economics"*, Prentice Hall editions, 1981.

[CHA89] R. Charette, *"Software Engineering Risk Analysis and Management"*, Mc Graw-Hill, 1989.

[KEM87] C. Kemerer, "An Empirical Validation of Software Cost Estimation Models", *Communications of the ACM*, 30 (5), May, 1987.

[DIL84] W. R. Dillon, *"Multivariate Analysis: Methods and Applications"*, Wiley and Sons, 1984

[SEL88] Selby and Porter "Learning from examples: Generation and Evaluation of Decision trees for Software Resource Analysis", *IEEE Trans. Software Eng.*, 1988

[BRE84] Breiman and al. "Classification And Regression Trees", *Wadsworth, Brooks/cole advanced books and softwares*, 1984.

[TOU74] Tou and Gonzalez "Pattern Recognition Principles", *Addison-Wesley Publishing Company*, 1974.

[COV67] Cover and Hart "Nearest Neighbor Pattern Classification", *IEEE Trans. Information Theory, IT-13: 21-27*, 1967

SECTION 3—SOFTWARE MEASUREMENT STUDIES

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

- "Paradigms for Experimentation and Empirical Studies in Software Engineering," V. R. Basili and R. W. Selby, *Reliability Engineering and System Safety*, January 1991

Reliability Engineering and System Safety **32** (1991) 171–191

Paradigms for Experimentation and Empirical Studies in Software Engineering

Victor R. Basili

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

&

Richard W. Selby

Department of Information and Computer Science, University of California, Irvine, California 92717, USA

ABSTRACT

The software engineering field requires major advances in order to attain the high standards of quality and productivity that are needed by the complex systems of the future. The immaturity of the field is reflected by the fact that most of its technologies have not yet been analyzed to determine their effects on quality and productivity. Moreover, when these analyses have occurred the resulting guidance is not quantitative but only ethereal. One fundamental area of software engineering that is just beginning to blossom is the use of measurement techniques and empirical methods. These techniques need to be adopted by software researchers and practitioners in order to help the field respond to the demands being placed upon it. This paper outlines four paradigms for experimentation and empirical study in software engineering and describes their interrelationships: (1) Improvement paradigm (2) Goal-question-metric paradigm, (3) Experimentation framework paradigm, and (4) Classification paradigm. These paradigms are intended to catalyze the use of measurement techniques and empirical methods in software engineering.

1 INTRODUCTION

We have been struggling with the problems of software development for many years.[1,2] Organizations have been clamoring for mechanisms to

Reliability Engineering and System Safety 0951-8320/91/$03·50 © 1991 Elsevier Science Publishers Ltd, England. Printed in Great Britain

improve the quality and productivity of software. We have evolved from focusing on the project, e.g. schedule and resource allocation concerns, to focusing on the product, e.g. reliability and maintenance concerns, to focusing on the process, e.g. improved methods and process models.[3-6] We have begun to understand that software development is not an easy task. There is no simple set of rules and methods that work under all circumstances. We need to better understand the application, the environment in which we are developing products, the processes we are using and the product characteristics required.

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

We have not fully accepted the need to understand the differences and learn from our experiences. We have been slow in building models of products and processes and people for software engineering even though we have such models for other engineering disciplines. Measurement and evaluation have only recently become mechanisms for defining, learning and improving the software process and product.[7,8]

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

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

2 THE IMPROVEMENT PARADIGM

Based upon our experiences in trying to evaluate and improve the quality in several organizations,[9-13] we have concluded that a measurement and analysis program that extends through the entire life cycle is a necessity. This program requires a long-term, quality-oriented, organizational *meta*-life-cycle model, which we call the Improvement Paradigm.[14,15] The paradigm has evolved over time, based upon experiences in applying it to improve various software related issues, e.g. quality and methodology. In its current form it has five essential aspects:

1. Characterizing the environment. This involves understanding the project and its context qualitatively and quantitatively so that the correct decisions can be made.

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

2. Planning. This involves articulating the specific qualities we expect from the process and product and their interrelationships. There are two integrated activities to planning that are iteratively applied:

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

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

3. Execution. This involves the construction of the products according to the process model chosen in step 2 and the collection and validation of the prescribed data. It is essentially the running of the experiment.

4. Analysis. This involves an analysis of the project relative to its goals to check for successes and failures.

We must conduct data analysis during and after the project. The information should be disseminated to the responsible organizations. The operational definitions of process and product goals provide traceability back and forth to metrics. This permits the measurement to be intepreted in context ensuring a focused, simpler analysis. The goal-driven operational measures provide a framework for the kind of analysis needed. During project development, analysis can provide feedback to the current project in real-time for corrective action.

5. Learning and feedback. This involves the synthesis of information gained from executing the project into models and other forms of structured knowledge so that we can better understand the nature of software development and can package that understanding for future projects.

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

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

As it stands, the Improvement Paradigm is a generic process whose steps need to be instantiated by various support mechanisms. It requires a mechanism for defining operational goals and transforming them into metrics (step 2a). It requires a mechanism for evaluating the measurement in the context of the goals (step 4). It requires a mechanism for feedback and learning (step 5). It requires a mechanism for storing experience so that it can be reused on other projects (steps 1,2b). It requires automated support for all of these mechanisms. In the following sections, we discuss mechanisms that can be used to support these activities.

3 THE GOAL/QUESTION/METRIC PARADIGM

The Goal/Question/Metric (GQM) Paradigm is a mechanism for defining and evaluating a set of operational goals, using measurement on a specific project (see Fig. 1). It represents a systematic approach for setting the project goals tailored to the specific needs of an organization, defining them in an operational, tractable way by refining them into a set of quantifiable questions that in turn implies a specific set of metrics and data for collection. It involves the planning of the experimental framework. It includes the development of data collection mechanisms, e.g. forms, automated tools, the collection and validation of data, and the analysis and interpretation of the collected data and computed metrics in the appropriate context of the questions and the original goals. In controlled experiments, the questions can be viewed as hypotheses. As such they can be formulated to the degree of formalization necessary for the experimental environment.

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.[15] These templates and guidelines reflect our

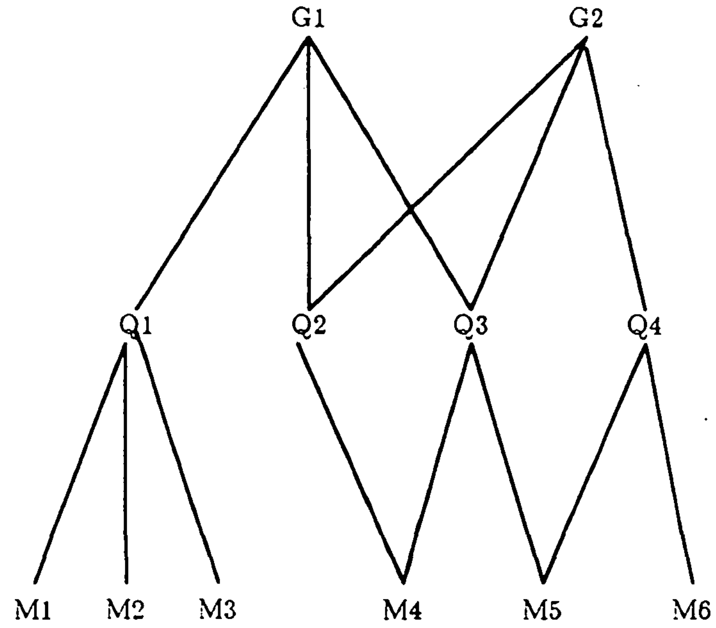


Fig. 1. The goal/question/metric paradigm. Goals = Gi. Questions = Qi. Metrics = Mi.

experience from having applied the GQM Paradigm in a variety of environments.

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

The GQM Paradigm provides a mechanism for supporting step 2a of the Improvement Paradigm, which requires a mechanism for defining operational goals and transorming them into metrics that can be used for characterization, evaluation, prediction and motivation. It supports step 3 by helping to define the experimental context and providing mechanisms for the data collection, validation and analysis activities. It also supports steps 4 and 5 by providing quantitative feedback on the achievement of goals.

The GQM Paradigm was originally used to define and evaluate goals for a particular set of projects in a particular environment, analyzing defects for a set of projects in the NASA/GSFC environment.[16] The application involved a set of case study experiments.

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

The GQM Paradigm was expanded to include various types of experimental approaches including controlled experiments.[14,17-20] This permits us to mix various types of formal experiments with actual project developments, so we can increase our understanding in more formal ways.

4 THE EXPERIMENTATION FRAMEWORK PARADIGM

An Experimentation Framework Paradigm for software engineering research is summarized in Fig. 2.[18] This framework represents a refinement

I. Definition					
Motivation	Object	Purpose	Perspective	Domain	Scope
Understand	Product	Characterize	Developer	Programmer	Single project
Assess	Process	Evaluate	Modifier	Program/project	Multi-project
Manage	Model	Predict	Maintainer		Replicated project
Engineer	Metric	Motivate	Project manager		Blocked subject-project
Learn	Theory		Corporate manager		
Improve			Customer		
Validate			User		
Assure			Researcher		

II. Planning		
Design	Criteria	Measurement
Experimental designs	Direct reflections of cost/quality	Metric definition
Incomplete block	Cost	Goal-question-metric
Completely randomized	Errors	Factor-criteria-metric
Randomized block	Changes	Metric validation
Fractional factorial	Reliability	Data collection
Multivariate analysis	Correctness	Automatability
Correlation	Indirect reflections of cost/quality	Form design and test
Factor analysis	Data coupling	Objective vs. subjective
Regression	Information visibility	Level of measurement
Statistical models	Programmer comprehension	Nominal/classificatory
Non-parametric	Execution coverage	Ordinal/ranking
Sampling	Size	Interval
	Complexity	Ratio

III. Operation		
Preparation	Execution	Analysis
Pilot study	Data collection	Quantitative vs. qualitative
	Data validation	Preliminary data analysis
		Plots and histograms
		Model assumptions
		Primary data analysis
		Model application

IV. Interpretation		
Interpretation context	Extrapolation	Impact
Statistical framework	Sample representativeness	Visibility
Study purpose		Replication
Field of research		Application

Fig. 2. Summary of the experimentation framework paradigm.

of the GQM Paradigm for experimentation. As defined in Ref. 18, it consists of four categories corresponding to phases of the experimentation process: (I) definition, (II) planning, (III) operation, and (IV) interpretation. The experiment definition phase is a formalization of the goal setting components of the GQM Paradigm, which corresponds to step 2a in the Improvement Paradigm. The experiment planning phase corresponds to the components of the GQM Paradigm for choosing the experimental design, the metrics, and the data collection forms (which also is part of step 2a in the Improvement Paradigm). The experiment operation phase corresponds to the analysis component of the GQM Paradigm and to the execution and analysis steps of the Improvement Paradigm (steps 3 and 4). The experiment interpretation phase corresponds to the interpretation component of the GQM Paradigm and to the learning and feedback step of the Improvement Paradigm (step 5). The following sections discuss the four phases of the Experimentation Paradigm in greater detail.

4.1 Experiment definition

The first phase of the experimental process is the study definition phase. The study definition phase contains six parts: (A) motivation, (B) object, (C) purpose, (D) perspective, (E) domain and (F) scope. Most study definitions contain each of the six parts; an example definition appears in Fig. 3.

There can be several motivations, objects, purposes, or perspectives in an experimental study. For example, the motivation of a study may be to understand, assess, or improve the effect of a certain technology. The 'object of study' is the primary entity examined in a study. A study may examine the

Definition element	Example
Motivation	To improve the unit testing process,
Purpose	characterize and evaluate
Object	the processes of functional and structural testing
Perspective	from the perspective of the developer
Domain: programmer	as they are applied by experienced programmers
Domain: program	to unit-size software
Scope	in a blocked subject-project study.

Fig. 3. Study definition example.

final software product, a development process (e.g. inspection process, change process), a model (e.g. software reliability model), etc. The purpose of a study may be to characterize the change in a system over time, to evaluate the effectiveness of testing processes, to predict system development cost by using a cost model, to motivate the validity of a theory by analyzing empirical evidence, etc. (For clarification, the usage of the word 'motivate' as a study purpose is distinct from the study 'motivation'.) In experimental studies that examine 'software quality', the interpretation usually includes correctness if it is from the perspective of a developer or reliability if it is from the perspective of a customer. Studies that examine metrics for a given project type from the perspective of the project manager may interest certain project managers, while corporate managers may only be interested if the metrics apply across several project types.

Two important domains that are considered in experimental studies of software are (i) the individual programmers or programming teams (the 'teams') and (ii) the programs or projects (the 'projects'). 'Teams' are (possibly single-person) groups that work separately, and 'projects' are separate programs or problems on which teams work. Teams may be characterized by experience, size, organization, etc., and projects may be characterized by size, complexity, application, etc. A general classification of the scopes of experimental studies can be obtained by examining the sizes of these two domains considered (see Fig. 4). Blocked subject-project studies examine

#Teams per project	#Projects	
	one	more than one
one	Single project	Multi-project variation
more than one	Replicated project	Blocked subject-project

Fig. 4. Experimentation scopes.

one or more objects across a set of teams and a set of projects. Replicated project studies examine object(s) across a set of teams and a single project, while multi-project variation studies examine object(s) across a single team and a set of projects. Single project studies examine object(s) on a single team and a single project. As the representativeness of the samples examined and the scope of examination increase, the wider-reaching a study's conclusions become.

4.2 Experiment planning

The second phase of the experimental process is the study planning phase. The following sections discuss aspects of the experiment planning phase: (A) design, (B) criteria and (C) measurement.

The design of an experiment couples the study scope with analytical methods and indicates the domain samples to be examined. Fractional factorial or randomized block designs usually apply in blocked subject-project studies, while completely randomized or incomplete block designs usually apply in multi-project and replicated project studies.[21,22] Multivariate analysis methods, including correlation, factor analysis and regression,[23-25] generally may be used across all experimental scopes. Statistical models may be formulated and customized as appropriate.[25] Non-parametric methods should be planned when only limited data may be available or distributional assumptions may not be met.[26] Sampling techniques[27] may be used to select representative programmers and programs/projects to examine.

Different motivations, objects, purposes, perspectives, domains and scopes require the examination of different criteria. Criteria that tend to be direct reflections of cost and quality include cost,[28-32] errors/changes,[33-38] reliability[39-46] and correctness.[47-49] Criteria that tend to be indirect reflections of cost and quality include data coupling,[12,50–53] information visibility,[54–56] programmer understanding,[57–60] execution coverage[61–63] and size/complexity.[64–66]

The concrete manifestations of the cost and quality aspects examined in the experiment are captured through measurement. Paradigms assist in the metric definition process: the goal–question–metric paradigm[17,67–69] and the factor–criteria–metric paradigm.[70,71] Once appropriate metrics have been defined, they may be validated to show that they capture what is intended.[29,72–76] The data collection process includes developing automated collection schemes[77] and designing and testing data collection forms.[67,78] The required data may include both objective and subjective data and differents levels of measurement: nominal (or classifacatory), ordinal (or ranking), interval or ratio.[26]

4.3 Experment operation

The third phase of the experimental process is the study operation phase. The operation of the experiment consists of (A) preparation, (B) execution and (C) analysis. Before conducting the actual experiment, preparation may include a pilot study to confirm the experimental scenario, help organize experimental factors (e.g. subject expertise), or inoculate the subjects.[19,60,74,79–81] Experimenters collect and validate the defined data during the execution of the study.[35,73] The analysis of the data may include a combination of quantitative and qualitative methods.[82] The preliminary screening of the data, probably using plots and histograms, usually proceeds the formal data analysis. The process of analyzing the data requires the investigation of any underlying assumptions (e.g. distributional) before the application of the statistical models and tests.

4.4 Experiment interpretation

The fourth phase of the experimental process is the study interpretation phase. The interpretation of the experiment consists of (A) interpretation context, (B) extrapolation and (C) impact. The results of the data analysis from a study are interpreted in a broadening series of contexts. These contexts of interpretation are the statistical framework in which the result is derived, the purpose of the particular study, and the knowledge in the field of research.[77] The representativeness of the sampling analyzed in a study qualifies the extrapolation of the results to other environments.[17] Several follow-up activities contribute to the impact of a study: presenting/publishing the results for feedback, replicating the experiment,[21,22] and actually applying the results by modifying methods for software development, maintenance, management and research.

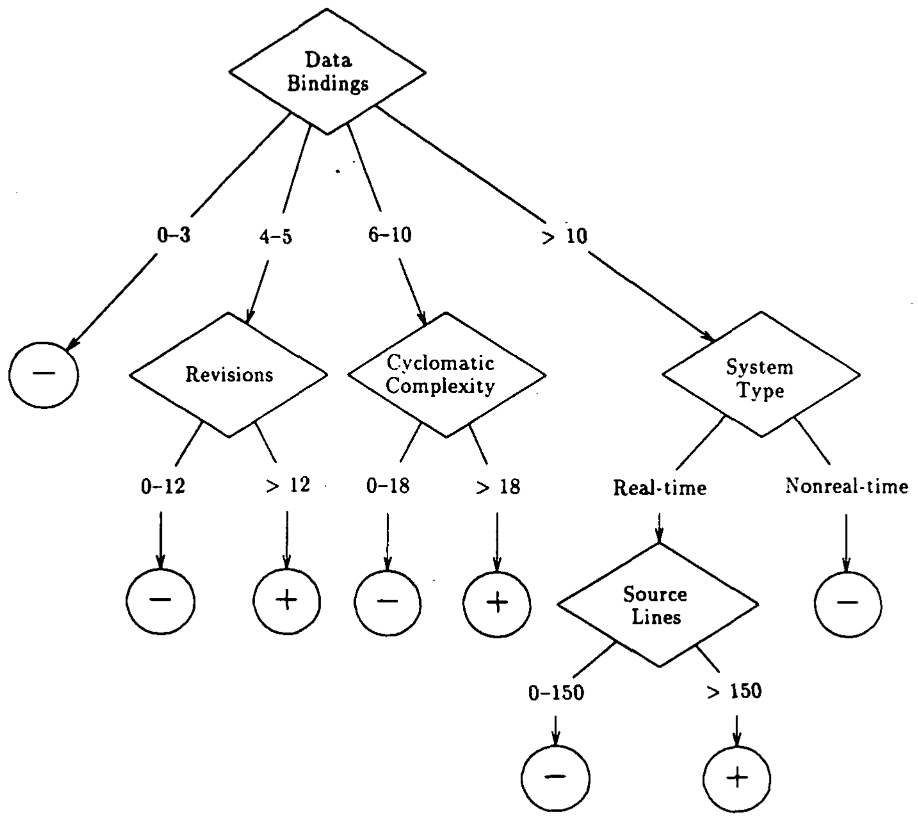
5 THE CLASSIFICATION PARADIGM

As stated earlier the Improvement Paradigm needs to be instantiated at further levels of detail and be automated whenever possible. One specific approach that can be automated for product assessment is the Classification Paradigm.[83] The Classification Paradigm provides input for what data are needed in the characterization phase of the Improvement Paradigm (step 1), focuses on specific types of goals (step 2a in the Improvement Paradigm), and automates the analysis based upon the specific product goals (step 4 in the Improvement Paradigm).

The Classification Paradigm is motivated by the '80:20 rule'. According to the rule, approximately 20% of a software system is responsible for 80% of the errors, human effort, changes, etc. The Classification Paradigm casts this phenomenon as a *classification* problem. Metric-based classification trees are constructed to identify those software components that are likely to be in the 'troublesome 20%' of the system. The classification trees are based on measurable attributes of software components and are automatically generated using data from past releases and projects. The trees provide a basis for forecasting which components on a current or future project are likely to share the same 'high-risk' properties. Examples of high-risk properties include components likely to be error-prone, change-prone, costly to develop, or contain errors in certain classes. Classification trees help localize the components likely to have these properties, and therefore enable developers to improve quality efficiently by focusing resources on high-payoff areas. Classification trees are tailorable to each development environment, using different metrics to classify different sets of components in different environments.

The Classification Paradigm supports a particular type of goal (corresponding to step 2a in the Improvement Paradigm), namely the identification of components likely to have certain properties based on historical data. The measurements used to characterize the components are determined by the classification tree generation algorithms (step 1 in the Improvement Paradigm). The metric data collected from the current project is automatically analyzed by the classification trees (step 4 in the Improvement Paradigm).

The classification trees use software metrics to characterize the software components. In other paradigms, metrics have primarily been used as barometers of goodness or badness with respect to quality and cost. This paradigm uses metrics to assess degrees of differentiation among software components. A simple example of a hypothetical metric-based classification tree is shown in Fig. 5. In the classification tree approach, the members of a set of software 'objects' (e.g. modules, subsystems) are classified as being



"+" = Classified as likely to have errors of type X

"−" = Classified as unlikely to have errors of type X

Fig. 5. Example (hypothetical) software metric classification tree. There is one metric at each diamond-shaped decision node. Each decision outcome corresponds to a range of possible metric values. Leaf nodes indicate whether or not an object is likely to have some property, such as high error-proneness or errors in a certain class.

inside or outside a 'target class' of objects. The objects inside and outside the target class are called positive and negative instances, respectively. A classification tree generation tool examines candidate metrics and recursively formulates a classification tree to identify all positive instances but no negative instances. The classification tree leaf nodes contain a probability (e.g. a simple 'yes' or 'no') to indicate whether a component is likely to be in the target class based on calibrations from previous releases and projects. The resulting classification tree then becomes the basis for forecasting whether an object, previously unseen, is a positive or negative instance.

An overview of the Classification Paradigm appears in Fig. 6. The paradigm has been applied in two validation studies using data from NASA[83] and Hughes.[84] The three central activities in the paradigm are: (i)

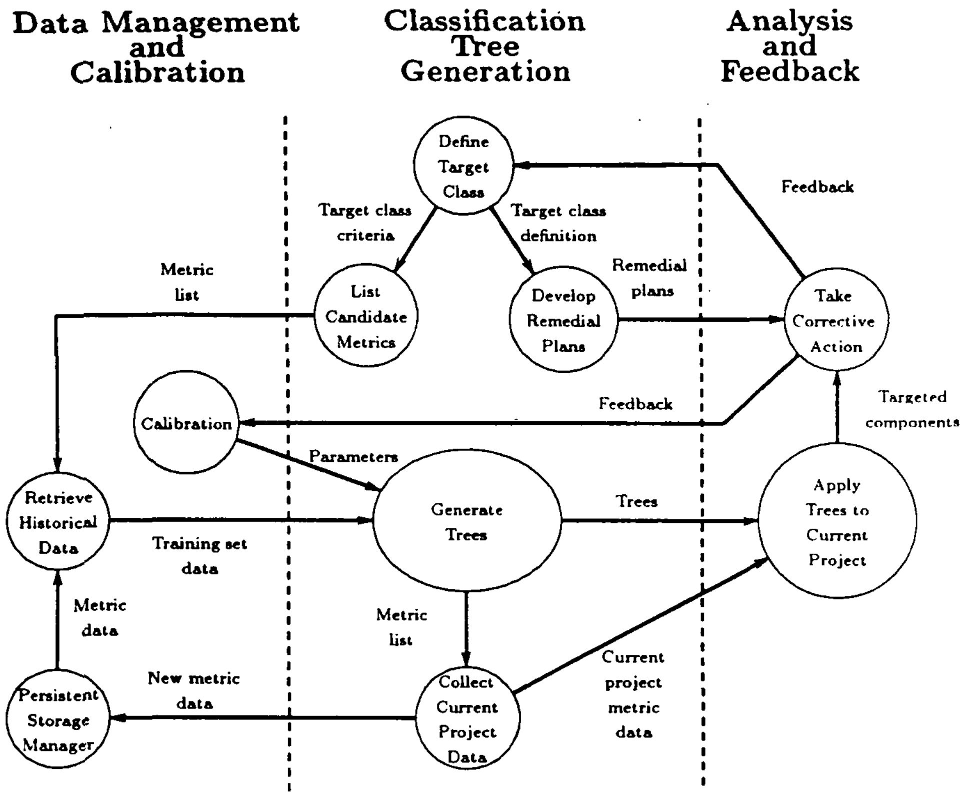


Fig. 6. Overview os classification tree approach.

data management and calibration, (ii) classification tree generation, and (iii) analysis and feedback of newly acquired information to the current project. Note that the process outlined in Fig. 6 is an iterative paradigm. The automated nature of the classification tree approach allows classification trees to be easily built and evaluated at many points in the lifecycle of an evolving software project, providing frequent feedback concerning the state of the software product.

5.1 Classification tree generation

This central activity focuses on the processes necessary to construct classification trees and prepare for later analysis and feedback. During this phase the target classes to be characterized by the trees are defined. Criteria are established to differentiate between members and non-members of the target classes. For example, a target class such as error-prone modules could be defined as those modules whose total errors are in the upper 10% relative to historical data. A list of metrics to be used as candidates for inclusion in the classification trees is passed to the historical data retrieval process. A common default metric list is all metrics for which data are available from previous releases and projects.

Importantly, one must determine the remedial actions to apply to those components identified as likely to be members of the target class. For example, if a developer wants to identify components likely to contain a particular type of error, then he should prescribe the application of testing or analysis techniques that are designed to detect errors of that type. Another example of a remedial plan is to consider redesign or reimplementation of the components. It is important to develop these plans early in the process rather than apply *ad hoc* decisions at a later stage.

Metric data from previous releases and projects as well as various calibration parameters are fed into the classification tree generation algorithms.[83] The tree construction process develops characterizations of components within and outside the target class based on measurable attributes of past components in those categories. Classification trees may incorporate metrics capturing component features and interrelationships, as well as those capturing the process and environment in which the components were constructed. Collection of the metrics used in the decision nodes of the classification trees should begin for the components in the current project. These data are stored for future use and passed, along with the classification trees, to the analysis and feedback activity.

5.2 Data management and calibration

Data management and calibration activities concentrate on the retention and manipulation of historical data as well as the tailoring of classification tree parameters to the current development environment. The tree generation parameters, such as the sensitivity of the tree termination criteria, need to be calibrated to a particular environment. For further discussion of generation parameters and examples of how to calibrate them, see Refs 83 and 84. Classification trees are built based on metric values for a group of previously developed components, which is called a 'training set'. Metric values for the training set, as well as those for the current project, are retained in a persistent storage manager.

5.3 Analysis and feedback

In this portion of the paradigm, the information resulting from the classification tree application is leveraged by the development process. The metric data collected for components in the current project are fed into the classification trees to identify components likely to be in the target class. The remedial plans developed earlier should now be applied to those targeted components. When the remedial plans are being applied, insights may result regarding new target classes to identify and further fine tuning of the generation parameters.

6 THE *TAME* PROJECT

The *TAME* project[15,63] recognizes the need to characterize, integrate and automate the various activities involved in instantiating the Improvement Paradigm for use on projects. It delineates the steps performed by the project and creates the idea of an experience base as the repository for what we have learned during prior developments. It recognizes the need for constructive and analytic activities and supports the tailoring of the software development process.

The *TAME* system offers an architecture for a software engineering environment that supports the goal generation, measurement and evaluation activities (see Fig. 7). It is aimed at providing automated support for managers and engineers to develop project specific goals and specify the appropriate metrics needed for evaluation. It provides automated support for the evaluation and feedback on a particular project in real-time as well as help prepare for post mortem analyses.

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

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

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

TAME provides mechanisms for instantiating the Improvement

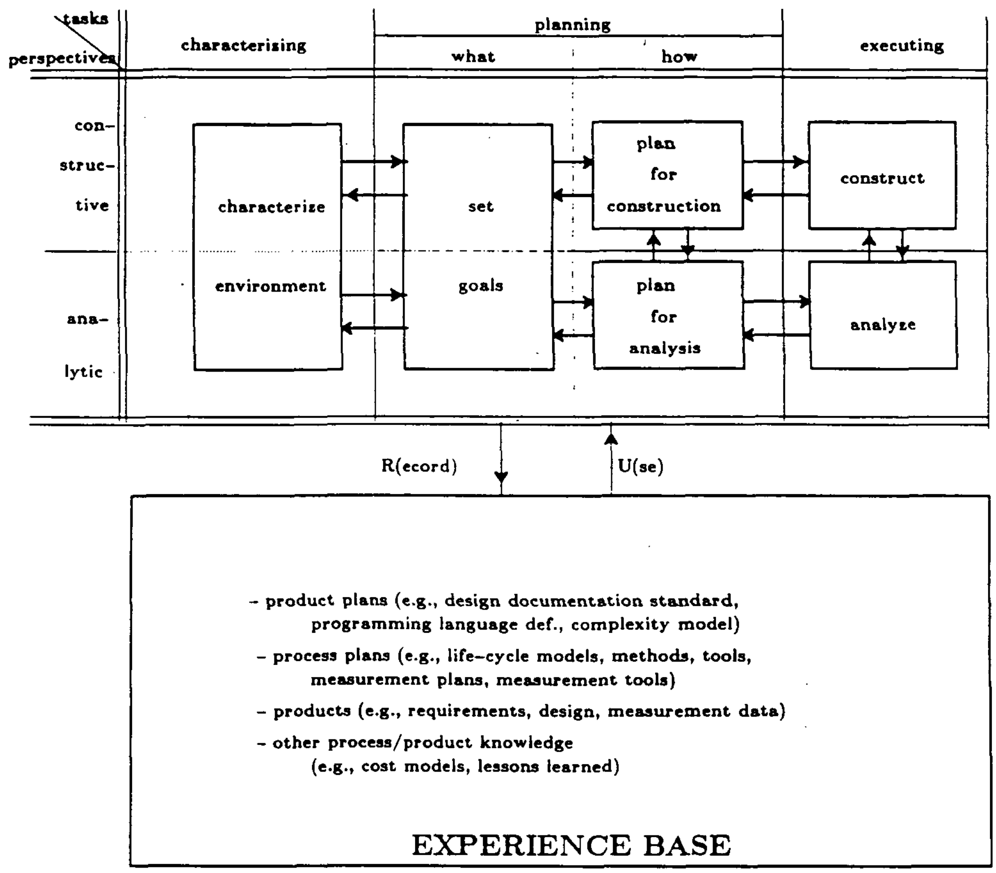

tasks
perspectives
characterizing
planning
what
how
executing
con-structive
ana-lytic
characterize environment
set goals
plan for construction
plan for analysis
construct
analyze
R(ecord)
U(se)
- product plans (e.g., design documentation standard, programming language def., complexity model)
- process plans (e.g., life-cycle models, methods, tools, measurement plans, measurement tools)
- products (e.g., requirements, design, measurement data)
- other process/product knowledge (e.g., cost models, lessons learned)
EXPERIENCE BASE

Fig. 7. The *TAME* system.

Paradigm by providing an experience base to allow the storing of experience so that it can be used on other projects (steps 1,2a), further defining the various steps to be performed (steps 1, 2, 4, 5), and automating whatever is possible.

7 CONCLUSION

Understanding the impact of software technologies is fundamental to the advancement of software research and practice. This understanding has suffered because of the lack of scientific assessment of their effect on software development and maintenance. The paradigms described in this paper are intended to help advance the use of measurement and empirical methods in software engineering. They offer a form of the scientific method for experimentation in the software domain. These paradigms have been used in a variety of environments. They permit a mix of experimental designs.

ranging from case studies to blocked subject-project studies, to live under the same framework. They provide mechanisms for integrating what has been learned from various types of experiments to help create formal bases of knowledge. They provide a framework for improving the experimental process as well as our understanding of the nature of the object of study.

ACKNOWLEDGEMENTS

This work was supported in part by the National Aeronautics and Space Administration under grant NSG-5123, and Institute for Advanced Computer Studies of the University of Maryland (UMIACS). Also supported in part by the National Science Foundation under grant CCR-8704311 with cooperation from the Defense Advanced Research Projects Agency under Arpa order 6108, program code 7T10; National Aeronautics and Space Administration under grant NSG-5123; and National Science Foundation under grant DCR-8521398.

REFERENCES

1. Boehm, B. W., Software engineering. *IEEE Transactions on Computers*, **C-25**(12) (Dec. 1976) 1226–41.
2. Zelkowitz, M. V., Yeh, R. T., Hamlet, R. G., Gannon, J. D. & Basili, V. R., Software engineering practices in the US and Japan. *IEEE Computer*, **17**(6) (June 1984) 57–66.
3. Basili, V. R. & Turner, A. J., Iterative enhancement: A practical technique for software development. *IEEE Transactions on Software Engineering*, **SE-1**(4) (Dec. 1975).
4. Boehm, B. W., A spiral model of software development and enhancement. *IEEE Computer*, **21**(5) (May 1988) 61–72.
5. Mills, H. D., Dyer, M. & Linger, R. C., Cleanroom software engineering. *IEEE Software*, **4**(5) (Sept. 1987) 19–25.
6. Royce, W. W., Managing the development of large software systems: Concepts and techniques. *Proc. WESCON*, Aug. 1970.
7. Basili, V. R., Data collection, validation, and analysis. In *Tutorial on Models and Metrics for Software Management and Engineering*, IEEE Computer Society, New York, 1980, pp. 310–13. IEEE Catalog No. EHO-167-7.
8. Boehm, B. W., Brown, J. R. & Lipow, M., Quantitative evaluation of software quality. *Proc. Second Int. Conf. Software Engng*, IEEE, New York, 1976, pp. 592–605.
9. Basili, V. R., Can we measure software technology: Lessons learned from 8 years of trying. *Proc. Tenth Annual Software Engineering Workshop*, NASA/GSFC, Greenbelt, MD, 1985.
10. Basili, V. R. & Weiss, D. M., Evaluation of a software requirements document by analysis of change data. *Proc. Fifth Int. Conf. Software Engng*, IEEE, New York, 1981, pp. 314–23.

11. Rombach, H. D. & Basili, V. R., A quantitative assessment of software maintenance: An industrial case study. *Proc. Conf. Software Maintenance*. IEEE, New York, 1987.
12. Selby, R. W. & Basili, V. R., Analyzing error-prone system coupling and cohesion. Technical Report TR-88-46, Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland, June 1988.
13. Weiss, D. M. & Basili, V. R., Evaluating software development by analysis of changes: Some data from the software engineering laboratory. *IEEE Transactions on Software Engineering*, **SE-11**(2) (Feb. 1985) 157–68.
14. Basili, V. R., Quantitative evaluation of software engineering methodology. *Proc. First Pan Pacific Computer Conf.*, Melbourne, Australia, 10–13 September 1985. (Also available as Technical Report TR-1519, Department of Computer Science, University of Maryland, College Park, July 1985.)
15. Basili, V. R. & Rombach, H. D., The tame project: Towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, **SE-14**(6) (June 1988) 758–73.
16. Basili, V. R. & Weiss, D. M., A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, **SE-10**(6) (Nov. 1984) 728–38.
17. Basili, V. R. & Selby, R. W., Data collection and analysis in software research and management. *Proceedings of the American Statistical Association and Biometric Society Joint Statistical Meetings*, Philadelphia, PA, 13–16 August 1984.
18. Basili, V. R., Selby, R. W. & Hutchens, D. H., Experimentation in software engineering. *IEEE Transactions on Software Engineering*, **SE-12**(7) (July 1986) 733–43.
19. Basili, V. R. & Selby, R. W., Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, **SE-13**(12) (Dec. 1987) 1278–96.
20. Selby, R. W., Basili, V. R. & Baker, F. T., Cleanroom software development: An empirical evaluation. *IEEE Transactions on Software Engineering*, **SI-13**(9) (Sept. 1987) 1027–37.
21. Box, G. E. P., Hunter, W. G. & Hunter, J. S., *Statistics for Experimenters*. John Wiley, New York, 1978.
22. Cochran, W. G. & Cox, G. M. *Experimental Designs*. John Wiley, New York, 1950.
23. Mulaik, S. A., *The Foundations of Factor Analysis*. McGraw-Hill, New York, 1972.
24. Neter, J. & Wasserman, W., *Applied Linear Statistical Models*. Richard D. Irwin, Inc., Homewood, IL, 1974.
25. SAS Institute, *Statistical Analysis System (SAS) User's Guide*. Box 8000, Cary, NC 27511, 1982.
26. Siegel, S., *Nonparametric Statistics for the Behavioural Sciences*. McGraw-Hill, New York, 1955.
27. Cochran, W. G., *Sampling Techniques*. John Wiley, New York, 1953.
28. Wolverton, R., The cost of developing large scale software. *IEEE Transactions on Computers*, **23**(6) (1974).
29. Walston, C. E. & Felix, C. P., A method of programming measurement and estimation. *IBM Systems Journal*, **16**(1) (1977) 54–73.

30. Putnam, L., A general empirical solution to the macro software sizing and estimating problem. *IEEE Transactions on Software Engineering*, **SE-4**(4) (1978).
31. Bailey, J. W. & Basili, V. R., A meta-model for software development resource expenditures. *Proc. Fifth Int. Conf. Software Engng*, San Diego, CA, 1981, pp. 107–16.
32. Boehm, B. W., *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981.
33. Endres, A. B., An analysis of errors and their causes in software systems. *IEEE Transactions on software engineering*, **SE-1**(2) (June 1975) 140–9.
34. Basili, V. R. & Weiss, D. M., Evaluation of a software requirements document by analysis of change data. *Proc. Fifth Int. Conf. Software Engng*, San Diego, CA, 9–12 March 1981, pp. 314–23.
35. Weiss, D. M. & Basili, V. R., Evaluating software development by analysis of changes: Some data from the software engineering laboratory. *IEEE Transactions on Software Engineering*, **SE-11**(2) (Feb. 1985) 157–78.
36. Albin, J. L. & Ferreol, R., Collecte et analyse de mesures de logiciel (collection and analysis of software data). *Technique et Science Informatiques*, **1**(4) (1982) 297–313; Rairo ISSN 0752-4072.
37. Ostrand, T. J. & Weyuker, E. J., Collecting and categorizing software error data in an industrial environment. *Journal of Systems and Software*, **4** (1984) 289–300.
38. Basili, V. R. & Perricone, B. T., Software errors and complexity: An empirical investigation. *Communications of the ACM*, **27**(1) (Jan. 1984) 42–52.
39. Currit, P. A., Dyer, M. & Mills, H. D., Certifying the reliability of software. *IEEE Transactions on Software Engineering*, **SE-12**(1) (January 1986) 3–11.
40. Jelinski, Z. & Moranda, P. B., Applications of a probability-based model to a code reading experiment. *Proc. IEEE Symposium on Computer Software Reliability*, New York, 1973, pp. 78–81.
41. Goel, A. L., Software reliability and estimation techniques. Technical Report RADC-TR-82-263, Rome Air Development Center, Griffiss Air Force Base, NY, October 1982.
42. Littlewood, B., Stochastic reliability growth: A model for fault renovation computer programs and hardware designs. *IEEE Transactions on Reliability*, **R-30**(4) (1981).
43. Littlewood, B. & Verrall, J. L., A Bayesian reliability growth model for computer software. *Applied Statistics*, **22**(3) (1973).
44. Musa, J. D., A theory of software reliability and its application. *IEEE Transactions on Software Engineering*, **SE-1**(3) (1975) 312–27.
45. Musa, J. D., Software reliability measurement. *Journal of Systems and Software*, **1**(3) (1980) 223–41.
46. Shanthikumar, J. G., A statistical time dependent error occurrence rate software reliability model with imperfect de-bugging. *Proc. 1981 National Computer Conference*, June 1981.
47. Floyd, R. W., Assigning meaning to programs. *Am. Math. Soc.*, **19** (1967).
48. Hoare, C. A. R., An axiomatic basis for computer programming. *Communications of the ACM*, **12**(10) (Oct. 1969) 576–83.
49. Linger, R. C., Mills, H. D. & Witt, B. I., *Structured Programming: Theory and Practice*. Addison-Wesley, Reading, MA, 1979.

50. Hutchens. D. H. & Basili. V. R., System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, **SE-11**(8) (Aug. 1985) 749–57.
51. Emerson. T., A discriminant metric for module cohesion. *Proc. Seventh Int. Conf. Software Engng*. IEEE. New York, 1984. pp. 294–303.
52. Stevens. W. P., Myers. G. J. & Constantine, L. L., Structured design. *IBM Systems Journal*. **13**(2) (1974) 115–39.
53. Myers. G. J., *Composite/Structured Design*. Van Notrand Reinhold, New York, 1978.
54. Parnas. D. L., A technique for module specification with examples. *Communications of the ACM*. **15** (May 1972).
55. Parnas. D. L., On the criteria to be used in decomposing systems into modules. *Communications of the ACM*. **15**(12) (1972) 1053–8.
56. Gannon. J. D., Katz. E. E. & Baisli. V. R., Measures for ada packages: An initial study. *Communications of the ACM*, **20**(7) (July 1986) 616–23.
57. Shneiderman. B., Mayer. R. E., McKay, D. & Heller, P., Experimental investigations of the utility of detailed flowcharts in programming. *Communications of the ACM*, **20**(6) (1977) 373–81.
58. Soloway. E., Ehrlich. K., Bonar. J. & Greenspan. J., What do novices know about programming? In *Directions in Human Computer Interactions*. ed. A. Badre & B. Shneiderman. Ablex. Inc., 1982.
59. Weinberg. G., *The Psychology of Computer Programming*. Van Nostrand Rheinhold. New York, 1971.
60. Weissman. L., Psychological complexity of computer programs: An experimental methodology. *SIGPLAN Notices*. **9**(6) (June 1974) 25–36.
61. Stucki. L. G., New directions in automated tools for improving software quality. In *Current Trends in Programming Methodology*. ed. R. T. Yeh. Prentice Hall. Englewood Cliffs. NJ. 1977.
62. Basili. V. R. & Ramsey. J. R., Structural coverage of functional testing. Technical Report TR-1442. University of Maryland. Department of Computer Science. College Park. MD. Sept. 1984.
63. Basili. V. R. & Rombach. H. D., Tailoring the software process to project goals and environments. In *Proc. Ninth Int. Conf. Software Engr*. IEEE. New York. 1987. pp. 345–57.
64. Basili. V. R. & Hutchens. D. H., An empirical study of a syntactic metric family. *IEEE Transactions on Software Engineering*. **SE-9**(6) (Nov. 1983) 664–72.
65. Halstead. M. H., *Elements of Software Science*. North Holland. New York. 1977.
66. McCabe. T. J., A complexity measure. *IEEE Transactions on Software Engineering*. **SE-2**(4) (Dec. 1976) 308–20.
67. Basili. V. R. & Weiss. D. M., A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*. **SE-10**(6) (Nov. 1984) 728–38.
68. Basili. V. R. & Selby. R. W., Four applications of a software data collection and analysis methodology. *Proc. NATO Advanced Study Institute: The Challenge of Advanced Computing Technology to System Design Methods*. Durham. July 29–Aug. 10. 1985.
69. Selby. R. W., Evaluations of software technologies: *testing, CLEAN-ROOM, and Metrics*. PhD thesis. Department of Computer Science. University of Maryland. College Park. 1985.

70. Cavano, J. P. & McCall, J. A., A framework for the measurement of software quality. *Proc. Software Quality and Assurance Workshop*, San Diego, CA, Nov. 1978, pp. 133–9.
71. McCall, J. A., Richards, P. & Walters, G., Factors in software quality. Technical Report RADC-TR-77-369, Rome Air Development Center, Griffiss Air Force Base, New York, Nov. 1977.
72. Basili, V. R., *Tutorial on Models and Metrics for Software Management and Engineering*. IEEE Computer Society, New York, 1980.
73. Basili, V. R., Selby, R. W. & Phillips, T. Y., Metric analysis and data validation across FORTRAN projects. *IEEE Transactions on Software Engineering*, **SE-9**(6) (Nov. 1983) 652–63.
74. Curtis, B., Sheppard, S. B. & Milliman, P. M., Third time charm: stronger replication of the ability of software complexity metrics to predict programmer performance. *Proc. Fourth Int. Conf. Software Engng*, IEEE, New York, 1979, pp. 356–60.
75. Feuer, A. R. & Fowlkes, E. B., Some results from an empirical study of computer software. In *Fourth Int. Conf. Software Engng*, IEEE, New York, 1979, pp. 351–5.
76. Zolnowski, J. C. & Simmons, D. B., Taking the measure of program complexity. *Proc. National Computer Conference*, 1981, pp. 329–36.
77. Basili, V. R. & Reiter, R. W., A controlled experiment quantitatively comparing software development approaches. *IEEE Transactions on Software Engineering*, **SE-7** (May 1981).
78. Basili, V. R., Zelkowitz, M. V., McGarry, F. E. Jr, Reiter, R. W., Truszkowski, W. F. & Weiss, D. L., The software engineering laboratory. Technical Report Rep. SEL-77-001, NASA/Goddard Space Flight Center, Greenbelt, MD, May 1977.
79. Curtis, B., Sheppard, S. B., Milliman, P., Borst, M. A. & Love, T., Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *IEEE Transactions on Software Engineering*, **SE-5**(2) (March 1979) 96–104.
80. Hwang, S-S. V., An empirical study in functional testing. Department of Computer Science, University of Maryland, College Park, Scholarly Paper 362, Dec. 1981.
81. Miara, R. J., Musselman, J. A., Navarro, J. A. & Shneiderman, B., Program indentation and comprehensibility. *Communications of the ACM*, **26**(11) (Nov. 1983) 861–7.
82. Bogdan, R. C. & Biklen, S. K., *Qualitative Research for Education: An Introduction to Theory and Methods*, 1982.
83. Selby, R. W. & Porter, A. A., Learning from examples: Generation and evaluation of decision trees for software resource analysis. *IEEE Transactions on Software Engineering*, **14**(12) (Dec. 1988) 1743–57.
84. Selby, R. W. & Porter, A. A., Software metric classification trees for guiding the maintenance of large-scale systems. *Proc. Conf. Software Maintenance*, IEEE, New York, 1989 (to appear).

SECTION 4—ADA TECHNOLOGY STUDIES

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

- "Object-Oriented Programming Through Type Extension in Ada 9X," E. Seidewitz, *Ada Letters*, March/April 1991
- "An Object-Oriented Approach to Parameterized Software in Ada," E. Seidewitz and M. Stark, *Proceedings of the Eighth Washington Ada Symposium*, June 1991
- "Designing Configurable Software: COMPASS Implementation Concepts," E. W. Booth and M. E. Stark, *Proceedings of Tri-Ada 1991*, October 1991

OBJECT-ORIENTED PROGRAMMING THROUGH TYPE EXTENSION IN ADA 9X

Ed Seidewitz
NASA Goddard Space Flight Center
Code 552.2
Greenbelt MD 20771

(301)286-7631
<eseidewitz%gsfcmail@ames.arc.nasa.gov>

1. INTRODUCTION

There have been a number of proposals for extensions to Ada to make it a "true" object-oriented programming language and there currently exist a number of object-oriented Ada preprocessors (for example, Classic-Ada from Software Productivity Solutions and DRAGOON from TXT). However, with the current Ada 9X effort there is a real possibility of adding object-oriented features to the new standard. A number of the submitted Ada 9X revisions requests asked for such features [Ada9X 89], and the draft Ada 9X requirements call out inheritance and polymorphism as "study issues" for the Mapping/Revision team [Ada9X 90].

However, in all this discussion, it is often unclear *why* such features should be added to Ada. It is important that Ada 9X not simply incorporate "object-orientation" because it is a currently popular buzzword, and it is also important that any new language features fit smoothly into the existing language "philosophy".

In this paper, I will indicate, through a simple running example, how the addition of object-oriented features might indeed be useful in Ada, and how they might be added to the language by building on existing Ada 83 features. This is presented in the hope that it will clarify some of the issues, stimulate thought and show that such features can be incorporated into Ada without doing violence to the current language design.

2. APPROACH

It has become accepted that Ada 83 is not really an "object-oriented programming language" (see, for example, [Seidewitz 87] and [Meyer 88]). This is because Ada does not support the features of inheritance, dynamic binding and polymorphism to the extent required of a "true" object-oriented programming language. Nonetheless, "object-oriented design" (such as in [Booch 88] and [Seidewitz 86]) is now widely accepted as the preferred approach to the design of software to be written in Ada. This is because Ada does provide encapsulation and abstract data types through the facilities of packages and private types, and traditional functionally-oriented methods do not provide proper guidance on the use of these features.

Now, it would certainly be possible to add a new object-oriented "class" construct to Ada. This is the approach taken by the preprocessor vendors. Proposals to introduce a "package type" capability (by analogy with "task types") have the same basic effect. However, such a new construct would strongly overlap the existing abstract data type capabilities provided by private types. Further, it would not be consistent with the fundamental approaches to encapsulation and typing taken in Ada 83.

In a typical object-oriented programming language, a *class* acts both as the type of a set of objects and as a program module [Meyer 88]. Inheritance, dynamic binding and polymorphism are provided through the type aspect of classes. Encapsulation and information hiding are provided through the module aspect of classes. In Ada, however, type definition is separated from program modularization (with the arguable exception of task types).

Object-oriented programming languages also generally require *reference semantics* [Meyer 88]. That is, names always contain *pointers* to objects, and two names may point to the same object. Ada, however, is based on *value semantics*, where each name contains an actual value and assignment requires an implicit copy operation. Reference semantics may be introduced in an Ada program only through explicit use of access types.

The approach I will take here is to provide basic object-oriented features by building on the existing Ada typing and encapsulation systems instead of adding features that overlap with them. I will use the Ada access type mechanism to introduce reference semantics when necessary for object-oriented extensions. In general, my approach is to add only those features necessary to replace the typical work-arounds, such as "call-throughs" and "case-selections", currently used when trying to program in an object-oriented style in Ada 83.

3. TYPE EXTENSION

Consider a simple monetary account data type expressed as a record with two fields, along with some obvious operations:

```
package Finance is

  type ACCOUNT_NUMBER is range 0..99999;
  type MONEY is delta 0.01 range -1_000_000.0..1_000_000.0;

  type ACCOUNT is
    record
      Identity : ACCOUNT_NUMBER;
      Balance  : MONEY;
    end record;

  procedure Deposit
    (Into_Account: in out ACCOUNT; The_Amount: in MONEY);
  procedure Withdraw
    (From_Account: in out ACCOUNT; The_Amount: in MONEY);
  function Current_Balance (Of_Account: ACCOUNT)
    return MONEY;

end Finance;
```

For the moment we will ignore issues of encapsulation and information hiding (we will return to these issues in Section 5).

Now, there are different types of accounts which we may wish to keep distinct. We can derive such types from the above ACCOUNT type:

```
with Finance;
package Bank is

  type RATE is delta 0.0001 range 0.0..1.0;
  type INTERVAL is new NATURAL;

  type SAVINGS_ACCOUNT is new Finance.ACCOUNT;

  function Interest_Earned
    ( On_Account : SAVINGS_ACCOUNT;
      At_Rate    : RATE;
      Over_Time  : INTERVAL )
    return MONEY;

end Bank;
```

The new SAVINGS_ACCOUNT type in effect "inherits" the ACCOUNT operations as derived subprograms (note that these operations must be immediately defined in the visible part of a package along with the type in order to be "derivable" [LRM 3.4]). In addition, we may define new operations which apply solely to the SAVINGS_ACCOUNT type, such as the function Interest_Earned, shown above.

Ada thus currently allows us to derive new types which inherit operations from their parent types, and to which we may add new operations. However, suppose we wish to add the appropriate interest rate as a *field* of the SAVINGS_ACCOUNT type, rather than as an argument to the Interest_Earned function? This kind of incremental extension is an important part of object-oriented approaches to design and programming.

Unfortunately, there is no easy way in Ada 83 to extend the Finance.ACCOUNT type with this new field. We could redefine the SAVINGS_ACCOUNT type from scratch, but this seems wasteful, since the new type is still mostly the same as the old one. Alternatively, we could define the new type as:

```
with Finance;
package Bank is

  type RATE is delta 0.0001 range 0.0..1.0;
  type INTERVAL is new NATURAL;

  type SAVINGS_ACCOUNT is
    record
      Parent        : Finance.ACCOUNT;
      Interest_Rate : RATE;
    end record;

  procedure Deposit ...
  procedure Withdraw ...
  function Current_Balance ...
  function Interest_Earned ...

end Bank;
```

However, we have now lost the automatic derivation of ACCOUNT operations. This forces us to write "call-through" subprograms such as:

```
separate(Bank)
procedure Deposit
  ( Into_Account : in out SAVINGS_ACCOUNT;
    The_Amount   : in Finance.MONEY ) is
begin
  Finance.Deposit (Into_Account.Parent, The_Amount);
end Deposit;
```

Such call-through subprograms are mechanical to provide, but tedious to code and maintain manually. A straightforward solution would be to provide for record type extension within Ada (this is the approach taken by Wirth in his new language Oberon [Wirth 88a,b,c]). For reasons which will become clearer as we go along, we will limit type extension to records explicitly declared as "extendible".

For example:

```
package Finance is

  type ACCOUNT_NUMBER is range 0..99999;
  type MONEY is delta 0.01 range -1_000_000.0..1_000_000.0;

  type ACCOUNT (<>) is
    record
      Identity :  ACCOUNT_NUMBER;
      Balance  :  MONEY;
    end record;

  ...

end Finance;
```

The syntax (which is not crucial to the discussion) is intended to indicate extension through similarity to parametrized record types and unconstrained arrays.

The basic idea of type extension is simple. We can define the Deposit, Withdraw and Current.Balance operations on the extendible Finance.ACCOUNT type. We may then derive new types which inherit these operations and *which may also add new fields to the parent type.* For example:

```
with Finance;
package Bank is

  type RATE is delta 0.0001 range 0.0..1.0;
  type INTERVAL is new NATURAL;

  type GENERAL_ACCOUNT is new Finance.ACCOUNT (null);

  type SAVINGS_ACCOUNT is
    new Finance.ACCOUNT (Interest_Rate: RATE);

  type CHECKING_ACCOUNT is
    new Finance.ACCOUNT
      ( Minimum_Balance : Finance.MONEY;
        Service_Charge  : Finance.MONEY );

  ...

end Bank;
```

Note how the list of new fields replaces the box <> in ACCOUNT (<>). The null in the definition of GENERAL_ACCOUNT indicates that no new fields are added. Note also that any of the above types could have been themselves declared extendible by including (<>) in their definition.

With the above definitions through type extension, derived versions of the ACCOUNT operations are inherited by the new types. No new code need be written or generated for these derived subprograms, since they can only refer to the common fields defined in ACCOUNT. We can then add new operations to the derived types, such as the Interest_Earned function for SAVINGS_ACCOUNT. We can redefine the basic ACCOUNT operations, for example by defining the CHECKING_ACCOUNT Withdraw operation to charge a service fee if the account balance falls below a minimum:

```
separate(Bank)
procedure Withdraw
  ( From_Account : in out CHECKING_ACCOUNT;
    The_Amount   : in Finance.MONEY) is
begin
  Finance.Withdraw (Finance.ACCOUNT(From_Account), The_Amount);
  if Finance.Current_Balance(Finance.ACCOUNT(From_Account))
       < From_Account.Minimum_Balance then
    Finance.Withdraw (Finance.ACCOUNT(From Account),
                      From_Account.Service_Charge);
  end if;
end Withdraw;
```

Note the use of type conversion to allow use of the parent operations.

4. POLYMORPHISM AND DYNAMIC BINDING

Suppose that we wish to store in an array the set of all accounts belonging to a single bank customer. Now, a customer may have an arbitrary number of accounts, each of which may be either a general account, a savings account or a checking account. We thus need a *supertype* which encompasses the union of all account types. Ignoring type extension for the moment, this is currently done in Ada 83 using a variant record:

```
type ACCOUNT_TYPE is (GENERAL, SAVINGS, CHECKING);
type ANY_ACCOUNT (Kind: ACCOUNT_TYPE := GENERAL) is
  record
    case ACCOUNT_TYPE is
      when GENERAL  => A_General_Account  : Bank.GENERAL_ACCOUNT;
      when SAVINGS  => A_Savings_Account  : Bank.SAVINGS_ACCOUNT;
      when CHECKING => A_Checking_Account : Bank.CHECKING_ACCOUNT;
    end case;
  end record;
```

Note the use of a default discriminant value in the definition of ANY_ACCOUNT. This allows us to define the required array type [LRM 3.7.2] as:

```
type CUSTOMER_ACCOUNTS is
  array (POSITIVE range <>) of ANY_ACCOUNT;
```

However, this means that in the general case each element of a CUSTOMER_ACCOUNTS array may have to be allocated space for the largest possible account record, in this case a CHECKING_ACCOUNT (though some compilers may be able to provide smart optimizations in special cases). For other variants, some space will be wasted. In this example, the wasted space is not too large, but in many cases, where the sizes of a variants differ more widely, this waste can be intolerable.

To avoid this, we could instead define a CUSTOMER_ACCOUNTS as an array of *pointers* to account records:

```
type ACCOUNT_REFERENCE is access ANY_ACCOUNT;
type CUSTOMER_ACCOUNTS is
  array (POSITIVE range <>) of ACCOUNT_REFERENCE;
```

Each record accessed through a CUSTOMER_ACCOUNTS array may now be dynamically allocated just the right amount of space. Note also that the default discriminant value is no longer useful, since it is illegal anyway to change the discriminant of a dynamically allocated record [LRM 4.8]. Thus we could just as well define the type ANY_ACCOUNT *without* the default:

```
type ANY_ACCOUNT (Kind: ACCOUNT_TYPE) is ...
```

In this case, the original definition of CUSTOMER_ACCOUNTS would be illegal, and the use of an access type would be *required* to construct a CUSTOMER_ACCOUNTS array [LRM 3.7.2].

The ANY_ACCOUNT variant allows us to define an effectively heterogeneous list such as CUSTOMER_ACCOUNTS, but all our operations are still defined on individual account types. Thus, to operate on any element of CUSTOMER_ACCOUNTS we must branch on the discriminant Kind to select the appropriate operation. This burden can be reduced somewhat by collecting such case selections into operations on the variant type ANY_ACCOUNT. For example:

```
procedure Deposit
  ( Into_Account : in out ANY_ACCOUNT;
    The_Amount   : in Finance.MONEY) is
begin
  case Into_Account.Kind is
    when GENERAL =>
      Bank.Deposit (Into_Account.A_General_Account, The_Amount);
    when SAVINGS =>
      Bank.Deposit (Into_Account.A_Savings_Account, The_Amount);
    when CHECKING =>
      Bank.Deposit (Into_Account.A_Checking_Account, The_Amount);
  end case;
end Deposit;
```

As with call-through subprograms, such case-selection subprograms are mechanical to write, but tedious to code and maintain. The definition of the account types as extensions of ACCOUNT in the last section does not solve this problem. The new types were defined there as *derived types*, and thus are all distinct (though mutually convertible) types. What we really want now is for, e.g., SAVINGS_ACCOUNT to be a *subtype* of ACCOUNT, in the same way that ANY_ACCOUNT(SAVINGS) is a subtype of ANY_ACCOUNT. The following definitions are consistent with the type extension notation of the last section:

```
with Finance;
package Bank is

  type RATE is delta 0.0001 range 0.0..1.0;
  type INTERVAL is new NATURAL;

  subtype GENERAL_ACCOUNT is Finance.ACCOUNT(null);

  subtype SAVINGS_ACCOUNT is
    Finance.ACCOUNT(Interest_Rate: RATE);

  subtype CHECKING_ACCOUNT is
    Finance.ACCOUNT
      ( Minimum_Balance : Finance.MONEY;
        Service_Charge  : Finance.MONEY );

  function Interest_Earned
    ( On_Account : SAVINGS_ACCOUNT;
      At_Rate    : RATE;
      Over_Time  : INTERVAL )
    return MONEY;

  procedure Withdraw
    ( From_Account : in out CHECKING_ACCOUNT;
      The_Amount   : in Finance.MONEY);

end Bank;
```

Just as in the case of an unconstrained variant record, it is now impossible to predict ahead of time what the size of a value of type ACCOUNT will be. Thus, ACCOUNT must be considered an unconstrained type. This is why extendible records must be explicitly declared as such. Extendible records are unconstrained and cannot be used directly in variable declarations. However, extensions which are not themselves extendible (such as the above account types, including the null extension) *are* constrained and may be used in variable declarations.

Following the rules for unconstrained types, an extendible type cannot be used directly in an array definition or record field definition, but it can be used in an access type definition or in the declaration of a subprogram parameter. Thus we can define a CUSTOMER_ACCOUNTS array type similarly to the case of an unconstrained variant record without discriminant defaults:

```
type ACCOUNT_REFERENCE is access Finance.ACCOUNT;
type CUSTOMER_ACCOUNTS is
  array (POSITIVE range <>) of ACCOUNT_REFERENCE;
```

Each component of a CUSTOMER_ACCOUNTS array may now point to a value of any subtype of ACCOUNT, just as we want.

The ACCOUNT operations Deposit, Withdraw and Current_Balance are still defined exactly as in the last section. However, now they may be considered *polymorphic*. That is, rather than conceptually having multiple, overloaded, derived Deposit subprograms, one single Deposit subprogram may act on values of any ACCOUNT subtype, and similarly for other operations. Thus, it is possible to make a Deposit call on an element of CUSTOMER_ACCOUNTS without any need to define a case-selection or any other additional subprogram:

```
Deposit (Fred_Accounts(3).all, 100.00);
```

where Fred_Accounts(3).all may be a GENERAL_ACCOUNT, SAVINGS_ACCOUNT or CHECKING_ACCOUNT.

As in the last section, we have defined two new subprograms in the Bank package. The function Bank.Interest_Earned is a *new* operation restricted to the subtype SAVINGS_ACCOUNT. The procedure Bank.Withdraw is a *redefinition* of the Withdraw operation for the subtype CHECKING_ACCOUNT. Any value of type ACCOUNT may be passed to these subprograms, but if that value is not of the appropriate subtype, then Constraint_Error will be raised at run time. Thus, for example, we must be careful to use Finance.Withdraw for values of subtypes GENERAL_ACCOUNT and SAVINGS_ACCOUNT, but to use Bank.Withdraw for values of subtype CHECKING_ACCOUNT. If, on the other hand, we attempt to make both Withdraw operations directly visible (say through a use Finance,Bank; clause), than an unqualified (or unrenamed) call to Withdraw will be ambiguous and therefore illegal, since overloading resolution does not consider the subtypes of parameters [LRM 8.7]. This is because Ada 83 requires static binding of subprograms to subprogram calls.

Static binding means that it is possible to determine at compile time exactly what subprogram will be called at each call statement. However, it is *not* possible to resolve at compile time a call to the overloaded Withdraw procedures based solely on the subtype of an argument, because the subtype of a value can in general only be determined at run time. Note that this was not an issue when CHECKING_ACCOUNT was a *derived type* of ACCOUNT, as in the last section, because arguments of a derived type are distinguishable at compile time from arguments of the parent type.

Thus, in Ada 83 we must explicitly check the subtype of an ACCOUNT before making a qualified call to either Finance.Withdraw or Bank.Withdraw. However, this means that a general method of handling withdrawals from any of the accounts in a CUSTOMER_ACCOUNTS array once again requires the introduction of case statements or case-selection subprograms. To avoid this, we need to allow *dynamic overloading* of operations of subtypes of extendible types. With the appropriate definitions of Bank.Withdraw for CHECKING_ACCOUNT and Finance.Withdraw for other subtypes of ACCOUNT, the following is *dynamically bound*:

```
Withdraw (Fred_Accounts(3).all, 100.00);
```

Instead of an explicit case selection on the subtype of Fred_Accounts(3).all, this call will implicitly and dynamically select the appropriate Withdraw subprogram for the subtype (the parent operation Finance.Withdraw if GENERAL_ACCOUNT or SAVINGS_ACCOUNT, the redefined operation Bank.Withdraw if CHECKING_ACCOUNT). This is truly dynamic, since, through reassignment, Fred_Accounts(3) may point to values of different subtypes of ACCOUNT at different times (assuming Fred_Accounts is a variable, not a constant).

5. ENCAPSULATION

Rather than defining ACCOUNT as a visible record type, it would be better to define it as a private type, and hide the type representation. To maintain ACCOUNT as an extendible type, however, it must still be declared as such:

```
package Finance is

  type MONEY is delta 0.01;
  type ACCOUNT (<>) is private;

  ...

private

  type ACCOUNT (<>) is
    record
      Identity :  ACCOUNT_NUMBER;
      Balance  :  MONEY;
    end record;

end Finance;
```

This is similar to including a discriminant in a private type definition.

Now, we could define a SAVINGS_ACCOUNT private type through type extension, but currently Ada requires private types to be new types, not subtypes [LRM 7.4.1]:

```
with Finance;
package Bank is

  type RATE is delta 0.0001 range 0.0..1.0;
  type INTERVAL is new NATURAL;

  type SAVINGS_ACCOUNT is private;

  procedure Initialize ...
  procedure Deposit ...
  procedure Withdraw ...
  function Current_Balance ...
  function Interest_Earned ...

private

  type SAVINGS_ACCOUNT is
    new Finance.ACCOUNT(Interest_Rate: RATE);

end Savings;
```

Type extension has simply been used here as a convenient way to implement the new private type. Outside the body of package Bank, all connection between SAVINGS_ACCOUNT and ACCOUNT has been lost. This is why all ACCOUNT operations needed to be redeclared in package Bank above.

Retaining the benefits of polymorphism and dynamic binding along with encapsulation requires the introduction of a "private extension" mechanism:

```
with Finance;
package Bank is

  type RATE is delta 0.0001 range 0.0..1.0;
  type INTERVAL is new NATURAL;

  subtype SAVINGS_ACCOUNT is Finance.ACCOUNT(private);

  procedure Initialize ...
  function Interest_Earned ...

private

  subtype SAVINGS_ACCOUNT is
    Finance.ACCOUNT(Interest_Rate: RATE);

end Bank;
```

The new type SAVINGS_ACCOUNT is now a subtype of ACCOUNT, just as before. Thus all operations on ACCOUNT immediately apply to SAVINGS_ACCOUNT, and only new or modified operations (such as Interest_Earned) need to be declared in the package specification. However, note that the representation of ACCOUNT is *not* visible within the body of package Bank. Thus, unlike most object-oriented programming languages, the supertype ACCOUNT presents the same interface to subtypes such as SAVINGS_ACCOUNT as to any other "client" (see also [Snyder 86]).

6. CONCLUSION

The combination of a package defining an extendible private type or subtype corresponds to the usual object-oriented concept of a class. There are, of course, a number of additional issues which I have not addressed at all in this paper. These include (at least):

- o The definition of "virtual" or "deferred" subprograms declared as operations of a supertype, but with bodies defined only for subtypes.
- o The interaction of object-oriented features with the Ada concepts of block structured scoping, generics and tasking.
- o The question of whether to allow dynamically bound subtype operations on supertype variables, raising `Constraint_Error` (or some other exception) at run-time if the operation is not possible.

Such issues must be resolved in any complete design for object-oriented features in Ada. However, they do not strongly impact on the basic discussion of whether not to include object-oriented features in Ada at all.

I personally think that inclusion of some object-oriented features in Ada 9X is crucial. I think it is valid to worry that lack of such features will make Ada less attractive relative to other languages (such as C++ [Stroustrop 86]). However, features should not be included in Ada 9X simply because they are popular elsewhere. Rather, we need to understand *why* these other languages are popular and whether they satisfy needs which could and should be satisfied by Ada 9X.

REFERENCES

[Ada9X 89]
Ada 9X Project, *Revision Request Report*, Office of the Under Secretary of Defense for Acquisition, August 1989

[Ada9X 90]
Ada 9X Project, *Draft Ada 9X Requirements Document (Version 3.3)*, Office of the Under Secretary of Defense for Acquisition, August 1990

[Booch 88]
Grady Booch, *Software Engineering with Ada (2nd Edition)*, Benjamin-Cummings, 1988

[LRM]
Department of Defense, *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A-1983, February 1983

[Meyer 88]
Bertrand Meyer, *Object-Oriented Software Construction*, Prentice-Hall, 1988

[Seidewitz 87]
Ed Seidewitz, "Object-Oriented Programming in Smalltalk and Ada" in *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, SIGPLAN Notices*, December 1987

[Seidewitz 86]
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" in *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, *SIGPLAN Notices*, November 1986

[Stroustrop 86]
Bjarne Stroustrop, *The C++ Programming Language*, Addison-Wesley, 1986

[Wirth 88a]
N. Wirth, "Type Extensions", *ACM Transactions on Programming Languages and Systems*, April 1988

[Wirth 88b]
N. Wirth, "From Modula to Oberon", *Software Practice and Experience*, July 1988

[Wirth 88c]
N. Wirth, "The Programming Language Oberon", *Software Practice and Experience*, July 1988

AN OBJECT-ORIENTED APPROACH TO PARAMETERIZED SOFTWARE IN ADA

Eighth Washington Ada Symposium, June, 1991

Ed Seidewitz and Mike Stark

Flight Dynamics Division
Code 552
Goddard Space Flight Center
Greenbelt MD 20771

(301)286-7631

1. INTRODUCTION

A *parameterized* software system is one that can be configured by selecting generalized models and providing specific parameter values to fit those models into a general design [Stark 1990]. This is in contrast to the top-down development approach where a system is designed first, and software is reused only when it fits into the design. The concept of parameterized software is particularly useful in a development environment such as the Goddard Space Flight Center Flight Dynamics Division (FDD), where successive systems have similar characteristics.

1.1 Parameterized Systems

Rather than being a monolithic piece of code, a parameterized system instead consists of a library of components which can be interconnected in both standard and novel ways. A *configuration process* produces a program for a specific application task from a parameterized system. This process involves the selection of appropriate components, their interconnection according to a standard architecture and the provision of values for relevant parameters. The configuration process replaces the normal development cycle for the program.

The packaging and generic features of the Ada language are particularly useful in creating the components of such a parameterized system. Ada packages allow tightly coupled data and functionality to be grouped together in a single component, while generics allow these components to be completely decoupled from other components. While parametrizing *all* the external dependencies of a component through generic parameters may seem extreme, it both allows the component to be defined in a completely stand-alone manner and interconnected with other components in flexible ways.

What we term parameterized systems are thus seen to be the kind of software which tends to evoke analogies with hardware. That is, the components of a parameterized system are analogous to standard hardware components with well defined interfaces which may be interconnected in various architectures. Achieving such a way of doing business in software development has been an extremely attractive, but so far mostly elusive, goal.

While there are a number of difficulties with achieving true parameterized software systems, perhaps the biggest problem is in constructing truly generalized components. Except in the simplest cases (such as standard data structures) it seems to be impossible to design the packaging and parametrization of a component *a priori* to any understanding of how that component might be applied. Thus, there is an increasing realization of the need for the design of components to be based on an analysis of the *domain* in which the component will be used [Prieto-Diaz 1990]. For example, a *domain analysis* approach was used to identify components for the Common Ada Missile Packages (CAMP) project [CAMP 1990]. In fact, we consider domain analysis to be exactly the process of specifying a parameterized software system, in much the same way that system analysis is the process of specifying a more traditional software system.

Currently, the FDD at Goddard has plans to develop a parameterized system known as the Combined Operational Mission Planning and Attitude Support System (COMPASS) [DeFazio et al. 1991]. The specification and implementation approaches planned for COMPASS are based on previous methodology work in the FDD and experience with reusable Ada simulator components [Booth and Stark 1989]. However, COMPASS is considerably larger in size and scope than any other Ada system developed in the FDD. Indeed, rather than specifying

10000174

COMPASS in terms of a single domain, we have found it necessary to consider COMPASS to cover three related domains, two *application* domains and a supporting *core* domain. The Ada code developed for these domains is expected to be on the order of 1 million lines (measured as terminal semicolons).

1.2 Overview

In this paper we present an overview of the specification and implementation approaches being used for COMPASS. While these approaches have been developed specifically for COMPASS, we feel that they can also provide a basis for more general methods for developing parameterized software. In the following sections we highlight the object-oriented nature of our approach and how we have adapted object-oriented concepts and techniques for COMPASS.

The domain analysis approach used for COMPASS has evolved out of work over the past two years in the Attitude Support application domain. The resulting specification concepts, discussed in Section 2, are strongly object-oriented [Seidewitz et al. 1991]. The object-oriented paradigm allows both the precise definition of required domain functionality and clear classification of that functionality. Classification is extremely important for organizing a problem domain and is also key to generalizing the specifications.

Another crucial benefit of the object-oriented approach is that the components of a parameterized system may be identified with a class specified during domain analysis. Thus the implementation approach for COMPASS, discussed in Section 3, uses Ada packages to implement classes [Booth and Stark 1991]. We note two important consequences of the use of Ada as our implementation language. First, our object-oriented specifications explicitly show the dependencies between classes. This is necessary to build a comprehensible "map" of the problem domain. However, as we stated earlier, our goal is to use Ada generics to completely parametrize all dependencies between components. Thus, instead of directly reflecting specified dependencies in the implemented components, the explicit dependencies in the specifications are used to identify the generic parameters for the implemented components.

The second consequence results from the lack of "full" object-oriented programming features in Ada: the Ada typing system cannot directly accommodate the object-oriented superclass/subclass hierarchy (see also the discussion in [Seidewitz 1987] and [Seidewitz 1991]). Rather than trying to closely simulate this hierarchy in some way in the Ada components, we have simply chosen to not implement superclasses as generalized components. Instead, categories can be generated in a straightforward way during the configuration process [Booth and Stark 1991].

2. SPECIFICATION APPROACH

Domain analysis is largely a process of *understanding* an application domain and then *documenting* that understanding in a clear, formal manner. To gain an understanding of an application domain, the domain analyst must first learn the fundamental terms and concepts of the domain. These concepts provide the basic vocabulary which may then be used to document the detailed functionality of the domain. This vocabulary is built based on an analysis of existing systems, future requirements and the knowledge of domain experts.

One of the COMPASS application domains involves ground operations to support the determination and control of the orientation (or *attitude*) of a spacecraft. A major area of this domain is driven by the processing of sensor data received from the spacecraft. Thus the concepts of *sensor* and *measurement* are crucial for understanding the domain. Further, there are several different types of sensors which may be on board a spacecraft, such as *sun sensors, star trackers* and *horizon scanners.* The processing of measurements from these sensors requires mathematical models of the sensors. These models in turn require models of the environment such as *star map, sun ephemeris,* and *horizon radiance.*

An object-oriented approach is very natural for formalizing an analysis such as this (see also [Shlaer and Mellor 1988; Coad and Yourdon 1991]). For our purposes, an *object* is a model of a specific concept in the application domain. This model includes both attributive data (or *data parameters*) that describe the object and operational functionality (provided by a set of *functions*) that specifies appropriate processing. Note that this is the key concept of the object-oriented approach: appropriate data are packaged together with all associated functionality to specify an object in the application domain.

2.1 Classes

A *class* is a group of objects specified together. All objects in a class share the same specifications of data and functionality. This class specification is effectively a "template" for individual objects of the class. For example, an individual star tracker is modeled as an object. The set of all star tracker objects is grouped together into a single class. While each star tracker object remains distinct, they all share a common specification model (each star tracker "works the same" as the others).

All objects of the same class thus have the same kinds of data parameters, and each parameter has a well-defined type which specifies its possible values. Each individual object of a class has specific values for each parameter within the appropriate types, though these values may vary over time.

In the COMPASS specification approach, an object may have any of the following three types of parameters:

- *Configuration parameters* are used to specialize a general algorithm to a particular mission These parameters are set during the configuration process, and do not change during the normal operation of a configured program.
- *Operation parameters* have values supplied by the user. The user can set and change these parameters during normal use of a configured program.
- *Internal data* defines the current "state" of the object. This state may evolve over time according to the specification of the processing associated with the object. (These parameters correspond to *instance variables* in object-oriented programming terminology.)

For example, consider a simple *Star Tracker* class. The following is part of the specification for this class giving examples of each of the above types of parameters:

Class Star Tracker

Configuration Parameters

Name	Type	Description
θ,φ,ψ	Radians	The nominal alignment angles of the sensor as mounted on the spacecraft body
...		

Operation Parameters

Name	Type	Description
$\bar{b}$	Unitless (3×1)	Measurement bias correction
...		

Internal Data

Name	Type	Description
H	Unitless	Horizontal position in the field of view of the current star being tracked
V	Unitless	Vertical position in field of view of the current star being tracked
...		

At any one point in time, a specific star tracker has specific values for each of these parameters. How these values may vary over time is different for each type of data parameter. In this example, the alignment of each star tracker object would be set when the parameterized system is configured for a specific use (at *configuration time*). However, the user could set specific bias corrections for each star tracker (at *run time*). Finally, the values of the internal data evolve as specified by the internal processing associated with a star tracker, such as the response to commands and requests for measurements.

Objects can also be more abstract. For instance, an *Analytical Orbit* class might specify elements of the orbit as operation parameters and maintain the position and velocity of a spacecraft as internal data. An *Estimator* might have configuration parameters indicating what is to be estimated and maintain the current estimation state error as internal data.

Classes provide a delineation and description of objects in the application domain. This structure also provides a natural framework for specifying the functionality required in the application domain. To do this, each class has an associated set of functions that specify the processing required for each object in that class. For example, a *Star Tracker* class might have functions that simulate the operation of a star tracker.

The specification of a class includes detailed specifications for each function associated with the class. Each function belongs to one of two groups. *Constructor functions* act as commands to an object to initiate some processing. This processing may involve alteration of internal data and or interaction with other objects. For example, *Star Tracker* constructors might include functions to *Enable* the sensor and *Break Track* of the current star.

Selector functions, on the other hand, do not alter internal data. Selectors simply respond to requests for an object to provide a result based on internal data. For example, the *Star Tracker* class could have a selector to provide a *Measurement* based on internal data on the currently tracked star.

We observe the convention that functions must be either constructors or selectors, but not both. That is, a function may not both alter internal data and provide a result based on that data. This convention helps make class specifications less complicated, more easily understood, and less likely to be overspecified.

The sole means for object interaction is for one object to request the processing specified in a function of another object. Such a request is a *message* that passes from the sending object to the receiving object. This message includes the name of the requested function and any required function arguments. In the case of a selector, the receiving object responds with an appropriate result.

One class *depends* on another if objects of the first class send messages to objects of the second class. Dependencies model the relationships between application domain concepts. Thus, a star tracker object would depend on a star map object, while a sun sensor object would depend on a sun model.

2.2 Categories

Since parameterized software must be highly generalized, its specification will include a large number of different classes. These classes must be organized in a manner which follows the conceptual hierarchy of the application domain. For example, the *Star Tracker* class is a *subclass* of a more general *Sensor* class. This *Sensor superclass* would also include such classes as *Sun Sensor* and *Horizon Scanner.*

In general, superclasses may themselves be subclasses of yet more general classes. However, in COMPASS we have found it sufficient to limit ourselves to a shallow two-level hierarchy. We refer to superclasses as *categories* and every class must belong to a category, even if it is (initially) the only member of its category.

In the general object-oriented approach, a superclasses also provides specifications of data and functionality which may be *inherited* by its subclasses. This provides a mechanism for factoring out common properties of subclasses so they do not have to be repeatedly respecified. Rather surprisingly, we have not found this to be a particularly important mechanism so far for COMPASS specifications. Instead, a COMPASS category simply specifies a common functional interface for its member classes. That is, all objects of any class in a category must respond to a common set of messages, although they will generally respond in different ways. (In object-oriented programming terms, categories are *abstract superclasses.*) For example, all *Sensor* classes must have a *Measurement* selector, but the *Star Tracker* class would specify different processing for this function than the *Sun Sensor* class.

The specification of a category includes a list of messages accepted by objects of every class in that category. For example, the following is a simplified specification for the *Sensor* category:

Category Sensor

Constructors

Message	Argument	Type	Description
Initialize	S	Sensor	Initialize sensor S
Power_On	S	Sensor	Turn on sensor S
Power_Off	S	Sensor	Turn off sensor S
Simulate	S	Sensor	Simulate sensor S observations at time t
	t	Seconds	
Output	S	Sensor	Store the current measurement of sensor S into the data base

Selectors

Message	Argument/ Response	Type	Description
Measurement	S	Sensor	v is the current measurement for sensor S
	$\bar{v}$	Real (3×1)	

The specification of each class in a category must include specifications for all functions necessary to respond to the messages listed for the category. This makes the classes in the category largely interchangeable from the point of view of other classes, as required for reconfiguration. However, sometimes a class will have functionality that is not readily generalized to other classes in its category. For example, the *Star Tracker* constructors *Set Track* and *Break Track* would not readily generalize to other *Sensor* classes. Thus, while each class must specify *at least* all functions required for its category, it may also specify additional functions unique to that class.

Generally, it is better for a class to depend on a category rather than specific classes within a category. For example, consider an *Estimator* class that specifies objects for estimating parameters of interest based on sensor data. Such an estimation technique may be generalized to only depend on functionality defined in the general class *Sensor*, not on the particulars of any subclass. Thus, such a specification is valid for any spacecraft, no matter what its specific complement of on-board sensors. This is the motivation for making all classes in a category "look the same" to other classes.

2.3 Subsystems

To organize the specifications for a large application domain, we divide the domain into groups of categories called *subsystems*. A subsystem contains all categories necessary to specify the functionality in a specific high-level area of processing. The goal is to group together categories containing classes that are operationally often needed together and are interdependent on each other. COMPASS subsystems are similar in some respects to "subjects" in [Coad and Yourdon 1991].

The formalization of dependencies provides a "map" of each subsystem in an application domain. This map may be presented graphically in a *dependency diagram* (which are comparable to the diagrams used in [Coad and Yourdon 1991; Shlaer and Mellor 1988]). Figure 1 shows a dependency diagram for an extremely simplified version of the COMPASS *Estimation* subsystem, which deals with parameter estimation. In this diagram, arrows represent dependencies between classes. To simplify the diagram, dependencies that are common to a number of classes or an entire category are shown by grouping the classes. Diagrams such as this, along with full textual specifications of each category and class, provide the formal specification of the application domain.

The *Estimation* subsystem shown in Figure 1 is an example of an *application subsystem*. An *application subsystem* specifies a major area of functionality in a specific COMPASS application domain. Further, each application subsystem has an associated set of *actions*. The actions of a subsystem are analogous, at a higher level, to the functions of an object. Actions identify the higher-level capabilities of the subsystem as a whole. The action specification shows what messages are sent to initiate the processing required for each capability.

Actions also provide the basis for specifying interaction with the user of a configured program. They define the smallest processing steps of direct interest to a user. The user may select from a specific set of actions at

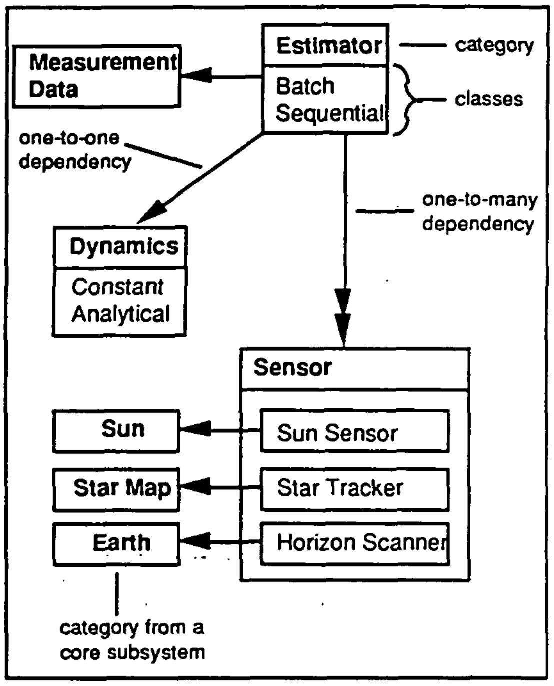


Figure 1 Dependency Diagram for the *Estimation* Subsystem

specified *decision points*. Actions act as "buttons" that, when pressed, initiate some activity by a subsystem and cause a transition to a new decision point. Thus, a dialog with the user may be specified using a state transition formalism. This formalism is used to specify a number of possible *drivers* for each application subsystem.

A second type of subsystem is a *core subsystem*. A core subsytem is a part of the specification for the common *core software* on which all COMPASS application software may be built. Core subsystems do not have actions and drivers. Instead, they provide functionality available to all application subsystems, while application subsystems are specific to certain application domains (this is somewhat akin to the domain hierarchies discussed in [Prieto-Diaz 1990]). The *Spacecraft Model* subsystem shown in Figure 2 is an example of a core subsystem.

The *Spacecraft Structure* category in Figure 2 is also an example of the specification of *aggregate* objects. In this example, the physical structure of a spacecraft is composed of a main body with a number of other components attached to it. These structural components are objects with their own attributes and functionality (e.g., it may be possible to command them to move relative to the spacecraft body).

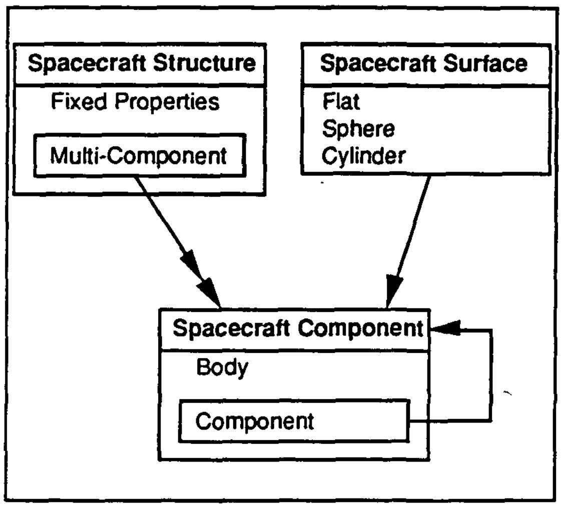


Figure 2 Dependency Diagram for the *Spacecraft Model* Subsystem

Rather than introducing a new notation specifically for such aggregation (such as the "whole-part structures" in [Coad and Yourdon 1991]), we have found it sufficient for COMPASS to simply model aggregation using dependencies. Thus, the *Spacecraft Structure* has a one-to-many dependency on the *Spacecraft Component* category (as shown in Figure 2). The *Spacecraft Structure* category specifies properties related to the spacecraft as a whole (such as the overall mass and moment-of-inertia) while the *Spacecraft Component* category deals with individual properties of components (such as component mass and orientation).

Even given the *Spacecraft Structure* aggregate, though, some classes may still depend directly on individual components. For instance, a model of all the external surfaces of a spacecraft is required to compute such things as the atmospheric drag on the spacecraft. Each surface in this model is a surface of some spacecraft component. This is shown in Figure 2 as a dependency between the *Spacecraft Surface* category and the *Spacecraft Component* category.

2.4 Configuration

As described above, application domain specifications are written in terms of classes of objects and their allowable interdependencies. To configure a parameterized system, however, one must specify specific objects of interest and their actual interdependencies. For example, a specific spacecraft may have one star tracker, and two sun sensors, modeled as *instances* of the general *Star Tracker* and *Sun Sensor* classes. These objects are specified by providing values for all configuration parameters (such as the alignment of the star tracker) and identifying objects to satisfy all dependencies (such as a star map object that may be used by the star tracker model). For COMPASS we use a tabular form for these specifications, with configuration parameter and dependency names from the class specification and a column of values for each class instance.

Categories also play an important role in the configuration process. To aid the process, each category has associated rules for using member classes for specific applications. For example, the *Sensor* category might have the usage rule "Use zero or more *Sensor* objects from appropriate classes for each mission". On the other hand, an *Orbit* category might have a usage rule to "Use one *Orbit* object for each spacecraft handled in a mission-support task". In effect, the analyst specifying an application program can follow the dependency arrows on the dependency diagram "maps" of the application domain, selecting objects from various categories following the usage rules.

Typically, an application of a parameterized system will require the configuration of a number of programs. In the case of COMPASS, such an application generally corresponds to the support of the "mission" of a single spacecraft. The support of a typical mission requires various analysis, data processing and operations programs. Since all these programs relate to the same spacecraft, they will share many of the same models, especially from the core domain. Thus, many objects may be specified on a "per mission" rather than "per program" basis. This set of objects provides a common repository for information on the mission as it evolves from early definition stages to launch. Additional objects may be specified on a "per program" basis.

Application programs are themselves specified based on a single driver from a specific application subsystem. This driver provides the basis for "gluing together" various objects specified from classes in the application subsystem and supporting core subsystems. For example, a program to provide real-time estimates of the spacecraft attitude might be specified based on a *Real-Time Estimation* driver of the *Estimation* subsystem, using an object from the *Sequential Estimator* class , various sensor objects, etc. On the other hand, a program to provide definitive (highest accuracy) estimates would be based on a *Non-Real-Time Estimation* driver, using the most accurate *Batch Estimation* algorithm, but the same sensor objects and other spacecraft models.

Note that the COMPASS configuration approach is fundamentally different from the more typical view of object-oriented system specification. In the latter view, classes specify objects that are created and destroyed when the specified system is run. In the COMPASS approach, in contrast, classes in a domain specification specify objects which are created at *configuration time*. No additional objects are created when the configured program is run. Instead, the already existing objects interact by passing data which are not themselves considered to be objects (for COMPASS this data tends to be such things as numbers, vectors, matrices, etc.). This is thus a "hybrid" rather than a "pure" object-oriented view. This view allows us to give a clear, precise definition to the configuration process and is particularly appropriate when the specified software is to be implemented in a modular language such as Ada.

Nevertheless, in a more general parameterized system approach, it may be useful to retain a more dynamic object-oriented run time viewpoint. In this case, it would be possible to specify some objects as "fixed" at configuration time, while others could be created and destroyed at run time. Alternatively, one could consider the data passed between objects created at configuration time to themselves be objects from a domain at an even lower level than the core domain. This low level "computational" domain would define standard classes of data types which are implicit in the application and core domains. Objects of these classes could be dynamically created and manipulated at run-time, while objects of the application and core domains would be fixed at configuration time. This latter approach might be particularly appropriate for software to be implemented in a "pure" object-oriented language.

3. IMPLEMENTATION APPROACH

The COMPASS implementation concepts are based on a tailored version of the general reuse model proposed by Booth and Stark [Booth and Stark 1989; Stark 1990]. This model separates reusable software components into a number of layers. The following three layers are the most relevant to the COMPASS implementation concepts:

- The *domain language* layer provides a virtual language for expressing the functionality and data of problem domain objects and classes. This layer defines basic utilities which are taken as given in the specifications (such as vector and matrix arithmetic in COMPASS).

- The *domain object* layer provides definitions for problem domain classes. Specified classes are directly implemented in this layer.

- The *architecture layer* binds the domain components and generalized services to a specific system architecture. This layer contains modules that combine the domain objects and classes with the services that make them work within a system architecture. It also provides the "glue" that dictates how modules are connected to form an executable system.

This section will focus mainly on the domain object and architecture layers for COMPASS.

3.1 Class Packages

Classes in the specification are implemented as parameterized components in the domain object layer. These *class packages* are actually parameterized in two ways. First, an outer generic package parameterizes dependencies on domain language layer utilities. This generic exports the abstract data type (private type) which actually defines the class. Second, one or more nested generics parameterize the dependencies on other classes that are given explicitly in the class specification. This structure is shown pictorially in Figure 3 and is discussed further in the following.

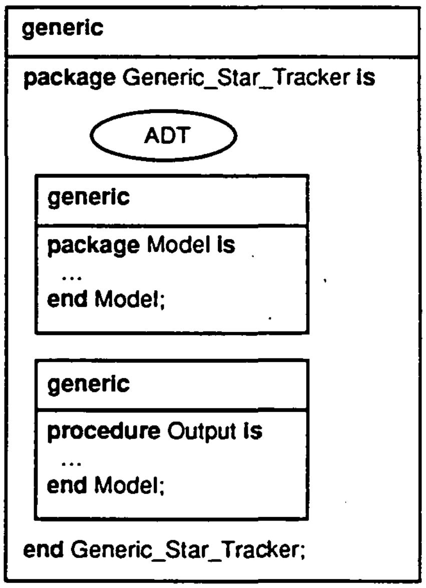

```
generic
package Generic_Star_Tracker is
    ADT
    generic
    package Model is
      ...
    end Model;
    generic
    procedure Output is
      ...
    end Model;
end Generic_Star_Tracker;
```

Figure 3 Structure of the Class Package for *Star Tracker*

Consider the class package that implements the *Star Tracker* class. The outer generic formal part of this component parameterizes dependencies on such types as vectors and matrices as well as operations on these types, such as matrix multiplication:

```
with Index_Types; -- import constrained indices
generic
  type TIME is private;
  type REAL is digits <>;
  type VECTOR3 is
    array (Index_Types.INDEX3) of REAL;
  type MATRIX33 is
    array(Index_Types.INDEX3, Index_Types.INDEX3)
      of REAL;

  with function "*" (M : MATRIX33; V : VECTOR3)
    return VECTOR3 is <>;
  ...

package Generic_Star_Tracker is
  ...
end Generic_Star_Tracker;
```

The types and subprograms used to instantiate *Generic_Star_Tracker* come from utility packages in the domain language layer. The utility packages can be for very general domains (such as linear algebra), or can be tailored to a specific domain (such as spacecraft orbit representations). The important idea is that *all* the types and functions needed to express the *Star Tracker* algorithms are imported through generic parameters. This allows COMPASS class packages to be instantiated in *any* system that provides the required types and subprograms and makes the COMPASS classes independent of the COMPASS system architecture.

When instantiated, the Star Tracker class exports the private type *ADT*, which defines all the data specified for the class:

```
generic
  ...
package Generic_Star_Tracker is

  type ADT is private;

  type PARAMETERS is record
    -- configuration parameters
    Theta            : REAL;
    Phi              : REAL;
    Psi              : REAL;
    ...
    -- operation parameters
    Bias_Correction  : VECTOR3;
    ...
  end record;

  procedure Set_Parameters (
    S                : in out ADT;
    To_The_Values    : in PARAMETERS);
  function Parameter_Values (S : ADT)
    return PARAMETERS;

  ...
private

  type ADT is
    record
      -- internal data
      H              : REAL := 0.0;
      V              : REAL := 0.0;
      Power_Is_On    : BOOLEAN := FALSE;
      ...
      -- parameters
      The_Parameters : PARAMETERS;
    end record;

end Generic_Star_Tracker;
```

Each item of internal data becomes a component of the record defined for *ADT*. The last component of *ADT* is of type *PARAMETERS*. This data type defines the configuration and operation parameters for the class. A constructor *Set_Parameters* and a selector *Parameter_Values* are defined to allow access to configuration and operation parameters.

The remainder of the generic package specification for *Generic_Star_Tracker* defines the functionality associated with the class. The procedures and functions specified in this package match the constructors and selectors specified for the class. Generally, a subset of these operations do not depend on any other class for their implementation. For example, the constructors *Power_On* and *Power_Off* affect only the *ADT* record component *Power_Is_On*. Operations such as this are *primitive operations* of the type *ADT*:

```
generic
  ...
package Generic_Star_Tracker is

  type ADT is private;
  ...

  -- constructors
  procedure Power_On (S : in out  ADT);
  procedure Power_Off (S : in out ADT);
  -- class specific operations
  procedure Set_Track (S : in out ADT);
  procedure Break_Track (S : in out ADT);

  -- selector
  function Measurement (S : ADT)
    return VECTOR3;

  ...
end Generic_Star_Tracker;
```

Of course, usually some of the operations defined in a class package will depend on other classes. As discussed in Section 2.1, such dependencies are given explicitly in the class specification. Class packages, however, completely parameterize dependencies using generic units nested within the class package. A dependency is parameterized using a generic formal private type and generic formal subprograms for any operations. The dependency itself is then coded as a generic formal parameter of the private type.

For example, the *Star Tracker* operations *Initialize* and *Simulate* depend on being able to access a *Star Map* catalog to get stars in the field of view of the sensor. Similarly, the *Output* operation depends on being able to *Write* data to a *Measurement Data* file. These dependencies are reflected in the *Generic_Star_Tracker* class package as follows:

```
generic
  ...
package Generic_Star_Tracker is

  type ADT is private;
  ...

  generic
    type STAR_MAP is private;
    type STARS_IN_VIEW
      is array (POSITIVE range <>) of VECTOR3;
    with function Stars_In_Field_Of_View (
      From_Map      : STAR_MAP;
      Around_Vector : VECTOR3;
      Cone_Angle    : REAL)
      return STARS_IN_VIEW;

    The_Map : STAR_MAP;

  package Model is
    procedure Initialize (S : in out ADT);
    procedure Simulate (
      S             : in out ADT;
      At_Time       : in TIME);
  end Model;

  generic
    type MEASUREMENT_DATA is private;
    with procedure Write (
      To_File       : in out MEASUREMENT_DATA;
      From_Sensor   : in ADT;
      At_Time       : in TIME;
      The_Data      : in VECTOR3);

    The_File : MEASUREMENT_DATA;

  procedure Output (S : in ADT);

  ...
end Generic_Star_Tracker;
```

Note that two nested generics are used above because the dependency of *Initialize* and *Simulate* on a *Star Map* is independent of the dependency of *Output* on a *Measurement Data* file. The basic *Star Tracker* abstract data type may be used along with instances of either or both nested generics, depending on what subset of *Star Tracker* capabilities is needed to meet the particular system requirements. One may even use the package without instantiating either of the generics, if that is sufficient for the system being configured.

The factoring of class dependencies into multiple nested generics provides the ability to define precise increments of functionality for a system configuration. In effect, instantiating different combinations of nested generics provides different subclass extensions of the basic class defined by the abstract data type and its primitive operations. Thus, this approach provides some of the combinatorial simplification of the "mixin" approach to multiple inheritance (see, for example, [Moon 1986; [Bracha and Cook 1990]). It may also be compared to the use of "friend classes" in C++ [Stroustrup 1986].

The dependencies in the *Generic_Star_Tracker* class above are all one-to-one dependencies. Objects in the *Estimator* category, however, have a one-to-many dependency on objects in the *Sensor* category (see Figure 1). Thus, this dependency must be implemented as a generic formal array of *Sensor* objects. For example, consider a simplified class package for the *Sequential Estimator* class:

```
generic
  -- domain language layer dependencies
  ...
package Generic_Sequential_Estimator is

  type ADT is private;

  -- primitive operations
  ...

  -- non-primitive operations
  generic
    type SENSOR is private;
    with function Measurement (S : SENSOR)
      return VECTOR3;

    type SENSOR_INDEX is (<>);
    type SENSOR_ARRAY is
      array (SENSOR_INDEX range <>) of SENSOR;
    The_Sensors : SENSOR_ARRAY;

  package Model is
    procedure Initialize (E : in out ADT);
    procedure Advance (E : in out ADT);
    procedure Update (E : in out ADT);
  end Model;

private
  ...
end Generic_Sequential_Estimator;
```

One-to-many dependencies require the addition of a generic formal array type and a generic formal array index type to the formal part of a nested generic. The *Sequential Estimator* operations defined in the nested generic can then loop through the array index. For instance, the *Update* operation calls the *Measurement* operation on each sensor in the array *The_Sensors* to gather the vectors needed by the estimation algorithm. Note that the fact that the specification indicated a dependency on a category rather than a specific class is irrelevant once the dependency is parameterized.

As the above two examples show, Ada packaging provides certain advantages over typical object-oriented class constructs, with powerful faculties for parameterizing and grouping the definition of an abstract data type, associated concrete data types, and increments of extended functionality. However, Ada does not provide any support for superclasses, and thus it is not possible to implement categories in a generalized way in Ada. Thus rather than implementing categories as generalized components, they must be constructed as part of the configuration process, as discussed in Section 3.3. (Note that various constructs proposed for Ada 9X language revision should alleviate this situation [Ada 9X 1991]).

3.2 Module Packages

For each class package defined in the domain object layer, there are one or more *module packages* defined in the architecture layer. Module packages serve two purposes. The first is to provide for the definition of a specific set of objects in a configured application program. The second purpose is to provide the capabilities needed to use the generalized objects within a specific system architecture. For example, the COMPASS architecture calls for operation parameters to be passed from the COMPASS user interface to the application program as a block of text. The *Star_Tracker_Module* is responsible for providing the subprograms needed to convert this text to data of the appropriate *PARAMETERS* type and passing the data to the proper *Star Tracker* object. The remainder of this subsection will focus on the first purpose.

Class packages define an abstract data type which may be used to define an arbitrary number of objects. In contrast, a module package is an abstract state machine which maintains only a *fixed number* of objects of a given class as required in a specific application program. These objects are indexed by an enumeration type given as a generic formal parameter. The module package then reexports functionality from the class package in terms of this enumeration type. Each module package defined for a class package provides the functionality obtained by instantiating a specific subset of the nested generics.

For example, the *Generic_Star_Tracker* class package described in Section 3.1 has two nested generics. Thus, there may be as many as four module packages associated with this class package, corresponding to instantiating neither nested generic, either one, or both. Each of these module packages must itself have as generic parameters all information necessary to instantiate the appropriate nested generics.

The following *Star Tracker* module package provides the functionality obtained by instantiating the nested generic package *Model* but not the nested generic procedure *Output*:

```
generic
  type TRACKER_NAME is (<>);

  type STAR_MAP is private;
  type STARS_IN_VIEW is
    array (POSITIVE range <>) of VECTOR3;
  with function Stars_In_Field_Of_View (
    From_Map       : STAR_MAP;
    Around_Vector  : VECTOR3;
    Cone_Angle     : REAL)
    return STARS_IN_VIEW;

  The_Map : STAR_MAP;

package Star_Tracker_Module is

  procedure Set_Parameters (
    S              : in TRACKER_NAME;
    To_The_Values  : in PARAMETER_VALUES);

  -- constructors
  procedure Initialize (S : in TRACKER_NAME);
  procedure Power_On (S : in TRACKER_NAME);
  ...
  procedure Simulate (
    S              : in TRACKER_NAME;
    At_Time        : in TIME);

  -- selector
  function Measurement (S : TRACKER_NAME)
    return VECTOR3;

end Star_Tracker_Module;
```

The generic formal part for this module package includes the *Star Map* dependency required by *Model*. It also declares operations that correspond to *Model* operations (*Initialize* and *Simulate*) as well as primitive abstract data type operations.

A module package is implemented with a state that is defined in terms of the abstract data type from a class package. This means that the module package must use an *instantiation* of the class package. In COMPASS, these instantiations are compiled into a library of generalized components, with the actual parameters being provided by components in the domain language layer. For example, the instantiation of *Generic_Star_Tracker* is:

```
with Generic_Star_Tracker, Calendar, Linear_Algebra;
use Calendar, Linear_Algebra;
package Star_Tracker is new Generic_Star_Tracker
  (TIME, REAL, VECTOR3, MATRIX33);
```

The domain language layer packages *Calendar* and *Linear_Algebra* export the subprograms needed to instantiate *Generic_Star_Tracker*. The use of "is <>" notation frees the human developer from writing the whole tedious list of utility functions.

The instantiated package *Star_Tracker* is imported into the body of the *Star_Tracker_Module*. The appropriate nested generics are then instantiated within this body:

```
with Star_Tracker;
package body Star_Tracker_Module is

  type MODULE_STATE is
    array (TRACKER_NAME) of Star_Tracker.ADT;
  State : MODULE_STATE;

  package Model is new Star_Tracker.Model
    (STAR_MAP, STARS_IN_VIEW,
      Stars_In_Field_Of_View, The_Map);

  procedure Initialize (S : in TRACKER_NAME) is
  begin
    Model.Initialize (State(S));
  end Initialize;
  ...
  procedure Break_Track ( S : in TRACKER_NAME) is
  begin
    Star_Tracker.Break_Track (State(S));
  end Break_Track;
  ...
end Star_Tracker_Module;
```

The state of the *Star_Tracker_Module* is defined as an array of star tracker objects, as defined by the type *Star_TrackerADT*. These data are accessed through calls to module operations that reexport the operations defined on *Star_TrackerADT*. A module subprogram such as *Break_Track* has the object name *S* as an input of the enumerated type *TRACKER_NAME*. This subprogram in turn calls *Star_TrackerBreak_Track* to act on object *State(S)*. The subprogram *Initialize*, on the other hand, calls *ModelInitialize*, using an instance of the nested package *Star_TrackerModel*. In either case, the actual data processing is performed by a subprogram from the class package, whether it is imported directly or is defined in a nested generic.

Note that the object name for the module operations is an "in" parameter, as it merely denotes the correct object. The object sent to a constructor operation from the class package, however, will have its state updated. Thus the corresponding class procedure takes the object as an "in out" parameter.

3.3 Configuration

To build an application program, one must first define the objects used, then instantiate the modules associated with these objects, then configure a driver to communicate with the user interface. This subsection focuses on the first two steps.

All the objects required in a configured application program are defined in a single enumeration type. The literals of this type name each object used in the application program. The class/category hierarchy is reflected in appropriate definitions of subtypes of this enumeration type.

For example, the following package defines the objects required to configure a simple estimation program with two sun sensors and one star tracker:

```
package Mission_Objects is

  type OBJECTS is (
    -- data read by estimator
    MEASUREMENT,
    -- dynamic model used in estimation
    DYNAMICS,
    -- models used by sensors
    SUN, STARS,
    -- sun sensor models
    SUN_SENSOR1, SUN_SENSOR2,
    -- star tracker model
    TRACKER,
    -- sequential estimator
    ESTIMATOR );

  -- Sensor category
  subtype SENSORS is
    OBJECTS range SUN_SENSOR1..TRACKER;
  type SENSOR_LIST is
    array (POSITIVE range <>) of SENSORS;

  -- Sun Sensor class
  subtype SUN_SENSORS is
    SENSORS range SUN_SENSOR1..SUN_SENSOR2;
  type SUN_SENSOR_LIST is
    array (POSITIVE range <>) of SUN_SENSORS;

  -- Star Tracker class
  subtype STAR_TRACKERS is
    SENSORS range TRACKER..TRACKER;
  type STAR_TRACKER_LIST is
    array (POSITIVE range <>) of STAR_TRACKERS;

end Mission_Objects;
```

In this package, the subtype *SENSORS* includes all the objects that are in the *Sensor* category, and the subtypes *SUN_SENSORS* and *STAR_TRACKERS* include the sensors in those respective classes.

The objects defined in *Mission_Objects* are used to instantiate the appropriate module packages. These instantiations provide the functionality required for each object. For example, the following is the instantiation of *Star Map* functionality for object *STARS*:

```
with Star_Map_Module;
with Mission_Objects; use Mission_Objects;
package Mission_Stars is
  new Star_Map_Module (Catalog_Name => STARS);
```

This package is in turn used to instantiate the *Star Tracker* module package:

```
with Star_Tracker_Module, Mission_Stars;
with Mission_Objects; use Mission_Objects;
package Mission_Star_Trackers is
  new Star_Tracker_Module (
    TRACKER_NAME   => STAR_TRACKERS,
    SKYMAP         => OBJECTS,
    The_Map        => STARS,
    STARS_IN_VIEW  => Mission_Stars.STAR_LIST,
    Stars_In_Field_Of_View
      => Mission_Stars.Stars_In_Field_Of_View);
```

The library instantiation *Mission_Sun_Sensor* is created in a similar way.

The sensor packages defined above are needed to instantiate the *Sequential_Estimator_Module*. However, both the sensor packages export *different Measurement* selectors. *Mission_Sun_Sensor.Measurement* must be used for *SUN_SENSOR1* and *SUN_SENSOR2*, while *Mission_Star_Tracker.Measurement* must be used for *TRACKER*.

Now, we would like to be able to write the instantiation for a *Sequential Estimator* in terms of a *Mission_Sensor_Category* package which defines a *Measurement* selector that works for all sensors:

```
with  Sequential_Estimator_Module,
   Mission_Sensor_Category;
with Mission_Objects; use Mission_Objects;
package Mission_Estimator is
   new Sequential_Estimator_Module (
      SENSOR             => SENSORS,
      SENSOR_INDEX  => POSITIVE,
      SENSOR_ARRAY => SENSOR_LIST,
      The_Sensors
         => (SUN_SENSOR1, TRACKER),
      Measurement
         => Mission_Sensor_Category.Measurement );
```

This package uses one sun sensor and one star tracker to provide two vector measurements. The estimator may use these measurements to generate estimates of, e.g, the spacecraft attitude.

Unfortunately, as discussed in Section 3.1, there is no generalized component from which *Mission_Sensor_Category* may be instantiated. Thus, we must *create* a mission-specific category package whose specification declares the common operations of the category, and whose body routes calls to modules of the appropriate class. Thus the implementation of the *Sensor* category for our estimator has the following package specification:

```
with Calendar, Linear_Algebra, Mission_Objects;
use Calendar, Linear_Algebra;
package Mission_Sensor_Category is

   subtype SENSOR is Mission_Objects.SENSORS;

   -- constructors
   procedure Initialize (S : in SENSOR);
   procedure Power_On (S : in SENSOR);
   procedure Power_Off (S : in SENSOR);
   procedure Simulate (
      S            : in SENSOR;
      At_Time   : in TIME);

   -- selector
   function Measurement (S : SENSOR)
      return VECTOR3;

end Mission_Sensor_Category;
```

The body of this category package imports all instantiations of module packages for objects in the category. Each subprogram in the package then has a case statement that routes calls to the corresponding subprogram in the appropriate module instantiation:

```
-- module instantiations
with Mission_Sun_Sensors, Mission_Star_Trackers;
--
package body Mission_Sensor_Category is
   ...

   function Measurement (S : SENSOR)
      return VECTOR3 is
   begin

      case S is
         when Mission_Objects.SUN_SENSORS =>
            return Mission_Sun_Sensors.Measurement(S);
         when Mission_Objects.STAR_TRACKERS =>
            return Mission_Star_Trackers.Measurement(S);
      end case;

   end Measurement;

end Mission_Sensor_Category;
```

In effect, the branch on mission-specific sensor type which might appear in the estimator in a monolithic, multi-mission support program has been instead moved outside of the generalized estimator and localized in the mission-specific *Mission_Sensor_Category* package.

Note that the interface provided by a category package is similar to that provided by a generic module package. In fact, the package specification is quite general except for the dependence on the mission-specific package *Mission_Objects*. However, the category package cannot simply be made generic, as the package body depends on the particular module package instantiations used for a given program.

It is possible, though, to provide a template for a category package that includes much of the code for the package. This template then includes slots for, e.g., the name of the object package and the name of the appropriate enumeration type, which must be filled in by editing the text, rather than through generic parameters. This approach is actually consistent and straightforward enough that it should be possible to automate the tedious process of generating such packages.

Actuallyr, our approach to classes and categories does seem to have some advantages of its own. The typical object-oriented implementation results in a tight coupling between classes and superclasses, defeating our goal of individually reusable parameterized components. In fact, the unfortunately common misuse of the inheritance features of object-oriented languages can result in superclass hierarchies which actually decrease reusability and increase maintenance costs (see, for example, [Wild 1990]).

In contrast, the COMPASS implementation concepts reflect a *constructive* approach to the implementation of superclasses. Our approach only couples the *instantiations* of components during system configuration, though this does require the generation of mission-specific category packages, which act as "glue" during this interconnection. Our approach also has trouble elegantly handling the implementation of common superclass functionality and rich, deep superclass hierarchies. However, we do not expect this to be a problem for COMPASS, since we have found the *specification* of common, generalized interfaces to be significantly more important than small scale code sharing during development. Further, we have also found that limiting the domain specifications to only two levels of classification (class and category) actually makes these specifications clearer and easier for our mission analysts to use.

4. CONCLUSION

Both the specification and implementation concepts for COMPASS are firmly based on our experiences to date with Ada and object-oriented methodology. These concepts then evolved through actual application on the COMPASS project. A considerable amount of actual domain analysis has already been carried out using the COMPASS specification concepts over the last year and a half. The implementation concepts have only been applied so far to a small amount of prototype code, but are based on considerable experience with generic simulation components. We plan to begin implementation of production COMPASS code late in 1992 with development complete in 1997.

During the evolution of the COMPASS concepts there has been a continuing tension between the need to provide powerful conceptual tools and the desire to keep the approach as simple as possible. If the concepts are not powerful enough, then it becomes prohibitively convoluted to specify and implement a parameterized system. On the other hand, if the approach is too complicated, it will be difficult to learn and apply.

The result of this tension has been a conscious decision on our part to adopt only those features of the object-oriented approach which seem crucial for our purposes. Thus, as discussed at various points earlier, our approach perhaps does not have the full generality of some object-oriented methodologies. On the other hand, what has been incorporated into our concepts is the result of real-world tradeoffs in our application domain and is tailored specifically for parameterized systems. Thus we feel that the concepts we are developing for COMPASS can provide a comprehensive approach to specifying and implementing parameterized systems in Ada. And even though our approach is still evolving, it already provides a firm basis for development of an architecture of parameterized systems in general.

REFERENCES

Ada 9X 1991 Ada 9X Project Report, *Draft Mapping Document*, Office of the Under Secretary of Defense for Acquisition, February, 1991

Booth and Stark 1989 E. Booth and M. Stark, "Using Ada Generics to Maximize Verbatim Software Reuse", *Proceedings of Tri-Ada '89*, November 1989

Booth and Stark 1991 E. Booth and M. Stark, *COMPASS Implementation Concepts*, Goddard Space Flight Center Flight Dynamics Division, 550-COMPASS-105 (Draft), February 1991

Bracha and Cook 1990 G. S. Bracha and W. Cook, "Mixin-based Inheritance", *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications / European Conference on Object-Oriented Programming*, SIGPLAN Notices, October, 1990

CAMP 1990 Common Ada Missile Packages, *Developing and Using Ada Parts in Real-Time Embedded Applications*, McDonnell Douglas Missile Systems Company, April 1990

Coad and Yourdon 1991 P. Coad and E. Yourdon, *Object-Oriented Analysis*, Yourdon Press, 1991

DeFazio et al. 1991 R. DeFazio., G. Meyers, C. Newman, R. Pajerski, K. Peters, E. Seidewitz and W. Weston, *COMPASS High-Level Requirements, Architecture, and Operation Concepts*, Goddard Space Flight Center, Flight Dynamics Division, 550-COMPASS-102, April 1991

Moon 1986 D. A. Moon, "Object-Oriented Programming with Flavors", *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, SIGPLAN Notices, November, 1986

Prieto-Diaz 1990 R. Prieto-Diaz, "Domain Analysis: An Introduction", *Software Engineering Notes*, April 1990

Seidewitz 1987 Seidewitz, E., "Object-Oriented Programming in Smalltalk and Ada", *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, SIGPLAN Notices, December, 1987

Seidewitz 1991 E. Seidewitz, "Object-Oriented Programming through Type Extension in Ada 9X", *Ada Letters*, March/April 1991

Seidewitz et al. 1991 E. Seidewitz, R. Bakos and G. Klitsch, *COMPASS Specification Concepts*, Goddard Space Flight Center, Flight Dynamics Division, 550-COMPASS-103, April 1991

Shlaer and Mellor 1988 S. Shlaer and S. Mellor, *Object-Oriented Systems Analysis: Modeling the World in Data*, Prentice-Hall, 1988

Stark 1990 M. Stark, "On Designing Parameterized Systems Using Ada", *Proceedings of the Seventh Washington Ada Symposium*, June 1990

Stroustrop 1986 B. Stroustrop, *The C++ Programming Language*, Addison-Wesley, 1986

Wild 1990 F. Wild, "A Comparison of Experiences with the Maintenance of Object-Oriented Systems: Ada vs. C++", *Proceedings of Tri-Ada '90*, December 1990

DESIGNING CONFIGURABLE SOFTWARE: COMPASS IMPLEMENTATION CONCEPTS

Eric W. Booth, Computer Sciences Corporation
Michael E. Stark, NASA/Goddard Space Flight Center

INTRODUCTION

The Flight Dynamics Division (FDD) of NASA's Goddard Space Flight Center has employed object-oriented methods and the Ada language on spacecraft attitude simulators since 1985. While software reuse trends on the first three projects were promising (see Figure 1), the level of reuse was not significantly higher than that on recent FORTRAN simulators developed using structured analysis and design techniques. Further, any productivity or quality improvements were lost in the cost of technology transition.

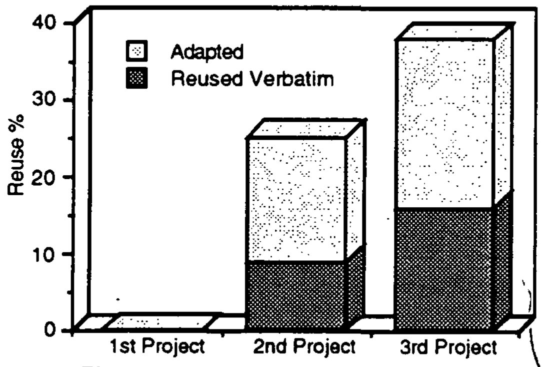


Figure 1. Initial Reuse Trends

The team developing the fourth Ada simulator project had the opportunity to use a different approach because another, very similar, system was known to be coming in the near future. With this in mind, the team designed the fourth simulator using a generic architecture that could be instantiated for the forthcoming simulator as well. The reusability effect of this generic architecture on the fifth *and* sixth Ada simulators is shown in Figure 2.

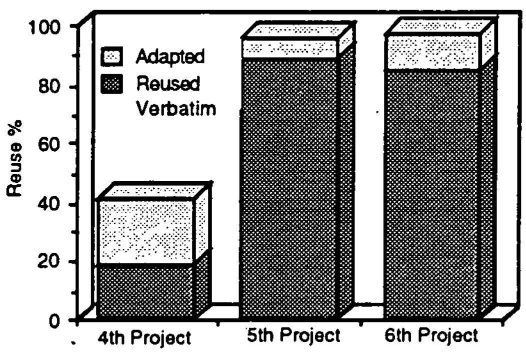


Figure 2. Impact of a Generic Architecture

The fifth simulator reused the entire generic architecture and achieved a reuse level of 90 percent. The sixth simulator achieved a still higher reuse level. Figure 3 shows the dramatic impact of the generic architecture on successive projects: the level of effort to develop a similar system has been reduced by more than 50 percent, the project duration has decreased by more than 60 percent, and the number of errors per delivered source instruction has been reduced by 90 percent [Booth and Luczak 1990, Groveman 1991].

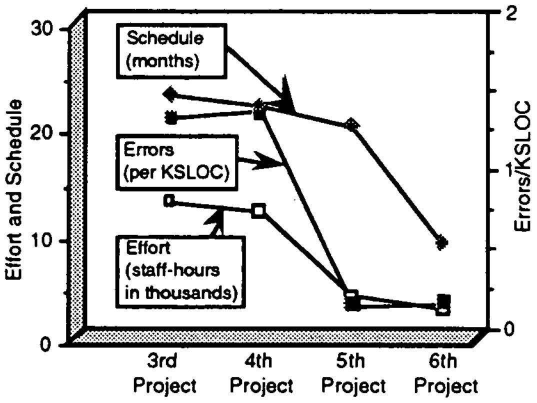


Figure 3. Benefits of a Generic Architecture

Because of the demonstrated success of using object-oriented methods and Ada to develop a generic architecture, the FDD is concentrating effort on capturing the most successful of these software reuse strategies and techniques and scaling them to a much larger software system.

This paper presents the high-level architectural approach taken by the FDD to specify and design this large software system: the Combined Operational Mission Planning and Attitude Support System (COMPASS).

COMPASS OVERVIEW

The specification and implementation approaches planned for COMPASS are based on an evolving object-oriented methodology in the FDD and on experience with reusable Ada software [Booth and Stark 1989]. However, COMPASS is considerably larger in size and scope that any other Ada system developed in the FDD. The Ada code to be developed for COMPASS is estimated to be on the order of 1 million lines (measured by terminal semicolons).

The high-level architecture for COMPASS [Defazio 1990] calls for the separation of the application software from the framework provided by the user interface/executive (UIX). The UIX includes the Operation Interface, from which the user runs an application; the Configuration Interface, from which applications are built; the Database services; and the Executive services. Furthermore, the application software is to be implemented using Ada, and the UIX is to be implemented using commercial standards such as Motif, TCP/IP, and UNIX. Figure 4 shows the relationship among the components of UIX framework and the components of the application software.

Application programs will be configured from a set of generalized components and executed on various hardware platforms in the FDD; the UIX processes will all execute on UNIX workstations. Application programs will communicate with each other and with the UIX through platform services. Application program interface (API) packages are provided as Ada interfaces to these platform services.

The implementation concepts for COMPASS [Booth and Stark 1991] define how the Ada application components are to be designed. These concepts are based on FDD's successful experiences with Ada, on research into large-scale software reuse, and on constraints imposed by the interrelated aspects of the COMPASS architecture. The most important of these constraints are that the implementation concepts must flow from the specification concepts for COMPASS [Seidewitz 1990] and that they must fit the architecture defined for the UIX [Seidewitz and Green 1990].

The specification concepts call for an object-oriented approach to producing generalized specifications. The specifications group the functionality and data associated with objects from the problem domain into *classes*. These classes can be physical objects such as a Sun sensor or abstract objects such as a numeric integrator. A collection of common classes represents a *category*. For example, all sensor classes provide the ability to produce a measurement and to

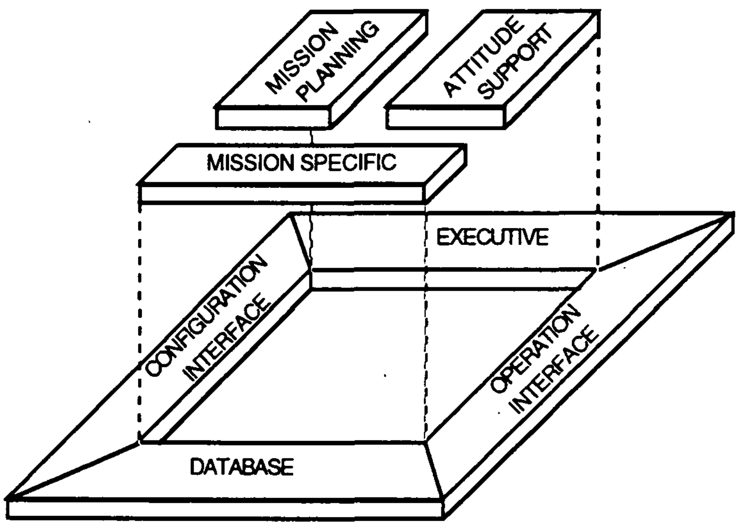


Figure 4. Application Software Within the Framework of the COMPASS Architecture

model the effects of sensor misalignment. Therefore, all sensors are specified within a sensor category. The category concept merely specifies the common interface to its member classes; the classes contain the implementation of data and methods.

A *subsystem* is an aggregation of related categories. For example, a flight-dynamics subsystem would contain categories for orbit dynamics and for force models. Subsystems can be configured into *application programs*. Subsystems include the specification of user actions such as "advance propagator to time t," and *drivers* that specify the interaction between the application program and the Operation Interface using a state transition model.

COMPASS IMPLEMENTATION CONCEPTS

The implementation concepts for COMPASS are based on the layered software reuse model defined in Stark 1990. Table 1 shows a further refinement of this model including examples specific to COMPASS.

Table 1. The Layered Software Reuse Model

LAYERS	LEVELS	EXAMPLES	PORTABILITY
Application	Application Architecture Components	UARS_Objects, UARS_Orbit_ Model_Category	
	Application Modules	Two_Body_ Orbit_Module	⇑
Problem Domain	Domain Classes	Two_Body_ Orbit_Class	PORTABLE
	Domain Language	Ephemeris_ Types, Linear_Algebra	LEVELS
Services	System-Independent Services	Interface Generics, Booch Components	PORTABILITY LEVEL
	System-Dependent Services	Application Program Interfaces (APIs)	NON-PORTABLE LEVEL ⇓

There are three major layers in this model:

- Application Layer
- Problem Domain Layer
- Services Layer

Each major layer is composed of two levels. Each level uses and builds on the functionality provided by lower levels; levels are not allowed to use (or in any way depend on) functionality provided in higher levels.

While this software reuse model is language independent, the generic unit feature of the Ada language provides the ability to parameterize component dependencies within and between levels. Components in the domain and service layers can therefore be used within different applications. Similarly, this approach allows domain objects and classes to be used with different sets of utility packages that provide domain-language functionality. Finally, all components in the portable layers can be used as a system on any platform that can fill in the required system-dependent services of the (lowest) non-portable level.

The next three sections of this paper describe the detailed implementation concepts associated with the three major layers, beginning with the lowest layer, services.

Services Layer

The services layer consists of two levels: the system-independent services level and the system-dependent services level. The system-independent services level contains the portable interface domain-independent utilities such as the Booch components [Booch 1987]. In the COMPASS concepts, the system dependent services are more interesting.

COMPASS applications access platform services through application program interfaces (APIs). These services are used to communicate with the Operation Interface for that system, and they are implemented as general-purpose C routines that access standard capabilities such as TCP/IP network communications software. The C APIs are imported into Ada using pragma Interface, and generic units tie strongly typed Ada to virtually untyped C.

Figure 5 shows these services components as a layered model. Use of the platform services is demonstrated by the example of passing orbit data (time, position, and velocity) to the Operation Interface for display.

Ada Platform Service Generic: Specification	PORTABILITY LEVEL
Ada Platform Service Generic: Body	NON-PORTABLE LEVELS ⇓
Ada API package specification pragma interface to C	
C API routines	

Figure 5. Implementation Concept for the Services Layer

The following is a partial specification for a generic data object:

```
with Communication_Services_Types;
generic
  type ITEM is private;
package Generic_Sequential_Transient_Data is
 type DATA_OBJECT is limited private;
  ...
  procedure Write (
   To_The_Object : in out DATA_OBJECT;
   Value : in ITEM);
  ...
private
  type DATA_OBJECT is record
       Socket
            : Communication Services_Types.SOCKET_ID;
       Time_Out_Interval
            : Communication_Services_Types. INTERVAL;
  end record;
  pragma Inline (Open, Close, Read, Write,...);
end Generic_Sequential_Transient_Data;
```

This package can be instantiated to allow an application program to transmit data to another process such as the Operation Interface. This package is similar to the predefined I/O packages in that, while its implementation is system dependent, its specification provides a system-independent visible part. Portable code can therefore be written using Generic_Sequential_Transient_Data, just as portable code can be written using package Direct_IO.

The private part of the package contains all of the system dependencies. The above example shows a package targeted to the 386/486 workstations that run SCO Unix. Therefore, it uses sockets for interprocess communications. The VAX version of this package could define type DATA_OBJECT in terms of VMS mailboxes, without changing the visible part.

The following shows the implementation of Generic_Sequential_Transient_Data:

```
with Communication_Services;
package body Generic_Sequential_Transient_Data is
  ...
  procedure Write (
     To_The_Object : in out DATA_OBJECT;
     Value : in ITEM) is
  use Communication_Services;
  begin
     Send_Data_Over_Socket (
       Socket_Identifier => To_The_Object.SOCKET,
       Data_Buffer => Value'ADDRESS,
       Length_Of_Buffer => Value'SIZE / 8,
       Time_Out_Interval
            => To_The_Object.Time_Out_Interval);
  end Write:
end Generic_Sequential_Transient_Data;
```

The write operation simply calls the Send_Data_Over_Socket procedure exported by the Ada API package Communication_Services. The Socket_Identifier and Time_Out_Interval come directly from To_The_Objects, while the address and amount of data come from the attributes of Value, which is the data being written. The following example shows the Ada API package; the pragma Interface_ Name is specific to the Alsys compiler under SCO Unix.

```
with System, Communication_Services_Types;
use Communication_Services_Types;
package Communication_Services is
  ...
  function Send_Data_Over_Socket (
     Socket_Identifier    : SOCKET_ID;
     Data_Buffer          : System.ADDRESS;
     Length_Of_Buffer     : NATURAL;
     Time_Out_Interval    : INTERVAL)
     return INTEGER;
  ...
private
 -- Alsys Ada interface pragmas to (routine S_SEND)
 pragma Interface (C, Send_Data_Over_Socket);
 pragma.Interface_Name
  (Send_Data_Over_Socket, "S_SEND");
  ...
end Communication_Services;
```

It is possible, but not straightforward, to write a single API package for interprocess communication in which the visible part would provide the operations read, write, open, and close and the hidden part would contain the data representation and interfaces. We chose the layered model to simplify the application programmer's use of the APIs. Note that the programmer need not remember that type ADDRESS is defined in package System, or that the 'SIZE attribute is measured in bits; the generic package hides these factors. The use of pragma Inline in package Generic_Sequential_Transient_Data guarantees that the extra layer of components does not affect performance.

This design of the platform services layer should be more easily maintained because the interface to the C code is contained in one package and the view presented to the higher levels is in a separate generic package.

Problem Domain Layer

The problem domain layer of the software reuse model (see Table 1) defines classes relevant to the problem domain and is divided into two levels: the domain language level and the domain classes level. The domain language level provides the domain-specific language, or utilities, to the domain classes level. The domain classes level defines the classes of actual objects from the problem domain. Moving through the reuse model from the lowest level to the higher levels highlights the recurring pattern of narrowing the domain while building on the lower levels.

Domain Language Level

The domain language level is implemented as a set of interdependent library instantiations of generic utility packages that export the types and operations needed to implement problem domain classes. The domain language level is divided into several sublevels with the lower levels supporting a wider range of applications and the upper levels being tailored to a narrower, more specific domain. Figure 6 shows the three major domain language sublevels.

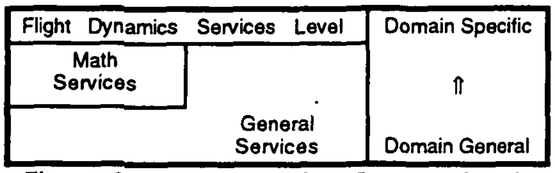


Figure 6. Implementation Concept for the Domain Language Level

The general services level contains packages such as Booch's String_Utilities and Floating_Point_Utilities. The math services level contains trigonometric functions, linear algebra, and a random-number generator. The flight dynamics services level contains packages exporting various representations of spacecraft orbit and attitude.

Packages from the general services and math services levels are reusable outside the flight dynamics domain because they are independent of the flight dynamics services level. However, the converse is not true; orbit types and operations in the flight dynamics services level depend on vector types and operations in the math services layer.

The generic formal parts of the Generic_Linear_Algebra and Generic_Math_Functions packages are shown in the following example:

```
generic
  type SCALAR is digits <>;
  Pi : in SCALAR
      := 4.14159_26535_89793_23846_26433;
  E : in SCALAR
      := 2.71828_18284_59045_23536_02874;
package Generic_Math_Functions is
  ...
end Generic_Math_Functions;

generic
  type REAL is digits <>;
  with function Sqrt (X : in REAL)
      return REAL is <>;
package Generic_Linear_Algebra is
  ...
end Generic_Linear_Algebra ;
```

The following example shows the interdependent library unit instantiations of these packages:

```
package Numeric_Types is
  type REAL is digits 15;
  ...
end Numeric_Types;

with Numeric_Types, Generic_Math_Functions;
package Real_Math_Functions is
  new Generic_Math_Functions
  (SCALAR => Numeric_Types.REAL);

with Numeric_Types, Generic_Linear_Algebra;
with Real_Math_Functions;
package Real_Linear_Algebra is
  new Generic_Linear_Algebra
  (  SCALAR      => Numeric_Types.REAL
     Sqrt        => Real_Math_Functions.Sqrt);
```

The advantage of this design technique is that the generic packages are independent implementations. The linear algebra package does not depend on this particular mathematical functions package; it only requires a real data type and a square root function. While the generic implementations are independent, the instantiations are *interdependent*. This means that the domain language packages need to be designed *as a set* to ensure that all generic units are compatible. The alternate design is to nest an instantiation of Generic_Math_Functions within the body of Generic_Linear_Algebra. However, this creates a dependency on a specific math package, reducing the generality of Generic_Linear_Algebra.

Another consideration in designing domain language level components is the use of exported types and operations in higher levels. In the domain objects level, the necessary types and operations are imported via generic formal parameters. The domain language level exports a mix of scalar, private, array, and record types. For scalar types, standard formal parameters such as "**type** X **is digits** <>" are sufficient. For private types, the generic formal parameter must also be a private type.

Array types can also be used as generic parameters. The only difficulty is that generic discrete types cannot be constrained to a fixed length, and there are instances where this would be useful. For example, COMPASS has a linear algebra package exporting 3-vectors and 3x3 matrices. To import the indices for these arrays, the package Index_Types is defined, and all packages using 3-vectors and 3x3 matrices must import Index_Types to get array indices. This introduces a modest but still undesirable coupling of generic class packages to a specific index types package.

Record types cannot be generic formal parameters, but they can be imported either as generic private types with subprograms to access record elements or by "withing" the package exporting the type. Again, using a generic formal private type parameter defers the coupling until instantiation at the cost of generating additional data access subprograms, both in the exporting package specification and the importing generic formal part. The tradeoff between these two approaches is being analyzed through prototyping.

Domain Classes Level

The domain classes level of the problem domain layer is implemented as a set of completely independent generic library unit packages that define the classes that correspond to actual objects in the problem domain. For example, a Sun sensor is a common object in the flight dynamics domain. The set of all Sun sensors is grouped together into the Sun-sensor class and implemented in Ada as a generic abstract data type (ADT) package. These library unit generic packages export various nested generic units as well as the ADT. This allows different instances (i.e., uses) of the class in different applications within the flight dynamics domain. The following partial specification provides a concrete example of how a Sun-sensor class might be implemented:

```
generic
  type TIME is private;
  type RADIANS is digits <>;
  type REAL is digits <>;
  type INDEX is range <>;
  type VECTOR is array (INDEX) of REAL;
  type MATRIX is array (INDEX, INDEX) of REAL;
  with function "*" (M : in MATRIX; V : in VECTOR)
      return VECTOR is <>;
  ...
package Generic_Sun_Sensor_Class is
  type ADT is private;
  type PARAMETERS is record
      Azimuth      : RADIANS;
      Elevation    : RADIANS;
        ...
  end record;
procedure Set_Parameters
  ( S : in out ADT; To : in PARAMETERS );
generic
  with function Body_to_Sensor_Frame
      (At_Time : in TIME) return MATRIX is <>;
  ...
package Model is
  procedure Initialize
      ( S : in out ADT; At_Time : in TIME );
  procedure Simulate
      ( S : in out ADT; At_Time : in TIME);
  ...
end Model;

private
  type ADT is record
      Enabled          : BOOLEAN;
      Power            : BOOLEAN;
      ...
      The_Parameters   : PARAMETERS;
  end record;
end Generic_Sun_Sensor_Class;
```

The design for each generic class package calls for three basic sections: the generic formal parameters section, the visible section, and the private section. The generic formal parameters section contains the generic formal type parameters necessary for the type declaration of the initial parameters (defined in the visible section), and common formal subprogram parameters. For COMPASS, this section includes standard types and subprogram parameters such as a real vector and matrix type and trigonometric functions.

The visible section contains a private type, a parameters type, and one or more nested generic package declarations. The operations parameter type is declared in the visible section because it is needed by other packages that interface with the user and handle input/output. The private section, of course, contains the complete definition of the ADT structure used throughout the generic class.

The library unit generic class package is instantiated in the domain classes level using the packages provided by the domain language level. However, the nested generic packages are left for the next highest level to instantiate as needed because the necessary instances depend on the type of application.

Just as the generic packages in the domain language classes level were required to be compatible, the generic packages in the domain classes level must be compatible with the lower levels of the reuse model. To ensure this, the problem domain layer must be specified, designed, and implemented as a set of compatible generic packages. That is, overall domain class architecture must be defined using domain analysis [Seidewitz and Stark 1991] and followed throughout the development.

In addition, the generic formal parameters for the Generic_Sun_Sensor_Class package are compatible with the types and subprograms from the domain language level. The instantiation of the Generic_Sun_Sensor_Class library unit might look something like that shown in the following example:

```
with Generic_Sun_Sensor
      Calendar, Numeric_Types,
      Index_Types, Linear_Algebra;
package Sun_Sensor_Class is
  new Generic_Sun_Sensor_Class (
      TIME         => Calendar.TIME;
      RADIANS      => Numeric_Types.RADIANS;
      REAL         => Numeric_Types.REAL;
      INDEX        => Index_Types.INDEX3;
      VECTOR       => Linear_Algebra.VECTOR3;
      MATRIX       => Linear_Algebra.MATRIX33;
      "*"          => Linear_Algebra."*" );
```

Note that the instantiation would not be compatible if either the VECTOR or MATRIX generic formal parameters were declared as unconstrained arrays. For example, assume that the generic formal array types were declared as follows:

```
  type VECTOR is array
    (INDEX range <>) of REAL;
  type MATRIX is array
    (INDEX range <>, INDEX range <>) of REAL;
```

The instantiation would not be valid with an actual array type parameter that was constrained. While this is a simplified example, the implication is that compatibility among generic units must exist *between* levels as well as *within* each level. The same implication applies when going from the domain classes layer to the application layer.

In general, incompatibility among a hybrid collection of generic units is often the reason that supposedly reusable components are not reused within the context of a system architecture. The use of consistent object-oriented specification and implementation concepts eliminates this problem in COMPASS.

Application Layer

As in the two lower layers, the application layer is composed of two separate levels: the application modules level and the application architecture components level.

Application Modules Level

The problem domain layer defines the classes for the domain but does not specify how or where the generic actual parameters of these classes are declared. The application modules level contains generic abstract state machines (ASMs), each defining an array of an ADT and declaring a state variable of that array type in the body. An ASM as described here is referred to as a *module* in COMPASS.

The following example shows the specification of the Generic_Sun_Sensor_Module. Notice that the module is divided into two basic sections, the generic formal parameters section and the visible section.

```
generic
  -- Module formal parameters
  type SENSOR_NAME is (<>);

  type COSINE_MATRIX is private;
  type ATTITUDE_DYNAMICS is (<>);
  with function Current_Attitude
      (A : in ATTITUDE_DYNAMICS; T : in TIME )
      return COSINE_MATRIX is <>;
  with function Sun_Position_GCI
     (At_Time : in TIME )
     return COSINE_MATRIX is <>;

package Generic_Sun_Sensor_Module is
   -- constructors
   procedure Initialize (
      The_Sensor : in SENSOR_NAME;
      At_Time : in TIME);
   procedure Simulate
      The_Sensor : in SENSOR_NAME;
      At_Time : in TIME);
   -- selectors
   function Measurement
      The_Sensor : in SENSOR_NAME;
      At_Time : in TIME);
      return SENSOR_MEASUREMENT;

end Generic_Sun_Sensor_Module;
```

The module's generic formal parameters section contains two kinds of generic formal parameters: module-specific parameters and formal parameters needed to instantiate the generic packages nested in the class package. The module-specific parameters will minimally include an enumerated type parameter that is used to name each object contained by the module (type SENSOR_NAME in this example).

The rest of the module's visible section contains the constructors and selectors that correspond to the class. While these appear very similar to the generic class, there is one subtle and important difference: the *names* of the objects are passed to the constructors and selectors in the module. The actual data is passed by the module to the instantiation of the generic class. This distinction becomes clearer in view of the structure of the module's body.

The following example shows the structure of a typical module body. The module body contains an array type definition, an array variable declaration, and, of course, subprogram bodies.

```
with Sun_Sensor_Class;
package body Generic_Sun_Sensor_Module is
  --Instantiation of nested generic Model
  package Model is new Sun_Sensor_Class.Model
        (COSINE_MATRIX, ATTITUDE_DYNAMICS);

  type SENSOR_ARRAY is   --STATE DATA:
     array (SENSOR_NAME)
     of Sun_Sensor_Class.ADT;

   State : SENSOR_ARRAY;    --An array of sensors
                            --indexed by names
  -- constructors
  procedure Initialize (
    The_Sensor : in SENSOR_NAME;    --Name is passed
    At_Time : in TIME ) is          --into the module
  begin                             --and used to ref
     Model.Initialize               --the state and
       (State(The_Sensor), At_Time);  --call the Model.
  end Initialize;
  procedure Simulate (  is
    The_Sensor : in SENSOR_NAME;
    At_Time : in TIME ) is
  begin
     Model.Simulate(State(The_Sensor), At_Time);
  end Simulate;

  -- selectors
  function Measurement (
    The_Sensor : in SENSOR_NAME;   --Name is passed
    At_Time : in TIME )            --into the module
    return SENSOR_MEASUREMENT is
  begin                            --and used to ref
     return Model.Measurement      --the state and
       (State(The_Sensor), At_Time);  --call the Model.
  end Measurement;
end Generic_Sun_Sensor_Module;
```

Each module depends on a corresponding instantiation of the generic class, which imposes a hard coupling between the module and the generic class. That is, this dependency is not a parameter; it is part of the overall COMPASS architecture.

The module body imports the instantiation of the the Generic_Sun_Sensor_Class package, which exports the Model package and ADT. The module state data is an array of ADT parameters indexed by name; the names are supplied as an enumeration type to the module at instantiation. The ADTs are provided by the instantiation of the generic class. For example, if there were two Sun sensors on a particular spacecraft, access to the current state is possible only within the module body. This is shown as State(The_Sensor) in the subprograms in the module body.

Different application programs within the same domain might use different module generics that each use the same class instantiation. The application modules level, therefore, specifies one or more modules per class instantiation, depending upon the number of different application programs. For a given application program in the domain (e.g., a simulator application in the flight dynamics domain) the application module level would define modules specific to the application but generic with respect to any given mission. The next level defines the mission-specific parameters to complete the instantiation of a given application program.

Application Architecture Components Level

The top-level components of a COMPASS application are contained in the application architecture components level. This level includes the mission-specific packages, such as the module instantiations, a mission objects package, category packages, and drivers.

Instantiation of a module is the first step toward configuring COMPASS software to a specific mission. The following example shows how a particular module might be instantiated:

```
with Generic_Sun_Sensor_Module,
   Mission_Objects,
   Attitude_Module;
package Sun_Sensor_Module is
   new Generic_Sun_Sensor_Module (
   SENSOR_NAME => Mission_Objects.SENSORS,
   ATTITUDE_DYNAMICS =>
        Mission_Objects.ATTITUDE_DYNAMICS,
   Current_Attitude => Attitude_Module.GCI_to_BCS );
```

Notice that mission-specific packages are introduced with context clauses. The following sample mission-specific package, Mission_Objects, describes the objects, classes, and categories used for a given mission in terms of an enumeration type and its subtypes. This particular package is used for module instantiation as well as for category packages in the implementation concepts. Similar missions may be able to share (reuse) some of these module instantiations, assuming that the mission characteristics required by the instantiations are identical.

```
package Mission_Objects is
  type OBJECTS is (
    ATTITUDE,
    SUN_SENSOR1, SUN_SENSOR2,
    EARTH_SENSOR1, EARTH_SENSOR2);
  subtype ATTITUDE_DYNAMICS is
    OBJECTS range ATTITUDE..ATTITUDE;
  subtype SENSORS is                      -- category
    OBJECTS range SUN_SENSOR1..EARTH_SENSOR2;
  subtype SUN_SENSORS is                  -- class
    SENSORS range SUN_SENSOR1..SUN_SENSOR2;
  subtype EARTH_SENSORS is                -- class
    SENSORS range EARTH_SENSOR1..EARTH_SENSOR2;
end Mission_Objects;
```

The dependency diagram for this particular module instantiation may be represented as shown in Figure 7. The mission-specific components of the diagram are all located in the application architecture components level.

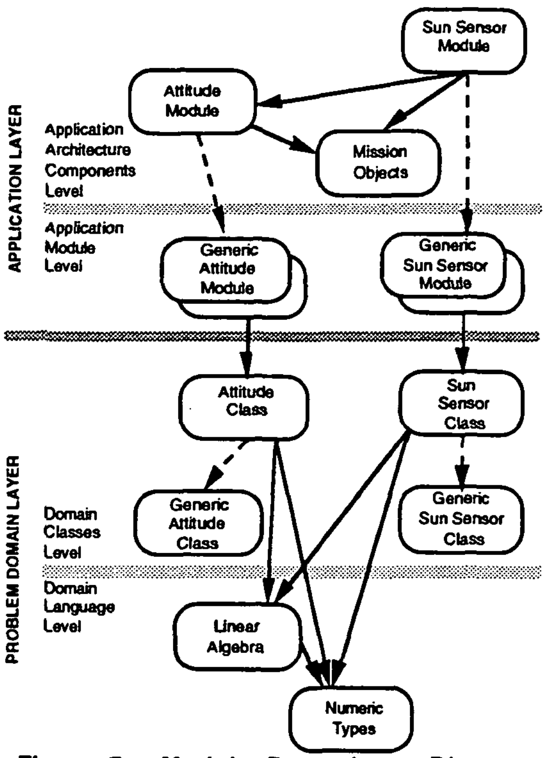


Figure 7. Module Dependency Diagram

A category is implemented as a nongeneric package that groups, by way of context clauses, a set of similar modules defining a common interface across member classes. For example, the Sun_Sensor_Module might be "withed" into a Sensor_Category package. The Sensor_Category package could also depend on such modules as Star_Tracker_Module or Earth_Sensor_Module. Member classes of a category will be mission specific because the context clauses are static, rather than dynamic, parameters. However, standard category packages may be defined that limit the changes to the category package body.

The package specification of a category package defines a common subprogram interface for its member classes. For example, all sensor classes must have a simulate function subprogram, but that subprogram would be implemented for the Sun-sensor class differently than for the Earth-sensor class. The following example shows a sample sensor category specification. Note that this specification may be reusable across multiple missions.

```
with Linear_Algebra, Calendar, Mission_Objects;
package Mission_Sensor_Category is
  subtype SENSORS is
    Mission_Objects.SENSORS;
  subtype TIME is Calendar.TIME;
  subtype VECTOR3 is Linear_Algebra.VECTOR3;
  --category constructors
  procedure Initialize
    ( A : in SENSORS; At_Time : in TIME );
  procedure Simulate
    ( A : in SENSORS; At_Time : in TIME );
  --category selectors
  function Measurement
    ( A : in SENSORS; At_Time : in TIME )
    return VECTOR3;
end Mission_Sensor_Category;
```

The category specification defines all subprograms common to each class in that category. However, a class will sometimes have functionality that is not readily generalized to other classes in the category. For example, the Star_Tracker class may have a procedure Break_Track that would not readily generalize to other Sensor classes. Therefore, while each class must implement *at least* all subprograms called out for its category, it may also implement additional subprograms unique to that class.

The following example shows the corresponding category package body. Notice that this particular mission has two kinds of sensor, Sun and Earth.

```
-- Actual module dependencies are shown here
with Sun_Sensor_Module;
with Earth_Sensor_Module;
package body Mission_Sensor_Category is
  procedure Initialize
      ( A : in SENSORS;
      At_Time : in TIME) is
  begin
    case A is
      when SUN_SENSORS =>
        Sun_Sensor_Module.Initialize (A, At_Time);
      when EARTH_SENSORS =>
        Earth_Sensor_Module.Initialize (A, At_Time);
    end case;
  end Initialize;
  ...
end Mission_Sensor_Category;
```

The Sun_Sensor_Module is architecturally identical to the Earth_Sensor_Module. However, the Sun_Sensor_Module implementation is different. The category package contains only subprograms common to both modules. A category subprogram calls the module subprogram for the appropriate subtype range that contains the sensor name. The actuator names and subtype range declarations are defined in the Mission_Objects package.

Figure 8 shows the dependency diagram for this particular mission's sensor category package. Lower-level details of the utility and module packages are omitted for clarity.

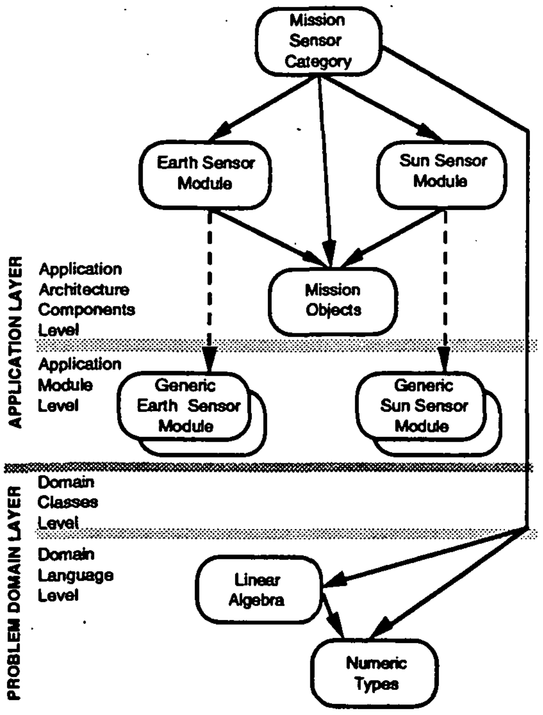


Figure 8. Category Dependency Diagram

Drivers manage the interaction of the application program and the Operation Interface through a generic package that drives the modules and a "dialog package" that manages the interaction with the user interface (see Figure 9). To accomplish this, the driver and dialog packages need to talk to each other. To avoid cyclic dependencies, the driver is instantiated using parameters exported by modules and the dialog package specification. The dialog package body then calls the driver instance to determine the current state of the system, calls modules to select data for user reports, and calls APIs to send the report data and request the user's next choice of action.

In this design, the generic driver's only function is to implement the state transition model specified in the generalized application specifications. The body of the dialog package implements the reports and user options required for a specific mission planning configuration. The package bodies for the dialog and the category packages constitute the only executable code that needs to be written for an application program.

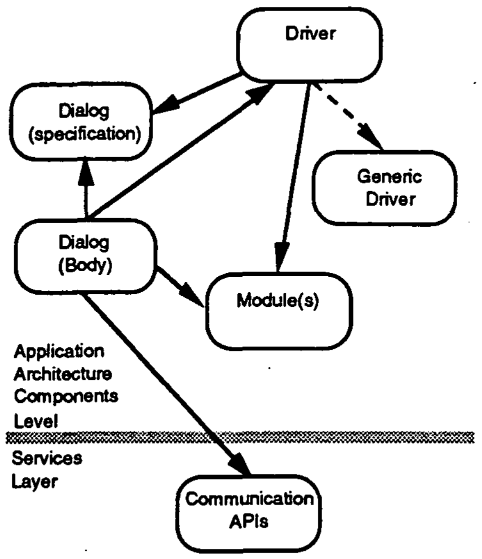


Figure 9. Driver Dependency Diagram

PROTOTYPING STATUS AND OBSERVATIONS

The implementation concepts are currently being prototyped by developing a small mission-planning system. The prototyping has shown that the generalized COMPASS specifications can be implemented as a set of generalized components. The single exception to this statement is the implementation of categories. Each category subprogram calls the corresponding routine in the appropriate module; "appropriate" is not defined until a set of modules is defined for a particular configuration. This means that a developer must write case statements to dispatch calls on a category. The Ada 9X type extension mechanisms described in [Ada 9X 1991] should allow generalized categories to be implemented as reusable components, at which point all generalized specifications will have corresponding generalized implementations.

The prototype team has observed that these concepts are not easily learned but are easily applied once they become understood. The first observation probably reflects the team's inexperience with Ada, coupled with the extensive use of generics. The second results from the direct mapping between the specification and the implementation of a problem domain class. The lessons learned from the prototyping will undoubtedly improve the implementation concepts and drive the FDD's Ada training curriculum.

The prototyping has shown that the techniques and models discussed in Booth and Stark 1989 can be used to define a workable approach to designing large reconfigurable systems. The first application prototype, completed in July of 1991, consists of an orbit propagator program and an operation interface. Approximately 85 percent of the propagator code is composed of generalized components that would be reused verbatim in an operational COMPASS program. This initial measurement shows that, so far, the implementation concepts are consistent with the project's reuse goals. Measurements such as this will be collected throughout the project to rate the success of the implementation concepts.

In "Planning the Software Industrial Revolution" [Cox 1990], Brad Cox extends the concepts of software "chips" to higher levels of integration such as "cards" or "racks." The Ada language provides a means of achieving "software ICs" through the package and generic unit features. The COMPASS implementation concepts, along with the COMPASS system architecture, are a means toward achieving the higher levels of integration that Cox envisions.

REFERENCES

Ada 9X 1991 Ada 9X Project Report, Draft Mapping Document, Office of the Undersecretary of Defense for Acquisition, February 1991

Booch 1987 G. Booch, *Software Components with Ada*, Menlo Park, CA: Benjamin Cummings, 1987

Booth and Stark 1989 E. Booth and M. Stark, "Using Ada Generics to Maximize Verbatim Software Reuse", *Proceedings of Tri-Ada '89*, October 1989

Booth and Luczak 1990 E. Booth and R. Luczak, *Extreme Ultraviolet Explorer (EUVE) Telemetry Simulator Software Development History*, FDD/552-90/045, June 1990

Booth and Stark 1991 E. Booth and M. Stark, *COMPASS Implementation Concepts*, Goddard Space Flight Center, Flight Dynamics Division, 550-COMPASS-105 (Draft), February 1991

Cox 1990 B. Cox "Planning the Software Industrial Revolution," IEEE Software, November 1990

DeFazio 1990 R. DeFazio, et. al., *COMPASS High-Level Requirements, Architecture, and Operation Concepts*, Goddard Space Flight Center, Flight Dynamics Division, 550-COMPASS-102, March 1990

Groveman 1991 B. Groveman, et. al., *Solar, Anomalous, Magnetospheric Particle Explorer (SAMPEX) Telemetry Simulator Software Development History*, Goddard Space Flight Center, Flight Dynamics Division, FDD-552-91/007, May 1991

Seidewitz 1990 E. Seidewitz et. al., *COMPASS Specification Concepts*, Goddard Space Flight Center, Flight Dynamics Division, 550-COMPASS-103, March 1990

Seidewitz and Stark 1991 E. Seidewitz and M. Stark, "An Object-Oriented Approach to Parameterized Software in Ada", *Proceedings of Eighth Washington Ada Symposium*, June 1991

Seidewitz and Green 1990 E. Seidewitz and D. Green, *COMPASS System Architecture Concepts*, Goddard Space Flight Center, Flight Dynamics Division, FDD/552-90/075, March 1990

Stark 1990 M. Stark, "On Designing Parameterized Systems Using Ada", *Proceedings of the Seventh Washington Ada Symposium*, June 1990

AUTHOR BIOGRAPHIES

Michael E. Stark, NASA, Goddard Space Flight Center. As a software engineer for GSFC's Flight Dynamics Division, Mr. Stark is is responsible for research activities on the Software Engineering Laboratory and is the Attitude Support team leader for the COMPASS project. In addition to leading the Attitude Support team, Mr. Stark also leads the development of the COMPASS implementation concepts and is involved in the prototyping of COMPASS application programs. Before becoming COMPASS team leader, he was a principle designer on GSFC's first Ada development project and coauthor of the Generalized Object-Oriented Development Methodology. He has also developed simulation software for the ERBS, COBE, and UARS projects. Mr. Stark holds a B.S. degree in mathematics and economics from Oberlin College and an M.S. degree in computer science from The Johns Hopkins University.

Eric W. Booth, Computer Sciences Corporation. As a software engineer for CSC, Mr. Booth is a task leader on the Systems, Engineering, and Analysis Support Program in the Flight Dynamics Technology Group, where he leads research activities on the Software Engineering Laboratory task supporting Goddard Space Flight Center's Flight Dynamics Division. Before becoming SEL task leader, he led the team that designed and developed a generic software architecture for the Upper Atmosphere Research Satellite telemetry simulator. Mr. Booth holds a B.S. degree in mathematics from the State University of New York at Oneonta and an M.S. degree in computer science from The Johns Hopkins University.

STANDARD BIBLIOGRAPHY OF SEL LITERATURE

The technical papers, memorandums, and documents listed in this bibliography are organized into two groups. The first group is composed of documents issued by the Software Engineering Laboratory (SEL) during its research and development activities. The second group includes materials that were published elsewhere but pertain to SEL activities.

SEL-ORIGINATED DOCUMENTS

SEL-76-001, *Proceedings From the First Summer Software Engineering Workshop*, August 1976

SEL-77-002, *Proceedings From the Second Summer Software Engineering Workshop*, September 1977

SEL-77-004, *A Demonstration of AXES for NAVPAK*, M. Hamilton and S. Zeldin, September 1977

SEL-77-005, *GSFC NAVPAK Design Specifications Languages Study*, P. A. Scheffer and C. E. Velez, October 1977

SEL-78-005, *Proceedings From the Third Summer Software Engineering Workshop*, September 1978

SEL-78-006, *GSFC Software Engineering Research Requirements Analysis Study*, P. A. Scheffer and C. E. Velez, November 1978

SEL-78-007, *Applicability of the Rayleigh Curve to the SEL Environment*, T. E. Mapp, December 1978

SEL-78-302, *FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 3)*, W. J. Decker, W. A. Taylor, et al., July 1986

SEL-79-002, *The Software Engineering Laboratory: Relationship Equations*, K. Freburger and V. R. Basili, May 1979

SEL-79-003, *Common Software Module Repository (CSMR) System Description and User's Guide*, C. E. Goorevich, A. L. Green, and S. R. Waligora, August 1979

SEL-79-004, *Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment*, C. E. Goorevich, A. L. Green, and W. J. Decker, September 1979

SEL-79-005, *Proceedings From the Fourth Summer Software Engineering Workshop*, November 1979

SEL-80-002, *Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation*, W. J. Decker and C. E. Goorevich, May 1980

SEL-80-003, *Multimission Modular Spacecraft Ground Support Software System (MMS/GSSS) State-of-the-Art Computer Systems/Compatibility Study*, T. Welden, M. McClellan, and P. Liebertz, May 1980

SEL-80-005, *A Study of the Musa Reliability Model*, A. M. Miller, November 1980

SEL-80-006, *Proceedings From the Fifth Annual Software Engineering Workshop*, November 1980

SEL-80-007, *An Appraisal of Selected Cost/Resource Estimation Models for Software Systems*, J. F. Cook and F. E. McGarry, December 1980

SEL-80-008, *Tutorial on Models and Metrics for Software Management and Engineering*, V. R. Basili, 1980

SEL-81-008, *Cost and Reliability Estimation Models (CAREM) User's Guide*, J. F. Cook and E. Edwards, February 1981

SEL-81-009, *Software Engineering Laboratory Programmer Workbench Phase 1 Evaluation*, W. J. Decker and F. E. McGarry, March 1981

SEL-81-011, *Evaluating Software Development by Analysis of Change Data*, D. M. Weiss, November 1981

SEL-81-012, *The Rayleigh Curve as a Model for Effort Distribution Over the Life of Medium Scale Software Systems*, G. O. Picasso, December 1981

SEL-81-013, *Proceedings of the Sixth Annual Software Engineering Workshop*, December 1981

SEL-81-014, *Automated Collection of Software Engineering Data in the Software Engineering Laboratory (SEL)*, A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-81-101, *Guide to Data Collection*, V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

SEL-81-104, *The Software Engineering Laboratory*, D. N. Card, F. E. McGarry, G. Page, et al., February 1982

SEL-81-107, *Software Engineering Laboratory (SEL) Compendium of Tools (Revision 1)*, W. J. Decker, W. A. Taylor, E. J. Smith, et al., February 1982

SEL-81-110, *Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics*, G. Page, F. E. McGarry, and D. N. Card, June 1985

SEL-81-205, *Recommended Approach to Software Development*, F. E. McGarry, G. Page, S. Eslinger, et al., April 1983

SEL-82-001, *Evaluation of Management Measures of Software Development*, G. Page, D. N. Card, and F. E. McGarry, September 1982, vols. 1 and 2

SEL-82-004, *Collected Software Engineering Papers: Volume 1*, July 1982

SEL-82-007, *Proceedings of the Seventh Annual Software Engineering Workshop*, December 1982

SEL-82-008, *Evaluating Software Development by Analysis of Changes: The Data From the Software Engineering Laboratory*, V. R. Basili and D. M. Weiss, December 1982

SEL-82-102, *FORTRAN Static Source Code Analyzer Program (SAP) System Description (Revision 1)*, W. A. Taylor and W. J. Decker, April 1985

SEL-82-105, *Glossary of Software Engineering Laboratory Terms*, T. A. Babst, M. G. Rohleder, and F. E. McGarry, October 1983

SEL-82-1006, *Annotated Bibliography of Software Engineering Laboratory Literature*, L. Morusiewicz and J. Valett, November 1991

SEL-83-001, *An Approach to Software Cost Estimation*, F. E. McGarry, G. Page, D. N. Card, et al., February 1984

SEL-83-002, *Measures and Metrics for Software Development*, D. N. Card, F. E. McGarry, G. Page, et al., March 1984

SEL-83-003, *Collected Software Engineering Papers: Volume II*, November 1983

SEL-83-006, *Monitoring Software Development Through Dynamic Variables*, C. W. Doerflinger, November 1983

SEL-83-007, *Proceedings of the Eighth Annual Software Engineering Workshop*, November 1983

SEL-83-106, *Monitoring Software Development Through Dynamic Variables (Revision 1)*, C. W. Doerflinger, November 1989

SEL-84-003, *Investigation of Specification Measures for the Software Engineering Laboratory (SEL)*, W. W. Agresti, V. E. Church, and F. E. McGarry, December 1984

SEL-84-004, *Proceedings of the Ninth Annual Software Engineering Workshop*, November 1984

SEL-84-101, *Manager's Handbook for Software Development (Revision 1)*, L. Landis, F. E. McGarry, S. Waligora, et al., November 1990

SEL-85-001, *A Comparison of Software Verification Techniques*, D. N. Card, R. W. Selby, Jr., F. E. McGarry, et al., April 1985

SEL-85-002, *Ada Training Evaluation and Recommendations From the Gamma Ray Observatory Ada Development Team*, R. Murphy and M. Stark, October 1985

SEL-85-003, *Collected Software Engineering Papers: Volume III*, November 1985

SEL-85-004, *Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics*, R. W. Selby, Jr., and V. R. Basili, May 1985

SEL-85-005, *Software Verification and Testing*, D. N. Card, E. Edwards, F. McGarry, and C. Antle, December 1985

SEL-85-006, *Proceedings of the Tenth Annual Software Engineering Workshop*, December 1985

SEL-86-001, *Programmer's Handbook for Flight Dynamics Software Development*, R. Wood and E. Edwards, March 1986

SEL-86-002, *General Object-Oriented Software Development*, E. Seidewitz and M. Stark, August 1986

SEL-86-003, *Flight Dynamics System Software Development Environment (FDS/SDE) Tutorial*, J. Buell and P. Myers, July 1986

SEL-86-004, *Collected Software Engineering Papers: Volume IV*, November 1986

SEL-86-005, *Measuring Software Design*, D. N. Card et al., November 1986

SEL-86-006, *Proceedings of the Eleventh Annual Software Engineering Workshop*, December 1986

SEL-87-001, *Product Assurance Policies and Procedures for Flight Dynamics Software Development*, S. Perry et al., March 1987

SEL-87-002, *Ada® Style Guide (Version 1.1)*, E. Seidewitz et al., May 1987

SEL-87-003, *Guidelines for Applying the Composite Specification Model (CSM)*, W. W. Agresti, June 1987

SEL-87-004, *Assessing the Ada® Design Process and Its Implications: A Case Study*, S. Godfrey, C. Brophy, et al., July 1987

SEL-87-008, *Data Collection Procedures for the Rehosted SEL Database*, G. Heller, October 1987

SEL-87-009, *Collected Software Engineering Papers: Volume V*, November 1987

SEL-87-010, *Proceedings of the Twelfth Annual Software Engineering Workshop*, December 1987

SEL-88-001, *System Testing of a Production Ada Project: The GRODY Study*, J. Seigle, L. Esker, and Y. Shi, November 1988

SEL-88-002, *Collected Software Engineering Papers: Volume VI*, November 1988

SEL-88-003, *Evolution of Ada Technology in the Flight Dynamics Area: Design Phase Analysis*, K. Quimby and L. Esker, December 1988

SEL-88-004, *Proceedings of the Thirteenth Annual Software Engineering Workshop*, November 1988

SEL-88-005, *Proceedings of the First NASA Ada User's Symposium*, December 1988

SEL-89-002, *Implementation of a Production Ada Project: The GRODY Study*, S. Godfrey and C. Brophy, September 1989

SEL-89-003, *Software Management Environment (SME) Concepts and Architecture*, W. Decker and J. Valett, August 1989

SEL-89-004, *Evolution of Ada Technology in the Flight Dynamics Area: Implementation/ Testing Phase Analysis*, K. Quimby, L. Esker, L. Smith, M. Stark, and F. McGarry, November 1989

SEL-89-005, *Lessons Learned in the Transition to Ada From FORTRAN at NASA/ Goddard*, C. Brophy, November 1989

SEL-89-006, *Collected Software Engineering Papers: Volume VII*, November 1989

SEL-89-007, *Proceedings of the Fourteenth Annual Software Engineering Workshop*, November 1989

SEL-89-008, *Proceedings of the Second NASA Ada Users' Symposium*, November 1989

SEL-89-101, *Software Engineering Laboratory (SEL) Database Organization and User's Guide (Revision 1)*, M. So, G. Heller, S. Steinberg, K. Pumphrey, and D. Spiegel, February 1990

SEL-90-001, *Database Access Manager for the Software Engineering Laboratory (DAMSEL) User's Guide*, M. Buhler, K. Pumphrey, and D. Spiegel, March 1990

SEL-90-002, *The Cleanroom Case Study in the Software Engineering Laboratory: Project Description and Early Analysis*, S. Green et al., March 1990

SEL-90-003, *A Study of the Portability of an Ada System in the Software Engineering Laboratory (SEL)*, L. O. Jun and S. R. Valett, June 1990

SEL-90-004, *Gamma Ray Observatory Dynamics Simulator in Ada (GRODY) Experiment Summary*, T. McDermott and M. Stark, September 1990

SEL-90-005, *Collected Software Engineering Papers: Volume VIII*, November 1990

SEL-90-006, *Proceedings of the Fifteenth Annual Software Engineering Workshop*, November 1990

SEL-91-001, *Software Engineering Laboratory (SEL) Relationships, Models, and Management Rules*, W. Decker, R. Hendrick, and J. Valett, February 1991

SEL-91-003, *Software Engineering Laboratory (SEL) Ada Performance Study Report*, E. W. Booth and M. E. Stark, July 1991

SEL-91-004, *Software Engineering Laboratory (SEL) Cleanroom Process Model*, S. Green, November 1991

SEL-91-005, *Collected Software Engineering Papers: Volume IX*, November 1991

SEL-91-102, *Software Engineering Laboratory (SEL) Data and Information Policy (Revision 1)*, F. McGarry, August 1991

SEL-RELATED LITERATURE

[4]Agresti, W. W., V. E. Church, D. N. Card, and P. L. Lo, "Designing With Ada for Satellite Simulation: A Case Study," *Proceedings of the First International Symposium on Ada for the NASA Space Station*, June 1986

[2]Agresti, W. W., F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," *Program Transformation and Programming Environments*. New York: Springer-Verlag, 1984

[1]Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," *Proceedings of the Fifth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1981

[8]Bailey, J. W., and V. R. Basili, "Software Reclamation: Improving Post-Development Reusability," *Proceedings of the Eighth Annual National Conference on Ada Technology*, March 1990

[1]Basili, V. R., "Models and Metrics for Software Management and Engineering," *ASME Advances in Computer Technology*, January 1980, vol. 1

Basili, V. R., *Tutorial on Models and Metrics for Software Management and Engineering*. New York: IEEE Computer Society Press, 1980 (also designated SEL-80-008)

[3]Basili, V. R., "Quantitative Evaluation of Software Methodology," *Proceedings of the First Pan-Pacific Computer Conference*, September 1985

[7]Basili, V. R., *Maintenance = Reuse-Oriented Software Development*, University of Maryland, Technical Report TR-2244, May 1989

[7]Basili, V. R., *Software Development: A Paradigm for the Future*, University of Maryland, Technical Report TR-2263, June 1989

[8]Basili, V. R., "Viewing Maintenance of Reuse-Oriented Software Development," *IEEE Software*, January 1990

[1]Basili, V. R., and J. Beane, "Can the Parr Curve Help With Manpower Distribution and Resource Estimation Problems?," *Journal of Systems and Software*, February 1981, vol. 2, no. 1

[9]Basili, V. R., and G. Caldiera, *A Reference Architecture for the Component Factory*, University of Maryland, Technical Report TR-2607, March 1991

[1]Basili, V. R., and K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," *Journal of Systems and Software*, February 1981, vol. 2, no. 1

[3]Basili, V. R., and N. M. Panlilio-Yap, "Finding Relationships Between Effort and Other Variables in the SEL," *Proceedings of the International Computer Software and Applications Conference*, October 1985

[4]Basili, V. R., and D. Patnaik, *A Study on Fault Prediction and Reliability Assessment in the SEL Environment*, University of Maryland, Technical Report TR-1699, August 1986

[2]Basili, V. R., and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Communications of the ACM*, January 1984, vol. 27, no. 1

[1]Basili, V. R., and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," *Proceedings of the ACM SIGMETRICS Symposium/Workshop: Quality Metrics*, March 1981

[3]Basili, V. R., and C. L. Ramsey, "ARROWSMITH-P—A Prototype Expert System for Software Engineering Management," *Proceedings of the IEEE/MITRE Expert Systems in Government Symposium*, October 1985

Basili, V. R., and J. Ramsey, *Structural Coverage of Functional Testing*, University of Maryland, Technical Report TR-1442, September 1984

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," *Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity, and Cost*. New York: IEEE Computer Society Press, 1979

[5]Basili, V. R., and H. D. Rombach, "Tailoring the Software Process to Project Goals and Environments," *Proceedings of the 9th International Conference on Software Engineering*, March 1987

[5]Basili, V. R., and H. D. Rombach, "T A M E: Tailoring an Ada Measurement Environment," *Proceedings of the Joint Ada Conference*, March 1987

[5]Basili, V. R., and H. D. Rombach, "T A M E: Integrating Measurement Into Software Environments," University of Maryland, Technical Report TR-1764, June 1987

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

[7]Basili, V. R., and H. D. Rombach, *Towards A Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment*, University of Maryland, Technical Report TR-2158, December 1988

[8]Basili, V. R., and H. D. Rombach, *Towards A Comprehensive Framework for Reuse: Model-Based Reuse Characterization Schemes*, University of Maryland, Technical Report TR-2446, April 1990

[9]Basili, V. R., and H. D. Rombach, *Support for Comprehensive Reuse*, University of Maryland, Technical Report TR-2606, February 1991

[3]Basili, V. R., and R. W. Selby, Jr., "Calculation and Use of an Environment's Characteristic Software Metric Set," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

Basili, V. R., and R. W. Selby, Jr., *Comparing the Effectiveness of Software Testing Strategies*, University of Maryland, Technical Report TR-1501, May 1985

[3]Basili, V. R., and R. W. Selby, Jr., "Four Applications of a Software Data Collection and Analysis Methodology," *Proceedings of the NATO Advanced Study Institute*, August 1985

[5]Basili, V. R., and R. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, December 1987

[9]Basili, V. R., and R. W. Selby, "Paradigms for Experimentation and Empirical Studies in Software Engineering," *Reliability Engineering and System Safety*, January 1991

[4]Basili, V. R., R. W. Selby, Jr., and D. H. Hutchens, "Experimentation in Software Engineering," *IEEE Transactions on Software Engineering*, July 1986

[2]Basili, V. R., R. W. Selby, and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," *IEEE Transactions on Software Engineering*, November 1983

[2]Basili, V. R., and D. M. Weiss, *A Methodology for Collecting Valid Software Engineering Data*, University of Maryland, Technical Report TR-1235, December 1982

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

[1]Basili, V. R., and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives," *Proceedings of the Fifteenth Annual Conference on Computer Personnel Research*, August 1977

Basili, V. R., and M. V. Zelkowitz, "Designing a Software Measurement Experiment," *Proceedings of the Software Life Cycle Management Workshop*, September 1977

[1]Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," *Proceedings of the Second Software Life Cycle Management Workshop*, August 1978

[1]Basili, V. R., and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," *Computers and Structures*, August 1978, vol. 10

Basili, V. R., and M. V. Zelkowitz, "Analyzing Medium Scale Software Development," *Proceedings of the Third International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1978

[9]Booth, E. W., and M. E. Stark, "Designing Configurable Software: COMPASS Implementation Concepts," *Proceedings of Tri-Ada 1991*, October 1991

[9]Briand, L. C., V. R. Basili, and W. M. Thomas, *A Pattern Recognition Approach for Software Engineering Data Analysis*, University of Maryland, Technical Report TR-2672, May 1991

[5]Brophy, C. E., W. W. Agresti, and V. R. Basili, "Lessons Learned in Use of Ada-Oriented Design Methods," *Proceedings of the Joint Ada Conference*, March 1987

[6]Brophy, C. E., S. Godfrey, W. W. Agresti, and V. R. Basili, "Lessons Learned in the Implementation Phase of a Large Ada Project," *Proceedings of the Washington Ada Technical Conference*, March 1988

[2]Card, D. N., "Early Estimation of Resource Expenditures and Program Size," Computer Sciences Corporation, Technical Memorandum, June 1982

[2]Card, D. N., "Comparison of Regression Modeling Techniques for Resource Estimation," Computer Sciences Corporation, Technical Memorandum, November 1982

[3]Card, D. N., "A Software Technology Evaluation Program," *Annais do XVIII Congresso Nacional de Informatica*, October 1985

[5]Card, D. N., and W. W. Agresti, "Resolving the Software Science Anomaly," *The Journal of Systems and Software*, 1987

[6]Card, D. N., and W. 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

Card, D. N., V. E. Church, W. W. Agresti, and Q. L. Jordan, "A Software Engineering View of Flight Dynamics Analysis System," Parts I and II, Computer Sciences Corporation, Technical Memorandum, February 1984

Card, D. N., Q. L. Jordan, and V. E. Church, "Characteristics of FORTRAN Modules," Computer Sciences Corporation, Technical Memorandum, June 1984

[5]Card, D. N., F. E. McGarry, and G. T. Page, "Evaluating Software Engineering Technologies," *IEEE Transactions on Software Engineering*, July 1987

[3]Card, D. N., G. T. Page, and F. E. McGarry, "Criteria for Software Modularization," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

[1]Chen, E., and M. V. Zelkowitz, "Use of Cluster Analysis To Evaluate Software Engineering Methodologies," *Proceedings of the Fifth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1981

[4]Church, V. E., D. N. Card, W. W. Agresti, and Q. L. Jordan, "An Approach for Assessing Software Prototypes," *ACM Software Engineering Notes*, July 1986

[2]Doerflinger, C. W., and V. R. Basili, "Monitoring Software Development Through Dynamic Variables," *Proceedings of the Seventh International Computer Software and Applications Conference*. New York: IEEE Computer Society Press, 1983

Doubleday, D., *ASAP: An Ada Static Source Code Analyzer Program*, University of Maryland, Technical Report TR-1895, August 1987 (NOTE: 100 pages long)

[6]Godfrey, S., and C. Brophy, "Experiences in the Implementation of a Large Ada Project," *Proceedings of the 1988 Washington Ada Symposium*, June 1988

Hamilton, M., and S. Zeldin, *A Demonstration of AXES for NAVPAK*, Higher Order Software, Inc., TR-9, September 1977 (also designated SEL-77-005)

[5]Jeffery, D. R., and V. Basili, *Characterizing Resource Data: A Model for Logical Association of Software Data*, University of Maryland, Technical Report TR-1848, May 1987

[6]Jeffery, D. R., and V. R. Basili, "Validating the TAME Resource Data Model," *Proceedings of the Tenth International Conference on Software Engineering*, April 1988

[5]Mark, L., and H. D. Rombach, *A Meta Information Base for Software Engineering*, University of Maryland, Technical Report TR-1765, July 1987

[6]Mark, L., and H. D. Rombach, "Generating Customized Software Engineering Information Bases From Software Process and Product Specifications," *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, January 1989

[5]McGarry, F. E., and W. W. Agresti, "Measuring Ada for Software Development in the Software Engineering Laboratory (SEL)," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988

[7]McGarry, F., L. Esker, and K. Quimby, "Evolution of Ada Technology in a Production Software Environment," *Proceedings of the Sixth Washington Ada Symposium (WADAS)*, June 1989

[3]McGarry, F. E., J. Valett, and D. Hall, "Measuring the Impact of Computer Resource Quality on the Software Development Process and Product," *Proceedings of the Hawaiian International Conference on System Sciences*, January 1985

National Aeronautics and Space Administration (NASA), *NASA Software Research Technology Workshop* (Proceedings), March 1980

[3]Page, G., F. E. McGarry, and D. N. Card, "A Practical Experience With Independent Verification and Validation," *Proceedings of the Eighth International Computer Software and Applications Conference*, November 1984

[5]Ramsey, C. L., and V. R. Basili, *An Evaluation of Expert Systems for Software Engineering Management*, University of Maryland, Technical Report TR-1708, September 1986

[3]Ramsey, J., and V. R. Basili, "Analyzing the Test Process Using Structural Coverage," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

[5]Rombach, H. D., "A Controlled Experiment on the Impact of Software Structure on Maintainability," *IEEE Transactions on Software Engineering*, March 1987

[8]Rombach, H. D., "Design Measurement: Some Lessons Learned," *IEEE Software*, March 1990

[9]Rombach, H. D., "Software Reuse: A Key to the Maintenance Problem," *Butterworth Journal of Information and Software Technology*, January/February 1991

[6]Rombach, H. D., and V. R. Basili, "Quantitative Assessment of Maintenance: An Industrial Case Study," *Proceedings From the Conference on Software Maintenance*, September 1987

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

[7]Rombach, H. D., and B. T. Ulery, *Establishing a Measurement Based Maintenance Improvement Program: Lessons Learned in the SEL*, University of Maryland, Technical Report TR-2252, May 1989

[6]Seidewitz, E., "Object-Oriented Programming in Smalltalk and Ada," *Proceedings of the 1987 Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1987

[5]Seidewitz, E., "General Object-Oriented Software Development: Background and Experience," *Proceedings of the 21st Hawaii International Conference on System Sciences*, January 1988

[6]Seidewitz, E., "General Object-Oriented Software Development with Ada: A Life Cycle Approach," *Proceedings of the CASE Technology Conference*, April 1988

[9]Seidewitz, E., "Object-Oriented Programming Through Type Extension in Ada 9X," *Ada Letters*, March/April 1991

[4]Seidewitz, E., and M. Stark, "Towards a General Object-Oriented Software Development Methodology," *Proceedings of the First International Symposium on Ada for the NASA Space Station*, June 1986

[9]Seidewitz, E., and M. Stark, "An Object-Oriented Approach to Parameterized Software in Ada," *Proceedings of the Eighth Washington Ada Symposium*, June 1991

[8]Stark, M., "On Designing Parametrized Systems Using Ada," *Proceedings of the Seventh Washington Ada Symposium*, June 1990

[7]Stark, M. E. and E. W. Booth, "Using Ada to Maximize Verbatim Software Reuse," *Proceedings of TRI-Ada 1989*, October 1989

[5]Stark, M., and E. Seidewitz, "Towards a General Object-Oriented Ada Lifecycle," *Proceedings of the Joint Ada Conference*, March 1987

[8]Straub, P. A., and M. V. Zelkowitz, "PUC: A Functional Specification Language for Ada," *Proceedings of the Tenth International Conference of the Chilean Computer Science Society*, July 1990

[7]Sunazuka, T., and V. R. Basili, *Integrating Automated Support for a Software Management Cycle Into the TAME System*, University of Maryland, Technical Report TR-2289, July 1989

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. D., and F. E. McGarry, "A Summary of Software Measurement Experiences in the Software Engineering Laboratory," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988

[3]Weiss, D. M., and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data From the Software Engineering Laboratory," *IEEE Transactions on Software Engineering*, February 1985

[5]Wu, L., V. R. Basili, and K. Reed, "A Structure Coverage Tool for Ada Software Systems," *Proceedings of the Joint Ada Conference*, March 1987

[1]Zelkowitz, M. V., "Resource Estimation for Medium-Scale Software Projects," *Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science*. New York: IEEE Computer Society Press, 1979

[2]Zelkowitz, M. V., "Data Collection and Evaluation for Experimental Computer Science Research," *Empirical Foundations for Computer and Information Science* (Proceedings), November 1982

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

[6]Zelkowitz, M. V., "Resource Utilization During Software Development," *Journal of Systems and Software*, 1988

[8]Zelkowitz, M. V., "Evolution Towards Specifications Environment: Experiences With Syntax Editors," *Information and Software Technology*, April 1990

Zelkowitz, M. V., and V. R. Basili, "Operational Aspects of a Software Measurement Facility," *Proceedings of the Software Life Cycle Management Workshop*, September 1977

NOTES:

[1]This article also appears in SEL-82-004, *Collected Software Engineering Papers: Volume I*, July 1982.

[2]This article also appears in SEL-83-003, *Collected Software Engineering Papers: Volume II*, November 1983.

[3]This article also appears in SEL-85-003, *Collected Software Engineering Papers: Volume III*, November 1985.

[4]This article also appears in SEL-86-004, *Collected Software Engineering Papers: Volume IV*, November 1986.

[5]This article also appears in SEL-87-009, *Collected Software Engineering Papers: Volume V*, November 1987.

[6]This article also appears in SEL-88-002, *Collected Software Engineering Papers: Volume VI*, November 1988.

[7]This article also appears in SEL-89-006, *Collected Software Engineering Papers: Volume VII*, November 1989.

[8]This article also appears in SEL-90-005, *Collected Software Engineering Papers: Volume VIII*, November 1990.

[9]This article also appears in SEL-91-005, *Collected Software Engineering Papers: Volume IX*, November 1991.