Using Measurement-Driven Modeling to Provide Empirical Feedback to Software Developers

Adam A. Porter

Department of Computer Science
University of Maryland
College Park, MD 20742
Internet: aporter@cs.umd.edu

October 1992

Abstract

Several authors have explored the application of classification methods to software development. These studies have concentrated on identifying modules that are difficult to develop or that have high fault density. While this information is important it provides little help in determining appropriate corrective action. This article extends previous work by applying one classification method, classification tree analysis (CTA), to more a fine-grained problem routinely encountered by developers. In this article we use CTA to identify software modules that have specific types of faults (i.e Logic, interface, etc.)

We evaluate this approach using data collected from six actual software projects. Overall, CTA was able to correctly differentiate faulty modules from fault-free modules in 72% of the cases. Furthermore, 82% of the faulty modules were correctly identified. We also show that CTA outperformed two simpler classification strategies.

1 Introduction

One fundamental problem confronting software process improvement is how to best allocate tightly constrained resources to obtain the highest quality product. This is particularly important for large scale systems that undergo continuous enhancement or when subcontractors provide independent validation and verification. In these cases, short release intervals and fixed resources, prohibit revolutionary changes to a system. Instead, resources must be used to maximize evolutionary improvement.

A common view of the software lifecycle is that each phase consists of a construction activity followed by a review of the resulting products (i.e testing, analysis, inspections). Review processes offer significant opportunities for incremental process improvement because they account for a large expenditure of resources[Mye79]. Also, improved review processes will not disrupt later activities, while providing added assurance that upstream processes are performed adequately.

To improve review processes, resources must be used as efficiently as possible. Several recent articles argue that review processes can be improved by systematically focusing effort on high-risk modules [She91][MIO87][BP84][Mye79]. In practice, managers frequently concentrate resources on modules (e.g. subsystems, components, threads) that they deem to be high-risk. To do this, however, they must rely on their subjective experience. In this article, we address the problem of automatically isolating high-risk software modules. In particular, we will use the term high-risk to denote software modules that possess some specific type of fault (i.e Interface, Logic, etc.) This definition is useful because it aids in selecting the right review methods for a particular environment.

Our general solution approach is called classification tree analysis (CTA). Classification tree analysis is a multivariate data analysis technique, that builds empirical models of faulty software modules. These models, called classification trees, are used to predict whether or not new modules will have certain types of faults. Suitable review techniques can then be applied to the software modules that are most likely to need them.

The remainder of this article examines the effectiveness of the classification tree approach for isolating faulty software modules. Section 2 provides some motivation and background information on classification tree analysis. Section 3 presents an empirical validation of CTA using data from several actual software developments. A summary of this work appears in Section 4.

2 Background

Classification tree analysis provides an automated approach for isolating problem areas in a software system. This approach analyzes historical data to model high-risk properties of interest to developers and managers. These properties might include: (a) modules that have 1 or more interface faults, (b) modules whose maintenance costs exceed 25 person-hours of effort, or (c) modules that require between 30 and 50 person-hours to construct. Each of these properties defines a "target class", which is a set of software modules likely to have that property. One classification model would be generated to classify objects that are in each of these target classes. A developer wishing to focus resources on high-payoff areas might use several classification trees to support his or her analysis processes.

Although several classification approaches have been reported in the literature, we have focused classification trees because the models are straightforward to build and interpret.

2.1 Classification Tree Analysis

Classification tree analysis is derived from the classification algorithms of ID3 [Qui86] and CART [BFOS84]. This approach provides an automated method for generating customizable, interpretable, models of software processes and objects. Unlike standard regression models, this approach places no constraints on its input data, providing a highly flexible framework for integrating symbolic and numeric attributes. The following paragraphs briefly outline the tree construction process. For a complete description of the classification tree algorithms, see [PS90].

The main goal of the classification tree process is to uncover relationships between a set of explanatory variables and a single dependent variable. The input to the tree construction process is a group of modules called the training set. For each training set module, we must have a dependent variable denoting target class membership or nonmembership. As a shorthand target class membership is often represented as "+", otherwise "-". We also require a set of explanatory variables. These are measurable attributes of the modules, their development process, or their development environment.

Classification trees are built by recursively subdividing the training set until a classification can be made. At each step, one attribute is selected as a partition. Based on the values or ranges of this attribute, the current set is partitioned into two or more subsets. An attribute will be chosen as the partition because it maximizes the homogeneity of the resulting subsets. In other words, the ideal attribute creates subsets that are composed entirely of modules with the same target class affiliation.) This process continues until no improvement can be made by further subdivision. At this point, a classification is made based on the composition of the final subsets.

Figure 1 exhibits a hypothetical, example classification tree. The internal nodes have associated with them a software attribute and the branches emanating from these nodes specify values for the attribute appearing in that node. Each leaf of the tree corresponds to a class assignment. The trees classify new software modules in the following manner: Starting at the root of the tree, a module is classified by determining its value for the attribute at the root and then following the appropriate branch. This process continues recursively until a leaf is encountered. At that time the class assignment that appears in the leaf is given to the module being classified.

In the following section we validate the the use of this approach, using data from actual software projects.

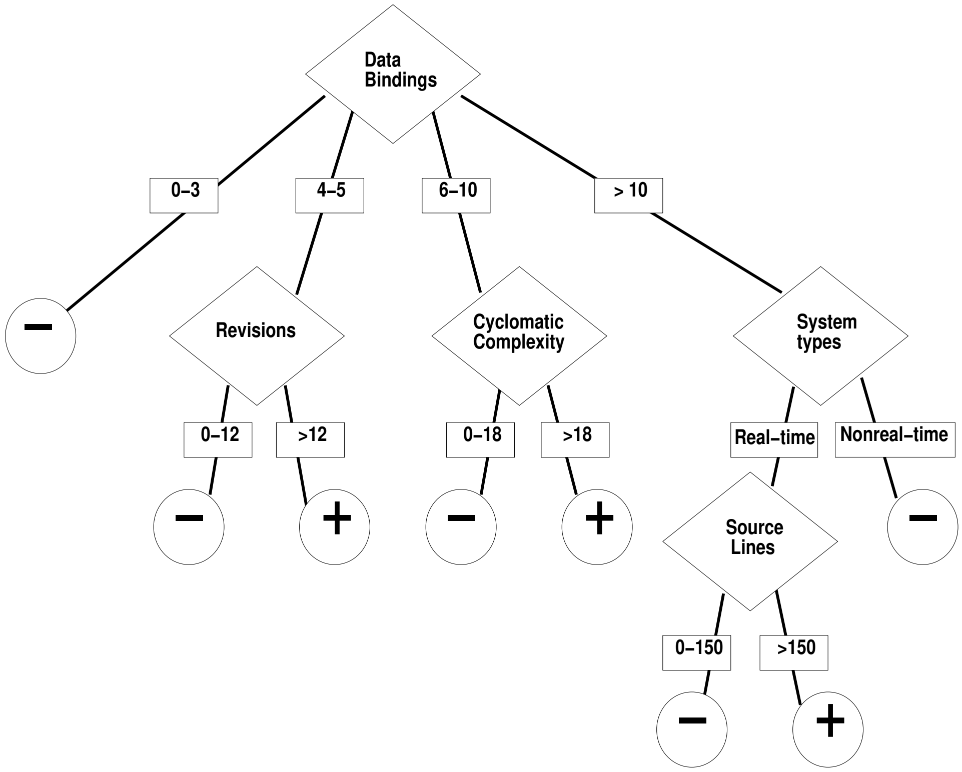

Figure 1: Hypothetical classification tree.

3 Validation Study

Several authors have explored the application of different classification methods to software development. (See [MS90],[SP88], [BBT91]) These efforts have concentrated on coarse grained problems such as identifying modules that are difficult to develop or have high fault density. While this information is important, it provides little help in determining appropriate corrective action. This article extends previous work by applying CTA to fine-grained problems routinely encountered by developers.

To assess the effectiveness of the CTA, we applied the approach to identify modules that will have some specific type of fault.

This study addresses the following major issues:

- Can classification trees correctly differentiate software modules with different types of faults?
- For a given type of fault, how well do the trees isolate all modules with that fault type?
- How many false alarms are raised?
- Would simpler strategies be more or less effective?

In this study we apply CTA to isolate modules with the following four fault types:

- Computational (fault in mathematical expression),
- Data (value or structure),

- Internal interface (module to module),
- Logic / Control structure (flow of control incorrect, missing logic), and

More complete definitions of these fault types can be found in [NAS90].

3.1 Software Environment

Data for this study was gathered from six moderate-sized projects developed in a NASA environment. [BZM+77][CMP+82][McG82]. The software is used to control and simulate the operation of unmanned spacecraft. The systems range in size from 7000 to 34000 lines of Fortran source code. Between 5 and 140 person months were expended on their development over a period of 5–24 months. The staff size ranged from 4 to 23 persons per project and there were between 99 and 366 *modules* in each system. The term *modules* refers to subroutines, functions, main programs and black data in the systems. We examined over 1,400 modules from these six systems. Four hundred eighty-two faults are cross-referenced to these modules. There are 19 attributes for each module. The attributes capture a wide range of information about the modules, including development effort, faults, changes, design style, implementation style, size and static analysis. The data spans the beginning of design specification to the end of acceptance testing. Appendix A lists the 19 attributes.

3.2 Data Collection Methods and Analysis Tools

The data collection and analysis for this study were based on a variety of data collection forms [BZM+77], programmer interviews [BW84], and static analysis tools [Dou87]. For a discussion of the data validation methods, see [BSP83] and [BW84].

3.3 Study Design

There are 24 possible pairings of projects with fault types. For each of these pairings we determined the number of software modules in the corresponding project with the indicated type of fault. In 7 of the 24 combinations, less than 14 of the total modules had the indicated fault. These combinations were not included in this study as the training sample was considered too small. Table 1 summarizes the fault statistics for the 6 projects.

One classification tree was then generated for each of the remaining 17 combinations of project p and fault type f. The generation process is as follows:

1. Training set selection. The large preponderance to low risk to high risk modules biases the algorithms towards classifying modules as low risk. To prevent this the number of low risk and high modules in the training set are made equal. First, all the modules in p with at least one fault of type f are placed into the training set. These modules are considered high-risk. From the remainder of p's modules and equal number are randomly selected and included in the training set. In this study, the number of modules in the training set ranged from 8 to 40 percent of the total modules in the project.
2. Classification tree generation. The trees are generated using the previously described algorithms.
3. Evaluation. Each module in the project is run through the classification tree. The tree predicts whether these modules are likely to have a fault of type f. The resulting predictions are compared against known fault statistics and several performance measures are calculated for the tree.

Project	Fault Type	Fault Free	Faulty
A	Logic	147	19
A	Data	132	34
B	Logic	300	22
B	Interface	302	20
B	Data	301	21
C	Logic	273	17
C	Data	255	35
C	Computation	271	19
D	Logic	81	18
D	Data	84	15
E	Logic	349	17
E	Interface	351	15
E	Data	313	53
E	Computation	337	29
F	Logic	144	18
F	Interface	148	14
F	Data	147	15

Table 1: Fault statistics

Tree Evaluation: There are two primary groups of performance measures calculated for each tree. They are:

- Accuracy: The most basic measure of model accuracy is correctness. How often does the model correctly predict whether or not a module will have a given type of fault? In practice, however, developers will be much more interested in correctly identifying the high-risk class, because corrective action will be taken on the modules. To capture these different perspectives, three accuracy measures are used in this study:
  1. **Correctness:** Percentage of modules whose target class membership was correctly identified.
  2. **Completeness:** Percentage of high risk modules correctly identified.
  3. **Consistency:** Percentage of modules predicted to be high risk that, in fact, were.
- Benchmarking. Another important question is "Could these results have been obtained by some simpler strategy?" We compared the results obtained using CTA to two simpler strategies. In both cases, we denote the number of modules predicted to be high risk by CTA as n and produce comparative samples of size n, using each of the following sampling procedures.
  - Random Sampling. Randomly select n modules.
  - Largest Modules. Select the n largest modules in terms of non-commentary source lines of code (NCSLOC).

3.4 Characterizing Model Accuracy

Correctness: The correctness measures for each tree appear in Figure 2. The mean correctness for the trees was 75.6%. Across all tree applications, 72.0% percent of the modules had their fault class correctly identified. The difference in these two numbers is due to differences in project size.

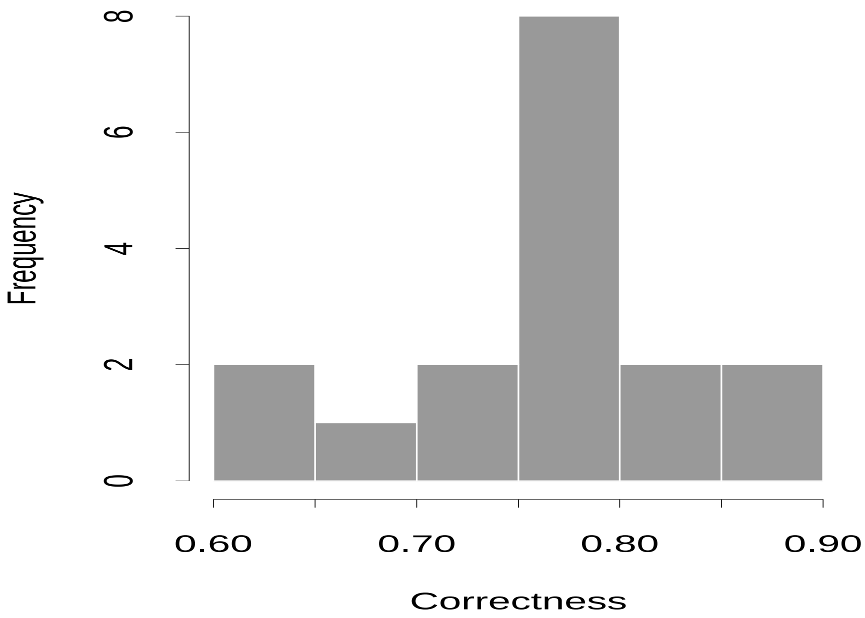

Figure 2: Classification tree correctness

Completeness: Because there is a significantly greater number of fault free modules than faulty modules. A high correctness can be achieved by always predicting that a module has no fault. Yet a developer will generally be most concerned with faulty modules. To determine how well CTA identifies faulty modules, we examine the completeness measure. The completeness measure can be thought of as correctness for the high-risk class. Completeness measures for the trees are shown in Figure 3. The mean completeness for the trees was 82.6%. Across all applications, 82% of the faulty modules were correctly identified. Again, these differences reflect differences in project size.

Consistency: Consistency is a measure of the number of false alarms raised by CTA. Here we are asking, "How many of the modules flagged as high-risk really are?" Consistency measures for the trees appear in Figure 4. The mean consistency for these trees was 27.18%. Over all the applications 17% of high risk predictions were on target.

3.5 Comparison with Simpler Methods

The accuracy measures discussed in the previous section provide insight into the performance of CTA. We still need to determine whether or not some simpler, probably cheaper, strategy would perform equally effectively. We address that question by comparing the performance of CTA against two very simple strategies.

As mentioned earlier, we are primarily concerned with effectively isolating faulty modules. We can view CTA as strategy for sampling n, hopefully faulty, modules from a given project. We would like

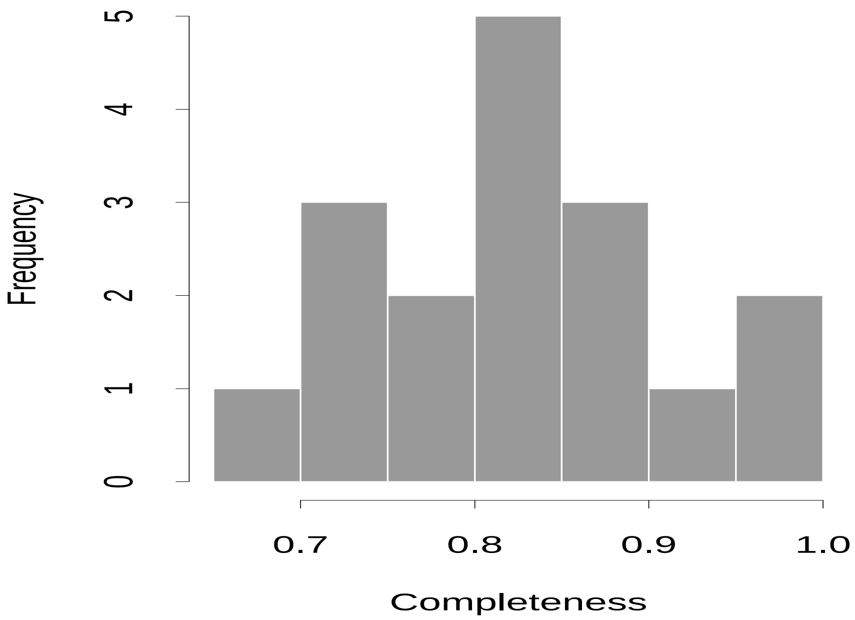

Figure 3: Classification tree completeness

to compare this approach with equal sized samples drawn with other strategies. We chose the following alternative strategies:

- Random Sampling: draw n modules at random.
- Largest Modules: draw the n largest modules in terms of non-commentary source lines of code (NCSLOC).

Figure 5 plots the completeness measures for CTA against those of the random sampling and largest modules strategies. For each trial, the completeness measure for CTA appears on the vertical axis, while the completeness for the other approaches appears on the horizontal axis. Points that lie above the 45 degree line indicate that the CTA outperformed the other method. In only one case did the largest modules strategy perform as well as CTA. In fact, for that case the classification tree generated the largest modules rule. In every case CTA outperformed random sampling. The mean completeness using random sampling was 31%. The mean completeness for the largest modules rule was 63.7% As was mentioned earlier, the mean completeness for CTA was about 82%

4 Conclusions

One central goal of our research is to use empirical methods to direct the execution of software processes. In this article we examined the application of classification tree analysis as a mechanism for providing empirically guiding software developers. To evaluate our approach, we gathered data from actual software

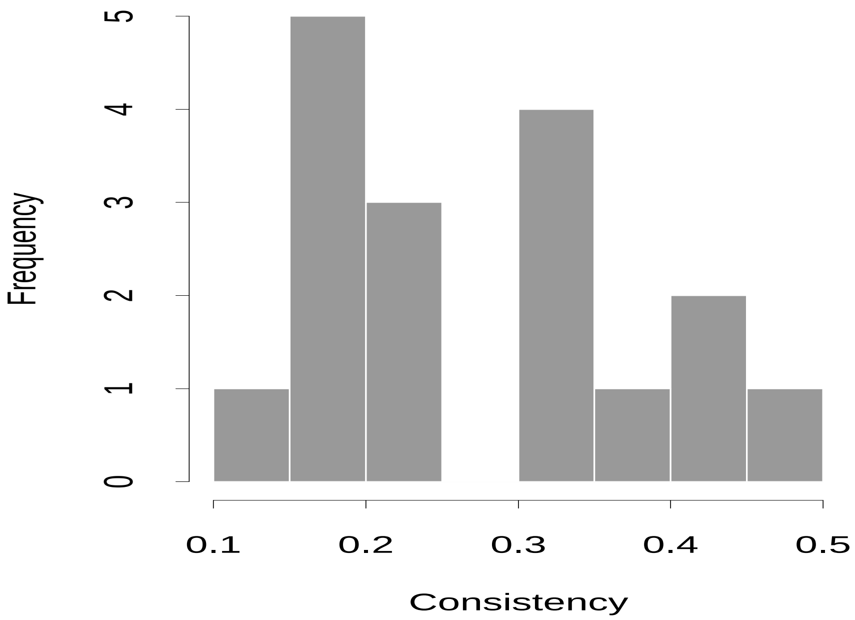

Figure 4: Classification tree consistency

projects and CTA to identify software modules that had one of four specific fault types. An analysis of the resulting models was used to validate the general approach for creating identifying tautly software modules. We also showed that CTA outperformed two simpler classification strategies.

General results that emerged from this work include:

- CTA is a feasible and effective mechanism for identifying software modules with specific types of faults.
- Classification trees are effective at identifying faulty modules even when they comprise a small portion of a software system.
- The high completeness measures indicate that the trees are successful at isolating most of the "high risk" modules. This implies that their high correctness is not a result of always classifying modules to be "low risk"
- Simpler classification strategies did not outperform CTA

The results presented in this article are based on the analysis of multiple projects in one environment. The empirical characterization is intended to provide the basis for validating the application of CTA. It is not implied that there is a direct extrapolation of these results to other projects and environments. Further work is underway to expand the scope of this analysis.

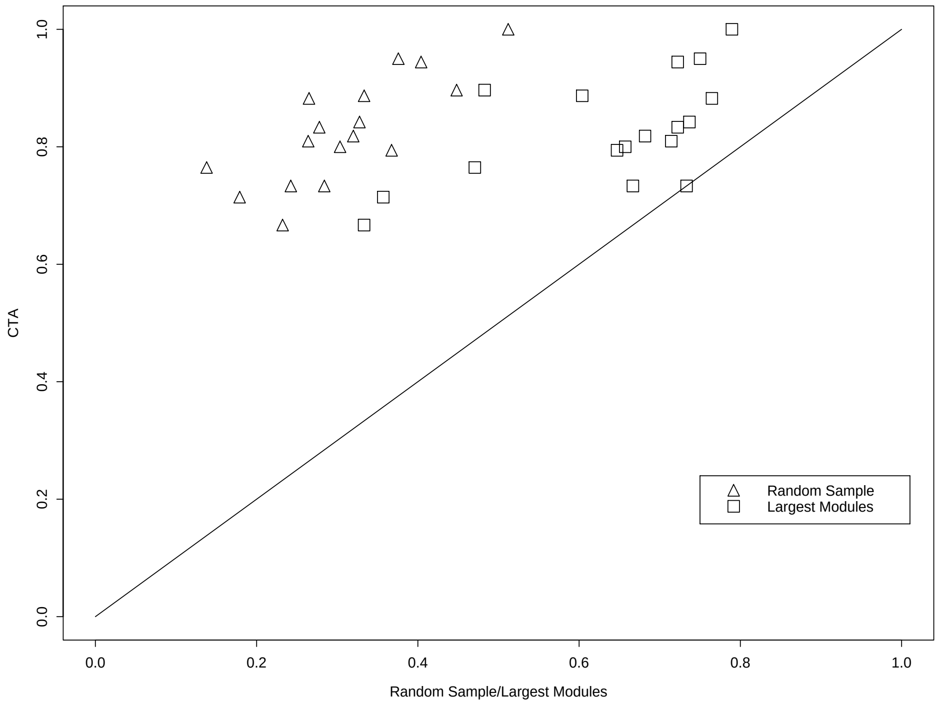

Figure 5: CTA performance vs. random sampling and largest module strategies

References

[BBT91] Lionel C. Briand, Victor R. Basili, and William M. Thomas. Recognizing patterns for software development prediction and evaluation. In *Conference on Software Engineering Economics*, McLean, Virginia, May 1991. Mitre Co.

[BFOS84] L. Breiman, J. Freidman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth, Monterey, CA, 1984.

[BP84] V. R. Basili and B. T. Perricone. Software errors and complexity: An empirical investigation. *Communications of the ACM*, 27(1):42–52, January 1984.

[BSP83] Victor R. Basili, Richard W. Selby, and Tsai Y. Phillips. Metric analysis and data validation across Fortran projects. *IEEE Transactions on Software Engineering*, SE-9(6):652–663, November 1983.

[BW84] V. R. Basili and D. M. Weiss. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, SE-10(6):728–738, November 1984.

[BZM+77] V. R. Basili, M. V. Zelkowitz, F. E. McGarry, Jr. R. W. Reiter, W. F. Truszkowski, and D. L. Weiss. The software engineering laboratory. Technical Report SEL-77-001, Software Engineering Laboratory, NASA/Goddard Space Flight Center, Greenbelt, MD, May 1977.

[CMP+82] D. N. Card, F. E. McGarry, J. Page, S. Eslinger, and V. R. Basili. The software engineering laboratory. Technical Report SEL-81-104, Software Engineering Laboratory, NASA/Goddard Space Flight Center, Greenbelt, MD, February 1982.

[Dou87] Dennis Lee Doubleday. Asap: An ada static source code analyzer program. Technical Report 1895, Department of Computer Science, University of Maryland, College Park, Maryland, August 1987.

[McG82] F. McGarry. Annotated bibliography of software engineering laboratory (sel)literature. Technical Report SEL-82-006, Software Engineering Laboratory, NASA/Goddard Space Flight Center, Greenbelt, MD, November 1982.

[MIO87] J. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, and Application*. McGraw Hill, New York, 1987.

[MS90] R. Kent Madsen and Richard W. Selby. Metric-driven classification models for analyzing large-scale software. Technical report, University of California, 1990. (submitted for publication).

[Mye79] G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, New York, 1979.

[NAS90] NASA. Software engineering laboratory (sel): Database organization and user's guide, revision 1. Technical Report SEL-89-101, Software Engineering Laboratory, NASA/Goddard Space Flight Center, Greenbelt, MD, February 1990.

[PS90] Adam A. Porter and Richard W. Selby. Empirically guided software development using metric-based classification trees. *IEEE Software*, 7(2):46–54, March 1990.

[Qui86] J. R. Quinlan. Induction of decision trees. *Journal of Machine Learning*, 1(1):81–106, 1986.

[She91] Susan Sherer. A cost-effective approach to testing. *IEEE Software*, 8(2):34–40, March 1991.

[SP88] Richard W. Selby and Adam A. Porter. Learning from examples: Generation and evaluation of decision trees for software resource analysis. *IEEE Transactions on Software Engineering*, SE-14(12):1743–1757, December 1988.

A

The 19 attributes for each software module are listed below.

Development Effort Attributes

design effort ; code effort;

Design and Implementation Style Attributes

comments ; source lines; non-commentary source lines; executable statements; cyclomatic complexity; assignment statements; module calls ; decisions; format statements; function calls ; input-output statements ; input-output parameters ; unique operands; unique operators; origin (new code, reused without modification, slightly modified, extensively modified; total operands; total operators.