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

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

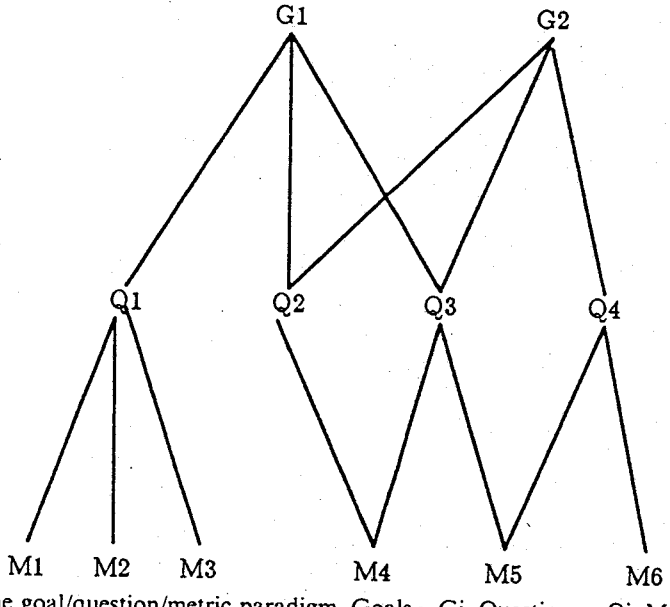

Fig. 1. The goal/question/metric paradigm. Goals = Gi, Questions = Qi, Metrics = Mi.

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

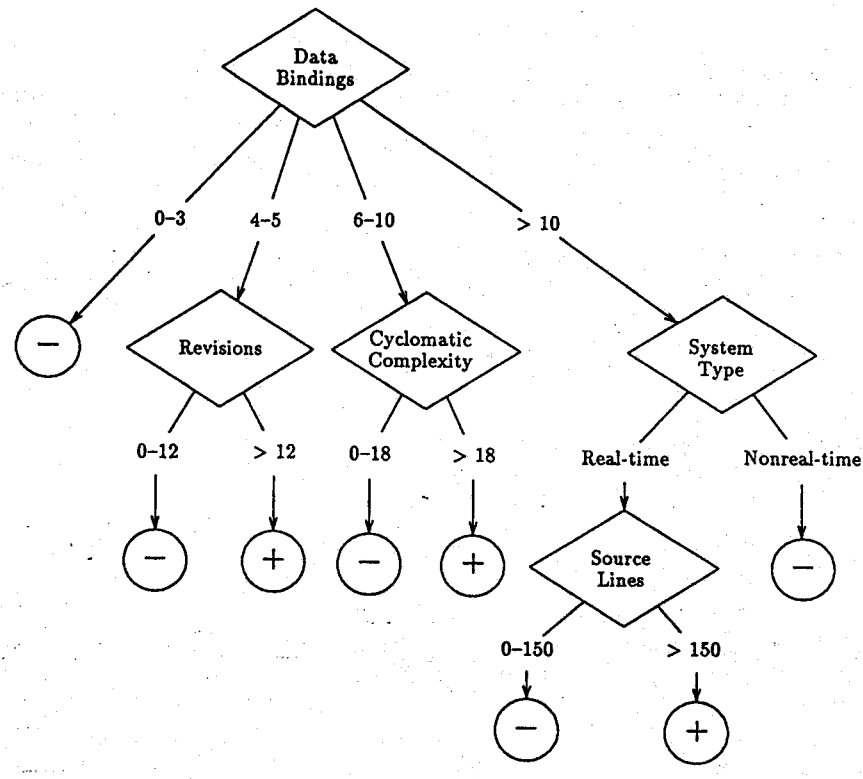

Fig. 5. Example (hypothetical) software metric classification tree. There is one metric at each diamond-shaped decision node. Each decision outcome corresponds to a range of possible metric values. Leaf nodes indicate whether or not an object is likely to have some property, such as high error-proneness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

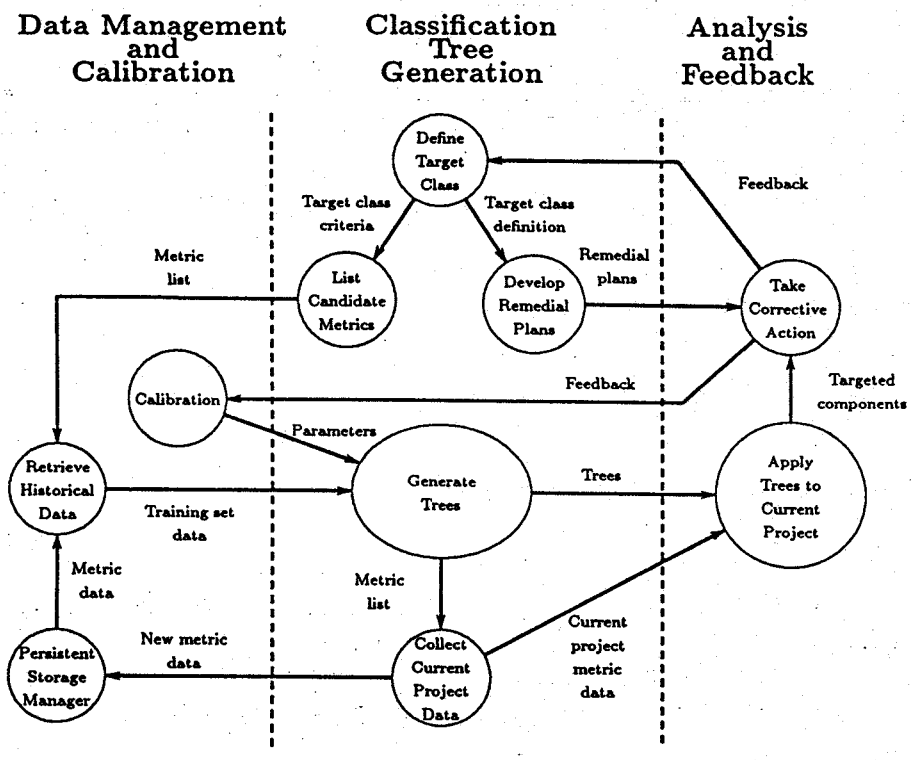

Fig. 6. Overview os classification tree approach.

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

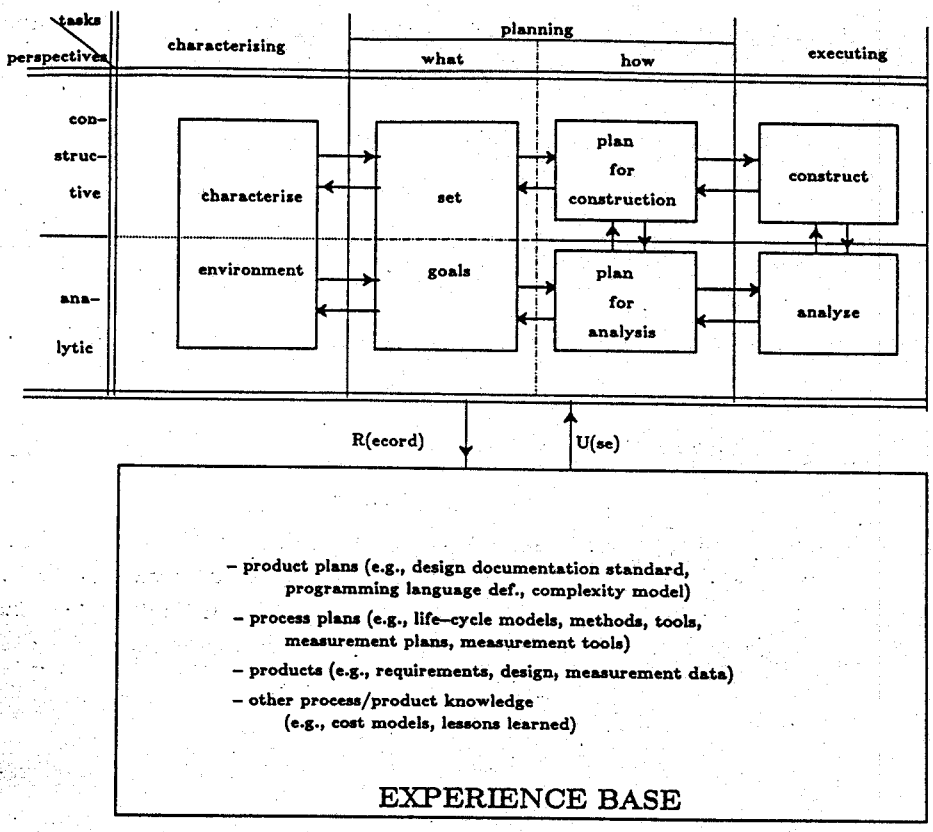

Fig. 7. The *TAME* system.

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).