Empirically guided software development using metric-based classification trees

ARTICLE *in* IEEE SOFTWARE · APRIL 1990

Impact Factor: 1.23 · DOI: 10.1109/52.50773 · Source: IEEE Xplore

Adam Porter

University of Maryland, College Park

121 PUBLICATIONS **3,592** CITATIONS

Empirically-Guided Software Development Using Metric-Based Classification Trees

Adam A. Porter and Richard W. Selby

Department of Information and Computer Science,[1]

University of California

Irvine, California 92717

(714) 856-6326

October 1989

[1]This work was 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; National Science Foundation under grant DCR–8521398; University of California under the MICRO program; Hughes Aircraft Company; and TRW.

Abstract

Metric-based classification trees provide an empirically-guided approach for identifying various classes of high-risk software components throughout the software lifecycle. Classification trees guide developers by focusing the application of specialized software analysis, testing, and construction techniques and tools. Focusing developer resources on high-payoff areas is intended to help improve quality efficiently. This paper motivates the use of classification techniques and summarizes one metric-based classification tree approach. We describe a multi-phase methodology for integrating classification tree techniques into large-scale, evolving software projects and outline an example of its use. The prototype tools supporting the methodology are environment independent and are calibrated to particular environments by measurements of past releases and projects. The feasibility and predictive accuracy of the metric-based classification tree approach has been validated using NASA and Hughes project data. The classification tree tools are part of the *Amadeus* measurement and empirical analysis system.

Contents

1 Introduction

According to the "80:20 rule", approximately 20 percent of a software system is responsible for 80 percent of its errors, costs, and rework. Boehm and Papaccio [BP88] assert:

> The major implication of this distribution is that software verification and validation activities should focus on identifying and eliminating the specific *high-risk* problems to be encountered by a software project, rather than spreading their available effort uniformly across trivial and severe problems.

Identifying problematic software components early in the development lifecycle, however, is a difficult task. A general solution approach, examined in this paper, casts this as a classification problem. Based on the measurable attributes of software components and their development processes, this approach derives models of problematic software components. The measurement-based models provide a basis for forecasting which components in the current project are likely to share the same high-risk properties. Examples of high-risk properties include error-proneness or high development cost. By employing classification techniques, software developers can localize the "troublesome 20 percent" of the system.

This paper outlines an automated method for generating measurement-based models of high-risk software modules and a methodology for its application on large-scale software projects. The proposed method, *automatic generation of metric-based classification trees*, employs software metrics from previous releases or projects to identify software components that are likely to have some high-risk property based on historical data. Software metrics are simply measures of software components and their development processes [Bas80] [BSP83]. Localizing likely software problem areas enables developers to focus their resources and tools on those software components that are likely to possess high-risk properties. Automatic classification tree generation was chosen over other classification methods because the resulting models are straightforward to build and interpret. Moreover, classification trees are customizable, using different metrics to classify different classes of components in differing software development settings.

This paper summarizes the software metric-based classification tree generation approach and outlines a multi-phase methodology for applying classification tree techniques to large-scale, evolving software systems. Classification trees are introduced in Section 2. Section 3 describes a methodology for integrating classification trees into the development process. Validation studies using the methodology are summarized in Section 4. An extended example applying the classification tree methodology appears in Section 5. Future work is outlined in Section 6.

2 Classification Trees

Classification trees provide an approach for identifying high-risk components (e.g., modules) [Boe81]. The trees use software metrics [Bas80] to classify modules according to their likelihood of having certain high-risk properties. Classification trees enable developers to orchestrate the use of several metrics, and hence, they serve as *metric integration frameworks*. Example metrics that may be used in a classification tree are: source lines, data bindings, cyclomatic complexity, data bindings per 100 source lines, and number of data objects referenced. A hypothetical tree appears in Figure 1. Developers can model those high-risk properties that interest them. For example, developers may want to identify those modules whose: (a) error rates are likely to be above 30 errors per 1000 source lines; (b) error rates are likely to be below 5 errors per 1000 source lines; (c) total error counts are likely to be above 10; (d) maintenance costs are likely to require between 25 to 50 person-hours of effort; (e) maintenance costs are likely to require between 0 to 10 person-hours of effort; or (f) error counts of error type X are likely to be above 0 (e.g., X = interface, initialization, control).

Each of these properties defines a "target class", which is the set of modules likely to have that property. A classification tree would be generated to classify those modules that are in each of these target classes. Classification trees are automatically generated using data from previous releases and projects. Generation of classification trees is based on a recursive algorithm that selects metrics that best differentiate between modules within a target class and those outside of it. [SP88]. A developer wishing to focus resources on high-payoff areas might use several classification trees to support his or her analysis process. Figure 2 provides an overview of the methodology for generating and using classification trees.

The metric-based classification tree approach has several benefits.

- Users can specify the target classes of modules to be identified.
- Classification trees are *generated automatically* using past data.
- The tree building approach is extensible — new metrics can be added.
- The trees serve as metric integration frameworks — they use multiple metrics simultaneously to identify a particular target class, and may incorporate any metric from all four measurement abstractions: nominal, ordinal, interval, and ratio.
- Classification tree analysis prioritizes data collection efforts and quantifies diminishing marginal returns.
- The tree generation algorithms are calibratable to new projects and environments using historical data called "training sets."
- The tree generation algorithms are applicable to large-scale systems, as opposed to being limited to small-scale applications.

We have developed a methodology for integrating automatic generation of metric-based classification trees into large-scale software development and evolution. This methodology is based on lessons learned in two validation studies using data from NASA [SP88] and Hughes [SP89]. A detailed discussion of this methodology appears in the following section.

3 Empirically-Based Classification Methodology

An overview of the classification methodology appears in Figure 2. The three central activities in the methodology are: (i) data management and calibration, (ii) classification tree generation, and (iii) analysis and feedback of newly acquired information to the current project. The subactivities (i.e., nodes) in Figure 2 are labeled with lower case letters, and for the remainder of this paper, these labels accompany references to the subactivities in the text. Note that the classification methodology outlined is an iterative process. 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.

3.1 Classification Tree Generation

This central activity focuses on the activities 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 [step a]. A target class such as error-prone modules, for example, could be defined as those modules whose total errors are in the upper 10 percent, relative to historical data. A list of metrics to be used as candidates for inclusion in the classification trees is created and passed to the historical data retrieval process [step c]. A common default metric list is all metrics for which data is available from previous releases and projects.

Importantly, one must determine the remedial actions to be applied to those modules classified as likely to be members of the target class [step b]. For instance, if a developer wants to identify modules likely to contain a particular type of error, then he or she should prescribe the application of testing or analysis techniques designed to detect errors of that type. Another example of a remedial plan is to consider redesign or reimplementation of the modules. Plans should be developed early in this process. This provides time to train personnel or build support tools and discourages the application of ad hoc remedies 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 [SP88] [steps d and e]. The tree construction process develops characterizations of those modules within and outside the target class based on measurable attributes of past modules [step f]. Classification trees may incorporate metrics capturing module features and interrelationships, as well as those capturing the process and environment in which the modules were constructed. Automatically collected metrics as well as subjective metrics may be included. A variety of automated metric collection tools have been developed (e.g., [DT82]). These collection tools can feed candidate metrics into the classification tree tools in order to minimize the impact of data collection on the development process. Once the trees have been generated, collection of the metrics used in the decision nodes of the classification trees should begin for the modules in the current project [step g]. This data is stored for future use and passed, along with the classification trees, to the analysis and feedback activity.

Target class definition and classification tree generation may be done at any stage of the software development and maintenance process. However, before a developer can apply a tree to classify modules on his or her current project, the metric data required by the tree needs to be collected on the project modules. E.g., a developer wants to apply the methodology at the end of the module coding and testing phase in order to identify components likely to have integration test errors. Then he or she would only include metrics in the candidate metric set that are available at or before the end of the module coding and testing phase (or else the tree may call for metrics that are not collectible on the current modules until some future phase).[2]

3.2 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. Tree generation parameters, such as the sensitivity of the tree termination criteria, need to be calibrated to a particular environment [step e]. For further discussion of generation parameters and examples of how to calibrate them[3], see [SP88] and [SP89]. Classification trees are built based on metric values for a group of previously developed modules, called a "training set" [step d]. Metric values for the training set, as well as those for the current project, are retained in a persistent storage manager [step h].

3.3 Analysis and Feedback

In this portion of the methodology, the information resulting from the classification tree application is leveraged by the development process. The metric data collected for modules in the current project is fed into the classification trees to identify modules likely to be in the target class [step i]. Remedial plans developed earlier should now be applied to those targeted modules [step j]. When the remedial plans are being applied, insights may result regarding new target classes to identify and

[2]As a starting point, developers may try applying the technique at the end of module specification or the end of module coding and testing.

[3]The calibration parameters examined in [SP88] that resulted in the most accurate trees were: the log-based, information theory evaluation function heuristic; either 25% or 40% tolerance in the tree termination criteria; octile ordinal groupings of metrics; including all available metrics in the candidate set; and including historical metric data on all available modules in the training set.

further fine tuning of the generation parameters. These insights are fed back to further target class definition and data calibration activities.

4 Validation Studies Using Classification Methodology

Preliminary tool prototypes have been developed to automatically generate metric-based classification trees. These tools embody the classification tree generation algorithms and supporting data manipulation capabilities. The tools are fragile prototypes and should be considered only preliminary versions.

One validation study has been conducted and another is underway using the classification tree methodology and supporting tool prototypes. The goal of the studies was to determine the feasibility of the approach and to analyze tree accuracy, complexity, and composition. The first study used project data from 16 NASA software systems [SP88]. In this study, the purpose of the classification trees was to identify two different target classes: those modules that had (a) high development effort or (b) high faults. In both cases, "high" was defined to mean the uppermost quartile relative to past data (i.e., the "top 25 percent"). A total of 9600 classification trees was automatically generated and evaluated based on several parameters. The 16 software systems ranged from 3000 to 112,000 source lines of Fortran. Seventy-four metrics were collected on the more than 4700 modules in the systems. The metrics captured perspectives of the modules: development effort, faults, changes, design style, and implementation style; for a complete metric list see [SP88]. On the average, the trees correctly classified 79.3 percent of the software modules according to whether or not they were in the target classes. A second study is underway to identify error-prone and change-prone components in a Hughes maintenance system (containing over 900 components and over 100,000 source lines) [SP89].

5 Example Application of Classification Tree Methodology

This section provides an extended example application of the classification tree methodology. In order to explain the example and underlying calculations completely, we have kept the number of modules in the training and test sets very small. Similarly, we have kept the number of metrics collected on each module very small. In actual practice, however, automated generation of classification trees is well suited to the volumes of data encountered in large-scale software development and evolution. The following illustrative example uses a training set of 12 modules and three candidate metrics collected on each — in [SP88], some of the training sets included over 4000 modules and over 70 metrics. See [SP88] and [SP89] for analyses using actual project data.

Our example scenario finds the management of project X concerned over a spate of newly uncovered interface errors in the latest version of the system. As project X is being developed iteratively, they decide to apply classification trees to localize modules in the forthcoming release that are likely to suffer from a large number of similar errors. The modules from earlier system releases comprise the training set. An examination of interface error data for these modules determines that three modules, **C**,**F** and **I**, each had a relatively high number of interface errors (see Table 1). The target class is defined as those modules that have at least 6 interface errors [step a]. Such modules are referred to as positive instances (or examples) of the target class. Modules having a smaller number of interface errors are labeled negative instances. At this time, management decides that they will apply additional testing and analysis to those modules in the current release that are identified as likely target class members. In particular, they decide to apply code reading by stepwise abstraction, to those future modules likely to have interface errors, since empirical studies [BS87] have indicated that that technique is effective for detecting interface errors [step b]. By outlining remedial plans prior to tree application, development personnel can receive training in the chosen technique, build necessary support tools, etc. Next, the list of candidate classification metrics should be chosen [step c]. Development personnel select three metrics. They are:

- *Module Function*,
- *Data Bindings*, a measure of module interrelationships[Sel88], and
- *Design Revisions*.

For each training set module, data for these three metrics are retrieved from an historical database [steps d and h]. These data appear in Table 2. As part of the calibration activity, the raw metric data is recoded with each metric having three possible values [step e] (see Table 3). This recoding groups the metric values into mutually exclusive and exhaustive ranges. In this example, metric values for the *Data Bindings* and *Design Revisions* metrics were statistically clustered to produce the three ranges. By definition, the *Module Function* metric returns one of three possible values. See [SP88] for further detail on calibration.

Classification tree generation now begins [step f]. Inputs to the classification tree generation algorithm include a set of modules to classify, called the training set, and a list of candidate metrics and their recoded values for the training set modules. The generation algorithm begins by selecting one metric from the candidate list. The algorithm then employs the candidate metric to generate a partition of the set of software modules. Supposing that the candidate metric has n possible values, then for $1 \leq k \leq n$, the k^{th} subset of the partition contains those modules with value k. The calibration activity for this project resulted in each metric having three outcomes, hence n=3 for each metric. After examining all the candidates metrics, one will be placed at the root of the tree and each subset created by that partition becomes a child of the root node. To determine which metric will be placed at the root of the tree, each candidate is evaluated by a metric selection function.

This function is intended to evaluate the homogeneity of the subsets resulting from using the metric as a partition. The selection function assesses the degree to which the metric partitions the modules into subsets that contain modules of the same class. Metrics yielding partitions with subsets of greater homogeneity are preferred because the purpose of classification trees is to distinguish target class members from non-members.

The metric selection function uses an evaluation function F to measure the homogeneity of a single subset. In our example,

$$F(p_i, n_i) := -\frac{p_i}{p_i + n_i} \log_2 \frac{p_i}{p_i + n_i} - \frac{n_i}{p_i + n_i} \log_2 \frac{n_i}{p_i + n_i}$$

where p_i and n_i correspond to the number of positive and negative instances in the subset[Qui86][SP88]. $F(p_i, n_i)$ returns values is the range [0,1] with smaller values indicating a greater homogeneity. Next, the metric selection function calculates a weighted average over each subset. The metric returning the lowest metric selection function value will be selected for the tree. The metric selection function used in our example, given a module set C and a candidate metric A yielding v subsets, is the following:

$$E(C, A) = \sum_{i=1}^{v} weight_i * F(p_i, n_i)$$

The variables p_i and n_i are the number of positive and negative instances of the target class in the i^{th} subset. The weighting factor, $weight_i$, is the fraction of modules in C that are elements of subset i,

$$weight_i = \frac{p_i + n_i}{|C|}$$

where $|C|$ is the number of modules in the set C.

Figure 3 exhibits a partially built tree with *Module Function* as the candidate metric. Modules having the i^{th} value are shown in the i^{th} child of the root. Modules in the target class are shown both bold-faced and underlined. *Module Function* has three possible values; which correspond to the three children of the root node.

Figures 4 and 5 display partially built trees with *Data Bindings* and *Design Revisions* as the candidate metrics. Each candidate metric is evaluated by the metric selection function. In our example, *Design Revisions* has the lowest metric selection function value and will be placed at the root of the tree. Table 4 shows the calculation of the metric selection function for *Design Revisions*, $E(\{A, B, .., L\}, Design\_Revisions)$. Figure 5 shows metric *Design Revisions* partitioning the training set into three subsets.

Each child node is now tested against a termination criteria. The simplest criteria, called zero percent tolerance, requires that tree construction end on this path when all the modules in the node are members of the same class. Our example uses the zero percent tolerance criteria. If the criteria is met, the node becomes a leaf and is labeled with the class of its members. Otherwise, the metric used at its parent node is deleted from the list of candidate metrics and the construction process is applied recursively to the node. In our example (see Figure 5), the leftmost child node meets the termination criteria, becomes a leaf, and is labeled "-". The "-" label indicates that modules following this path are not likely to be members of the target class.

Figures 6 and 7 depict the generation process continuing on the second child of the root node. Metric *Data Bindings* is selected at this node (Figure 7). We also see that each child nodes meets the termination criteria. Tree construction then continues on the third child of the root. The candidate metrics are evaluated in Figures 8 and 9. Metric *Module Function* is chosen (Figure 8), its children meet the termination criteria, and tree construction halts. Figure 10 exhibits the completed tree. Note that a metric (except for the root metric) may appear in the tree more than once, and that some of the available candidate metrics may not be used in the tree at all.

Project developers are now ready to apply the classification tree to the version of the system currently under development [step i]. When the data for the three metrics in the tree are collected for the current version of the system [step g], it is recoded, and run through the classification tree to target those modules likely to suffer from a large number of interface errors. Tables 5 and 6 list the raw and recoded data from the current system modules. To determine those modules in the current version likely to be high risk, start at the root of the tree and based on the value of the metric at that node follow one of the tree branches. Repeat this step until a leaf is encountered. A module is classified as likely to be a member of the class labeled at the leaf node. Figure 11 shows the path taken when classifying module **N** of the current system. Module **N** is targeted as likely to have a high number of interface errors, based on past project training data. Modules **M** and **O** are classified as unlikely to have a high number of interface errors. Subsequently, the corrective action planned earlier, which in this example is the application of code reading by stepwise abstraction, can be initiated on the targeted modules (i.e., **N**) [step j].

6 Concluding Remarks and Future Work

Empirically-based approaches can provide added visibility into software systems and their development processes. One manner in which this visibility can be realized is by characterizing various classes of software modules. By creating valid descriptions of module classes (i.e., those modules that are error-prone, have a high development cost, require rework, or contain some specified class of error), we have a basis for identifying similar modules in a current project. Information concerning potentially high-risk modules can then be fed back into development and maintenance processes to more effectively focus limited resources, improving the overall quality and cost of the software produced. In this paper

- we motivated the use of classification techniques in the software development process,

- discussed one classification approach,
- outlined a methodology for integrating this technique into software development and evolution processes, and
- described an extended example application of the methodology.

Further refinement of the proposed approach is underway. Some issues related to the use of the classification tree methodology that remain include:

- exploring classification tree effectiveness on a wider range of module classes,
- fine-tuning the classification tree algorithms and parameter selection, and
- integrating the classification tree toolset into the *Amadeus* system.

Amadeus is a measurement and empirical analysis system under development that will support automated techniques for measuring, modeling, and empirically analyzing large-scale software systems and processes.

Further research in these areas will aid in improving measurement-based software development and evolution processes, helping software researchers and practitioners to understand and control software development effectively.

References

[Bas80] V. R. Basili. *Tutorial on Models and Metrics for Software Management and Engineering.* IEEE Computer Society, New York, 1980.

[Boe81] B. W. Boehm. *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[BP88] B. W. Boehm and P. N. Papaccio. Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, SE-14(10):1462–1477, October 1988.

[BS87] V. R. Basili and R. W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Trans. Software Engr.*, SE-13(12):1278–1296, December 1987.

[BSP83] V. R. Basili, R. W. Selby, and T. Y. Phillips. Metric analysis and data validation across fortran projects. *IEEE Transactions on Software Engineering*, SE-9(6):652–663, Nov. 1983.

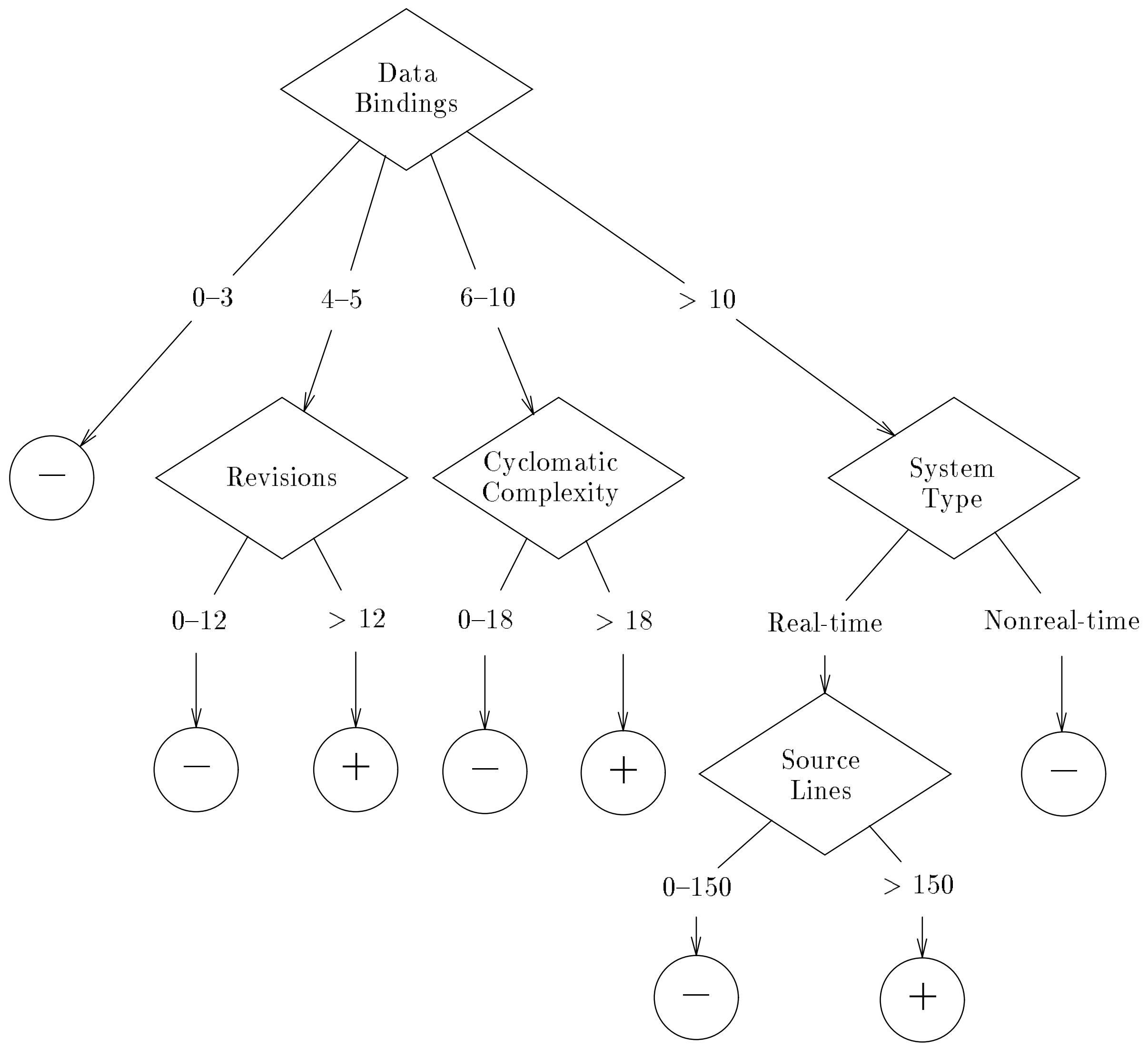
[DT82] W. J Decker and W. A. Taylor. Fortran static source code analyzer program (sap) user's guide (revision 1). Technical Report SEL-78-102, Software Engineering Laboratory, NASA/Goddard Space Flight Center, Greenbelt, MD, May 1982.

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

[Sel88] Richard W. Selby. Generating hierarchical system descriptions for software error localization. In *Proceedings of the Third Workshop on Software Testing, Verification, and Validation*, Banff, Alberta, Canada, July 1988.

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

[SP89] Richard W. Selby and Adam A. Porter. Software metric classification trees for guiding the maintenance of large-scale systems. In *Proceedings of the Conference on Software Maintenance*, Miami, FL, October 1989.



$''+''$ = Classified as likely to have errors of type X

$''-''$ = Classified as unlikely to have errors of type X

Figure 1: 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 a module is likely to have some property, such as high error-proneness or errors in a certain class.

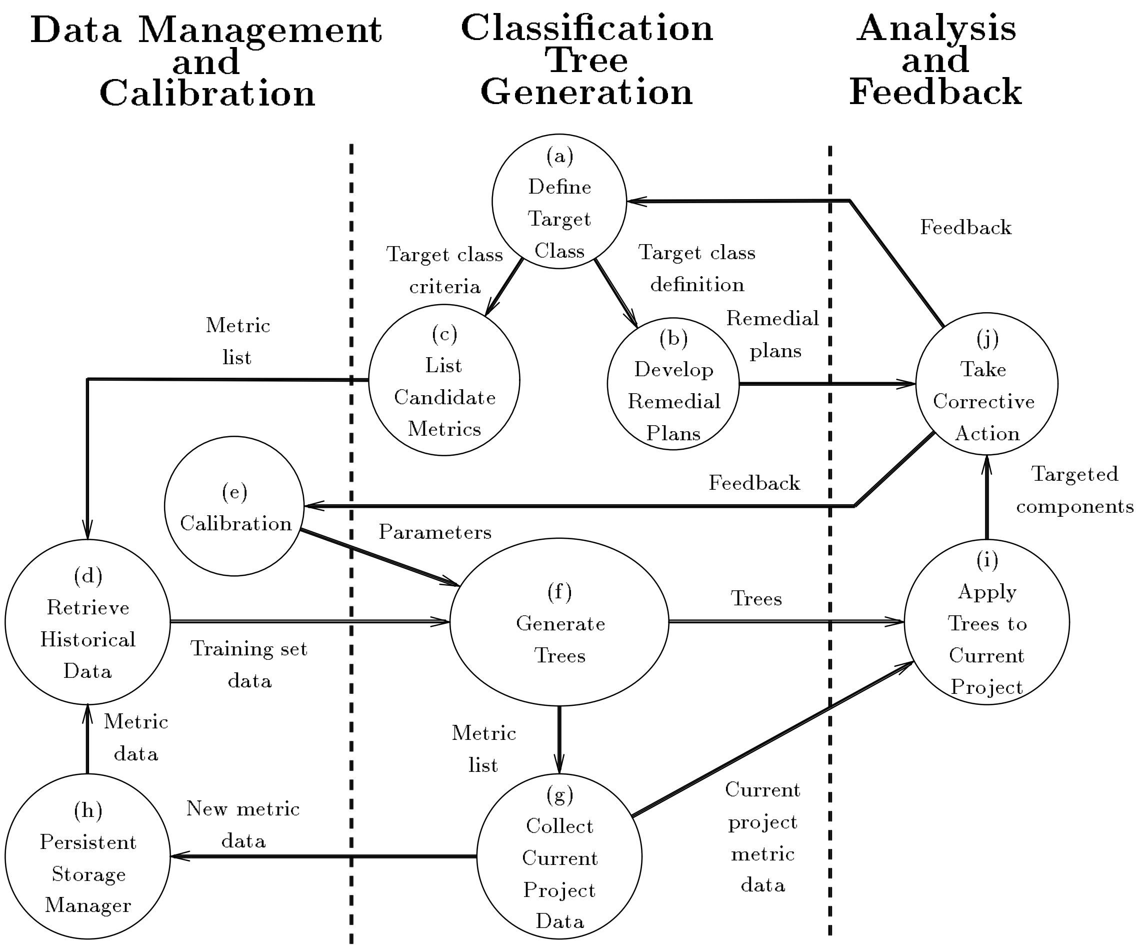

Data Management and Calibration
Classification Tree Generation
Analysis and Feedback
(a) Define Target Class
(b) Develop Remedial Plans
(c) List Candidate Metrics
(d) Retrieve Historical Data
(e) Calibration
(f) Generate Trees
(g) Collect Current Project Data
(h) Persistent Storage Manager
(i) Apply Trees to Current Project
(j) Take Corrective Action
Target class criteria
Target class definition
Remedial plans
Feedback
Metric list
Feedback
Targeted components
Parameters
Trees
Training set data
Metric data
Metric list
New metric data
Current project metric data

Figure 2: Overview of classification tree methodology.

	Module											
Metric	A	B	C	D	E	F	G	H	I	J	K	L
Interface Errors	3	2	10	1	2	9	1	3	6	2	3	0
Class	−	−	+	−	−	+	−	−	+	−	−	−

Table 1: Interface error data.

	Module											
Metric	A	B	C	D	E	F	G	H	I	J	K	L
Module Function	I	I	F	I	F	I	P	P	P	I	F	F
Data Bindings	2	9	6	13	10	15	6	15	20	4	17	16
Design Revisions	11	9	11	0	5	4	2	10	5	7	1	0
Class	−	−	+	−	−	+	−	−	+	−	−	−

Table 2: Raw training set data.

	Module											
Metric	A	B	C	D	E	F	G	H	I	J	K	L
Module Function	β	β	α	β	α	β	γ	γ	γ	β	α	α
Data Bindings	α	β	α	β	β	γ	α	γ	γ	α	γ	γ
Design Revisions	γ	γ	γ	α	β	β	α	γ	β	β	α	α
Class	−	−	+	−	−	+	−	−	+	−	−	−

Module Function	α = File management (F)	β = User interface (I)	γ = Process control (P)
Data Bindings	$\alpha = 0 \leq x \leq 7$	$\beta = 8 \leq x \leq 14$	$\gamma = x \geq 15$
Design Revisions	$\alpha = 0 \leq x \leq 3$	$\beta = 4 \leq x \leq 8$	$\gamma = x \geq 9$

Table 3: Recoded training set data.

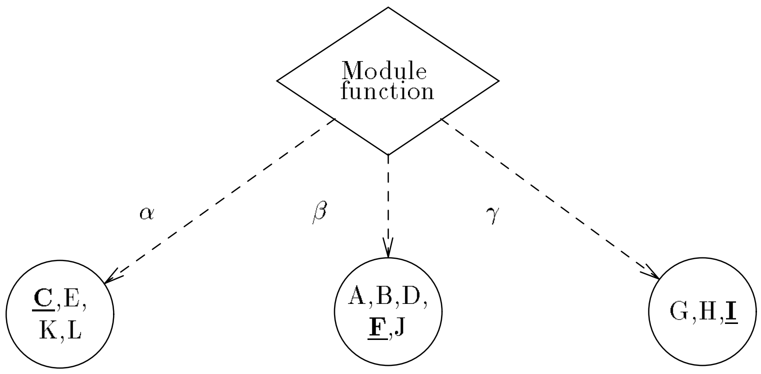


Figure 3: Partial tree using *Module Function* as the candidate metric. Metric selection function E({A,B,..,L},*Module Function*) returns 0.801. Positive target class instances are underlined.

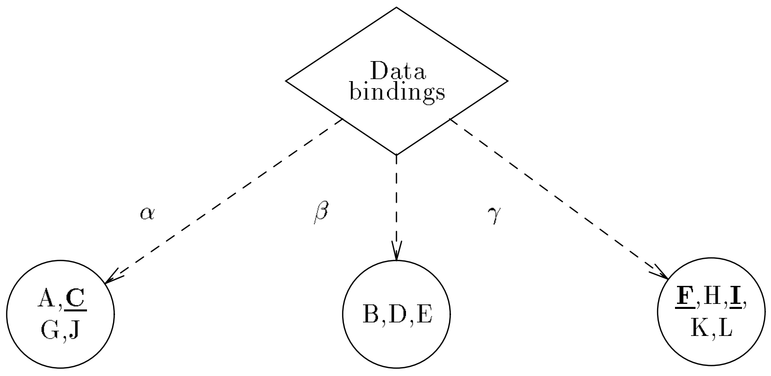


Figure 4: Partial tree using *Data Bindings* as the candidate metric. Metric selection function E({A,B,..,L},*Data Bindings*) returns 0.675. Positive target class instances are underlined.

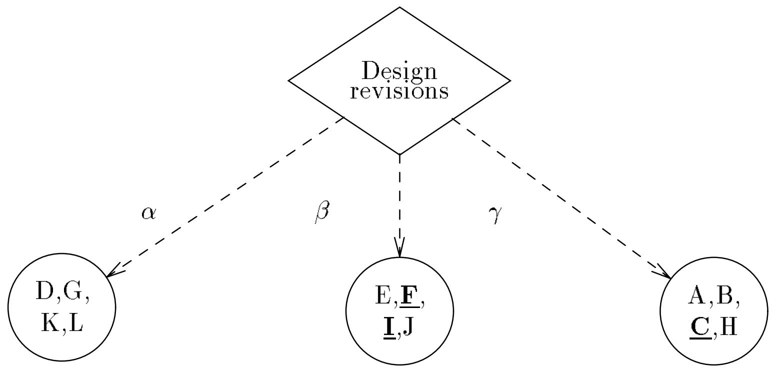


Figure 5: Partial tree using *Design Revisions* as the candidate metric. Metric selection function E({A,B,..,L},*Design Revisions*) returns 0.603. Positive target class instances are underlined. This metric is selected and the leftmost child will become a leaf node labeled "–".

	p	n	total	weight	F(p,n)	$weight * F(p,n)$
Child 1	0	4	12	.333	0.0	0.0
Child 2	2	2	12	.333	1.0	.333
Child 3	1	3	12	.333	.811	.270
Sum						.603

Table 4: Calculation of metric selection function using the *Design Revisions* metric to partition the modules.

	Module		
Metric	M	N	O
Module Function	P	I	I
Data Bindings	3	16	9
Design Revisions	0	7	12

Table 5: Raw test set data

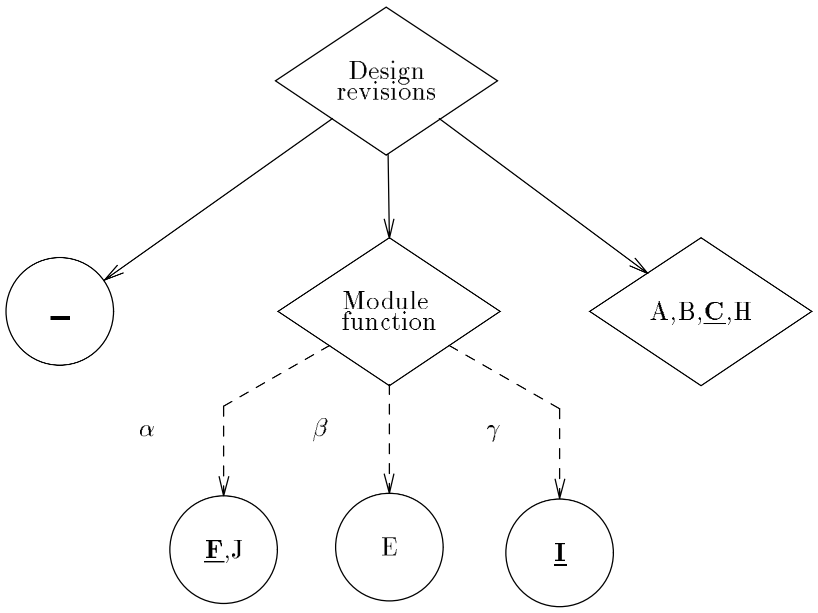


Figure 6: Partial tree using *Module Function* as the candidate metric. Metric selection function E({E,F,I,J},*Module Function*) returns 0.500. Positive target class instances are underlined.

	Module		
Metric	M	N	O
Module Function	γ	β	β
Data Bindings	α	γ	β
Design Revisions	α	β	γ

Table 6: Recoded test set data

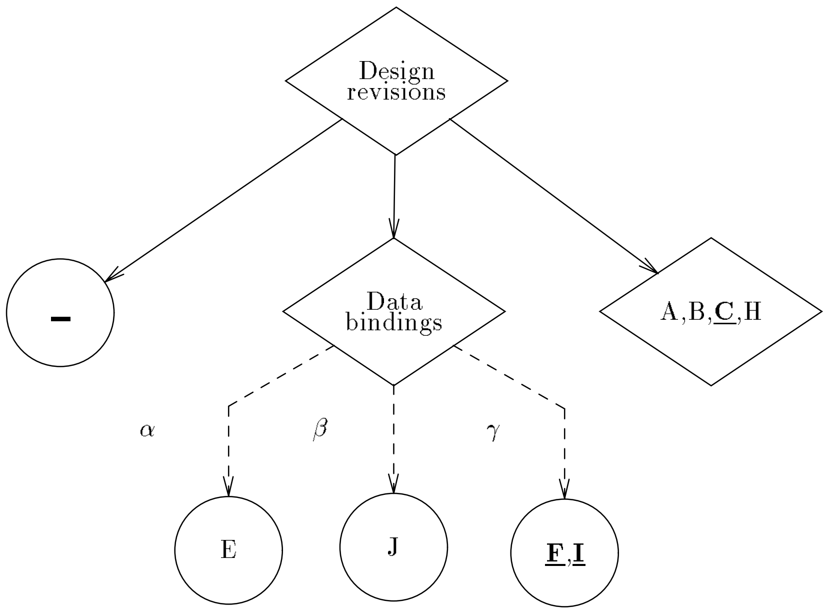


Figure 7: Partial tree using *Data Bindings* as the candidate metric. Metric selection function E({E,F,I,J},*Data Bindings*) returns 0. Positive target class instances are underlined. Metric *Data Bindings* is selected, yielding three leaves labeled "–", "–" and "+", reading from left to right.

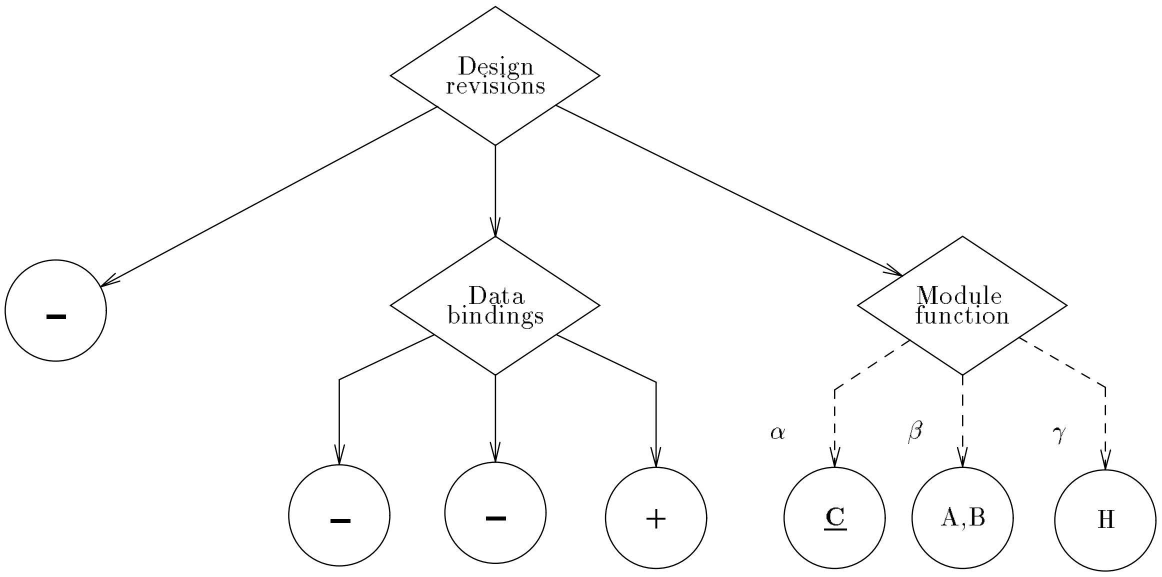


Figure 8: Partial tree using *Module Function* as the candidate metric. Metric selection function E({A,B,C,H},*Module Function*) returns 0. Positive target class instances are underlined.

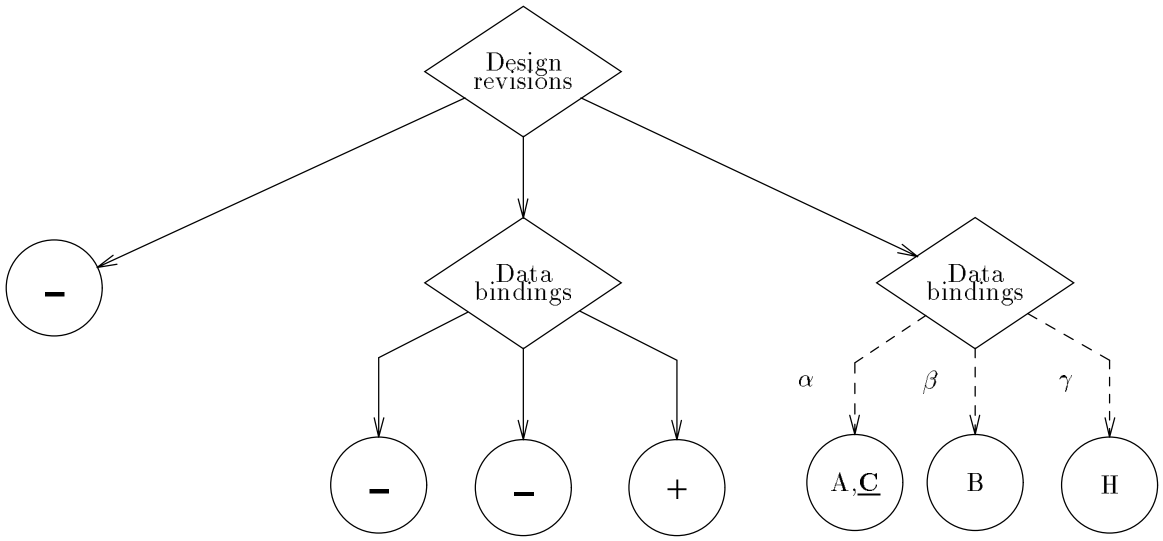


Figure 9: Partial tree using *Data Bindings* as the candidate metric. Metric selection function E({A,B,C,H},*Data Bindings*) returns 0.500. Positive target class instances are underlined. For this example metric *Module Function* is selected (See Figure 8). Metric *Module Function* produces three children labeled "+," "-," and "-.". The completed classification tree appears in Figure 10.

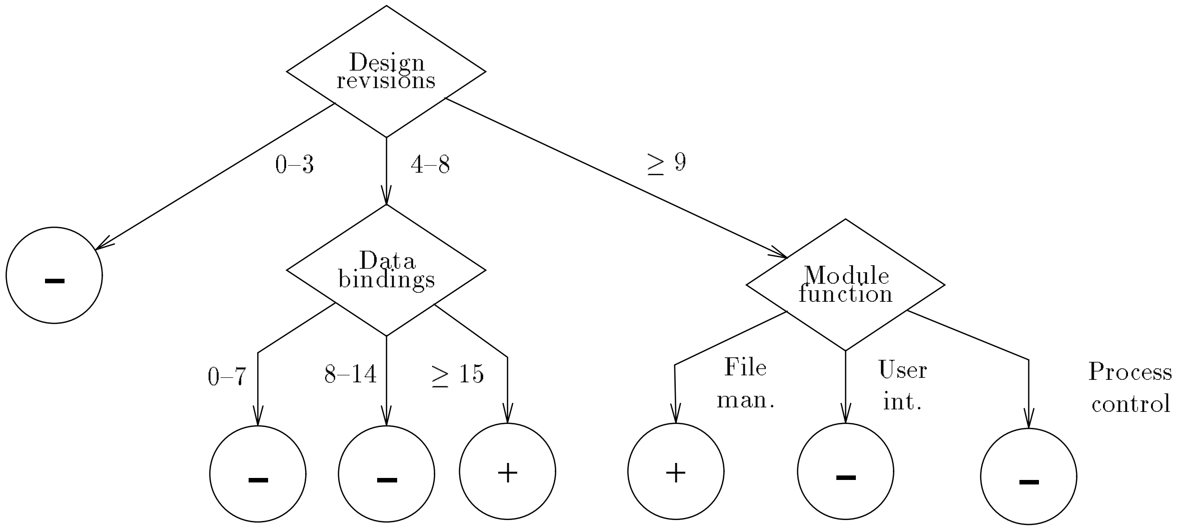


Figure 10: Completed classification tree.

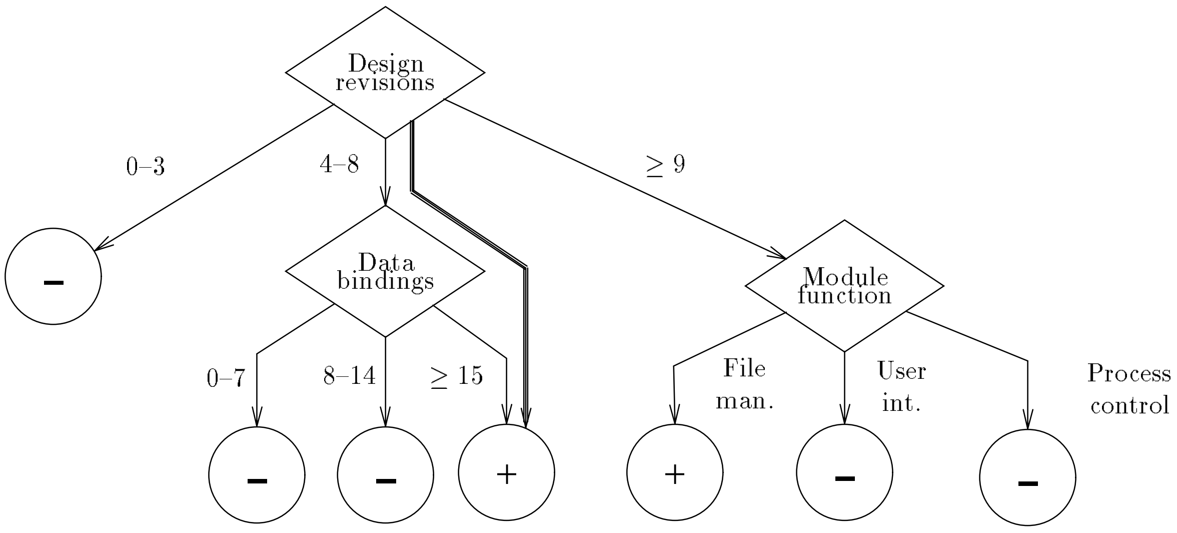


Figure 11: Classification tree application on module **N**.