TAILORING THE SOFTWARE PROCESS TO PROJECT GOALS AND ENVIRONMENTS*

Victor R. Basili and H. Dieter Rombach

Department of Computer Science
University of Maryland
College Park MD 20742
(301) 454-2002

ABSTRACT

This paper presents a methodology for improving the software process by tailoring it to the specific project goals and environment. This improvement process is aimed at the global software process model as well as methods and tools supporting that model. The basic idea is to use defect profiles to help characterize the environment and evaluate the project goals and the effectiveness of methods and tools in a quantitative way. The improvement process is implemented iteratively by setting project improvement goals, characterizing those goals and the environment, in part, via defect profiles in a quantitative way, choosing methods and tools fitting those characteristics, evaluating the actual behavior of the chosen set of methods and tools, and refining the project goals based on the evaluation results. All these activities require analysis of large amounts of data and, therefore, support by an automated tool. Such a tool - TAME (Tailoring A Measurement Environment) - is currently being developed.

KEYWORDS: software process, methods, tools, measurement, evaluation, improvement, tailoring, goals, environment, errors, faults, failures.

INTRODUCTION

One of the major problems in software projects is the lack of management's ability to (1) find criteria for choosing the appropriate process (global process model and methods and tools supporting those models), (2) evaluating the goodness of the software process, and (3) improve it. In a survey of the software industry Thayer et al.[20] listed the twenty major problems reported by software managers. Of these twenty, over half (at least thirteen) delineated the need of management to find selection criteria for the choice of technology or to be able to judge the quality of the existing software process.

In many cases, there does exist a fair amount of technology available for software projects. However, it is not always apparent to the manager which of these methods or tools to invest in, and whether or not they are working as predicted for the particular project. What is needed in almost all cases is a quantitative approach to software management and engineering.

We have been working on a methodology for choosing and improving the software process and resulting products. A particular software process is defined by a global process model and methods and tools that support it. In this paper we emphasize choosing and improving the set of methods and tools in the context of a given global process model. The criteria for improvement of the set of methods and tools to be used in a project is the degree to which they support the achievement of given project quality and productivity goals in this particular project environment. Such an evolutionary improvement process requires the tailoring of methods and tools to (constantly changing) project goals and environment characteristics. Sound tailoring requires the ability to characterize the project goals to be achieved, the environment in which those goals are to be achieved, and the effect of methods and tools on achieving those goals in a particular environment. Quantitative characterization is preferred because it gives more credibility to characterizations and better justification to the improvement recommendations based upon these characterizations.

There are various approaches to characterization, one of which is to use defects (errors, faults, and failures). Project goals are characterized by the number and type of defects (deviations from the optimum), environments are characterized by the number and type of defects they impose on projects, and methods and tools can be characterized by the number and type of defects related to their use. We can think of alternative approaches to characterize the impact of goals, environments, methods and tools. Instead of defects during development we could use defects during operation or some measure of customer's/user's satisfaction. A completely different approach would be to characterize a project environment and methods and tools by a set of factors such as 'what are the abilities of humans involved and how are those abilities supported by candidate methods and tools', 'what are characteristics of the software process model and

* Research for this study was supported in part by the National Aeronautics and Space Administration grant NSG-5123 to the University of Maryland. Computer time was supported in part through the facilities of the Computer Science Center of the University of Maryland.

Reviewed and recommended by:

Barry Boehm

how are those aspects supported by candidate methods and tools', etc. These different characterization approaches are not necessarily exclusive but might be used together. However, in this paper we are basing our tailoring on defect profiles during development.

As indicated in figure 1 the characterization mechanism 'defects during development' is applied to a particular environment, the NASA/SEL* environment. Therefore, all the defect profiles and characterizations presented in this paper are specific to those NASA/SEL projects. The fact that the impact of methods and tools may vary substantially across environments does not affect the message to be sent by this paper: There are ways to support a project manager's need to evaluate and improve the software process based on quantitative information. The approach can be transferred to other environments, the specific characterizations derived from NASA/SEL projects cannot; they have to be revalidated in the new environment.

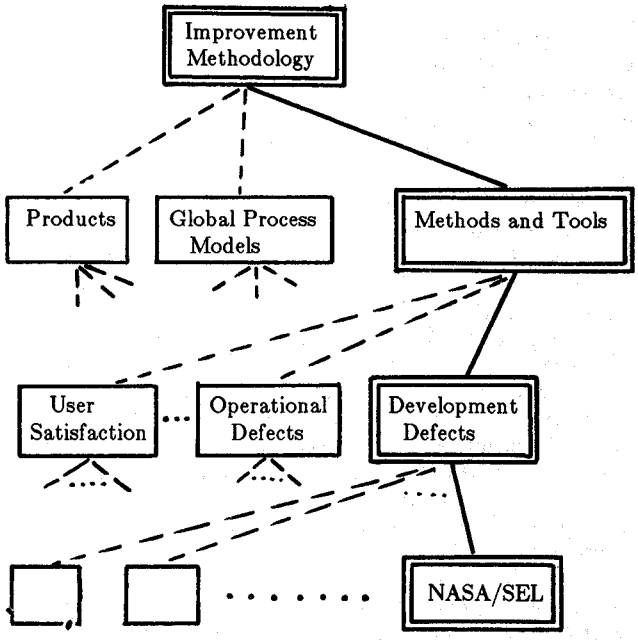

Fig. 1. Improvement Methodology Framework

Figure 1 clearly decribes the overall scope of the improvement and tailoring approach presented in this paper. Based upon a general improvement methodology we (1) emphasize choosing and tailoring the set of methods and tools to specific project goals and environments, (2) use one particular mechanism (development defects) for quantitatively characterizing project goals, environments, and the effect of methods and tools, and (3) validate the approach in one particular environment (NASA/SEL).

* The SEL (Software Engineering Laboratory) is a joint project between NASA Goddard Space Flight Center, the University of Maryland, and Computer Sciences Corporation. The objective of this project is to evaluate and improve the software process and its resulting products.

In the following sections we introduce the improvement methodology; the relationship of errors, faults, and failures with a (most process model underlying) problem-solution model; classification schemes for errors, faults, and failures; the tailoring approach including characterizations of the impact of methods and tools in the NASA/SEL environment; an application of the improvement methodology to a characteristic NASA/SEL project; and, finally, the TAME (Tailoring A Measurement Methodology) project intended to support all measurement and evaluation tasks required by the presented improvement methodology.

IMPROVEMENT METHODOLOGY

The improvement methodology requires a mechanism for characterizing the project environment and the candidate process models, methods, and tools. The process requires an organized mechanism for determining the improvement goals; defining those goals in a traceable way into a set of quantitative questions that define a specific set of data for collection. The improvement goals flow from the needs of the current project and, as far as possible, knowledge from previous projects. Based on a check to what degree the established improvement goals can be met by candidate process models, methods, and tools in the particular project environment, the most promising ones are chosen for the current project. Throughout the project the set of prescribed data is collected, validated, fed back into the current project, and subsequently evaluated for the purpose of improving future projects. This evaluation determines the degree to which the stated improvement goals were met by the chosen software process. Based on these findings recommendations for improvement are made as input for the next project.

This whole improvement process[6] is structured into five major steps:

1. Characterize the approach/environment.
This step requires an understanding of the various factors that will influence the project development. This includes the problem factors, e.g. the type of problem, the newness to the state of the art, the susceptibility to change, the people factors, e.g. the number of people working on the project, their level of expertise, experience, the product factors, e.g. the size, the deliverables, the reliability requirements, portability requirements, reusability requirements, the resource factors, e.g. target and development machine systems, availability, budget, deadlines, the process and tool factors, e.g. what methods and tools are available, training in them, programming languages, code analyzers.

2. Set up the goals, questions, data for successful project development and improvement over previous project developments.
It is at this point the organization and the project manager must determine what the goals are for the project development. Some of these may be specified from step 1. Others may be chosen based upon the needs of the organization, e.g. reusability of the code on another project, improvement of the quality, lower cost.

3. Choose the appropriate methods and tools for the project.
Once it is clear what is required and available, methods

and tools should be chosen and refined that will maximize the chances of satisfying the goals laid out for the project. Tools may be chosen because they facilitate the collection of the data necessary for evaluation, e.g. configuration management tools not only help project control but also help with the collection and validation of error and change data.

4. **Perform the software development and maintenance, collect the prescribed data and validate it, and provide feedback to the current project in real time.**
This step involves the transfer of new technologies chosen in the previous step into the current project environment, and the application of the new software process. Throughout the project data have to be collected by forms, interviews, and automated collection mechanisms. The advantages of using forms to collect data is that a full set of data can be gathered which gives detailed insights and provides for good record keeping. The drawback to forms is that they can be expensive and unreliable because people fill them out. Interview can be used to validate information from forms and gather information that is not easily obtainable in a form format. Automated data collection is reliable and unobtrusive and can be gathered from program development libraries, program analyzers, etc. However, the type of data that can be collected in this way is typically not very insightful and one level removed from the issue being studied. Besides the post mortem analysis in step 5 for the purpose of suggesting improvements for future projects, we are also interested in tuning the software process of the ongoing project based on real time feedback from measurement activities.

5. **Analyze the data to evaluate the current practices, determine problems, record the findings, and make recommendations for improvement for future projects.**
This is the key to the mechanism. It requires a post mortem evaluation of the project. Project data should be analyzed to determine how well the project satisfied its goals, where the methods were effective, where they were not effective, whether they should be modified and refined for better application, whether more training or different training is needed, whether tools or standards are needed to help in the application of the methods, or whether the methods or tools should be discarded and new methods or tools applied on the next project. Proceed to step 1 to start the next project, armed with the knowledge gained from this and the previous projects.

This procedure for developing software has a corporate learning curve built in. The knowledge is not hidden in the intuition of first level managers but is stored in a corporate data base available to new and old managers to help with project management, method and tool evaluation, and technology transfer.

As indicated earlier the effectiveness of this improvement methodology depends on the ability to quantitatively characterize the improvement criteria 'environment' (see step 1) and 'goals' (see step 2) as well as the effectiveness of our improvement vehicles 'process models', 'methods', and 'tools' in meeting those criteria (see step 2). The following sections of this paper propose an approach to support the activities in figure 2 marked with '*' by analysis of error, fault, and failure profiles. The steps concerned with setting up improvement goals, data collection and validation, data analysis, and interpretation are performed according to a separate evaluation methodology[2]; these steps are marked with '+' in figure 2. The choosing of appropriate models, methods, and tools (see steps marked with '**' in figure 2) is made based on characteristics of the of improvement goals and the environment and sound knowledge concerning the qualification of models, methods, and tools of meeting those characteristics[9].

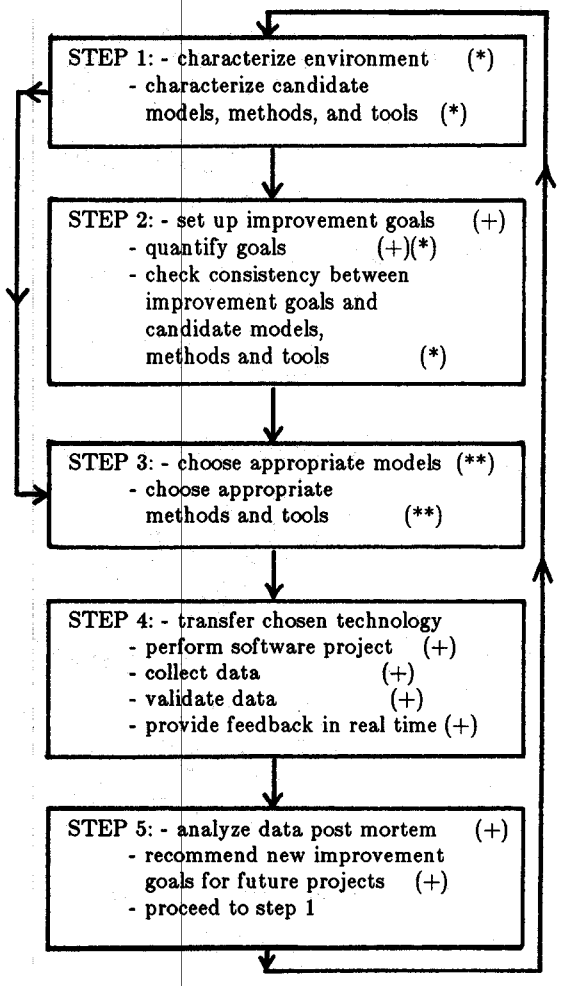

Fig. 2. Improvement Methodology

LIFE CYCLE OF DEFECTS

The use of methods and tools is supposed to improve software quality and productivity by reducing the number of defects. To make effective use of methods and tools one has to be aware of the nature of defects.

Defects exist in three different instances according to[15]: **Errors** are defects in the human thought process made while trying to understand given information, to solve problems, or to use methods and tools. **Faults** are the concrete

manifestations of errors within the software. One error may cause several faults; various errors may cause identical errors. **Failures** are the departures of the software system from software requirements (or intended use respectively). A particular failure may be caused by several faults together; a particular failure may be caused by different faults alternatively; some faults may never cause a failure (difference between reliability and correctness).

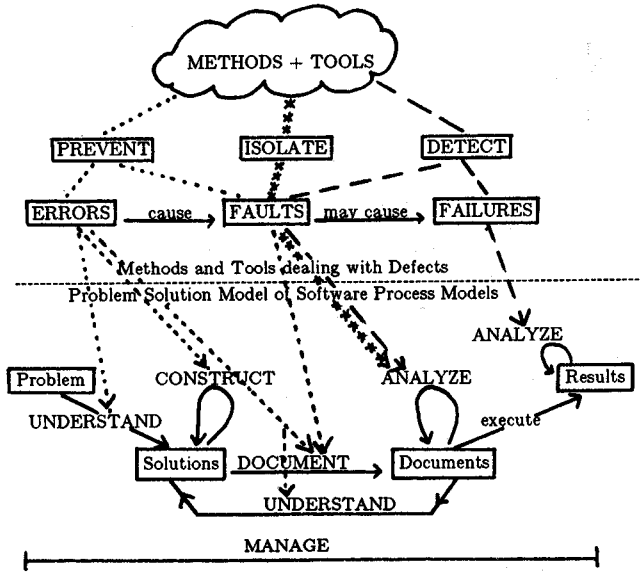

Fig. 3. Methods, Tools - Defects - Process Model

In Figure 3, the above defined relationship between errors, faults, and failures, their relationship with a general problem solving model incorporated in each concrete process model, and their relationship with prevention, isolation, or detection methods and tools is outlined.

A general problem solving model incorporated in each process model consists (or should consist) of an iteration* of the following sequence of general activities:

- **Understanding** of given information such as problem, requirements or design documents
- **Constructing** some new (in general more concrete) solution
- **Documenting** the new solution
- **Analyzing** the new solution, and possibly starting a new iteration of development or executing the product
- **Managing** the development and maintenance process and all resulting documents (data)

The relationships between errors, faults, failures on the one hand and the prevention or detection approach on the other hand are as follows:

- **Errors can be prevented** (e.g. by training).
- **Faults can be prevented** from entering a software product (e.g. by a syntax directed editor).
- **Faults can be detected** during non-operational analysis, all related faults can be isolated and corrected.
- **Failures can be detected** during execution (test or operation), all related faults can be isolated and corrected.

CLASSIFICATION OF DEFECTS

The effectiveness of the introduced improvement methodology depends on the availability of defect classification schemes allowing us to characterize quality and productivity aspects, as well as the impact of a particular environment on quality and productivity, and to distinguish between methods and tools based on the degree to which they can prevent, detect, isolate and correct various defect classes. Numerous classification schemes for defects were proposed for various purposes. In this section several classification schemes for errors, faults, and failures will be presented which are expected to allow us to do a good job in tailoring methods and tools towards project improvement goals and environments. Most of these schemes were presented in the literature already, some are refinements of earlier schemes.

The usefulness of each classification scheme for the purpose of tailoring methods and tools to improvement goals and environments is evaluated with respect to three criteria: 1) is it possible to decide the defect class for each defect, 2) can the information necessary for the decision be collected easily, and 3) for each class, are there methods and tools that can either prevent or detect, isolate, and correct the defects in that class. The first criterion determines whether a scheme is of any practical use, the second criterion just formulates the characteristics of a real classification scheme (for each defect there exists one and only one class it belongs to), whereas the third criterion defines the goal of schemes in this context.

ERROR CLASSIFICATION

The criterion for a classification of errors in this context is, to define classes of errors by the ease with which they can be prevented by different (types of) methods and tools. The presented error classification schemes all try to allow the identification of certain problem areas within the project environment. The first classification scheme indicates the phases in which errors occurred; the second classification scheme indicates domains of the project environment which resulted in errors. There exist many more schemes in the literature[1, 5], most of them being refinements of the following two schemes; refinements of these two schemes might be appropriate in order to represent specific environment characteristics. or 'problem-solution'.

The practical use of error classification schemes in general is tricky because error data can't be collected by analyzing documents. By nature, identifying errors means to understand the defect in the thought process of a human being after the fact[16]. The problems, and consequently sources for misclassification, lie in the attempt to reconstruct the thought process of human beings as well as in the fact that this classification of errors is usually done after the fact.

* The number of iterations depends on the chosen global process model.

The usual procedure is, that fault data are collected, and error data are derived based on interviews with the original programmer or subjective guesses. An additional problem lies in the complex interrelationship between errors and faults: One error can result in different faults (an application error might result in a control fault as well as in a computation fault), one fault might be caused by different errors (a computation fault can be caused by an application error as well as by a clerical error), one error can result in a number of faults at the same time. Faults are classified depending on how they were corrected. It is well-known that a given fault in many cases might be corrected in different ways (changing a control construct or changing a computation) what would put it into different fault classes. Trying to reconstruct the underlying error based on such vague fault classification might be an impossible

Error Scheme 1 (by Time of Error Occurrence). Classification of errors by the time of their occurrence allows you to attribute certain errors to methods and tools used at this time. Because methods and tools are usually used during certain phases or activities according to some process model, the virtual time scale used for error classification is phases. E.g., for NASA projects monitored by the University of Maryland errors were classified, according to NASA's process model, as **1) requirements, 2) specification, 3) design, 4) code, 5) unit test, 6) system test, 7) acceptance test, and 8) maintenance errors**. Whenever one of the classes in such a classification scheme shows an above average number of errors we know what phase to emphasize for the purpose of error prevention. This classification scheme fulfills all three criterion for being useful.

Error Scheme 2 (by Domains which are Causing the Errors). Classification of errors by the project aspects that caused problems allows you to attribute certain errors to methods and tools dealing with these aspects of the software project. Typical problem domains can be the application area, the methodology to be used, the environment of the software to be developed, etc. The following classification is a slight modification of the scheme developed by Basili and others[5]:

- **Application** errors are due to a misunderstanding of the application or problem domain. Application errors are possible during all life cycle phases, but are more likely during early development phases.
- **Problem-Solution** errors are due to not knowing, misunderstanding, or misuse of problem solution processes. This kind of errors occur in the process of finding a solution for a stated and well-understood problem; this solution is then going to be represented using the syntax and semantic rules of some language. Practically, these problem-solution errors can occur in the process of specifying, designing or coding a problem.
- **Semantics** errors are due to a misunderstanding or misuse of the semantic rules of a language (for representing code, designs, specifications, or requirements).
- **Syntax** errors are due to a misunderstanding or misuse of the syntactic rules of a language (for representing code, designs, specifications, or requirements).
- **Environment**: errors are due to a misunderstanding or misuse of the hardware or software environment of a given project. Environment comprises all hardware and software used but not developed within a given project (for example, operating systems, devices, data base systems).
- **Information Management** errors are due to a mishandling of certain procedures.
- **Clerical** errors are due to carelessness while performing mechanical transcriptions from one format to another or from one medium to another. No interpretation or semantic translation is involved. Examples are typing errors using an editor.

This classification scheme has its problems with respect to criterion 1. It is not always easy to decide whether an error is of type 'application' or of type 'problem-solution'.

FAULT CLASSIFICATION

The criterion for a classification of faults in this context is, to define classes of faults by the ease with which they can be detected or isolated by different (types of) methods and tools. The presented fault classification schemes try to allow the identification of certain problem areas within the project environment. The first classification scheme indicates the phases in which faults are detected; the second scheme indicates whether a fault was due to omission or commission; the third classification scheme indicates various software aspects affected by faults. A number of fault classifications exist[1, 17, 18, 19].

Fault Scheme 1 (by Time of Fault Detection). Classification of faults by the time of their detection allows you to attribute certain faults to methods and tools used up to this time. Because methods and tools are usually used during certain phases or activities according to some process model, the virtual time scale for fault classification is phases or activities. In the case of the NASA/SEL environment the same classification scheme is used as in error scheme 1. This classification scheme fulfills all three criteria for being useful.

Fault Scheme 2 (by Omission/Commission). Classification of faults depending on whether something is missing completely (**omission**) or whether something is incorrect (**commission**) proved to be a very helpful classification with respect to classifying methods and tools. It is obvious that omission errors are harder to detect by detection methods and tools solely based on the source code such as structural testing, whereas functional testing or code reading are more successful based on the fact that these methods include the corresponding specifications into the detection process[8]. This classification scheme is useful according to our three criteria.

Fault Scheme 3 (by Software Aspects Affected by Faults). Classification of faults by the product aspects affected allows us to attack certain faults by methods and tools aiming at exactly those aspects. It is obvious that a large number of control flow faults is better detected by a detection method or tool which is based on dynamic simulation of the program (such as testing) rather than static checks (such as code reading by stepwise abstraction)[8]. How many classes exist depends heavily on the language used. It doesn't make sense to create classes for faults that cannot be identified easily because the corresponding aspects are not represented by language features explicitly. One example is that in Fortran environments it is harder to identify control flow faults of global character (affecting more than one program unit) than it is in Ada, where interfaces are explicit. Therefore, the following classification scheme, used in the

NASA/SEL Fortran environment is of higher granularity (especially as far as interface or global faults are concerned) than the corresponding scheme for an Ada environment would be:

- **Control Flow** faults are related to incorrect control flow within one module. Examples are incorrect sequences of statements, incorrect branching, use of incorrect branching condition, or incorrect computation of branching condition.
- **Interface** faults are related to problems affecting more than one module. Examples are incorrect module interfaces, incorrect implementation in more than one module due to a bad design decision, or incorrect definition or initialization of global data. An interface fault might require corrections in only one or in more than one module.
- **Data** faults are related to incorrect data handling. One can distinguish between three types:
  - **Data Definition** faults are related to incorrect name, type, or memory specification.
  - **Data Initialization** faults are related to incorrect initialization of a variable.
  - **Data Use** faults are related to wrong use of a variable.
- **Computation** faults are related to incorrect mathematical expression (if not a branching condition).

This classification scheme is useful with respect to our three criteria. If a fault seems to fit into more than one class, the first applicable one is to be chosen.

FAILURE CLASSIFICATION

The criterion for a classification of failures in this context is, to define classes of failures by the ease with which they can be detected by different methods and tools. The presented failure classification schemes allow the identification of the failure time and the impact of failures on the production of a system.

Failure Scheme 1 (by Time of Failure Detection). Classification of failures by the time of their detection allows you to attribute certain failures to methods and tools used up to this time. Because methods and tools are usually used during certain phases or activities according to some process model, the virtual time scale for failure classification is phases or activities. In the case of the NASA/SEL environment a subset of the classification scheme in 4.1.1. is used; only those phases or activities are used which include execution: (1) **unit test**, (2) **system test**, (3) **acceptance test**, and **maintenance**. This classification scheme is useful according to all three of our criteria.

Failure Scheme 2 (by Severity of Failures). Classification of failures by their impact on the environment of the system under consideration allow us to decide on the degree to which those failures can be tolerated. A possible classification scheme is (1) **stops production completely**, (2) **impacts production significantly**, (3) **prevents full use of features, but can be compensated**, and (4) **minor or cosmetic**. This classification scheme is useful for characterizing the impact of failures, but it does not allow the classification of methods and tools with respect to the ease with which those failures can be detected.

TAILORING TO PROJECT GOALS AND ENVIRONMENT

Supporting the improvement methodology for the purpose of tailoring the set of methods and tools to be used in a project, requires quantification of how to characterize (1) project improvement goals, (2) the particular project environment, and (3) the effect of candidate methods and tools on those goals and environment. The approach chosen in this paper is to utilize error, fault, and failure analysis. As discussed as part of the introduction section and reflected in figure 1, utilizing defects is only one possibility for characterizing improvement goals, environments, methods, and tools for the purpose of tailoring. However, it is an approach that guarantees that we take all factors possibly affecting the outcome of a project into consideration. The advantage of this approach is that we can use data from previous similar projects in the same environment; the disadvantage is that we take an indirect characterization approach (by measuring the impact of environments) rather than a direct approach (by measuring factors of the environments themselves). Indirect approaches allow precise characterizations of environments by conducting post-mortem analysis of the impact of environments and methods and tools in this environment on quality and productivity; direct approaches allow better characterization of new environments before any projects are completed in this new environment.

CHARACTERIZING IMPROVEMENT GOALS

The approach to the quantification of goals is the goal/question/metric (GQM) paradigm[2, 4, 6, 7] developed to help us define the areas of all kinds of studies, in particular studies concerned with improvement issues, and help in the interpretation of the results of the data collection process. The paradigm does not provide a specific set of goals but rather a framework for stating goals and refining them into specific questions about the software development process and product that provide a specification for the data needed to help answer the goals.

Using this paradigm, the **process of quantifying improvement goals** consists of three steps:

1. **Generate a set of goals based upon the needs of the organization.**
   The first step of the process is to determine what it is you want to improve. This focuses the work to be done and allows a framework for determining whether or not you have accomplished what you set out to do. Sample goals might consist of such issues as on how to improve the set of methods and tools to be used in a project with respect to high quality products, customer satisfaction, productivity, usability, or that the product contains the needed functionality.
2. **Derive a set of questions of interest or hypotheses which quantify those goals.**
   The goals must now be formalized by making them quantifiable. This is the most difficult step in the process because it often requires the interpretation of fuzzy terms like quality or productivity within the context of the development environment. These questions define the goals of step 1. The aim is to satisfy the intuitive notion of the goal as completely and consistently as possible.

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

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

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

- **Perspective**:
Examine the (cost, effectiveness, correctness, errors, changes, product metrics, reliability, etc.) from the point of view of the (developer, manager, customer, corporate perspective, etc) E.g. Examine the effectiveness from the developer's point of view.

- **Environment**:
The environment consists of the following: process factors, people factors, problem factors, methods, tools, constraints, etc. E.g. The product is an operating system that must fit on a PC, etc.

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

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

- **Product Questions**
For each product under study there are several subgoals that need to be addressed. These include the definition of the product (characterize the product quantitatively) and the evaluation of the product with respect to a particular quality (e.g. reliability, user satisfaction)

The definition of the product consists of:

1. Physical Attributes. e.g. size (source lines, number of units, executable lines), complexity (control and data), programming language features, time space.
2. Cost. e.g. effort (time, phase, activity, program)
3. Changes. e.g. errors, faults, failures and modifications by various classes.
4. Context. e.g. customer community, operational profile. The improvement is relative to a particular quality e.g. correctness. Thus the physical characteristics need to be analyzed relative to these.

The improvement goals and questions in the appendix were derived by applying this template.

The idea of basing sound software development on precise formulation of project goals or objectives is not new; it is related to a number of approaches, e.g., Boehm's 'Goal-Oriented Approach to Life-cycle Software (GOALS)'[13] and Gilb's 'Multi-Element Component Comparison and Analysis Method (MECCA)'[14]. However, there are major differences between these approaches and our improvement methodology based on the GQM paradigm. In Boehm's approach, major project goals are identified by using a 'software engineering goal structure' and means for achieving those goals are defined. This approach corresponds to the setting up of goals in our improvement methodology (see step 2). The GQM approach provides support for generating goals in a more formal way (see our goal templates) and deriving quantifiable questions and metrics (see our process and product related templates). Gilb's approach is closer to our GQM approach. However, the two major differences are that the GQM approach formalizes the refinement of high-level goals into metrics, and permits the interpretation of measurement results in the context of a particular project environment by allowing for subjective metrics in addition to objective metrics.

CHARACTERIZING THE ENVIRONMENT

Characterizing the environment was one of the subgoals in applying the GQM paradigm to characterizing improvement goals. The environment was characterized in terms of subjective metrics such as 'to which degree were certain methods or tools used by the project personnel'. The problem with these subjective metrics is that it is hard to choose methods and tools based solely on such unprecise criteria.

It is our goal to characterize the project environment as objectively as possible. The approach chosen in this paper is use error, fault, and failure profiles for characterizing the environment in a quantitative way. We are actually measuring the impact of the environment on the quality of the software process and its resulting products. This indirect characterization has the advantage of objectivity. We can either use actually measured defect profiles or, if measurement results are not available, hypothesized defect profiles. All changes in a project environment can expected to be reflected in changing defect profiles. Unfamiliarity with the application domain can be expected to result in more application errors, using a set of new concepts for structuring software, e.g. using Ada as implementation language, can be expected to result in more problem-solution errors.

Assuming we know the effect of certain methods and tools on defect profiles, it should be relatively easy to tailor

the set of methods and tools to cope with defect profiles of a particular environment.

CHARACTERIZING METHODS AND TOOLS

The effectiveness of the improvement methodology depends on the amount of knowledge we have on the impact of methods and tools on defect profiles. Unfortunately, we do not have enough such knowledge yet. Most of the available knowledge is extremely environment dependent.

We have to start creating environment specific knowledge concerning the effect of methods and tools. Where not enough knowledge is available in terms of measured results, we have to add hypotheses in order to start using the proposed methodology effectively. As we apply the improvement methodology we increase our initial knowledge based on analysis results derived during step 4 of our methodology. Our goal must be the refinement of existing knowledge and the substitution of actual analysis results for hypotheses.

Tables 1, 2, and 3 describe the impact of a small set of methods and tools on preventing errors and detecting faults. This knowledge is mostly based on actual measurement results as far as detection is concerned[8], and hypotheses as far as prevention is concerned. We selected methods and tools which are either currently used or are candidates for future use in the NASA/SEL environment. In most case the names of the methods and tools are self-explanatory. However, the reuse method employed at NASA/SEL needs some explanation: In the NASA/SEL environment applications of similar type are developed over and over again; therefore, not only code modules but especially whole specifications and designs are reused with modifications. Without knowing these specifics of the reuse method used at NASA/SEL the impact of reuse in table 3 might look much too positive. Both measurement results and hypothesis (see tables 1, 2, and 3) are NASA/SEL specific. Therefore, the characterizations in these tables may vary significantly for different environments. However, we expect the general pattern to be more or less preserved.

Table 1
Fault Detection classified according to Scheme 2

METHODS + TOOLS	Fault Classes	
	Omission	Commission
Functional Testing	+	+
Structural Testing	-	o
Code Reading (by stepwise abstraction)	+	+
Syntax Directed Editor	-	o

The impact of methods and tools is determined on a subjective scale (--, -, o, +, ++). Characterizing the effect of a method or tool with respect to a particular defect class as '--' means that this method or tool is never able to detect or prevent defects of this type, as '-', that it is unlikely that this method or tool will detect or prevent defects of this type, as 'o', that it is possible that this method or tool will detect or prevent defects of this type, as '+', that it is likely that this method or tool will detect or prevent defects of this type, and as '++' that it is certain that this method or tool will detect defects of this type. It is evident that only the effect of (automated) tools can be classified as '--' or '++'; for all (non-automated methods there is never a guarantee that they will never or always detect or prevent certain types of defects due to the fact that the ability of human beings is a deciding factor.

Table 2
Fault Detection classified according to Scheme 3

METHODS + TOOLS	FAULTS						
			Data			Interface	
	Control	Comput.	Def.	Init.	Use	Global Data	Other
Functional Testing	+	+	-	+	-	-	-
Structural Testing	o	o	-	o	-	-	-
Code Reading (by stepwise abstraction)	o	+	-	+	-	o	o
Syntax Directed Editor	-	-	-	+	o	-	-
Tool for keeping track of common data + references	-	-	-	-	-	+	-

Table 3
Error Prevention classified according to Scheme 2

METHODS + TOOLS	Error Classes						
	Appl.	Prob.-Sol.	Sem.	Synt.	Env.	Inf.Magmt.	Clerical
Training wrt. Application	+	+	o	o	o	o	-
Training wrt. Language/ Environment	-	-	+	+	+	-	-
Chief Programmer Team	+	+	o	o	o	+	+
Document Library	-	-	-	-	o	+	+
Configuration Control (automated)	-	-	-	-	+	+	+
Reuse	+	+	+	+	o	-	o
PDL Design Lang _e		o	+	+	-	+	o
PDL Processor	-	+	+	+	-	+	+
Syntax Directed Editor	-	-	o	++	-	-	+
Data Abstraction	-	+	o	-	o	o	-

APPLICATION OF THE TAILORING PROCESS

The presented improvement methodology including the approach to characterizing goals, environment, and methods and tools by defect profiles was applied to a characteristic project in the NASA/SEL environment. The project was analyzed after completion, and based on the analysis results recommendations were made for future projects of the same class.

Some of the results of this improvement process are presented according to the five steps of the improvement methodology:

- **Step 1**: The project is characteristic for the class of ground support systems developed at NASA. Projects of this class were built several times before; therefore, a very high amount of code was reused from these previous projects. The software process for these class of systems is well established; whereas the process model was not changed over time, the set of methods and tools was fine-tuned to the application from time to time. The management personnel (first line managers and above) are extremely experienced in this class of projects, whereas lower-level personnel frequently changes. Based on the continuity at the management level, managers understand the design of the systems very well. The development process is not supported by a very high number of automated tools; this fact is currently changing in the NASA environment. An important characteristic of this class of projects is the fact that the managers are very familiar with the future use of their systems. As a consequence, a testing method for system and acceptance test was established, whose termination criterion is not decreasing mean-time-between-failures but just the completion of the set of test cases derived from this knowledge concerning future use of the system.

Table 4
Error Profile according to Scheme 2

ERROR CLASS	PERCENTAGE
Application	5%
Problem-Solution	58%
Semantics	8%
Syntax	3%
Environment	2%
Information Management	5%
Clerical	17%

Looking at the error profiles in table 4, we recognize a low number of application errors, a high number of problem-solution errors, and a high number of clerical errors. The number of application errors reflects the extreme familiarity with the application; the number of problem-solution and clerical errors can be explained by the relative inexperience of the the lower-level project personnel. The high number of errors occurring during the design or coding of a single component (see table 5) supports the hypothesis that the high number of problem-solution errors in table 1, can, in fact, be linked to the inexperience of the lower-level personnel.

Table 5
Error Profile according to Scheme 1

ERROR CLASS	PERCENTAGE
Requirements	5%
Specification	3%
Design or Implementation	
- of a single component	78%
- of more than one component	4%
Use of Language	8%

[This classification scheme is slightly different from error scheme 1; data for error scheme 1 were not available for this project. As opposed to classifying errors by the time of their occurrence, here they are classified by the project aspects affected: requirements, specification, design or implementation, and use of language.]

Table 6
Fault Profile according to Scheme 2

FAULT CLASS	PERCENTAGE
Omission	22%
Commission	76%

Table 7
Fault Profile according to Scheme 3

FAULT CLASS	PERCENTAGE
Control	13%
Computation	16%
Data	30%
Interface	
- global data	13%
- other	20%

The fault profile in table 6 reveals a percentage of omission faults (22%) which is lower than the average in this class of projects (this base line data is not included in the tables). One explanation is the very high percentage of reuse in this project.

The fault profile in table 7 supports findings reported by Basili and Perricone[8], that reuse results in a lower number of control flow faults. According to the same study, the high percentage of data faults is due to the inappropriate method for writing specifications; these specifications made it hard to understand differences between old algorithms (from previous projects) and new algorithms (required for the current project). The number of global data faults, even in a Fortran project, seems to be unnecessarily high.

Failure profiles could not be measured for this class of projects. NASA manages to have almost no failures during operation. This fact is due to a very thorough testing process and the perfect knowledge concerning future use of those systems.

• **Step 2**: The project goals for this class of systems in the NASA/SEL environment are to produce highly reliable systems and to produce them on time. The improvement goals are to decrease error and fault classes which were identified as overrepresented in step 1 by changing the set of methods and tools.

• **Step 3**: Recommendations for future projects based on lessons learned from the analysis of this project are: A number of recommendations for future projects could be made based on lessons learned from the analysis of this particular project. The fact that these recommendations are not very surprising does not affect the importance of the analysis results. The objectivity of quantitative analysis results, even if they are not surprising, increase the credibility of these results and the justification of the improvement recommendations based upon these analysis results. Another advantage of quantitative analysis results is that they might allow evolutionary improvement by revealing the problem sources rather than improper improvement recommendations or revolutionary improvement.

- Train (lower-level) personnel better with respect to algorithms and technologies to be used; use studies of solutions of this class of problem. This approach promises to lower the number of problem-solution errors.
- Integrate more automated tools into the software process for preventing clerical errors; candidate tools (according to table ??) are configuration control tools, PDL processors, and syntax-directed editors.
- Indications that reuse lowers the number of omission faults suggest to encourage the implementation of reuse strategies in future projects. The detection of omission faults is very difficult; therefore, reuse as a prevention method is even more important.
- Better specification methods and tools should be introduced in order to decrease the number of data faults due to misunderstanding of the specifications written according to the currently used method.
- The high number of global data faults is mostly due to changes in common data structures without updating all references. It should be easy to implement a tool keeping track of all common data structures and related references. In the case of changing data structures all affected references could be updated.

These recommendations promise to improve the development of future systems of the same class. This assumption has to be verified by performing steps 4 and 5 of the improvement methodology in future projects.

TOOL SUPPORT FOR THE IMPROVEMENT METHODOLOGY

All steps of the methodology for choosing, evaluating, and improving process models and their support by methods and tools require automated support. In 1986 we started the TAME (Tailoring A Measurement Environment) project which aims at the development of a prototype environment to support all kinds of quantitative evaluations.

The objective of the TAME prototype is to support quantitative and qualitative evaluation of Ada projects (process and product aspects) in the framework of the GQM paradigm. This includes (1) setting up the environment for evaluation (deriving goals, questions, metrics, establishing protection mechanisms), (2) conducting the actual measurement and evaluation activities, and (3) maintaining a historical database. In the long-run such a system could become an integral part of a comprehensive Software Development Environment.

The requirements for the TAME system provide for many features which assist the user in all kinds of measurement activities, including those required in the context of this methodology. These features include:

- generating evaluation goals, questions, and metrics.
  Goal-oriented evaluation will be conducted in the context of the GQM paradigm. The formulation of specific goals and corresponding questions is not an easy task; the TAME system will give assistance in performing this task.
- collecting data.
  The metrics or distributions necessary for addressing particular evaluation questions may originate from different sources, e.g., forms filled out by development or maintenance personnel, source code, all kinds of documents, running systems. The computation of the metrics is performed by a set of measurement tools analyzing these raw data, such as static code analyzers. The TAME system will support inputing and storing the raw data and computing the metrics required for evaluation purposes.
- validating collected data.
  All collected data (especially those collected by forms) are subject to errors. The system cannot guarantee completeness and correctness in a strict way. For example, how should the system judge whether the reported schedule for completing some development task is correct or not? However, it can guarantee partial completeness and consistency; e.g., it can check that the schedule for completing all modules of a system is consistent with the schedule of the whole system.
- storing data in a data repository.
  All data have to be stored in a data repository as soon as collected. Data have to be identifiable according to various criteria, e.g., when collected, from which source (type of document, version, product name, etc.), time period covered. In addition, the system has to maintain consistency of the data repository.

- retrieving information for answering particular evaluation questions.
  The TAME system will provide a basis for answering the user's evaluation questions based on information available in the data repository.
- evaluating data.
  The TAME system will provide goal-directed interpretation and evaluation of data according to an a priori established framework (see the first feature)).
- running statistical analysis.
  The TAME system will provide statistical analysis packages for computing statistical significance of evaluation results.
- maintaining a historical knowledge base.
  The TAME system will create and maintain a historical data base over time. The purpose of this data base is to allow better interpretations of analysis results relative to historical baselines reflecting the characteristics of a particular environment. Whereas all input into the database (see the fourth feature) is related to data regarding individual systems, maintaining a historical database requires an additional dimension by creating base-lines across systems or even environments.

A macroscopic view of the TAME architecture shows the system divided into four hierarchically organized layers:

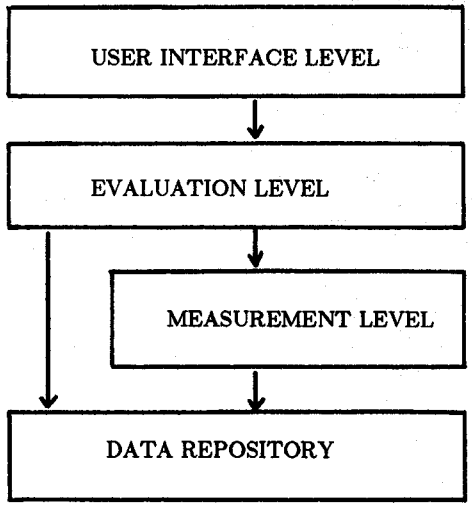

Fig. 4. TAME Architecture

1. The **User Interface Level** implements the appropriate means of interaction between users and TAME. In addition, the user interface level contains a tool for setting up the measurement and evaluation environment for each individual user (= creating or tuning an appropriate instance of the evaluation level). An important part of this measurement and evaluation environment is the actual set of goals, questions, and metrics.
2. The **Evaluation Level** implements the appropriate environment (probably set up by the user interface level). Such an environment is characterized by goals, questions, metrics, and interpretation procedures, as well as a protection profile which defines legal access paths to the measurement and data repository level for this particular user. This level triggers the computation of the appropriate metrics (by either activating the appropriate measurement tools or by accessing the data repository level), and provides adequate interpretation. A separate instance of this level might exist for each individual user.
3. The **Measurement Level** consists of tools for computing metrics. Examples of such tools are tools for computing data binding metrics, structural coverage metrics, or complexity metrics.
4. The **Data Repository Level** provides the infra-structure for various types of evaluation. This level allows storing and retrieving all kinds of software related data. This level should be as independent as possible of a particular data base management system or a concrete data base structure; the data repository should be implemented as an abstract data type hiding all these implementation details.

Another important general requirement for this TAME data repository is to be **flexible** in various ways; the data repository must allow

- changing (if possible extending) the data base structure of the repository level,
- changing the access procedures to the repository level,

without affecting existing 'user' programs (measurement tools, evaluation programs) more than absolutely necessary. To make it clear, by flexibility of the repository level we do not mean that the repository level may not be changed in the case of data base changes; what we mean is, that in this case ONLY the repository level has to be changed, while retaining the 'user' programs (measurement tools, etc.) without changes.

An interesting aspect from a research perspective is the fact that this project requires and provides opportunities for cooperation between software engineering, data base, and artificial intelligence. For more details concerning the TAME project the reader is referred to TAME project reports[10,11]

SUMMARY AND FUTURE RESEARCH

Various versions of the improvement methodology have been applied in several industrial settings. The basic approach has evolved from the work performed in the NASA/SEL environment, where most of the defect classification schemes presented in this paper were developed and applied to improve the development environment. This methodology is expected to be refined in the future based on experience from future applications.

Using defect profiles for characterizing improvement goals and environments and tailoring methods and tools towards these quantified goals and environments has proven to be feasible and beneficial. However, as indicated in the introduction, defects are only one approach of many to characterize improvement goals, environments, and the impact of methods and tools. We will continue to improve defect classification schemes for the purpose of characteriza-

tion as well as investigating alternative approaches to characterization.

The TAME prototype is expected to support all kinds of measurement, analysis, and evaluation needed in the context of this tailoring approach. In the long-run the TAME project is expected to derive guidelines for future software development environments. Those future software development environments are expected to include the software process itself as one variable. Those environments are expected to recognize the fact that quality software can only be built in a productive manner if the process for building the software is tailored to the particular project goals and environments in a natural way. Future software development environments will not provide just a set of construction tools (as most state of the art development environments to today) or support one particular process model. Instead they will provide (1) the flexibility of choosing the appropriate global process model and tailoring it to specific project goals and environment characteristics, (2) the flexibility of choosing methods and tools (for construction and evaluation) which fit into the defined process model framework, (3) support for tracing quality and productivity throughout the process in a quantitative way, and (4) support for evaluating the effectiveness of the chosen software process model as well as individual methods and tools as far as meeting quality and productivity goals are concerned. The latter evaluation activity can be performed on-line for the purpose of providing feedback into ongoing project or post-mortem for the purpose of learning for future projects.

A first TAME prototype is currently built for an Ada environment[12]. There are many reasons for this decision. (1) NASA is considering Ada as the language for building Space Stations, (2) there is a thrust towards developing programming support environments for Ada, and (3) it is believed that more and more environments will move from traditional languages to Ada as the implementation language and have to confront the problem of choosing an appropriate process model including methods and tools. In this context the tailoring of software process models will be very important; it can be expected that Ada environments will not only differ from traditional environments in the sense that different methods and tools are going to be used, they might also require completely different process models.

Future use of this methodology will result in accumulating more and more knowledge concerning the impact of methods and tools on various defect types; this in turn will make the tailoring methodology more effective. In addition, the TAME prototype will be an incentive and vehicle for applying the methodology in various industrial environments.

ACKNOWLEDGEMENTS

The authors would like to thank Frank McGarry of NASA/Goddard Space Flight Center and Dr. David M. Weiss of the Office of Technology Assessment for their helpful comments on earlier versions of this paper, and the reviewers for helping us better express our ideas.

REFERENCES

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

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

[3] V. R. Basili, B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," Communications of the ACM, vol. 27, no. 1, January 1984, pp. 42-52.

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

[5] V. R. Basili, E. E. Katz, N. M. Panlilio-Yap, C. Loggia Ramsey, S. Chang, "Characterization of an Ada Software Development," IEEE Computer, vol. 18, no. 9, September 1985, pp. 53-65.

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

[7] V. R. Basili, "Measuring the Software Process and Product: Lessons Learned in the SEL," in Proc. Tenth Annual Software Engineering Workshop, NASA Goddard Space Flight Center, Greenbelt MD 20771, December 1985.

[8] V. R. Basili, R. W. Selby, Jr., "Comparing the Effectiveness of Software Testing Strategies," Technical Report TR-1501, Dept. of Computer Science, University of Maryland, College Park, May 1985.

[9] V. R. Basili, A. J. Turner, "Software Development Process Models," Technical Report, Department of Computer Science, University of Maryland, College Park, MD, forthcoming.

[10] V. R. Basili, H. D. Rombach, "The TAME Project: Motivation, Background, Ideas and Objectives," Technical Report TR-1764, TAME Report TAME-TR-1-1987, Department of Computer Science, University of Maryland, January 1987.

[11] V. R. Basili, M. Daskalantonakis, A. Delis, D. Doubleday, L. Mark, K. Reed, H. D. Rombach, D. Stotts, J. A. Turner, S. Wang, L. Wu, S. Xiao-Hong, "The TAME Project: Requirements and System Architecture," Technical Report TR-1765, TAME Report TAME-TR-2-1987, Department of Computer Science, University of Maryland, January 1987.

[12] V. R. Basili, H. D. Rombach, "TAME: Tailoring an Ada Measurement Environment," Proc. of the Joint Ada Conference, Arlington, VA, March 16-19, 1987.

[13] B. W. Boehm, "Software Engineering Economics," Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[14] T. Gilb, "Software Metrics," Winthrop Publishers, Inc., Cambridge, Massachusetts, 1977.

[15] "IEEE Standard Glossary of Software Engineering Terminology," IEEE, 342 E. 47th St., New York, Rep.IEEE-Std-729-1983, 1983.

[16] W. L. Johnson, St. Draper, E. Soloway, "Classifying Bugs is a Tricky Business," in Proc. Seventh Annual Software Engineering Workshop, NASA, Goddard Space Flight Center, Greenbelt MD 20771, December 1982.

[17] M. Lipow, "Prediction of Software Failures," The Journal of Systems and Software, vol. 1, no. 1, 1979, pp. 71-76.

[18] T. J. Ostrand, E. J. Weyuker, "Software Error Data Collection and Categorization," in Proc. Seventh Annual Software Engineering Workshop, NASA, Goddard Space Flight Center, Greenbelt MD 20771, December 1982.

[19] "Software Engineering Laboratory (SEL): Data Base Organization and User's Guide," NASA, Goddard Space Flight Center, Greenbelt MD 20771, SEL-81-102, July 1982.

[20] R. H. Thayer, A. Pyster, and R. C. Wood, "The Challenge of Software Engineering Project Management," IEEE Computer Magazine, vol. 13, no. 8, August 1980, pp 51-59.