Part

3

Emerging Techniques

Chapter

12

Software Metrics for Reliability Assessment

John C. Munson
University of Idaho

Taghi M. Khoshgoftaar
Florida Atlantic University

12.1 Introduction

Software development is a complicated process in which software faults are inserted in code by mistakes on the part of program developers. For the purpose of software reliability engineering, it is important to understand this fault insertion phenomenon. The pattern of the faults has been shown to be related to measurable attributes of the software. Consider the case of a large software system. Typically, this system will contain many subunits or modules. Each of the modules may be characterized in terms of a set of attribute measures. It would be quite useful to be able to construct predictive models for software faults based on these attribute values.

Among various software attributes, we focus on complexity metrics. This is based on the observation that software complexity has a direct impact on its quality. Some programs are easy to understand, easy to modify, and account for little of the expense in the development of the software systems of which they are components. Other programs seem almost beyond comprehension, even to their authors. These programs are nearly impossible to modify without inserting multiple faults, and they account for much of the expense in the development of the software systems of which they are components. Between these extremes lies a range of programs of intermediate complexity. Many program attributes, considered together, account for this observed variability. A large set of interrelated software complexity metrics quantify these static program complexity attributes. The size of this set, and the inter-

relationships among its elements, cause difficulties in understanding software complexity.

We also note that the initial reliability of a software system is largely determined during program design. One distinct aspect of the software design process that lends itself to measurement is the decomposition of the functionality of a system into modules and the subsequent interaction of the modules under a given set of inputs to the software. The reliability of a software system may be characterized in terms of the individual software modules that make up the system, as well as their executions. The likelihood that a component will fail is directly related to the complexity of that module. If it is very complex the fault probability is also high. Furthermore, a typical large software system might consist of many hundreds of distinct software modules. When the software is executing any one of many possible functionalities, only a small subset of the code is actually executing. The reliability of a software system, then, is a function of software modules that are actually executing and the fault density of these modules. In order to model the reliability of a software system, we must be able to characterize the dynamic characteristics of the software.

For each possible design outcome in a software design effort there will be a set of expected execution profiles, one for each of the anticipated program functionalities. The reliability of the system can be thought of in terms of the exposure of the program to complex modules while the program is running. This complexity will vary in relation to the execution profiles induced by the operating environment of the program. At the design stage, quantitative measures correlated to fault-proneness and product failure provide a means to begin to exert statistical process control techniques on the design itself as an ongoing quality-control activity. Rather than merely documenting the increasing operational complexity of a software product and therefore its decreasing reliability, you can also monitor the operational complexity of successive design adaptations during the maintenance phase. This effort can ensure that subsequent design revisions do not increase operational complexity and, especially, do not increase the variance among individual module's functional complexity.

In this chapter we describe the software metrics that can be obtained early in the life cycle for software reliability assessment. In particular, we introduce the measure of complexity attributes to predict the quality of software. We first present the techniques that reduce a large set of interrelated complexity metrics, respectively, to a smaller set of orthogonal metrics, and to a single metric. We then describe a method for quantifying execution profiles. Furthermore, we combine the concepts of static complexity and execution profile to produce dynamic complexity metrics, that is, metrics that quantify the complexity of

software systems as they operate in a given environment. Finally, the usage of software complexity metrics in software reliability models is presented.

12.2 Static Program Complexity

Our ultimate objective in the software measurement process is to be able to characterize the quality of a software system in terms of some measurable software attributes. These software quality attributes might include such domains as the number of embedded faults in the software or the number of changes made to the software over its useful life. The problem with software quality attributes is that these attributes are known only at the end of the useful life of the software. It is a moot point whether or not a system has a low or a high fault density after the software has been taken out of service at the end of its career. In order to understand the quality of software, we need to examine different complexity metrics.

12.2.1 Software metrics

We have, over a period of time, observed a relationship between measures of software complexity and measures of software quality [Khos90]. An example of this would be the direct relationship and high correlation between the lines of code (LOC) metric and software faults [Khos92a]. There are many software complexity metrics. A number of these metrics are highly correlated with measures of quality such as fault count or change count. Measures of software complexity can be used as good predictors of software quality: for example, complex software modules are those likely to have a high fault count. Further, if a complex module is executed, it has a much higher likelihood of failing than a module that is not so complex. The important fact here is that measures of software complexity can be obtained very early in the software life cycle. Some may be obtained by measuring the source code, such as LOC. Some may be obtained from the high-level design, such as measures relating to the flow of program control. Some measures may even be taken on the software specifications themselves. Thus we can use software complexity measures as leading indicators of measures of software quality.

Since most of the existing metrics have common elements and are linear combinations of these common elements, it seems reasonable to investigate the structure of the underlying common factors or components that make up the raw metrics. The technique we have chosen to use to explore this structure is a procedure called *principal components analysis*. Principal components analysis is a decomposition technique

that may be used to detect and analyze the relationships among the software metrics. When confronted with a large number of metrics measuring a single construct, it may be desirable to represent the set by some smaller number of variables that convey all, or most, of the information in the original set. Principal components are linear transformations of a set of random variables that summarize the information contained in the variables. The transformations are chosen so that the first component accounts for the maximal amount of variation of the measures of any possible linear transform; the second component accounts for the maximal amount of residual variation; and so on. The principal components are constructed so that they represent transformed scores on dimensions that are orthogonal [Muns89].

12.2.2 A domain model of software attributes

Through the use of principal components analysis, it is possible to have a set of highly related software attributes mapped into a small number of uncorrelated attribute domains. This solves the problem of multicollinearity in subsequent regression analysis [Khos90]. There are many software metrics in the literature, but principal components analysis reveals that there are few distinct sources of variation, i.e., dimensions, in this set of metrics. It would appear perfectly reasonable to characterize the measurable attributes of a program with a simple function of a small number of orthogonal metrics, each of which represents a distinct software attribute domain. Still, some metrics measure distinct program attributes. For example, Halstead developed a number of metrics now known as the software science metrics [Hals77]. Four of these metrics cannot be decomposed into other metrics:

- N_1, the total number of operators in a program
- N_2, the total number of operands in a program
- η_1, the number of unique operators in a program
- η_2, the number of unique operands in a program

From these primitive metrics, Halstead composed nonprimitive metrics, including

- $N = N_1 + N_2$, program length
- $V = N \log_2 (\eta_1 + \eta_2)$, program volume
- $\hat{E} = V [\eta_1 N_2 / 2\eta_2]$, estimated effort

While these metrics are sensitive to program size, they are not sensitive to program control flow; that is, programs with vastly different

control flow structure can have identical Halstead metric values. Thus, Halstead's metrics do not measure complexity due to control flow.

On the other hand, McCabe developed a nonprimitive metric, the cyclomatic number, which does measure some aspects of control flow complexity [McCa76]. Given a strongly connected graph G, the cyclomatic number of G is the number of independent paths in G. This is given by $V(G) = e - n + p$, where e is the number of edges, n is the number of nodes, and p is the number of connected components. McCabe applied this graph theory by constructing a program control flow graph. In this directed graph, nodes represent entry points, exit points, segments of sequential code, or decisions in the program. Edges represent control flow in the program. Strong connectivity is satisfied with the addition of an edge from the exit node to the entry node. McCabe observed that, for a structured program with single entry and exit constructs, $V(G)$ is equal to the number of predicates in the program plus one.

While $V(G)$ is sensitive to program control flow complexity, it is not necessarily related to program size, that is, programs with vastly different counts for operators and operands can have identical cyclomatic numbers. Thus, Halstead's metrics and McCabe's metric measure two distinct program attributes. Each of these program attributes represents a source of variation underlying the measured complexity metrics.

12.2.3 Principal components analysis

A statistical technique like principal components analysis may be used quite effectively to isolate the distinct sources of variation underlying the set of software complexity metrics describing a software system. A multivariate data set might, for example, consist of values for each of m software attribute measures for a set of each of n program modules. These data can be represented by an n by m matrix. When applying principal components analysis, you typically seek to account for most of the variability in the m attributes of this matrix with $p < m$ linear combinations of these attributes. Each linear combination represents an orthogonal source of variation underlying the data set. Let $\mathbf{\Sigma}$ be the covariance matrix for the metric data set. Then $\mathbf{\Sigma}$ is a real symmetric matrix and, assuming that it has distinct roots, can be decomposed as

$$\mathbf{\Sigma} = \mathbf{T}\,\mathbf{\Lambda}\,\mathbf{T}'$$

where $\mathbf{\Lambda}$ is a diagonal matrix with the eigenvalues, $\lambda_1, \lambda_2, \ldots, \lambda_m$ on its diagonal

$\sum_{j=1}^{m} \lambda_j = \text{trace}(\Sigma)$

$\mathbf{T}$ is an orthogonal matrix where column j is the eigenvector associated with λ_j

$\mathbf{T}'$ is the transpose of $\mathbf{T}$

The m eigenvectors in T give the coefficients that define m uncorrelated linear combinations of the original complexity metrics. These orthogonal linear combinations are the principal components of Σ. The ratio $\lambda_j/\text{trace}(\Sigma)$ gives the proportion of complexity metric variance that is explained by the jth principal component. The first few principal components typically explain a large proportion of the total variance. Thus, restricting attention to the first few principal components can achieve a reduction in dimensionality with an insignificant loss of explained variance. A stopping rule selects $p < m$ principal components such that each one contributes significantly to the total explained variance, and the p selected components collectively account for a large proportion of this variance. A typical stopping rule selects principal components with associated eigenvalues greater than one.

The standardized transformation matrix, T*, is constructed from T to produce p domain metrics for each of the n programs comprising the software system. An element t_{ij} of T* gives the coefficient, or weight of the ith complexity metric, $i = 1,2, \ldots, m$, for the jth domain metric, $j = 1,2, \ldots, p$. Thus, given $\mathbf{z}_k$, the vector of standardized metrics for program module $k = 1,2, \ldots, n$, $\mathbf{D}_k = \mathbf{z}_k\text{T}*$ is a new vector of orthogonal domain metrics for the kth program module. The p domain metric values for this program module, $D_{k1}, D_{k2}, \ldots, D_{kp}$, represent variation due to the p orthogonal complexity domains underlying the complexity data. The n values for the jth domain metric $D_{1j}, D_{2j}, \ldots, D_{nj}$ are distributed with a mean of 0 and variance of 1. Since domain metrics are not directly observable, they are best interpreted in terms of domain loadings, that is, in terms of their correlations with the complexity metrics.

Example 12.1 Consider a sample metric data set consisting of metric values for N, V, $\hat{E}$, and $V(G)$ for the programs comprising a software system. Assume that principal components analysis of this data set reveals two principal components having eigenvalues greater than 1. Thus, two complexity domains represent significant sources of variation underlying the complexity metrics. Table 12.1 gives the pattern of domain loadings for these domains along with their associated

TABLE 12.1 Domain Pattern

Metric	Domain 1	Domain 2
N	**0.98**	0.11
V	**0.97**	0.09
$\hat{E}$	**0.91**	0.13
$V(G)$	0.12	**0.99**
Eigenvalue	2.75	1.02
% variance	68.83	25.61
Cumulative % variance	68.83	94.44

eigenvalues. The three Halstead metrics correlate strongly with the first domain, while $V(G)$ correlates strongly with the second domain. This observation leads to the interpretation that the first domain is related to size complexity, while the second domain is related to control flow complexity. These two domains account for, respectively, about 68.8 and 25.6 percent of the variance observed in the metric data. Table 12.2 gives the standardized transformation matrix, $\mathbf{T}*$.

Consider program module k that has a vector of standardized metrics

$$\mathbf{z}_k = [1.284\ 1.408\ 0.777\ -0.415]$$

The domain metric values for this program module may be obtained by postmultiplying the standardized metric values by the transformation matrix in Table 12.2 as follows:

$$\mathbf{d}_k = [1.284\ 1.408\ 0.777\ -0.415] \begin{bmatrix} 0.364 & -0.046 \\ 0.361 & -0.061 \\ 0.334 & -0.015 \\ -0.119 & 1.020 \end{bmatrix}$$

$$= [1.286\ -0.581]$$

In most linear modeling applications with software metrics, such as regression analysis and discriminant analysis, the independent variables, or metrics, are assumed to represent some distinct aspect of variability not clearly present in other measures. In software development applications, the independent variables (in this case, the complexity metrics) are strongly interrelated or demonstrate a high degree of multicollinearity. In cases like this, it will be almost impossible to establish the unique contribution of each metric to the model. One distinct result of multicollinearity in the independent measures is that the regression models developed using independent variables with a high degree of multicollinearity have highly unstable regression coefficients. Such models may be subject to dramatic changes due to additions or deletions of variables or even discrete changes in metric values.

12.2.4 The usage of metrics

Our objective is to build and extend a model for software attributes. This model will contain a set of orthogonal attribute domains. Once we

TABLE 12.2 Standardized Transformation Matrix, T*.

Metric	Domain 1	Domain 2
N	0.364	−0.046
V	0.361	−0.061
$\hat{E}$	0.334	−0.015
$V(G)$	−0.119	1.020

have such a model in place we would then like to identify and select from the attribute domain model those attributes that are correlated with a software quality measure, such as number of faults. Each of the orthogonal attributes will have an associated metric value that is uncorrelated with any other attribute metrics. Each of these attributes may potentially serve to describe some aspect of variability in the behavior of the software faults in a program module. This further suggests that constructing a composite metric consisting of Halstead's metrics and McCabe's metric can lead to a better fault prediction capability [Lyu94a].

Some ill-considered attempts have been made to design software systems reflecting the complexity of the object being designed. The most notable of these attempts relates to the use of McCabe's measure of cyclomatic complexity $V(G)$. Magic values of cyclomatic complexity are being incorporated into the requirements specifications of some software systems. For example, we might choose to specify that no program module in the software system should have a cyclomatic complexity greater than an arbitrary value of, say, 15, which is used as a guideline in the design process.

Potentially catastrophic consequences may be associated with this univariate design criterion. First, there is little or no empirical evidence to suggest that a module whose cyclomatic complexity is greater than 15 is materially worse than one whose cyclomatic complexity is 14. Second, and most important, is the fact that if, in the process of designing a software module, we find that the module has a cyclomatic complexity greater than 15, the most obvious and common solution to the problem is to divide the software module into two distinct modules. Now we will certainly have two modules whose cyclomatic complexity is less than 15. The difficulty here is that instead of one program module we have created two, or possibly three, in its place. This will increase the macro complexity of measures related to complexity. In other words, we have decreased *cyclomatic complexity,* but we have increased *coupling complexity.* The result of this shortsighted decision may well be that the total *system complexity* will increase. This in turn will likely lead to a concomitant increase in total faults.

12.2.5 Relative program complexity

In order to simplify the structure of software complexity even further it would be useful if each of the program modules in a software system could be characterized by a single value representing some cumulative measure of complexity. The objective in the selection of such a function, g, is that it be related in some linear manner to software faults such that $g(x) = ax + b$, where x is some unitary measure of program com-

plexity. The more closely related x is to software faults, the more valuable the function g will be in the anticipation of software faults. Previous research has established that the relative complexity metric ρ has properties that would be useful in this regard. The relative complexity metric, ρ, is a weighted sum of a set of uncorrelated attribute domain metrics [Muns90a, Muns90c]. This relative complexity metric represents each raw metric in proportion to the amount of unique variation contributed by that metric.

For an analysis concerned with the relative contributions of each program complexity domain to the complexity of each program, reduction to domain metrics is sufficient. For these applications, it is possible to compute the relative complexity metric for each program module k by forming the weighted sum of the domain metrics as follows: $\rho_k = \mathbf{d}_k \Lambda'^*$ where Λ^* is the vector of eigenvalues associated with the selected domains, Λ'^* is the transpose of this vector, and $\mathbf{d}_k$ is the vector of domain metrics for program module k [Muns90b]. The n values of ρ are distributed with a mean of 0 and a variance of

$$V(\rho) = \sum_{j=1}^{p} \lambda_j^2$$

where p is the number of selected domains, and λ_j is the eigenvalue associated with the jth domain. Thus ρ will take both positive and negative values. A scaled version of this metric

$$\rho_k' = \frac{10\rho_k}{\sqrt{V(\rho)}} + 50$$

is more easily interpreted. The scaled metric is distributed with a mean of 50 and a standard deviation of 10.

Example 12.2 Consider a program module k having the vector of domain metric values $\mathbf{d}_k = [1.286 \; -0.581]$. The relative complexity for program module k may be computed by the multiplication of the transpose of a vector containing the eigenvalues from Table 12.1 as follows:

$$\rho_k = [1.286 \; -0.581] \begin{bmatrix} 2.753 \\ 1.024 \end{bmatrix} = 2.946$$

The scaled relative complexity is simply,

$$\rho_k' = \frac{(10)(2.946)}{\sqrt{2.753^2 + 1.024^2}} + 50 = 60.03$$

From this scaled value it is easy to see that the scaled relative complexity of module k is roughly one standard deviation (10.03) above the mean relative complexity, 50, for all modules in the total software system.

12.2.6 Software evolution

From the standpoint of early reliability prediction, we would like to be able to use the measurements from past software development efforts to perform a preliminary assessment of fault density or reliability of current or active software development projects. In other words, we would like to have the ability to use a past system to serve as a baseline for a current development project. In essence, the objective is to use an existing database as a baseline for subsequent measures in a software system currently in development. We might choose, for example, to take the first build of a real-time control software system that was developed in the past and use this for a real-time control software system currently being developed. In this sense, all subsequent software measures on new systems will be transformed relative to the baseline system.

The ability to use information from past development projects in current design work is most important. This is due to the fact that many of the software quality and reliability attributes of a system can be measured only after the system has been in service for some time. If a software system under current development is directly comparable to one that has demonstrated quality and/or reliability problems in the past, there is evidence to suggest that the design had better be modified.

The attribute measures presented so far are *static* measures of the program. They measure such features of the program as its size and the complexity of its control structure. If the functionality of a program was extremely restricted, these static measures might well be sufficient to describe the program entirely. Most modern software systems, however, have a broad range of functionality. Consider, for example, the software system for a typical spreadsheet program. The number of distinct functions in such a system and the number of ways that these functions might be exercised are both very large numbers. In addition to static measures of program attributes, we must also be concerned with dynamic measures of programs as well.

In order to describe the complexity of an evolving system at any point in time, it will be necessary to know which version of each of the modules was a constituent in the program that failed. Consider a software system composed of n modules as follows:

$$m_1, m_2, m_3, \ldots, m_n$$

Now, let m_j^i represent the ith version of the jth module. With this nomenclature, the first build of the system would be described by the set of modules:

$$<m_1^1, m_2^1, m_3^1, \ldots, m_n^1>$$

We can represent this configuration more succinctly simply by recording the superscripts as vectors. Thus a system under development might look like the following sequence of module version sets:

$$\nu^1 = < 1, 1, 1, 1, \ldots, 1 >$$

$$\nu^2 = < 1, 2, 2, 1, \ldots, 1 >$$

$$\nu^3 = < 2, 2, 3, 1, \ldots, 1 >$$

$$\nu^4 = < 2, 3, 3, 2, \ldots, 2 >$$

.

.

.

Thus, the ith entry in the vector ν^n would represent the version number of the ith module in the nth build of the system.

A natural way to capture the intermediate versions of the software is to have the system development occur under a configuration management system. For a system running under configuration management, all versions of all modules can be reconstructed from the time the program was placed in the system. That is, the precise nature of ν^n can be determined from the configuration management system.

The computation of the relative complexity of various releases or versions of a software system will occur as follows: for an initial build of a software system described by ν^1, transformation matrix will map the raw complexity metrics onto a set of reduced orthogonal domain metrics. From these, relative complexity values may be computed for the modules represented by the vector ν^1. The transformation coefficient matrix derived from the first build will not change subsequently, but will serve as a baseline for measuring changes in program complexity.

Associated with the ith program module, m_i^1, at the first build of a program, there is a corresponding relative complexity value of ρ_i^1. By definition, the adjusted relative complexity, ρ, of the program system at this first build will be

$$\rho^1 = \sum_i \rho_i^{\nu_i^1} = 50$$

As the system progresses through a series of builds, system complexity will tend to rise [Muns90b]. Thus, the system relative complexity of the

nth version of a system may be represented by a nondecreasing function of module relative complexity as follows:

$$\rho^n = \sum_i \rho_i^{v_i^n} \geq 50$$

where v_i^n represents an element from the configuration vector $\mathbf{v}^n$ described earlier.

This change in the overall relative complexity of an example system over time is presented pictorially in Fig. 12.1. It can be seen from this figure that the relative complexity of a system will rise fairly rapidly shortly after the first build of the system. It would further appear that the relative complexity becomes asymptotic to a value of, say, 55. This is not the case. The system complexity continues to rise, albeit more slowly, throughout the life of the software system.

12.3 Dynamic Program Complexity

The relative complexity measure, ρ, of a program is a measure of the program at rest. When a program is executing, the level of exposure of its modules depends on the execution environment. Consequently, both the static complexity of a software system and the system's operational environment influence its reliability [Khos93a]. The complex programs comprising a software system often contain a disproportionate number of faults. However, if in a given environment the complex modules are rarely exercised, then few of these faults are likely to become expressed as failures. Different environments will exercise a system's programs differently. The dynamic complexity is a measure of the complexity of the subset of code that is actually executed as a system is performing a

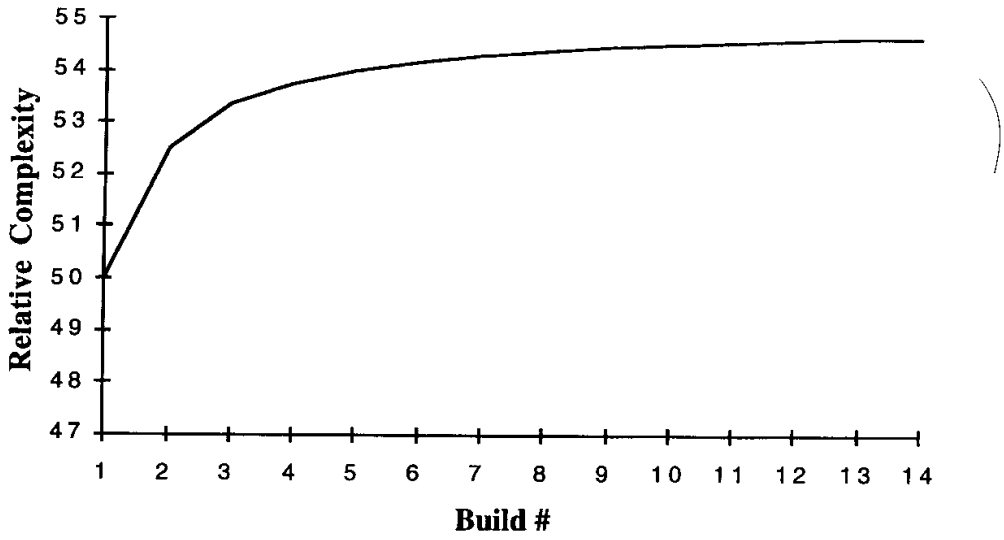

Figure 12.1 Change to relative complexity over time.

given function. A system's dynamic complexity is high in an environment that exercises the system's complex programs with high probability. It is likely that one or more potential scenarios induce inordinately large dynamic complexity. Identifying these scenarios is the first step in assuring that they receive the testing time that they warrant. This is evident in the concept of execution profile.

12.3.1 Execution profile

A software system is written to fulfill a set of functional requirements. It is designed in such a manner that each of these functionalities is expressed in some code component. In some cases a direct correspondence exists between a particular program module and a particular functionality. That is, if the program is expressing that functionality, it will execute exclusively in the module in question. In most cases, however, there will not be this distinct traceability of functionality to modules. The functionality will be expressed in many different code modules.

As a program is exercising any one of its many functionalities, it will apportion its time across one to many program modules. This temporal distribution of processing time is represented by the concept of the execution profile. In other words, if we have a program structured into n distinct modules, the execution profile for a given functionality will be the proportion of time spent in each program module during the time that the function was being expressed.

Another way to look at the execution profile is that it represents the probability p_i of execution occurring in module m_i at any point in time. When a software system is running a fixed function there is an execution profile for the system represented by the probabilities p_1, p_2, p_3, . . . , p_n. For our purposes, p_i represents the probability that the ith module in a set of n modules is in execution at any arbitrary time.

Each functionality will have its own, possibly unique, execution profile. For a set of 10 hypothetical program modules, the execution profiles of two functionalities are shown in Fig. 12.2. From this example, we can see that for program module 6, there is a low probability of finding this module in execution at any time in the duration of the execution of function 1. For function 2, on the other hand, this same module shows a relatively high rate of use.

12.3.2 Functional complexity

The functional complexity ϕ of the system running an application with an execution profile is defined as

$$\phi = \sum_{j=1}^{n} p_j \rho_j$$

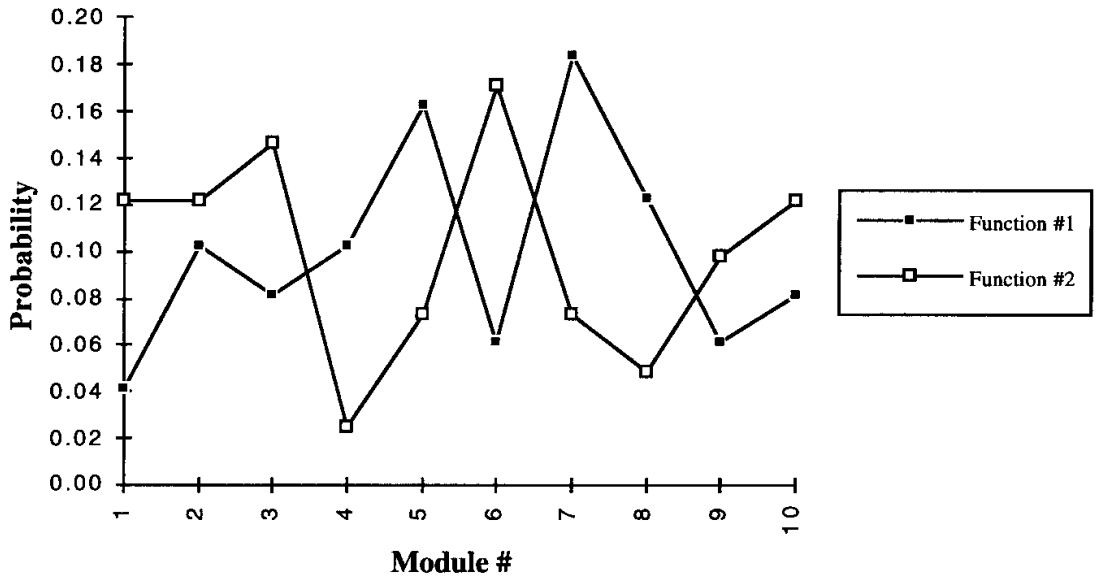

Figure 12.2 Two different execution profiles.

where ρ_j is the relative complexity of the jth program module and p_j is the execution probability of this module. This is simply the expected value of relative complexity under a particular execution profile. The execution profile for a program can be expected to change across the set of program functionalities. In other words, for each functionality, f_i, there is an execution profile represented by the probabilities p_1^i, $p_2^i, \ldots, p_n^i$. As a consequence, we can observe a functional complexity ϕ_i for each function, f_i execution, where

$$\phi_i = \sum_{j=1}^{n} p_j^i \rho_j$$

This is distinctly the case during the test phase when the program is subjected to numerous test suites to exercise differing aspects of its functionality. The functional complexity of a system will vary greatly as a result of these different test suites. A bar chart demonstrating the relationship of execution profile and relative complexity of a program running a scenario of low functional complexity is shown in Fig. 12.3. In Fig. 12.4 a high functional complexity test scenario is presented.

Given the relationship between complexity and embedded faults, we would expect the failure intensity to rise as the functional complexity increases. If an application is chosen in such a manner that high execution probabilities are associated with the complex modules, then the functional complexity will be large and the likelihood of a failure event during this interval would be relatively high. In Fig. 12.5, the operation of a hypothetical software system executing various functionalities across time is presented. From this figure, we would expect the software failures to be directly related to those periods when the functional complexity is high.

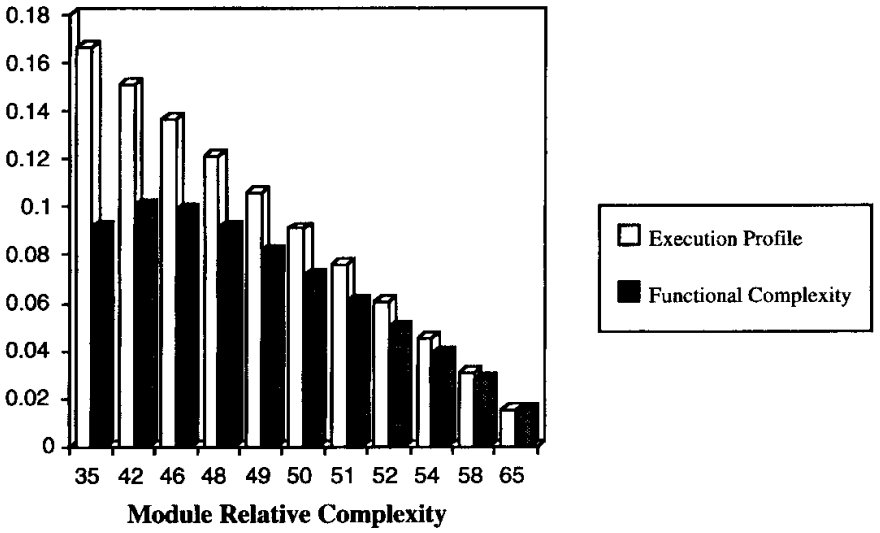

Figure 12.3 Low functional complexity.

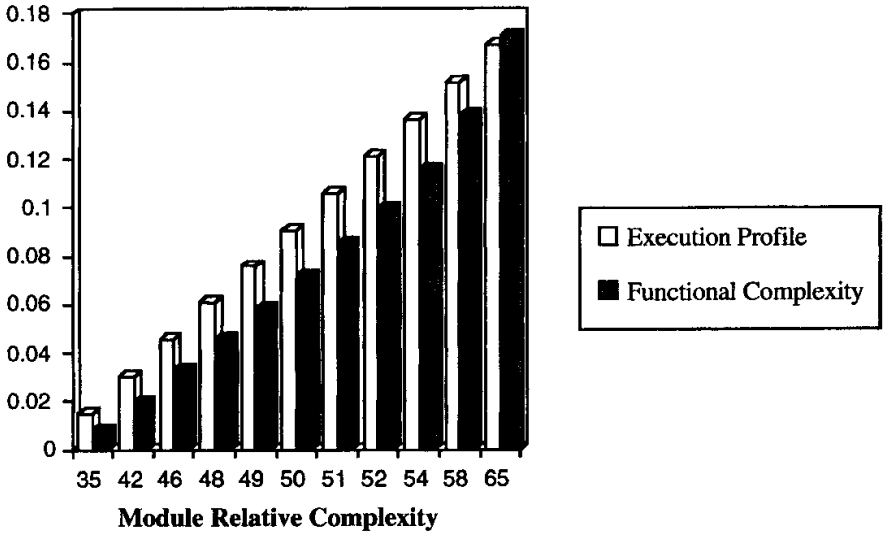

Figure 12.4 High functional complexity.

12.3.3 Dynamic aspects of functional complexity

System functional complexity can be determined in two dimensions. First, the functional complexity will vary in accordance with the function being executed. It will also vary as a function of the reconstitution of the software over builds. Thus, the functional complexity for function f_i at the jth build represented by ν^j will be

$$\phi_{ij} = \sum_{j=1}^{n} p_k^i \rho_k^{\nu_k^j}$$

It is possible to determine the functional complexity for various execution profiles at varying stages of software maturity and thus to indi-

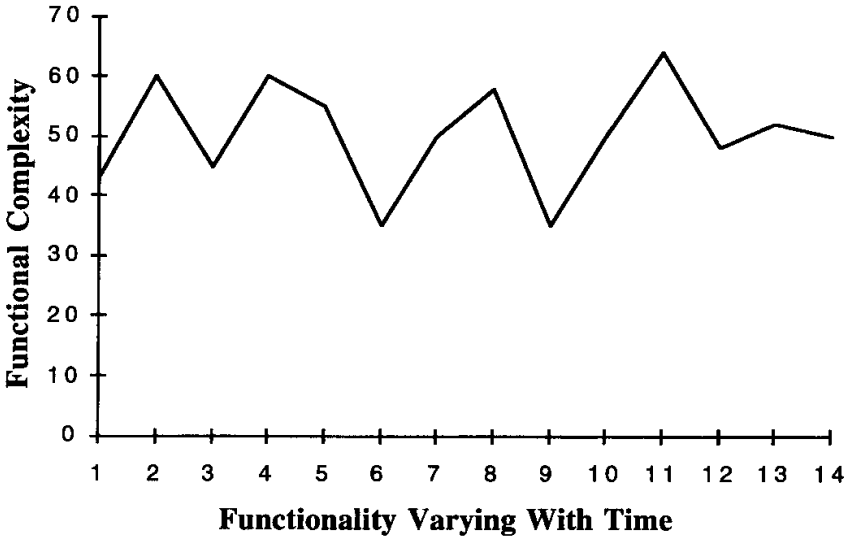

Figure 12.5 Varying functional complexity.

cate their likely failure intensity. In order to model correctly the reliability of a software system we have to determine the functionality of the program and how these functions will interact as the program executes. The latter information not only directs the formation of test suites but also provides the information necessary to formulate execution profiles. The functionalities that imply execution profiles which cause the functional complexity to increase merit our attention since these are the conditions that will increase failure rates for a given design.

Example 12.3 Programs may be seen to differ in terms of the variability of the system functional complexity. Some software systems will be fairly homogeneous in terms of their functionalities. There will be very little diversity in the range of things that the software will do. As a consequence, little variability will be observed in functional complexity from one application to another. This scenario is represented graphically in Fig. 12.6 for a sample set of 14 program modules under four distinct tests of the program. The execution profiles induced by each of these tests are similar. Thus little variation can be seen in the functional complexity from one test to another. It will be very easy to characterize the functional complexity of this system from a statistical perspective.

Figure 12.7 shows a very different software system where there is a substantial difference in functional complexity from one test case to another. It will not be nearly so easy to characterize the behavior of this program in terms of its functional complexity over varying test scenarios. Great diversity can be identified from the execution profiles. Some tests will result in high functional complexity while others will result in low functional complexity. Given the relationship between software complexity and software faults, some tests will lead to consistent failures while others will not. The most important message here is that a software system will fail in direct relationship to its functionality. This is a very predictable and understandable relationship.

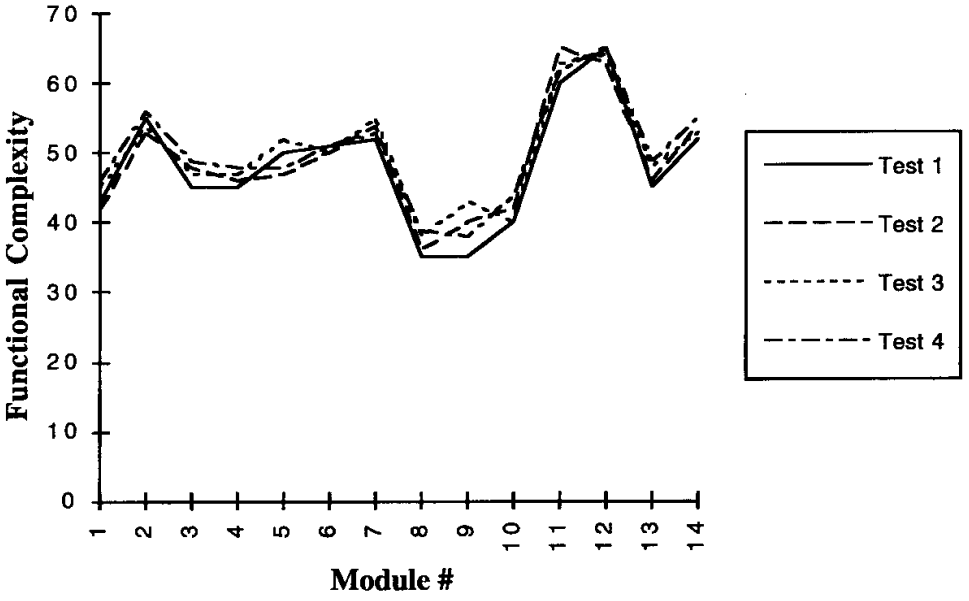

Figure 12.6 Low variability in functional complexity.

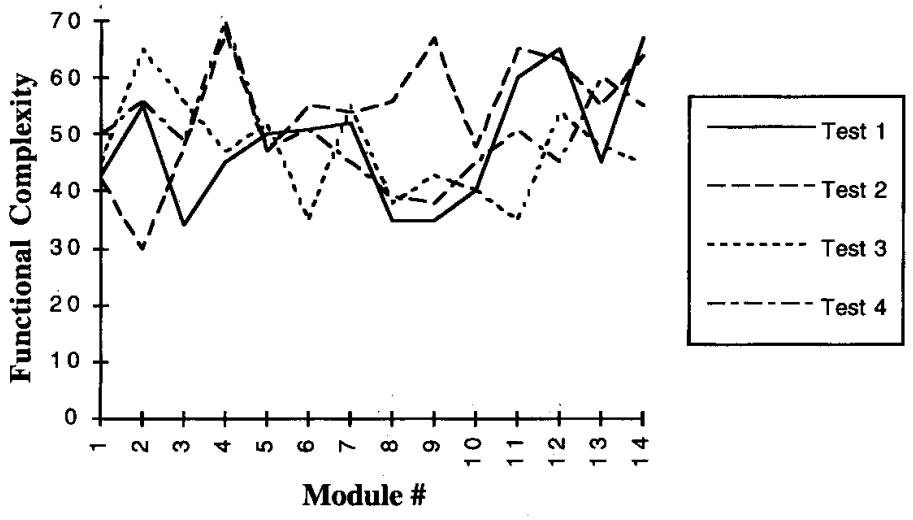

Figure 12.7 High variability in functional complexity.

12.3.4 Operational complexity

We must now come to terms with the fact that whenever a program executes, the changing functionalities will cause it to select only a subset of possible paths from the set of all possible paths through the control flow graph representation of the program. As each distinct functionality f_i of a program is expressed, a subset or *subgraph* of the program and each of its modules m_i will execute. The source code subset of the original program represented by the subgraph m_i will have a relative subgraph complexity ξ_i, where $\xi_i \leq \rho_i$. This relative subgraph complexity value will represent the complexity of just the code that was executed, which clearly cannot be more complex than the original program.

The *operational complexity* ω of a system running an application that generates an execution profile P is

$$\omega = \sum_i p_i \xi_i$$

The execution profile for a program will change over time as a function of system inputs. The relative subgraph complexity of a program will also change over time as a function of the same inputs. For each functionality f_i, there is an operational profile represented by the probabilities

$$\mathbf{P}^i = < p_1^i, p_2^i, \ldots, p_n^i >$$

For each functionality, f_i, there is also a complexity of module m_k on the jth build represented by ξ_k^j. For this functionality, the operational complexity of a system may be represented as

$$\omega_{ij} = \sum_k p_k^i \xi_k^{v\,j}$$

The complexity, then, of a program will vary in direct relationship to the particular subgraph selected by each element in the set of program functionalities. Each subgraph has its own distinct complexity attributes.

12.4 Software Complexity and Software Quality

12.4.1 Overview

A distinct relationship exists between software faults and measurable program attributes. This information can yield specific guidelines for the design of reliable software. In particular, software complexity measures are distinct program attributes that have this property [Khos90]. Generally, if a program module is measured and found to be complex, then it will have a large number of faults. These faults may be detected by analytical methods, e.g., code inspections. The faults may also be identified based on the failures that they induce when the program is executing. A program may preserve a number of latent faults over its lifetime in that the particular manner that it is used may never cause the complex code sequences to execute and thus never expose the faults. Alternatively, a program may be forced to execute its complex code segments early in its life cycle and thus fail frequently early on, followed by reliable service after repair.

Code faults are not inserted by some random process. Faults occur in direct relationship to the complexity of the programming task. A pro-

grammer is faced with the task of converting a complex requirement into a complex algorithm in a rich programming language. It is quite reasonable to expect that the programmer will make mistakes. These mistakes will express themselves as faults in the program. From a maintenance perspective, it will be very expensive and time-consuming to find and to fix these faults. The real problem is to identify design rules that will restrict code faults from being introduced in the first place.

Typically, a small fraction of the modules comprising a software system tend to be complex. For example, [LeGa90] observed several software systems and found that only 4 to 6 percent of the modules were complex, 32 to 36 percent were simple, and the remaining modules fell between these extremes. The complex modules will cause great problems during software development and test. To take early actions that increase software reliability, software engineers must understand the relationships between software attributes that are measurable early in the product life cycle and the software quality characteristics that are not measurable until late in this cycle. Multivariate analytic techniques are necessary when approaching these relationships statistically, since both software complexity and software quality are multidimensional concepts. Software quality models exploit the relationship between static software complexity and software quality metrics. Software engineers fit software quality models to data collected from past projects. With a fitted model, engineers predict the number of faults that testing and operation will reveal in the modules of a similar ongoing project, or identify fault-prone programs of this project [Bria92, Henr91, Khos92b, Khos93c, Khos94a, Lyu95b].

In this section, we consider two multivariate techniques that are useful in developing software quality models: multiple discriminant analysis and multiple regression analysis. *Multiple discriminant analysis* is an analytic technique used to classify objects into two or more mutually exclusive and exhaustive groups based upon a set of independent variables. The independent variables are p measures of object attributes. For software quality models, the objects are program modules and the independent variables are software complexity measures. The classes are defined a priori based upon some criterion. For example, a program module could be classified as high- or low-risk, based upon the number of changes required to remove faults from the module. In this case, some value of change count, say, five, is the criterion. Thus if the module requires more than five changes to remove faults, then it is high-risk, otherwise it is low-risk. Discriminant analysis derives a linear combination of the independent variables that discriminates between the a priori groups such that misclassification error rates are minimized. Section 12.4.3 applies discriminant analysis to software quality control.

Multiple regression analysis is an analytic technique used to assess the relationship between a dependent variable and a set of independent variables. Again, the independent variables are p measures of object attributes, software complexity measures for software quality models. The dependent variable is a measure of some interesting quality attribute that is believed to vary with some linear combination of the measurable software attribute. For example, one might suspect that the number of changes required to remove faults from a module will vary with the static complexity of this module. Regression analysis is concerned with predicting the mean value of the dependent variable using known values of the independent variables. Section 12.4.4 applies multiple-regression modeling to software quality control.

[Schn92a] noted that metric-based models can give inconsistent results across development projects due to variations in product domains and other product characteristics, as well as variations in process maturity levels, development environments, and the skill and experience of people. To minimize the risk in applying quality models, validated criterion values of metrics obtained from one project are applied to another project. This procedure is performed at the completion of one effort and the inception of the other to determine the aptness of a quality model fitted to data from the completed effort for predicting results of the new effort.

12.4.2 The application and its metrics

Data collected during the development and maintenance of a Medical Imaging System (MIS) provide for examples in Secs. 12.4.3 and 12.4.4. MIS is a commercial software system consisting of approximately 4500 routines written in about 400,000 lines of Pascal, FORTRAN, and PL/M assembly code. MIS development took five years, and the system has been in commercial use at several hundred sites for three years. [Lind89] collected the number of changes made to each module due to faults discovered during system testing and maintenance, as well as 11 software complexity metrics for each of the modules that comprise MIS. The following list describes the software complexity metrics:

- *LOC* is the number of lines of code, including comments.
- *CL* is the number of lines of code, excluding comments.
- *TChar* is the number of characters.
- *TComm* is the number of comments.
- *MChar* is the number of comment characters.
- *DChar* is the number of code characters.

- $N = N_1 + N_2$ is program length, where N_1 is the total number of operators and N_2 is the total number of operands [Hals77].
- $\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$ is an estimated program length, where η_1 is the number of unique operators and η_2 is the number of unique operands.
- $N_F = (\log_2 \eta_1)! + (\log_2 \eta_2)!$ is Jensen's estimator of program length [Lind89].
- $V(G)$, McCabe's cyclomatic number, is one more than the number of decision nodes in the control flow graph [McCa76].
- BW is Belady's bandwidth metric [Lind89], where $BW = 1/n \sum_i i L_i$ and L_i represents the number of nodes at level i in a nested control flow graph of n nodes. This metric is indicative of the average level of nesting or width of the control flow graph representation of the program.

The techniques we employ to analyze these data are not limited to this particular selection of metrics. The process would certainly benefit from a richer set than was available. Our goal is to demonstrate the modeling technique, not to justify the use of any particular selection of metrics. Before applying any modeling technique, software engineers must select a set of metrics that is suitable in their unique environment. The process of selecting and validating of metrics has been well studied in the software metrics literature [Muns89, Schn92a, Zuse91, Fent91].

We consider an MIS subset of 390 modules written in Pascal and FORTRAN for modeling (see `MIS.DAT` on the Data Disk). These modules consist of approximately 40,000 lines of code. When applying any modeling technique, an assessment of predictive quality is important. *Data-splitting* is a modeling technique that is often applied to test predictive quality. Applying this technique, one randomly partitions the data set to produce two data sets. The first data set is used for fitting the models. The remaining data set, the validating data set, provides for quantifying the predictive quality of the fitted models. Data-splitting randomly partitioned the 390 observations of MIS data into two sets, two-thirds of the observations comprising the fitting data set, and the remaining one-third comprising the validating data set.

Both multiple-regression modeling and discriminant analysis carry the assumption that no multicollinearity exists among independent variables. Violations of this assumption result in unstable models. For models based upon complexity data, domain metrics typically outperform correlated complexity metrics producing models with greater stability and predictive quality. The principal components analysis technique discussed in Sec. 12.2.3 will be used to create orthogonal domain metrics to eliminate potential problems of multicollinearity among the selected metrics.

The principal components analysis of the MIS-fitting data set produced two complexity domains. Table 12.3 gives the domain patterns of these two domains. This table shows the degree of relationship between each of the complexity metrics and the two domains. Domain 1 is strongly correlated with all of the metrics except *BW*. Each of these metrics are related to program size. Domain 2 is correlated with *BW*, a metric related to program control flow. Together, these two domains account for about 90 percent of the variability seen in the complexity metrics.

Table 12.4 gives the standardized transformation matrix, $\mathbf{T}*$, for the domains. This matrix and the vectors of standardized complexity measures from the fitting and validating data sets yield two domain metric values for each of the program modules in these data sets. This transformation matrix was produced during the principal components analysis of the fitting data set and then used as a baseline transformation for the validation data set. These domain metric values and the *changes* associated with each module provide data for illustrating the techniques presented in the following sections.

12.4.3 Multivariate analysis in software quality control

In the application of multiple discriminant analysis to program risk classification, the independent variables are p complexity domain metrics, and the criterion for class membership is a given value of a software quality measure related to program faults [Khos94a, Muns92]. A model is derived to optimally classify a collection of modules with known domain metric values and known class memberships. It is

TABLE 12.3 Principal Components Domain Patterns

Metric	Domain 1 Size	Domain 2 Control
TChar	**0.965**	0.192
LOC	**0.964**	0.213
CL	**0.941**	0.265
DChar	**0.932**	0.285
N_F	**0.917**	0.269
$\hat{N}$	**0.916**	0.273
N	**0.909**	0.306
TComm	**0.899**	0.107
MChar	**0.833**	-0.010
V(G)	**0.799**	0.481
BW	0.145	**0.964**
Eigenvalues	8.291	1.650
% variance	75.373	15.000
Cumulative % variance	75.373	90.037

TABLE 12.4 Standardized Transformation Matrix

	Domain 1	Domain 2
Metric	Size	Control
TChar	0.136	–0.074
LOC	0.131	–0.053
CL	0.112	0.004
DChar	0.105	0.026
N_F	0.106	0.014
$\hat{N}$	0.105	0.019
N	0.095	0.053
TComm	0.147	–0.140
MChar	0.166	–0.238
V(G)	0.025	0.256
BW	–0.235	0.912

expected that this model will achieve a low misclassification rate in a similar software development environment for untested program modules with known complexity metric values.

A two-class model classifies modules as either high-risk or low-risk based upon known domain metric values. A module is known to be high-risk or low-risk after a suitable period of testing and field experience reveals the number of changes required to remove faults discovered in it, that is, the criterion variable is *changes.* The criterion divides the set of modules into two classes, that is, modules having values of *changes* greater than some selected number are high-risk, while those having no more than this number are low-risk. Software engineers select this criterion variable value based upon the history of similar projects. An analysis of historical data will reveal the criterion value that isolates the set of modules that were considered troublesome in past development experience. The size of this set, and thus the appropriate value of the criterion variable, will vary with aspects of the product under development and the software development process.

One of several discriminant techniques may be appropriate for a given analysis. The choice lies with the types of the independent variables used in the analysis. The independent variables may be all quantitative, all qualitative, or a mixture of these two types. Applicable models in each of these cases use, respectively, linear, discrete, and logistic discrimination techniques [Dill84]. Since all of the software complexity metrics in this study are quantitative, we use a linear discriminant model and restrict our discussion to techniques of this type.

In the linear discriminant model that we develop, an observation, $\mathbf{x}$, is a vector of software complexity metrics. Let $\bar{\mathbf{x}}_j$ represent the mean of two classes, $j = 1, 2$. Then the generalized squared distance from an observation to the mean of each class is

$$D_j^2(\mathbf{x}) = (\mathbf{x} - \bar{\mathbf{x}}_j)^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \bar{\mathbf{x}}_j)$$

where Σ is the pooled covariance matrix. Thus the posterior probability of membership of $\mathbf{x}$ in class j is

$$p_j(\mathbf{x}) = \frac{e^{-1/2D_j^2(\mathbf{x})}}{e^{-1/2D_1^2(\mathbf{x})} + e^{-1/2D_2^2(\mathbf{x})}}$$

The model assigns an observation $\mathbf{x}$ to the class j having greater posterior probability of membership, that is, the model selects j such that $p_j(\mathbf{x}) = \max(p_1(\mathbf{x}), p_2(\mathbf{x}))$.

We observe two aspects of quality in discriminant models. First, a model must be successful in classifying program modules having known complexity data, but unknown fault data. Second, a model must be able to perform this classification with little uncertainty. The misclassification rates of a model measure its success in classifying program modules. A model can commit two types of classification errors. A type 1 error occurs when a low-risk module is classified as high-risk. This could result in some wasted attention to low-risk modules. A type 2 error occurs when a high-risk module is classified as low-risk. This could result in an extension of the scheduled release date as more effort is required than planned for, or the release of a lower-quality product. The nature of the impacts of these error types suggests that the type 2 error rate is more important than the type 1 error rate in considering the quality of a classification model.

The model assigns a module to one of the two classes based upon some function of the module's complexity domain metric data. Modules with values of this function above some cutoff value fall in one class; the remaining modules fall in the other class. For some modules the function value will fall far from the cutoff value that determines class membership. These modules have a high probability of falling in the assigned class. For other modules the function value will fall close to the cutoff value. These modules have a relatively low probability of falling in the assigned class. For correct classifications, the model probability of membership in the opposite class is the uncertainty in the classification.

Example 12.4 We develop a discriminant model for classifying each MIS module as either high- or low-risk. The domain metric and change data values for the 260 modules in the MIS-fitting data set serve to fit this model. The value of the criterion variable, *changes,* is one. Modules requiring one or fewer changes to remove faults are low-risk, those requiring more than one change are high-risk. To magnify the difference between the high- and low-risk modules, we biased the training data set before training the models [Muns92]. We achieved this by removing all of the modules with values of *changes* between 2 and 9. This left 156 modules in the fitting data set: 126 low-risk modules and 30 high-risk modules.

Application of the fitted model to classify the 130 modules in the MIS testing data set serves to test the predictive quality of this model. The model classifies

these modules based upon their domain metric values. Since the modules have known values of *changes,* the model's misclassification rates are known for this data set. Tables 12.5 and 12.6 give the results of the discriminant model for modules in the upper and lower extremes with regard to those with one or fewer fault-correcting changes and those with 10 or more fault-correcting changes. The modules appearing in Table 12.5 belong to the low-risk class. Those appearing in Table 12.6 belong to the high-risk class. These two tables give the domain metric values, the relative complexity metric value, the number of changes, and the model classification for the modules in the testing data set. Tables 12.5 and 12.6 also include the model uncertainty in correct classifications. Table 12.5 shows

TABLE 12.5 Classification Data for Modules with One or Fewer Changes

Program number	D_1	D_2	ρ'	Changes	Pred. group	Uncertainty
1	−0.63	−0.59	42.6	0	1	0.04
2	0.48	−1.02	52.7	0	2	—
3	−0.67	−0.26	42.8	0	1	0.04
4	−0.95	1.27	43.1	0	1	0.33
5	−0.33	0.98	48.6	0	1	0.35
6	−0.37	1.04	48.3	0	1	0.31
7	−0.80	0.52	43.1	0	1	0.07
8	−0.39	−0.08	45.9	0	1	0.06
9	0.05	−0.68	49.2	0	2	—
10	−0.38	−0.88	44.5	0	1	0.06
11	−0.85	−0.13	41.3	0	1	0.07
12	−0.76	−0.63	41.2	0	1	0.06
13	−1.43	2.75	41.3	0	2	—
14	−0.58	−0.85	42.5	0	1	0.04
15	−0.48	−0.73	43.7	0	1	0.04
16	−0.18	0.30	48.7	0	1	0.33
17	−0.70	−0.57	41.9	0	1	0.04
18	−0.42	−0.94	43.9	1	1	0.05
19	−0.66	−0.73	42.0	1	1	0.04
20	−0.48	−0.04	45.1	1	1	0.04
21	−0.55	0.17	44.8	1	1	0.04
22	−0.73	1.18	45.0	1	1	0.16
23	0.45	−0.47	53.5	1	2	—
24	−0.28	−0.22	46.7	1	1	0.09
25	−0.39	−0.80	44.5	1	1	0.05
26	−0.59	0.61	45.3	1	1	0.06
27	−0.23	−0.76	46.2	1	1	0.12
28	−0.59	−0.85	42.5	1	1	0.04
29	−0.75	−0.86	40.9	1	1	0.07
30	−0.85	−0.46	40.7	1	1	0.08
31	−0.41	−0.79	44.3	1	1	0.05
32	−0.23	−0.74	46.2	1	1	0.12
33	−0.67	−0.02	43.3	1	1	0.04
34	−0.23	0.65	48.9	1	1	0.38
35	−0.37	−1.12	44.1	1	1	0.07
36	−0.56	−0.75	42.9	1	1	0.04
37	−0.88	−0.29	40.7	1	1	0.09
38	−0.15	−0.15	48.1	1	1	0.26

that the model misclassified 4 of the 38 low-risk modules, yielding a type 1 error rate of about 10 percent. Table 12.6 shows that the model misclassified 4 of the 30 high-risk modules, giving a type 2 error rate of about 13 percent. The average uncertainty for high- and low-risk classifications is about 11 and 3 percent, respectively. Overall, this model misclassified about 12 percent of the modules in this study. This demonstrates an average uncertainty of about 7 percent.

12.4.4 Fault prediction models

In the application of multiple regression modeling to fault prediction, the independent variables are p complexity domain metrics, and the dependent variable is a software quality measure related to program faults [Khos90, Khos93b]. The first step in multiple regression modeling is model selection. In this step, one selects a subset of the p attributes to include in the regression. Several techniques are available for selecting this subset. These include stepwise regression, forward selection, backward elimination, the R^2 criterion, and C_p criterion.

TABLE 12.6 Classification Data for Modules with 10 or More Changes

Program number	D_1	D_2	ρ′	Changes	Pred. group	Uncertainty
101	2.40	1.46	76.4	10	2	0.00
102	−0.27	0.25	47.7	10	1	—
103	1.12	0.50	61.9	11	2	0.00
104	−0.04	−0.12	49.2	11	2	0.42
105	−0.28	1.01	49.2	11	1	—
106	1.53	−0.85	63.3	11	2	0.00
107	1.36	1.98	67.2	12	2	0.00
108	−0.07	−0.23	48.8	12	1	—
109	1.37	1.78	66.9	12	2	0.00
110	0.38	0.81	55.4	13	2	0.00
111	0.60	2.15	60.1	14	2	0.00
112	−0.20	0.97	49.9	14	2	0.33
113	2.57	−1.32	72.6	15	2	0.00
114	0.34	0.66	54.6	15	2	0.00
115	−0.53	−0.36	44.0	16	1	—
116	1.52	2.13	69.1	16	2	0.00
117	1.58	2.53	70.4	17	2	0.00
118	2.23	3.28	78.3	19	2	0.00
119	1.06	−0.28	59.8	20	2	0.00
120	0.25	−1.00	50.5	22	2	0.00
121	0.59	−1.21	53.4	25	2	0.00
122	0.96	0.01	59.4	28	2	0.00
123	3.98	−0.77	87.6	30	2	0.00
124	1.66	1.83	69.9	30	2	0.00
125	1.78	0.18	67.8	34	2	0.00
126	1.34	−0.34	62.5	38	2	0.00
127	2.53	−0.97	72.9	40	2	0.00
128	4.90	2.03	102.1	42	2	0.00
129	1.09	−1.12	58.5	46	2	0.00
130	4.28	0.08	92.2	98	2	0.00

In the *stepwise regression* analysis procedure, an initial model is formed by selecting the independent variable with the highest simple correlation with the dependent variable. In subsequent iterations new variables are selected for inclusion based on their partial correlation with variables already in the regression equation. Variables in this model may be removed from the regression equation when they no longer contribute significantly to the explained variance. There must be an a priori level of significance chosen for the inclusion or deletion of variables from the model. The second stepwise procedure is *forward inclusion.* In the case of this procedure, a variable once entered in the regression equation may not be removed. The third technique, *backward elimination,* forms a regression equation with all variables and then systematically eliminates variables, one by one, which do not contribute significantly to the model. For further details concerning model selection, refer to [Myer90].

Stepwise procedures for the selection of variables in a regression problem should be used with caution. These are useful tools for variable selection only in the circumstances of noncollinearity. We recommend a different set of procedures in the presence of collinearity. Once collinearity has been identified, a set of new variables, principal components, can be formed by using principal components analysis (see Sec. 12.2.3). These new variables will not be collinear. Then, stepwise procedures are used to select the factors which are important for predicting the dependent variable, which in our case will be an enumeration of programming faults.

Traditionally, the R^2 statistic is used almost exclusively in empirical studies in software engineering. Some distinct problems are associated with the use of R^2, which is defined as follows:

$$R^2 = \frac{\text{regression sum of squares}}{\text{sum of squares about the mean}}$$

Alternatively,

$$R^2 = \frac{\Sigma(\hat{Y}_i - \bar{Y})^2}{\Sigma(Y_i - \bar{Y})^2}$$

In that $\Sigma(Y_i - \overline{Y})^2$ is constant for all regression models, R^2 can increase only as independent variables are added to a regression equation, whether or not they will account for a significant amount of variance in the dependent variable. It is important to note that the R^2 statistic does not assess the quality of future prediction. If a model is sufficiently tailored to fit the noise and other aberrations in the data, then it is quite possible to develop a model that fits the data well but is worthless for future prediction. While we are interested in the fact that

the model fit the data, our primary focus should be on the ability of the chosen model to render worthwhile future predictions.

The case for the C_p statistic is very different. C_p may be defined in terms of R_p^2 as follows:

$$C_p = \frac{(1 - R_p^2)(n - T)}{1 - R_T^2} - (n - 2p)$$

where n represents the number of observations and T represents the total number of parameters in the complete model. The statistic C_p is a measure of the total squared error in a regression. Thus, a researcher should choose a model with the smallest value of C_p. This statistic is to be preferred to R^2 because a penalty is introduced for overfitting the model with excess independent variables, which bring with them an additional noise component.

After selecting a model, we must fit it to the observed data. Some notation becomes helpful here. For N observations, the dependent variable values y_i will have corresponding predicted value, $\hat{y}_i$, produced by the model and a residual value $\varepsilon_i = y_i - \hat{y}_i$ representing the difference between the value predicted by the model and the observed value. For a model with p independent variables, $x_1, x_2, \ldots, x_p$, the least-squares estimation technique yields estimated model parameters, $\beta_0, \beta_1, \ldots, \beta_p$, such that, for $1 \le i \le N$,

$$\hat{y}_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip} \qquad \text{and the sum of} \quad \sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

is minimized.

Example 12.5 Applying these concepts to the MIS data, we select a model for predicting *changes* for each MIS module given the domain metric values for these modules. The domain metric and change data values for the 260 observations in the MIS fitting data set serve to fit this model. The best model revealed by all model selection methods discussed above includes both domain metrics. The fitted model is given by

$$Changes = 7.15 + 7.07D_1 + 1.36D_2$$

The domain metric and change data values for the 130 observations in the MIS testing data set serve to measure the predictive quality of the fitted model. The average absolute error of the model on these data is 5.32.

12.4.5 Enhancing predictive models with increased domain coverage

One aspect of program variability that is not currently represented by any code metrics in the domain model of program complexity is a measure of the complexity of data structures in a program. For example, consider the case of a program to find the average value of a set of real

numbers. This program may be written using only scalar values to accumulate the developing sum and a count of the numbers processed so far. On the other hand, the numbers may be copied into an array as they are read. After all of the numbers have been read, the contents of the array may be tallied to compute the sum. Clearly, these two programs will differ as a result of the complexity of the data structures that they will contain. They will also employ completely different algorithms to compute the sum. Hence, measures of their algorithmic complexity will also be different.

[Muns93] offered a measure of data structure complexity that is based upon a specification of the complexity of each data structure in the implementation language. The data structure complexity of a module, *DS,* is evaluated as the sum of the complexities of the data structures used in a program module.

Central to the idea of validating a new measure is the notion that it is measuring an attribute of programs not already being measured by another metric. That is, the new metric must map into a new orthogonal domain not already in the domain model. There must be some aspect of variability between programs that is attributable to the new metric that is not present to a large extent in other measures taken on the program. The utility of the new metric will be assessed by its ability to contribute new, distinct, and meaningful information to our understanding of program complexity.

For the present study, our metric analyzer for Ada code was augmented to compute data structure values for Ada source programs. This analyzer computes 16 complexity metrics for Ada program modules at the package level. In addition to *DS,* this analyzer computes a subset of the metrics defined in Sec. 12.4.2, including N_1, N_2, η_1, η_2, $V(G)$, and *BW,* along with the metrics defined in the following list:

- *Stmts,* the number of Ada statements
- *Paths,* the number of unique paths in the control flow graph
- *Cycles,* the number of cycles in the control flow graph
- *Max–Path,* the longest path in the control flow graph
- $\overline{Path}$, the average length of a path in the control flow graph
- $V(G)$, McCabe's cyclomatic complexity
- *CO,* the number of calls out of a package
- *CI,* the number of calls into a package
- *Global,* the number of global data references [Nav187]
- $\overline{Span}$ the number of levels up the procedure tree that the analyzer had to go to find the definition of each data item being referenced divided by the number of references [Nav187]

A total of 240 separately compilable Ada units or packages were measured. First, we were interested in the structure of the underlying complexity domains without the data structure metric, *DS*, present. The domain structure for this analysis is shown in Table 12.7. For the set of 15 metrics shown, there were four distinct domains. The resulting domain structure had four domains in it with a stopping rule for the analysis that the associated eigenvalues for each new domain be greater than or equal to 1.

Domain 1 shows a high correlation with *Stmts*, N_1, N_2, and *Paths*. These metrics all relate to size or volume of a program. Hence, we will identify this domain as relating to a *size* domain in our domain model. Similarly, domain 2 could be called a *control flow* domain, domain 3 an *action / reusability* domain, and domain 4 a *modularity* domain.

For the second analysis, the data structures metric, *DS*, was added to the set of 15 primitive metrics and the data were reanalyzed. The domain structure for this second analysis is shown in Table 12.8. In this case the resulting domain structure had five domains in it, again, with a stopping rule of 1 for the eigenvalues. The essential domain structure has been preserved from the first analysis to the second. That is, with only one exception, the size domain contains the same set of metrics in both analyses. Similarly, the control flow, action/reusability, and modularity domains contain the same sets of metrics in both

TABLE 12.7 Domain Pattern Without Data Structure Metric

Metric	Domain1 Size	Domain2 Control flow	Domain3 Action/ reuseability	Domain4 Modularity
N_1	**0.960**	0.202	0.086	0.007
N_2	**0.951**	0.142	0.153	0.026
Stmts	**0.893**	0.350	0.092	–0.008
Global	**0.887**	0.278	0.027	–0.050
CO	**0.877**	0.189	0.029	–0.024
η_2	**0.644**	–0.042	0.443	0.095
Paths	**0.600**	0.374	–0.286	–0.126
Max–Path	0.328	**0.906**	0.068	–0.007
Path	0.349	**0.892**	0.069	–0.023
Cycles	0.164	**0.772**	0.026	–0.045
Band	0.102	**0.765**	0.372	0.104
η_1	0.118	**0.746**	0.480	0.057
Span	–0.012	0.199	**0.713**	–0.162
V(G)	0.319	0.431	**0.546**	0.077
CI	–0.028	0.035	–0.087	**0.968**
Eigenvalue	5.345	4.026	1.515	1.016
% variance	35.633	26.840	10.100	6.773
Cumulative % variance	35.633	62.473	72.573	79.347

analyses. A new domain, however, is present in the second analysis that was not present in the first. This is the new domain containing the *DS* metric.

With the set of primitive metrics employed in this study, *DS* has served to identify another complexity domain. In this regard, it is important to note the eigenvalues reported in Tables 12.7 and 12.8. These eigenvalues represent the relative proportion of variance accounted for by each of the domains. The larger the eigenvalue the more variance attributable to each domain. The sum of the eigenvalues for the domains in each of the tables represents the total variance accounted for by that domain structure. The four domains displayed in Table 12.7 account for about 79 percent of the variance explained by the underlying metrics, while the five domains displayed in Table 12.8 account for about 84 percent of the variance explained by the underlying metrics. Thus, the increase in domain coverage by the addition of the data structures domain has increased the ability of the domain model to describe differences among the programs being measured.

12.5 Software Reliability Modeling

Current software reliability modeling approaches are, in some cases, simply extensions of hardware reliability models. Our view of complex

TABLE 12.8 Domain Pattern with Data Structure Metric

Metric	Domain1 Size	Domain2 Control flow	Domain3 Data structures	Domain4 Action/ reuseability	Domain5 Modularity
N_1	**0.922**	0.197	0.111	0.153	−0.003
N_2	**0.899**	0.284	0.195	0.165	0.023
Stmts	**0.897**	0.119	0.135	0.099	0.006
Global	**0.871**	0.181	0.425	0.044	0.002
CO	**0.818**	0.141	0.527	0.059	0.006
η_2	**0.706**	0.295	−0.104	−0.108	−0.067
Paths	0.350	**0.894**	0.095	0.095	−0.000
Max–Path	0.372	**0.878**	0.095	0.099	−0.015
Path	0.131	**0.811**	0.180	−0.050	−0.075
Cycles	0.087	**0.741**	0.187	0.437	0.046
Band	0.144	**0.730**	−0.013	0.452	0.128
DS	0.117	0.267	**0.860**	−0.027	−0.034
η_1	0.353	0.042	**0.841**	0.144	0.011
Span	0.011	0.116	0.005	**0.840**	−0.117
V(G)	0.270	0.409	0.146	**0.546**	0.102
CI	−0.026	0.013	−0.019	−0.054	**0.980**
Eigenvalues	4.912	3.841	2.095	1.524	1.017
% variance	30.700	24.006	13.094	9.525	6.356
Cumulative % variance	30.700	54.706	67.803	77.328	83.684

software systems is colored by our experience with complex mechanical or electronic systems. The dynamic complexity of the software system will depend on the inputs to the system. The net effect of differing inputs to the system is that the operational or functional complexity of the system will change in response to the varying inputs. Given the association between module complexity and faults, it follows that as applications change over time intervals, so too will the likelihood of faults change with respect to time.

12.5.1 Reliability modeling with software complexity metrics

For the purpose of demonstration, we will now explore how the notions of relative complexity and functional complexity may be incorporated into a Bayesian model for reliability measurement. There are good reasons for employing a Bayesian technique in this modeling process. In this model, successive execution times between failures are independent random variables $T_1, T_2, \ldots, T_i, \ldots$, where T_i is the execution time of the software system being modeled, from the time of the repair of the $(i-1)$st failure to the ith failure. By convention, t_i will represent an observation of the random variable T_i. The T_i's are assumed to have an exponential probability density function (pdf) with a parameter λ of

$$f(t_i,\lambda(i)) = \lambda(i)e^{\lambda(i)t_i} \qquad t > 0 \qquad \lambda > 0$$

The parameter λ is related to the failure rate of a program. As the program progresses through the testing process, traditional hardware reliability modeling theory would have the failure rate diminish as fixes are made to the system, i.e.,

$$\lambda(i-1) > \lambda(i)$$

In software development applications, we cannot always assume that the system failure rate will improve as a result of a fix. Sometimes a fix will introduce faults of its own.

In the Littlewood-Verrall (Bayesian) model (see Sec. 3.6.1), the parameter λ is assumed to have a pdf of its own. Let the pdf of $\lambda(i)$ be denoted by $g(l, i, \alpha)$, where α is a parameter or vector of parameters. From the standpoint of mathematical tractability, the pdf of $\lambda(i)$ is chosen to have a gamma distribution in two parameters, α and $\psi(i)$ as follows:

$$g(l, i, \alpha) = \frac{\psi(i)[\psi(i)l]^{\alpha-1}e^{-\psi(i)l}}{\Gamma(\alpha)} \qquad t > 0 \qquad \lambda > 0$$

The parameter, $\psi(i)$, is essentially a scaling factor and is a monotonically increasing function of i. This assumption will guarantee the

ordering of the distribution functions in i. Further, $\psi(i)$ is not estimated but is completely determined as a measure of a debugger's *behavior* at time i. The main problem here is that we have left the well-defined realm of mathematics and entered the realm of psychology. We would like to derive more substantive reliability models than those based solely on the attitudes or competencies of programmers.

It would seem reasonable that some measure of system complexity may be used for the ill-defined parameter, ψ. If, on the one hand, it is important that $\psi(i)$ be a monotonically increasing function of i, then the system relative complexity measure, ρ_s^i, would meet the monotonically increasing property as a function of time. Hence,

$$g(l, i, \alpha) = \frac{\rho_s^i (\rho_s^i l)^{\alpha-1} e^{-\rho_s^i l}}{\Gamma(\alpha)} \qquad t > 0$$

The property that $\psi(i)$ be a monotonically increasing function of i is an unnecessary restriction. This property will certainly ensure that the failure rate is a decreasing function of time. Empirical observations, however, do not support this view of failure rate. In many cases, we do observe an increase in failure rate during some time intervals.

Many dynamics operate on a program during the period it is measured for reliability modeling. As the functional complexity induced by test scenario on a program increases, so too will its exposure to code likely to contain faults. The failure rate at any time, i, ought to reflect which functions the program is executing during this time. Hence, we feel that the functional complexity ϕ_i at time i is a much better parameter for this model, in which case,

$$g(l, i, \alpha) = \frac{\phi_i (\phi_i l)^{\alpha-1} e^{-\phi_i l}}{\Gamma(\alpha)} \qquad t > 0$$

Consequently, two distinct variants in the Bayesian case are observed: one with relative complexity and one with dynamic complexity. In these modeling approaches some rather intensive measurements must be made on the software. It is not sufficient to record the time that the system failed; it is also necessary to measure the incremental versions of the systems in terms of their complexity as well as the operational profiles generated by test scenarios over time.

To complete the discussion of the Bayesian models, we will use the case where $\psi(i) = \phi_i$. It can be shown (see Prob. 12.13) that the maximum likelihood estimate for α, $\hat{\alpha}$, is obtained as

$$\hat{\alpha} = \frac{n}{\sum_{i=1}^{n} \ln\left(1 + \frac{t_i}{\phi_i}\right)}$$

Subsequently, the estimate for the current reliability of the software at the present time, n, is given by

$$\hat{R}(t_n) = 1 - \hat{F}(t_n) = \left(\frac{\phi_n}{t_n + \phi_n}\right)^{\hat{\alpha}}$$

As we can see from this functional relationship, the reliability of the system is directly dependent on the functional complexity induced by the operational profile of the test scenario during this nth time interval. The greater the functional complexity of the test, the less reliable will the software appear to be.

12.5.2 The incremental build problem

In the previous section we have taken a look at how software complexity attributes can be introduced into software reliability models. None of these approaches reflect the fact that most modern software systems are developed incrementally. This is yet another aspect of software development that must be incorporated into our thinking about software reliability modeling.

Functionality will be added incrementally to a developing core system. At any point in time the system is composed of a fixed set of modules. The precise status of the system at any particular build i will be given by the vector $\mathbf{v}^i$ introduced in Sec. 12.2.6. If we examine the contents of this vector, we will see that some modules have received extensive revision. Other modules have not been modified in some time. Those modules that have received continual changes will be substantially more failure-prone than those modules that have a longer period of stability. With this consideration in mind, we can observe that the granularity of the reliability modeling should be at the software module level and not on the software system as a whole [Schn92b].

Each program module in a total system is more than likely at a different stage of maturity as a system is developed. Most modern software systems begin with a nucleus of legacy code. This is code that has been ported from an older application. It has probably been run for some time. It may also be fairly fault-free. On the other hand, new program modules will be added to the system representing enhanced functionality of the software. These new modules will be added to the system over a period of time. At any particular build, the total system will consist of program modules at varying levels of maturity and, consequently, reliability.

Let $T^{\mathbf{v}^i_j}$ be the estimated time to failure of each module m_j at the ith build of the software. The estimated time to failure of the total system, T_{SYS} on the ith build may be represented by

$$T^i_{SYS} = \frac{1}{\frac{1}{T^{\mathbf{v}^i_1}} + \frac{1}{T^{\mathbf{v}^i_2}} + \frac{1}{T^{\mathbf{v}^i_3}} + \cdots + \frac{1}{T^{\mathbf{v}^i_n}}}$$

This means of computing the reliability of a modular software system has been successfully used in the development of reliability estimates for the space shuttle onboard flight software system [Schn92b] (see Sec. 11.7.2).

12.6 Summary

This chapter examines software complexity measurement and software quality as these issues relate to software reliability. We describe the measurement of software attributes for early prediction of software quality. These attributes are primarily software complexity measures, which include relative program complexity and dynamic program complexity. Various strategies have been examined to exploit the relationship between software quality and software complexity. We also demonstrate how software complexity metrics can be included in software reliability models for the enhancement in their reliability predictions, and how reliability estimates for incrementally built systems can be obtained.

Problems

12.1 Software systems are characteristically different from hardware systems. From the standpoint of reliability modeling, exactly how does software differ from hardware?

12.2 System functional complexity is a dynamic measure of complexity. What is the relationship between the static complexity of a system as measured by relative complexity and the dynamic complexity of a system as measured by functional complexity?

12.3 What is the difference between functional complexity and operational complexity?

12.4 How would measurements be taken on a system to compute operational complexity?

12.5 How might existing models of software reliability be enhanced to incorporate measures of dynamic complexity?

12.6 List all the software metrics that strongly correlate with the second domain in Table 12.9.

12.7 Given a program module with the following standardized measurements,

$$
\begin{aligned}
N2 &= -0.38485 \\
N1 &= -0.40943 \\
LOC &= -0.41983 \\
ELOC &= -0.45814 \\
ETA2 &= -0.42401 \\
ETA1 &= -0.13556
\end{aligned}
$$

TABLE 12.9 Domain Pattern

Rotated factor pattern (varimax rotation method)		
	Factor1	Factor2
N2	0.90650	0.40314
N1	0.90241	0.42068
LOC	0.88923	0.39994
ELOC	0.87983	0.44468
ETA2	0.76635	0.58308
ETA1	0.40602	0.90972
Eigenvalues	3.953055	1.864772

TABLE 12.10 Standardized Transformation Matrix, T1

Standardized scoring coefficients		
	Factor1	Factor2
N2	0.37179	–0.24400
N1	0.34825	–0.20546
LOC	0.35964	–0.23069
ELOC	0.30054	–0.13353
ETA2	0.04072	0.26228
ETA1	–0.65691	1.30095

and the transformation matrix in Table 12.10, what are the domain metrics for this module?

12.8 Given the data in Table 12.11, what is the relative complexity metric, ρ_k, and the scaled relative complexity metric, ρ'_k, for each program module?

12.9 Why is multicollinearity an important issue in software quality modeling?

12.10 For a discriminant model that classifies high-risk and low-risk modules, what is a type 1 error and what is a type 2 error?

12.11 What is a measure of the quality of predictions of a multiple linear regression model?

TABLE 12.11 Domain Metrics

Factor scores			
Module	Factor1	Factor2	Factor3
1	–0.505	–0.508	–2.752
2	–0.493	0.874	0.445
3	6.754	–0.322	1.787
Eigenvalues	7.048	2.560	2.547

12.12 List the steps of model development related to the *data-splitting* technique. Why is this technique used?

12.13 Solve the Bayesian model in Sec. 12.5.1. Namely, (1) Express the pdf of t_i given α and ϕ_i; (2) solve $\hat{\alpha}$; (3) obtain the reliability estimate at the present time n.