Chapter 5

# Metrics Classification Techniques and an Integrated Framework for Risk Management

## 5.1 Introduction

In the previous chapters, we described some simple queries that can be answered directly from retrieving pertinent raw metrics data stored in the metrics database, as well as more complex queries requiring analytical techniques. As argued in Chapter 4, for some complex queries, more sophisticated metrics classification and analysis techniques may be required to aid in obtaining the query results. In this chapter, we describe two metrics classification techniques, namely *classification trees* and *neural networks*, that can be used to determine the level of risk of software components based on knowledge of past historical data. We describe how the techniques can be used to determine historical project trends and provide indications of potential problem areas that aid in management decision making. Finally, at the end of this chapter we present a framework where all the techniques of chapters 4 and 5 are integrated to provide a unique environment for software risk management.

An important objective of software test programs is to identify high-risk components. If such components can be identified early in a software development program, the manager can redirect resources to support preventive actions. Metrics classification techniques are used to determine the level of risk associated with management objectives by analyzing past project trends. Questions that may be asked by management include:

- Which components in the software development process are error-prone?
- Which components in the software development process require high development effort?

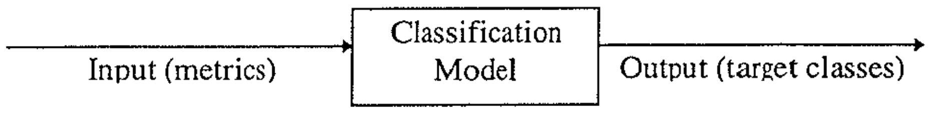


**Figure 5.1: Black Box of Classification Model**

Answering these queries can be regarded as a metrics-based classification problem. A black box model for such a classification problem is shown in Figure 5.1. The inputs to the model are the metrics and the associated empirical data, while the outputs are the target classes. Based on these target classes, the model uses several different software metrics to categorize past components as being within or outside a target class. The model is then applied to a project for which information about the desired property is unknown, and based on the past data, it identifies the component as being in the target class or not.

The application of such a classification technique using target classes is a three step process:

1. Initially, a target class is defined which is a binary membership function on the objects or components being classified.
2. The model is then trained to accurately classify a set of training objects, depending on whether or not they are in the target class. These training objects need to be representative of the population from which future objects will be selected, in order

that those objects, whose target classes membership is unknown, will be predicted by the model.

3. Lastly, the model is applied to previously unseen objects and predicts whether or not they belong to the target class.

## 5.2 Classification Trees

In this section, we introduce the use of classification trees to evaluate two factors of risk in a software development environment:

- software reliability
- software development productivity.

### 5.2.1 What is a Classification Tree?

A *classification tree* is essentially a decision tree in which each internal node represents a partitioning function for a metric [PAU94d]. Each link which emanates from an internal node represents a partition of a metric value of the node. The leaf nodes do not partition objects, they label the objects they encounter as members or non-members of the target class. Each path from the root node to a leaf node, therefore, represents a unique vector whose elements correspond to the partitioned values associated with each link in that path.

The classification tree has a functional architecture which resembles a tree. The top branches represent the input statements, each of which defines a software module with specific characteristics. The trunk represents a series of decision nodes which perform further classification of the software modules using the metrics data which is reported for each module. The roots of the tree represent the subsequent classification of the software modules according to the level of risk associated with the management objectives. Software components are input through the top of the classification tree, classified at each inner node by the metrics values, and identified by level of risk at the bottom nodes. Because classification trees allow managers to orchestrate the use of several metrics, the trees also serve as one type of metrics integration framework.

The *target class* in a classification tree define the conditions which the manager wants to identify in software components. In order to identify which software components are within the target class, a series of metrics must be applied which measure those characteristics which are related to the target class. These characteristics are defined as the *contributing factors* (metrics). Once the target class and the contributing factors have been identified, a corresponding classification tree can be generated.

The *training set* is a baseline of historic data which has been collected on the contributing factors and the target classes. The training set provides an empirical basis to infer rules on the dependencies of contributing factors and target classes. The classification tree algorithm implements an evaluation function to build a tree in which every path from the root node leading to a leaf is a rule. Each node in the tree is a contributing factor (software metric) and each branch of that node represents a value or possible range of values for the metric. Therefore, a path represents a condition, or a Boolean conjunction, of the values of the metrics at each node which the path may take. The conjunction is expressed in the form "metric X falls in range BX1 and metric Y falls in range RY2 and ...". The leaf node represents the classification when the condition is satisfied. If the metric values of a new component satisfy the condition of any rule, the leaf classifies whether or not the component falls in the target class. In this way, once the tree has been generated from historic data, it can be used during the development program to identify components which may be error-prone. Based on such predictions, appropriate corrective measures can be taken well in advance. Similarly, knowing the development risks in advance can support a decision to change the project schedule to more efficiently achieve the project goals.

### 5.2.2 Constructing a Classification Tree

We now describe the method for constructing a classification tree. For the purpose of this description, we assume that the training set of components and a given set of contributing factors have already been determined.

#### 5.2.2.1 Choosing the Training Set

An evaluation is performed on the complete set of software components for which historic data exists. The objective is to identify and eliminate those components that do not belong to a meaningful representative sample. For example, if two different components have the same metric values, but belong to opposite target classes, one or both of the components may be discarded. The resulting subset of components is the baseline training set.

#### 5.2.2.2 Partitioning into Ranges

The possible range of metric values for the training set of components is partitioned into a set of ranges. Partitioning can be done in different ways. One way is to first sort the values of a metric, then partition the metric into contiguous ranges in such a way that each partition covers an approximately equal number of components. For example, the Size metric (number of lines of code) is partitioned into ranges: (27,500), (501,1500), (1501,2147) such that each range consists of approximately equal number of components.

### 5.2.2.3 Choosing a Metric at a Node

As the tree is built, at each node of the tree, a metric is selected from the given set of metrics and assigned to the node. The metric which is selected is that metric (contributing factor) which incurs the minimum "cost". Cost is a measure of the homogeneity of the range of metric values within the component partitions. The cost factor measures the homogeneity of the different range values of the metrics between the components. Values of cost are computed by an evaluation function that measures the value of the "partial tree" formed when a metric is assigned, and each partition of its values as branches of the node. This process of choosing the metric with minimum cost is repeated for every node. The set of metrics at a node is obtained by discarding the metric that was assigned to the parent from the set of metrics that were available at the parent.

### 5.2.2.4 Partial Tree Construction

Beginning with the root node, a partial tree is constructed for each metric. The partial tree construction procedures are the same for every metric. All the components are separated according to the partitions of a metric's range of values, each of which denotes a branch. The selection criteria is whether or not the corresponding metric value of the component falls within the range of a partition. The next step is to derive the values of "P" and "N" for each set of components corresponding to the branch. The value of P is the number of components which belong to the target class (positive class) and N is the number of components that do not belong.

### 5.2.2.5 Recursive Metric Assignment

Assign the metric with the least value of E (the least cost) to the current node. Once a metric is selected at a node, discard it from further consideration for all the children of the node. This process is applied recursively until at least one of the "termination criteria" is satisfied. Termination criteria for the recursion may be divided into three cases which assist in deciding leaf nodes:

1. The primary termination case occurs when all the components at a node belong to the same target class. The node is marked as a leaf node. The leaf node is also marked as having either a positive or a negative target class.
2. Termination occurs if there are no components that can be filtered into a particular branch. This occurs when no components fall within a particular partition of a metric's range. The child node in that branch is marked as a leaf node with negative target class.
3. Termination also occurs if all components do not fall under the same target class, even though some components filter through a branch. In this termination case, all the metrics may have been assigned before arriving at the child node and there are no metrics which can be assigned to the child node. The child node is marked as a leaf node with negative target class.

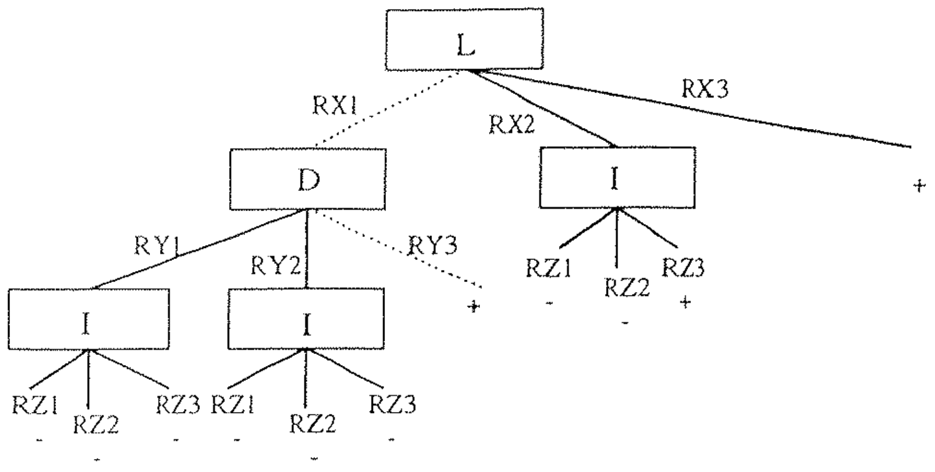
In cases (2) and (3), the leaf nodes are assigned a negative target class because nothing can be said in such a situation. An alternative may be to assign a new target class labeled "Unknown".

### 5.2.3 An Example of Classification Tree

In this section, we present an example of how classification trees can be constructed and used for analysis of software metrics based on the methodology described in the previous section. Consider a situation where a user wants the classification tree to predict whether a component might require extensive labor effort during development. Available metrics for this example include [POR90]:

- the size of the component
- the number of decisions
- the number of input/output variables.

For these selected metrics, a typical classification tree is constructed as illustrated in Figure 5.2 with L (Lines of Code or Size) as the root node. The figure illustrates a classification tree constructed with the metrics of Size, Number of Decisions, and Number of Input-Output Variables. These metrics are respectively partitioned into the ranges (RX1, RX2, RX3), (RY1, RY2, RY3), and (RZ1, RZ2, RZ3). The "+" and "-" symbols at the leaf nodes indicate whether or not the component is expected to require a high level of effort during development. The tree can also be used for predicting trends of new components. For example, if a new component has a size (number of lines) of 300, number of decisions of 100, and the number of input/output variables is 130, then that component will require a high level of effort during development, according to the tree. This is shown by the path traced by broken lines in Figure 5.2. Similarly, other components fall under the categories which are shown in Table 5.1.



| | | |
|---|---|---|
| L = size of component | | + high effort in development |
| D = number of decisions | | - low effort in development |
| I = number of I/O variables | | |
| RX1: $27<L\leq500$ | RY1: $0<D\leq50$ | RZ1: $0<I\leq100$ |
| RX2: $501<L\leq1500$ | RY2: $51<D\leq90$ | RZ2: $101<I\leq120$ |
| RX3: $1501<L\leq2147$ | RY3: $91<D\leq162$ | RZ3: $121<I\leq238$ |

**Figure 5.2: A Simple Classification Tree**

| Component | LINES | DECISIONS | I/O VARIABLES | High Development Effort? |
|---|---|---|---|---|
| 1 | 2000 | 20 | 35 | Yes |
| 2 | 1000 | 65 | 150 | Yes |
| 3 | 100 | 65 | 110 | No |

**Table 5.1: Categories of Components In Classification Tree**

| Component Number | Number of Decisions | I/O Variables | Lines | Effort | Class |
|---|---|---|---|---|---|
| 92 | 20 | 8 | 121 | 45 | 0 |
| 93 | 15 | 15 | 148 | 87 | 0 |
| 94 | 9 | 11 | 95 | 162 | 1 |
| 95 | 11 | 7 | 90 | 100 | 0 |
| 96 | 34 | 31 | 229 | 313 | 1 |
| 97 | 8 | 14 | 150 | 115 | 0 |
| 98 | 7 | 5 | 86 | 62 | 0 |
| 101 | 11 | 21 | 108 | 145 | 1 |
| 102 | 12 | 11 | 124 | 162 | 1 |

**Table 5.2: Data for Construction of Classification Tree**

| I/O VARIABLES Range | LINES Range | DECISIONS Range | Target Class |
|---|---|---|---|
| 1 | | | "-" |
| 2 | 1 or 3 | | "+" |
| 2 | 2 | | "-" |
| 3 | 1 | | "+" |
| 3 | 2 | | "+" |
| 3 | 3 | 1 or 3 | "+" |
| 3 | 3 | 2 | "-" |

**Table 5.3: Determining If A Component Is Within A Target Class**

### 5.2.4 Advantages and Drawbacks of Classification Trees

Classification trees are considered superior to other classification methods because the resulting models are straightforward to build and interpret. The generation process is extensible and new metrics can be added during construction of the tree. Classification trees can also be customized by using various metrics to classify different types of components in multiple development environments. Classification trees serve as metric integration frameworks and can be used to relate various high-risk properties of components to different software metrics. Classification trees can also be used for inter-project evaluation. One of the most significant advantages of classification trees is that the tree-generation algorithm can be applied to large systems.

Classification trees suffer from one major drawback: the target classes must be linearly separable, that is, the software components must fall into one of the defined target classes. An advantage of the neural network-based classification technique as described in the next section is that target classes do not have to be linearly separable.

## 5.3 Neural Networks

Neural networks can also be used to classify high-risk software components [MER96,PAU94c]. Using historical data, the neural network learns the relationship between certain metrics and a particular classification. A neural network can select the classification which best fits the input metrics. A neural network represents a series of relationships between the values of the metrics data and the desired classification. The neural network relationships can be learned through the correlation's of metric values to result in classes which are demonstrated through operation of the classification tree.

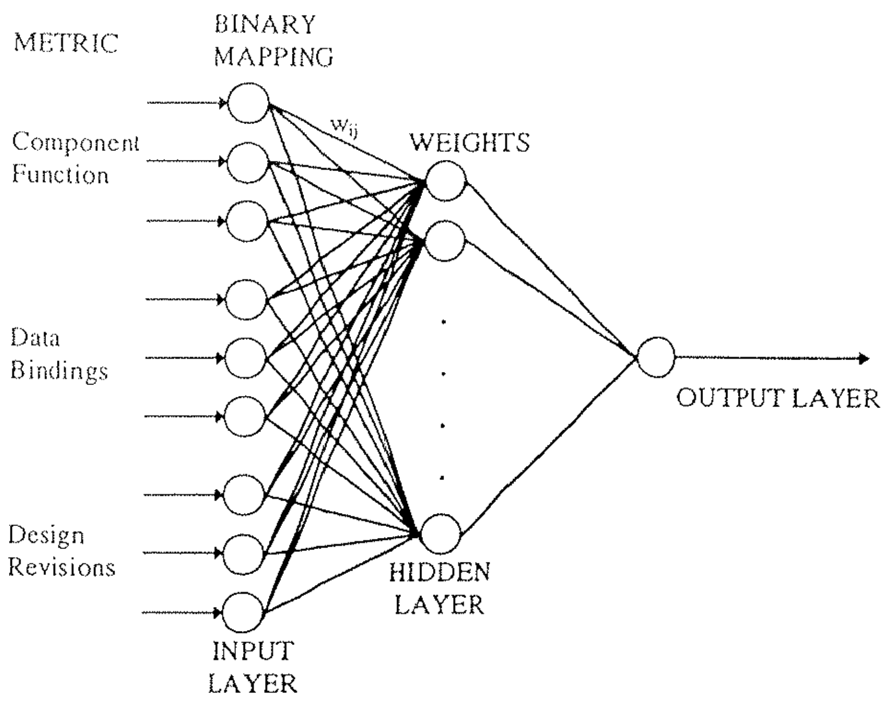


**Figure 5.3: Schematic of Neural Network-Based Classification Model**

The principle underlying neural network-based models is pattern matching similar to that for linear networks. These models treat the training objects as sets of paired input-output patterns. The model contains links that map part of the input pattern to part of the output pattern. Based on past metrics data, the network is trained to learn a set of link "weights" which are used to predict user-specifiable properties of software components. Examples of user-specified properties are components that are likely to be fault prone, units that have high development effort, or software functions that have faults in a certain class. These weights are "learned" by iteratively modifying their empirical mapping of input patterns to output patterns during their classification of a set of training data.

The schematic of the neural network-based classification model is shown in Figure 5.3. The model's structure and function is described in two phases. The first phase is the learning phase and the second is the output phase. The basic structure of the model consists of an input layer, an output layer and a number of hidden layers. The number of hidden layers is dependent on the user.

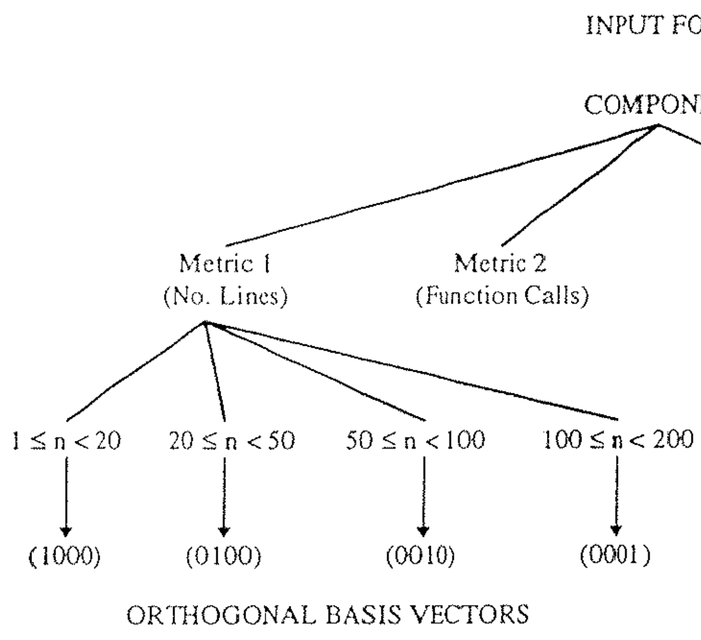


**Figure 5.4: Input Form for Neural Network-Based Classification Model**

To understand its function, consider the input to the model. The raw training-set data is re-coded into relatively small groups of discrete values. The re-coding process maps metrics values into mutually exclusive, and collectively exhaustive, ranges by transforming all data into nominal values. This grouping into discrete ranges can be done using non-parametric methods such as quartiles, distribution-sensitive least-weight-subsequent functions, or clustering methods. We now describe a three layer, error back-propagation neural network for classifying software metrics using this schematic.

### 5.3.1 Input Phase

Figure 5.4 shows the input form for the network model in Figure 5.3. Each of the software components used from the past metrics data has information about several different metrics, such as number of lines of code, function calls, design effort, etc. Consider a single such metric, the number of lines of code. Different software components have different values for this metric. Rather than handle with a large number of different values, the entire set of possible values is partitioned into discrete ranges and the value of the metric is quantified as falling into a certain range of values. For example, Figure 5.3 shows the following partitions:

- $1 \leq n < 20$
- $20 \leq n < 50$
- $50 \leq n < 100$
- $100 \leq n < 200$

where n is the number-of-lines of code metric. A component which has 43 lines would fall into the second range. This pre-processing is uniform for all metrics considered. The key difference is that the number of such partitions for a given metric can be specified by the user. Furthermore, the partitions are mutually exclusive. This provides the basis for generating orthogonal basis vectors for each partition. In this example, the orthogonal basis vectors for the number-of-lines of code metric are (1,0,0,0), (0,1,0,0), (0,0,1,0) and (0,0,0,1). Since the particular component has 43 lines, it would fall into partition block 2 and the basis vector is (0,1,0,0). These basis vectors, which are formed for each of the metrics, are the input to the network. The next stage in the network is a set of "weights" which have to be learned using the past metric data. At the start, these weights are initialized to be 0. A weighted sum is obtained and a bias is added, giving the output value. The bias is set to 0 initially. Note that the chosen initial weights is 0. However, any other random weight can be used as a starting weight.

### 5.3.2 Learning Phase

This phase involves training the network using past metric data to learn the set of weights. For each set of past metric data of a component, a "target class" is specified that reflects the goals and circumstances of a particular project. These user-specifiable target classes define the properties that developers want to predict about the system and its components. Examples of these properties are components likely to have a high number of faults, high development effort, or faults in a certain class. The model building process characterizes software artifacts from previous systems (components, subsystems, or processes) which meet the target class criteria.

The orthogonal basis vectors generated for a component are input to the network model and the output value is obtained. This output value is compared with the target class specification. A positive instance is

- if the output value is greater than 0.9, and if the component is in the target class, or
- if the output value is less than 0.1, and if the component is not in the target class.

Otherwise, there is a negative instance. In a negative instance, the set of weights, along with the bias, have to be corrected. In case of a positive instance, the weights remain the same.

The predicted mapping is correct if the calculation made on the network's current set of link weights ($w_{ij}$) classifies the component correctly. In this case, the link weights are not modified. The predicted mapping is incorrect if the calculation based on the network's current set of link weights ($w_{ij}$) classifies the component incorrectly. In this case the link weights are modified. In general, a link weight contributes to a miss-classification if it maps a 1 in the input pattern to an incorrect 1 (in the target class) or 0( not in the target class) in the output. This weight change is in increments of $+\delta$ or $-\delta$, corresponding respectively to whether the network incorrectly classified a component as 0 (the component is predicted as not in the target class but actually is), or 1 (the component is predicted as in the target class but actually is not). The bias is considered as a weight whose input is always a 1.

This process is repeated for each iteration through the training data. The training ends when the network can correctly classify 100% of the components in the training set or no further improvement is achieved.

### 5.3.3 Output Phase

Once the learning phase is complete, a set of weights exists for the network. The basis vectors are generated for a new component and its target class predicted using the learned weights. The re-coded data is given as input to train the model to learn the link weights.

### 5.3.4 Experiences with Neural Network-Based Classification Methodology

The capability of neural networks to classify non-linearly separable problem spaces gives neural networks an advantage over tree-based and linear network-based classification methods [PAU94c]. Linearly separable means that there is a straight line that separates the two pattern classes as being in the target class and not in the target class. Figure 5.5 shows patterns that are linearly separable while Figure 5.6 shows patterns that are not linearly separable.

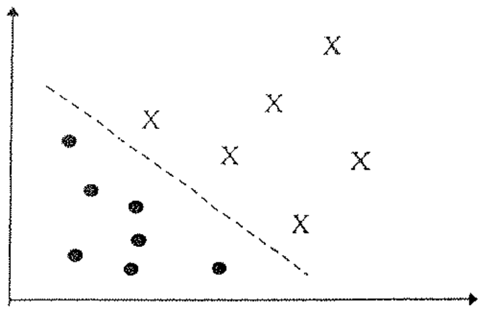


**Figure 5.5: Patterns That Are Linearly Separable**

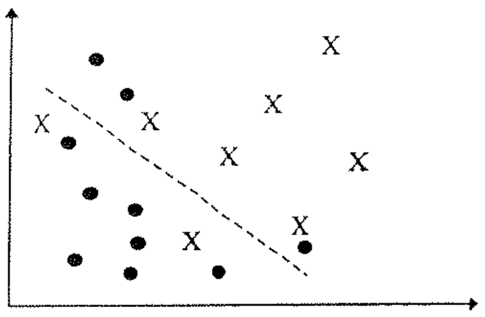


**Figure 5.6: Patterns That Are Not Linearly Separable**

In an our recent study [MER96] we have reported several important findings about our experience with neural network for their use in software development process. In particular, the following factors stand out clearly in this study:

1. Preprocessing is important. It can remove scaling effect of the projects by transforming the metrics into normalized or percentage data whenever possible. This can improve the similarity of projects of the same type with different scale.

2. Using similar or comparable projects for training data is important. This can help the accuracy of the prediction process. In [MER96] although the concept of metric analysis has been shown, the efficacy of the system in a real data environment has not

been evaluated. Training of the neural network on past projects does not guarantee good generalizations of future performance. At worst, there can no correlation in the training sets used, and so by definition no valid projections can be made. In that case, even though the network can be trained to correctly project the historical data, the output will exhibit only random performance on new data. However, this is not expected, since the goal of metrics is to provide information for prediction that can be usefully correlated with project status. In particular, the software metrics set was carefully evaluated and was chosen to include metrics that have descriptive utility. The only critical requirement for the system proposed in [MER96] and is not already in place is the recording of project milestone dates. This is a recommended addition to the software metrics, and is required to perform the functions described above on any other metric set.

3. The Neural network evaluation cannot be trusted blindly. The framework proposed in [MER96] suggests that a combination of neural nets and Fuzzy-based expert system would be desirable since it can allow re-evaluation of the metrics analysis and prediction by embedding the human knowledge in the expert system.

## 5.4 An Integrated Software Project Management Framework

Impressive gains have been made in having a multitude of techniques and tools work together towards a common goal of optimal software production processes. However, the state of the art tools still seem to be ad hoc by today's standards of full integration. Such integration should include the levels of presentation, data, control, and process. Presentation level integration concerns itself with how an end-user interacts with the target system. This would include such things as the GUI, ease of use of the products by the intended market, transparent resource impact, and reducing usage nonuniformities among different tools and techniques. This can have a significant impact on the learning curve of users, a topic discussed in the following chapter.

Another level of integration concerns data itself. Hence the techniques, when faced with different types or formats of data should still use same segments of data to derive the results. If inter-utilization of data among independent tools involves a lot of work, then integration at the data level is weak. Similarly, it should be easy for well-integrated techniques to cooperate in performing and maintaining integrity checks on data. Control integration refers to how easily functionalities of one technique are available from within another technique. This would also mean one technique is able to easily employ services it uses or requires from another technique in the environment.

Process integration ensures that different techniques interact to support processes, including different units of work, incident conditions that may arise, as well as the management of different types of constraints on the processes.

The variance in software environments can be large, and it may not be appropriate to impose the same detailed procedures upon all of them. Irrespective of the environment, a suite of comprehensive analytic techniques is needed to help in making important quality control decisions. The primary objective is to extract knowledge and gain meaningful insights into the development process based on raw metrics data. Such insight is essential to identify problematic components or processes during all phases of project development and provide answers to complex queries regarding *data probing*, current *assessment*, and *predictions* about the quality of the product. Possible corrective actions must also be assessed in terms of their cost, benefits, and consequences bearing upon the quality of the product.

With the above facets of integration in view, we present an integration framework of analytical techniques and elaborate how these techniques can be coupled with software metrics to provide a comprehensive and efficient management environment for the entire life-cycle of a software project [PAU96b,PAU96c].

In Table 5.4, we first summarize the objective of all the techniques presented in Chapters 4 and 5. The environment depicted in Figure 5.7 shows the overall integration of the above techniques in the big picture. The overall process, from the beginning of the software metrics collection, data analysis, rule-based decision analysis and process improvement via feedback, is depicted both in Figure 4.6 and 5.7. Most of the current CASE tools such as 001/DBTF [HAM90] and LOGICASE can be enhanced by using such integration to provides a framework to overcome inherent limitation discussed before.

| **Tool** | **Objective** |
|---|---|
| Multivariate Statistics<br>• Outlier Analysis<br>• Regression<br>• Principal Component Analysis<br>• Principal Component Regression | • Data Screening<br>• Dependency Relationship<br>• Reduce Dimensionality<br>• Reduce Dimensionality of Reduction |
| Classification Trees<br>• Metric Integration Framework<br>• Interproject Evaluation<br>• Training Set | • Reliability Identify Error Prone Components<br>• Identify High Development Efforts Productivity<br>• Classify high risk components |

| | |
|---|---|
| MultiResolution Analysis<br>• Convolution Computation with the scaling Function | • Problem Detect via fluctuations of data at deferent scales data resolution analysis for diagnosis of problem |
| Neural Networks<br>• Learning Set<br>• Pattern Matching | • Predict and classify high risk fault Components<br>• Predict and classify Development Efforts<br>• Predict and classify Unstability Requirements<br>• Predict and classify Unstability Design<br>• Predict Reliability |
| Influence Diagrams<br>• Model Relationships among metrics, risk items and Decision Alternative | |
| Singularity-Based Analysis | • Predict and estimate Productivity |
| Time Series Analysis<br>• Graphical Techniques | • Diagnose for Understanding Data Classifying and Cross Relating Exceptions |

**Table 5. 4 Analytical Techniques For Software Metrics Data**

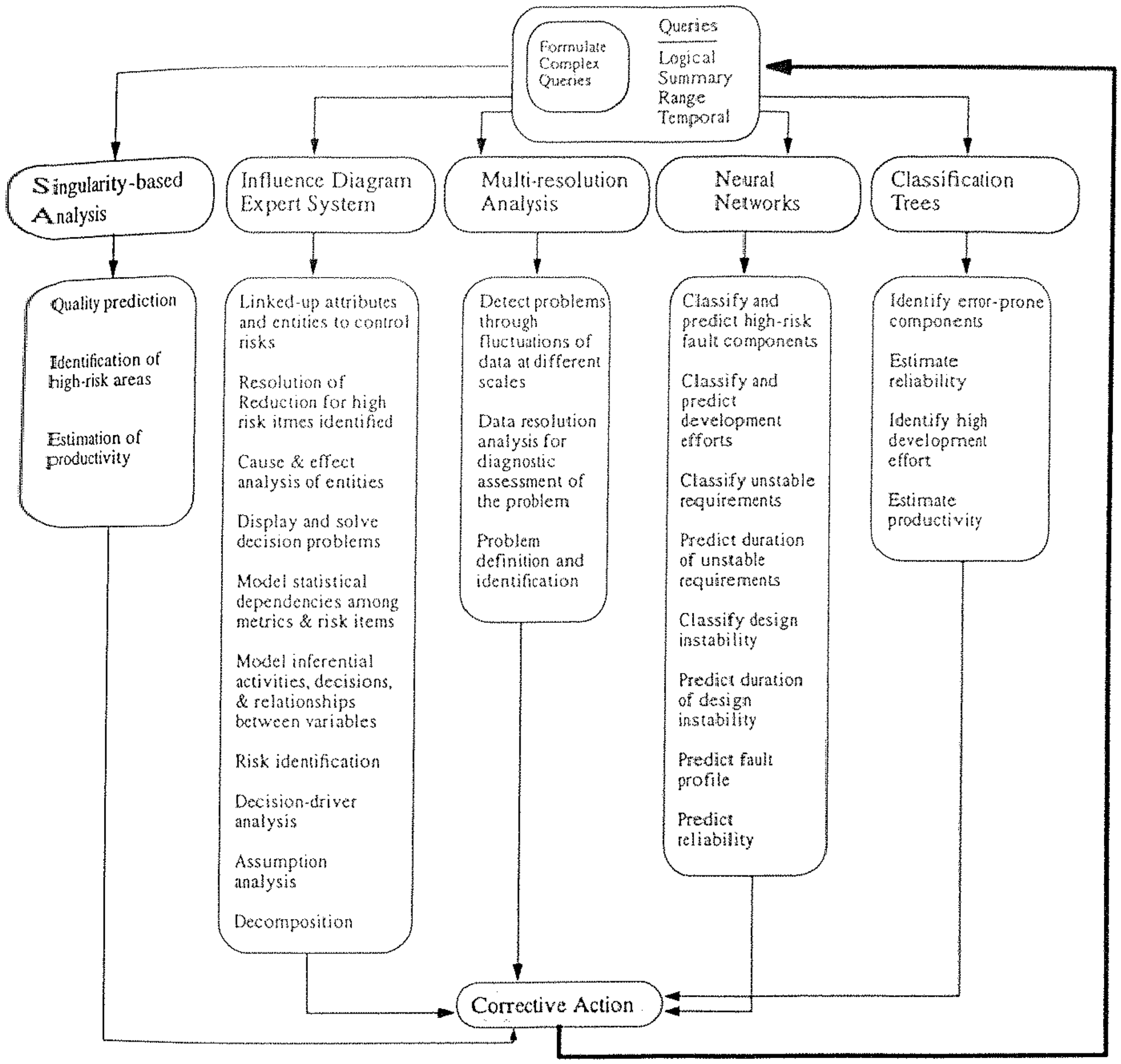

Formulate Complex Queries
Queries
Logical
Summary
Range
Temporal
Singularity-based Analysis
Influence Diagram Expert System
Multi-resolution Analysis
Neural Networks
Classification Trees
Quality prediction
Identification of high-risk areas
Estimation of productivity
Linked-up attributes and entities to control risks
Resolution of Reduction for high risk itmes identified
Cause & effect analysis of entities
Display and solve decision problems
Model statistical dependencies among metrics & risk items
Model inferential activities, decisions, & relationships between variables
Risk identification
Decision-driver analysis
Assumption analysis
Decomposition
Detect problems through fluctuations of data at different scales
Data resolution analysis for diagnostic assessment of the problem
Problem definition and identification
Classify and predict high-risk fault components
Classify and predict development efforts
Classify unstable requirements
Predict duration of unstable requirements
Classify design instability
Predict duration of design instability
Predict fault profile
Predict reliability
Identify error-prone components
Estimate reliability
Identify high development effort
Estimate productivity
Corrective Action

**Figure 5.7 The Integrated Framework for Corrective Actions and Decision Making**

## 5.5 Conclusion

Current techniques and CASE tools do not employ environments that provide sufficiently automated means to solve the problem at hand. In the context of software project management, this includes the recognition of the root problems and determination of procedures to fix them now and prevent them in the future. A proper choice of basic software metrics combined with a selection of powerful techniques, and true integration of these metrics and techniques forms the foundation of efficient and robust software project management. In this chapter, we have presented an integrated framework needed for current CASE environments to achieve these goals.

We have demonstrated that software metrics along with appropriate analytical tools can be an effective aid for CASE environments at all levels. Cumulatively, these techniques can help automate detection and correction of problems in a project. Besides automated trouble shooting, this framework also provides opportunities for further automation of processes in current CASE environments via tighter integration of techniques, tools and metrics. Other advantages include the control of risks via identification and removal of feedback loops. The comprehension of a project is also enhanced which helps modularization and component reuse. Such a framework can assist in optimizing management of large software projects by reducing low-level human intervention. It can thus reduce risk and increase quality of the final product, without increased cost. It can also increase reliability and greatly reduce the occurrence of costly mistakes in the future.

In conclusion, software engineering today, unlike other engineering sciences, lacks rigorous mathematical foundations and there are few experimental results available to form any concrete basis. The mathematical framework presented in this and previous chapters along with the proposed software metrics can lead to mature software science for the research and development community. In essence, our framework can be applied to any software development process once the appropriate metrics data is available.