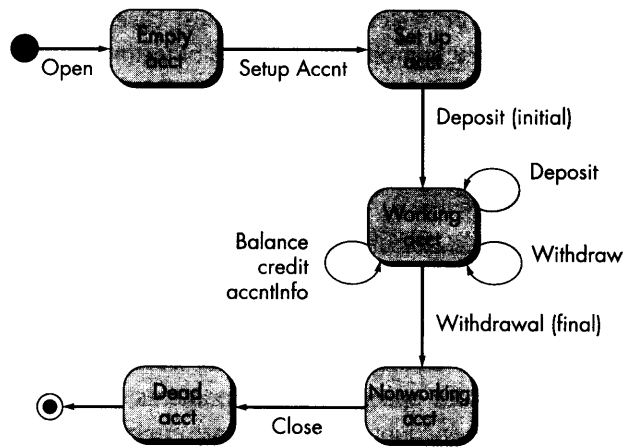


FIGURE 14.12 State diagram for the Account class (adapted from [KIR94])

The tests to be designed should achieve all state coverage [KIR94]. That is, the operation sequences should cause the **Account** class to make a transition through all allowable states:

Test case s_1: **open•setupAccnt•deposit (initial)•withdraw (final)•close**

It should be noted that this sequence is identical to the minimum test sequence discussed in Section 14.9.1. Adding additional test sequences to the minimum sequence,

Test case s_2: **open•setupAccnt•deposit(initial)•deposit•balance•credit•withdraw (final)•close**

Test case s_3: **open•setupAccnt•deposit(initial)•deposit•withdraw•accntInfo•withdraw (final)•close**

Still more test cases could be derived to ensure that all behaviors for the class have been adequately exercised. In situations in which the class behavior results in a collaboration with one or more classes, multiple state diagrams are used to track the behavioral flow of the system.

The state model can be traversed in a "breadth-first" [MGR94] manner. In this context, breadth first implies that a test case exercises a single transition. When a new transition is to be tested only previously tested transitions are used.

Consider a **CreditCard** object that is part of the banking system. The initial state of **CreditCard** is *undefined* (i.e., no credit card number has been provided). Upon reading the credit card during a sale, the object takes on a *defined* state; that is, the attributes **card number** and **expiration date**, along with bank-specific identifiers are defined. The credit card is *submitted* when it is sent for authorization, and it is *approved*

when authorization is received. The transition of **CreditCard** from one state to another can be tested by deriving test cases that cause the transition to occur. A breadth-first approach to this type of testing would not exercise *submitted* before it exercised *undefined* and *defined*. If it did, it would make use of transitions that had not been previously tested and would therefore violate the breadth-first criterion.

14.10 Testing for Specialized Environments, Architectures, and Applications

The testing methods discussed in preceding sections are generally applicable across all environments, architectures, and applications, but unique guidelines and approaches to testing are sometimes warranted. In this section we consider testing guidelines for specialized environments, architectures, and applications that are commonly encountered by software engineers.

14.10.1 Testing GUIs

A testing strategy similar to random or partition testing (Section 14.8) can be used to design UI tests.

Graphical user interfaces (GUIs) present interesting challenges for software engineers. Because of reusable components provided as part of GUI development environments, the creation of the user interface has become less time consuming and more precise (Chapter 12). But, at the same time, the complexity of GUIs has grown, leading to more difficulty in the design and execution of test cases.

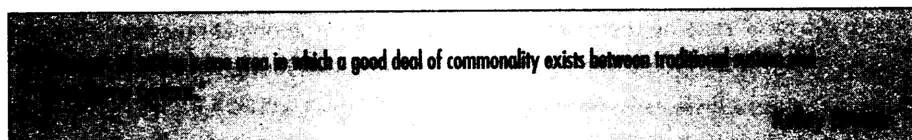
Because many modern GUIs have the same look and feel, a series of standard tests can be derived. Finite state modeling graphs may be used to derive a series of tests that address specific data and program objects relevant to the GUI.

Due to the large number of permutations associated with GUI operations, testing should be approached using automated tools. A wide array of GUI testing tools has appeared on the market over the past few years. For further discussion, see Chapter 12.

14.10.2 Testing of Client/Server Architectures

WebRef

Client/server architectures represent a significant challenge for software testers. The distributed nature of client/server environments, the performance issues associated with transaction processing, the potential presence of a number of different hardware platforms, the complexities of network communication, the need to service multiple clients from a centralized (or in some cases, distributed) database, and the coordination requirements imposed on the server all combine to make testing of client/server software architectures considerably more difficult than standalone applications. In fact, recent industry studies indicate a significant increase in testing time and cost when client/server environments are developed.



In general, the testing of client/server software occurs at three different levels: (1) individual client applications are tested in a "disconnected" mode; the operation of the server and the underlying network are not considered; (2) the client software and associated server applications are tested in concert, but network operations are not explicitly exercised; (3) the complete client/server architecture, including network operation and performance, is tested.

Although many different types of tests are conducted at each of these levels of detail, the following testing approaches are commonly encountered for client/server applications:

What types of tests are conducted for client/server systems?

- **Application function tests.** The functionality of client applications is tested using the methods discussed earlier in this chapter. In essence, the application is tested in standalone fashion.
- **Server tests.** The coordination and data management functions of the server are tested. Server performance (overall response time and data throughput) is also considered.
- **Database tests.** The accuracy and integrity of data stored by the server is tested. Transactions posted by client applications are examined to ensure that data are properly stored, updated, and retrieved. Archiving is also tested.
- **Transaction tests.** A series of tests are created to ensure that each class of transactions is processed according to requirements. Tests focus on the correctness of processing and also on performance issues (e.g., transaction processing times and transaction volume).
- **Network communication tests.** These tests verify that communication among the nodes of the network occurs correctly and that message passing, transactions, and related network traffic occur without error. Network security tests may also be conducted as part of these tests.

To accomplish these testing approaches, Musa [MUS93] recommends the development of *operational profiles* derived from client/server usage scenarios.[9] An operational profile indicates how different types of users interoperate with the client/server system. That is, the profiles provide a "pattern of usage" that can be applied when tests are designed and executed.

14.10.3 Testing Documentation and Help Facilities

The term *software testing* conjures images of large numbers of test cases prepared to exercise computer programs and the data that they manipulate. Recalling the definition of software presented in the first chapter of this book, it is important to note

9 It should be noted that operational profiles can be used in testing for all types of system architectures, not just client/server.

that testing must also extend to the third element of the software configuration—documentation.

Errors in documentation can be as devastating to the acceptance of the program as errors in data or source code. Nothing is more frustrating than following a user guide or an on-line help facility exactly and getting results or behaviors that do not coincide with those predicted by the documentation. It is for this reason that documentation testing should be a meaningful part of every software test plan.

Documentation testing can be approached in two phases. The first phase, review and inspection (Chapter 26), examines the document for editorial clarity. The second phase, live test, uses the documentation in conjunction with the use of the actual program.

INFO

Documentation Testing

The following questions should be answered during documentation and/or help facility testing:

- Does the documentation accurately describe how to accomplish each mode of use?
- Is the description of each interaction sequence accurate?
- Are examples accurate?
- Are terminology, menu descriptions, and system responses consistent with the actual program?
- Is it relatively easy to locate guidance within the documentation?
- Can troubleshooting be accomplished easily with the documentation?
- Are the document table of contents and index accurate and complete?
- Is the design of the document (layout, typefaces, indentation, graphics) conducive to understanding and quick assimilation of information?
- Are all software error messages displayed for the user described in more detail in the document? Are actions to be taken as a consequence of an error message clearly delineated?
- If hypertext links are used, are they accurate and complete?
- If hypertext is used, is the navigation design appropriate for the information required?

The only viable way to answer these questions is to have an independent third party (e.g., selected users) test the documentation in the context of program usage. All discrepancies are noted and areas of document ambiguity or weakness are defined for potential rewrite.

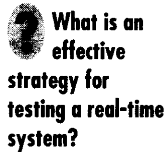
14.10.4 Testing for Real-Time Systems

The time-dependent, asynchronous nature of many real-time applications adds a new and potentially difficult element to the testing mix—time. Not only does the test case designer have to consider conventional test cases but also event handling (i.e., interrupt processing), the timing of the data, and the parallelism of the tasks (processes) that handle the data. In many situations, test data provided when a real-time system is in one state will result in proper processing, while the same data provided when the system is in a different state may lead to error.

For example, the real-time software that controls a new photocopier accepts operator interrupts (i.e., the machine operator hits control keys such as RESET or DARKEN)

with no error when the machine is making copies (in the *copying* state). If these same operator interrupts are input when the machine is in the *jammed* state, a display of the diagnostic code (indicating the location of the jam) will be lost (an error).

In addition, the intimate relationship that exists between real-time software and its hardware environment can also cause testing problems. Software tests must consider the impact of hardware faults on software processing. Such faults can be extremely difficult to simulate realistically.



Comprehensive test case design methods for real-time systems continue to evolve. However, a four-step strategy can be proposed:

- **Task testing.** The first step in the testing of real-time software is to test each task independently. That is, conventional tests are designed and executed for each task. Each task is executed independently during these tests. Task testing uncovers errors in logic and function, but not timing or behavior.
- **Behavioral testing.** Using system models created with automated tools, it is possible to simulate the behavior of a real-time system and examine its behavior as a consequence of external events. These analysis activities can serve as the basis for the design of test cases that are conducted when the real-time software has been built.
- **Intertask testing.** Once errors in individual tasks and in system behavior have been isolated, testing shifts to time-related errors. Asynchronous tasks that are known to communicate with one another are tested with different data rates and processing load to determine if intertask synchronization errors will occur. In addition, tasks that communicate via a message queue or data store are tested to uncover errors in the sizing of these data storage areas.
- **System testing.** Software and hardware are integrated and a full range of system tests (Chapter 13) are conducted in an attempt to uncover errors at the software/hardware interface. Most real-time systems process interrupts. Therefore, testing the handling of these Boolean events is essential. Using the state diagram and the control specification (Chapter 8), the tester develops a list of all possible interrupts and the processing that occurs as a consequence of the interrupts. Tests are then designed to assess the following system characteristics:

  —Are interrupt priorities properly assigned and properly handled?

  —Is processing for each interrupt handled correctly?

  —Does the performance (e.g., processing time) of each interrupt-handling procedure conform to requirements?

  —Does a high volume of interrupts arriving at critical times create problems in function or performance?

In addition, global data areas that are used to transfer information as part of interrupt processing should be tested to assess the potential for the generation of side effects.

14.11 Testing Patterns

WebRef
Pointers to over 70 testing patterns can be found at www.rbsc.com.

In earlier chapters, we have discussed the use of patterns as a mechanism for describing software building blocks or software engineering situations. These building blocks or situations are encountered repeatedly as different applications are built or different projects are conducted. Like their counterparts in analysis and design, *testing patterns* describe often-encountered building blocks or situations that software testers may be able to reuse as they approach the testing of some new or revised system.

Not only do testing patterns provide software engineers with useful guidance as testing activities commence, they also provide three additional benefits described by Marick [MAR02]:

KEY POINT
Testing patterns can help a software team communicate more effectively about testing and better understand the forces that lead to a specific testing approach.

1. They provide a vocabulary for problem-solvers. "Hey, you know, we should use a Null Object."
2. They focus attention on the forces behind a problem. That allows [test case] designers to better understand when and why a solution applies.
3. They encourage iterative thinking. Each solution creates a new context in which new problems can be solved.

Although these benefits are "soft," they should not be overlooked. Much of software testing, even during the past decade, has been an ad hoc activity. If testing patterns can help a software team communicate about testing more effectively, understand the motivating forces that lead to a specific approach to testing, and approach the design of test cases as an evolutionary activity, they have accomplished much.

Testing patterns are described in much the same way as analysis and design patterns (Chapters 7 and 9). Dozens of testing patterns have been proposed in the literature (e.g., [BIN99], [MAR02]). The following three testing patterns (presented in abstract form only) provide representative examples:

WebRef
Patterns that describe testing organization, efficiency, strategy, and problem resolution can be found at [illegible]/[illegible]/techpapers/[illegible]/papers/[illegible].

Pattern name: **pair testing**

Abstract: A process-oriented pattern, **pair testing** describes a technique that is analogous to pair programming (Chapter 4) in which two testers work together to design and execute a series of tests that can be applied to unit, integration, or validation testing activities.

Pattern name: **separate test interface**

Abstract: There is a need to test every class in an object-oriented system, including "internal classes" (i.e., classes that do not expose any interface outside of the component that used them). The **separate test interface** pattern describes how to create "a test in-

terface that can be used to describe specific tests on classes that are visible only internally to a component" [LAN01].

Pattern name: **scenario testing**

Abstract: Once unit and integration tests have been conducted, there is a need to determine whether the software will perform in a manner that satisfies users. The **scenario testing** pattern describes a technique for exercising the software from the user's point of view. A failure at this level indicates that the software has failed to meet a user visible requirement [KAN01].

A comprehensive discussion of testing patterns is beyond the scope of this book. The interested reader should see [BIN99] and [MAR02] for further information on this important topic.

14.12 SUMMARY

The primary objective for test case design is to derive a set of tests that have the highest likelihood of uncovering errors in software. To accomplish this objective, two different categories of test case design techniques—applicable to conventional and object-oriented systems—are used: white-box testing and black-box testing.

White-box tests focus on the program control structure. Test cases are derived to ensure that all statements in the program have been executed at least once during testing and that all logical conditions have been exercised. Basis path testing, a white-box technique, makes use of program graphs (or graph matrices) to derive a set of linearly independent tests that will ensure coverage. Condition and data flow testing further exercise program logic, and loop testing complements other white-box techniques by providing a procedure for exercising loops of varying degrees of complexity.

Black-box tests are designed to validate functional requirements without regard to the internal workings of a program. Black-box testing techniques focus on the information domain of the software, deriving test cases by partitioning the input and output domain of a program in a manner that provides thorough test coverage. Equivalence partitioning divides the input domain into classes of data that are likely to exercise specific software function. Boundary value analysis probes the program's ability to handle data at the limits of acceptability. Orthogonal array testing provides an efficient, systematic method for testing systems with small numbers of input parameters.

Although the overall objective of object-oriented testing—to find the maximum number of errors with a minimum amount of effort—is identical to the objective of conventional software testing, the strategy and tactics for OO testing differ somewhat. The view of testing broadens to include the review of both the analysis and design model. In addition, the focus of testing moves away from the procedural component (the module) and toward the class. The design of tests for a class uses a variety of methods: fault-based testing, random testing, and partition testing. Each of these methods exercises the operations encapsulated by the class. Test sequences

are designed to ensure that relevant operations are exercised. The state of the class, represented by the values of its attributes, is examined to determine if errors exist.

Integration testing can be accomplished using a use-based strategy. Use-based testing constructs the system in layers, beginning with those classes that do not use server classes. Integration test case design methods can also use random and partition tests. In addition, scenario-based testing and tests derived from behavioral models can be used to test a class and its collaborators. A test sequence tracks the flow of operations across class collaborations.

Specialized testing methods encompass a broad array of software capabilities and application areas. Testing for graphical user interfaces, client/server architectures, documentation and help facilities, and real-time systems each require specialized guidelines and techniques.

Experienced software developers often say, "Testing never ends, it just gets transferred from you [the software engineer] to your customer. Every time your customer uses the program, a test is being conducted." By applying test case design, the software engineer can achieve more complete testing and thereby uncover and correct the highest number of errors before the "customer's tests" begin.

REFERENCES

[AMB95] Ambler, S., "Using Use Cases," *Software Development*, July 1995, pp. 53–61.

[BEI90] Beizer, B., *Software Testing Techniques*, 2nd ed., Van Nostrand-Reinhold, 1990.

[BEI95] Beizer, B., *Black-Box Testing*, Wiley, 1995.

[BIN94] Binder, R. V., "Testing Object-Oriented Systems: A Status Report," *American Programmer*, vol. 7, no. 4, April 1994, pp. 23–28.

[BIN99] Binder, R., *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley, 1999.

[DEU79] Deutsch, M., "Verification and Validation," in *Software Engineering* (R. Jensen and C. Tonies, eds.), Prentice-Hall, 1979, pp. 329–408.

[FRA88] Frankl, P. G., and E. J. Weyuker, "An Applicable Family of Data Flow Testing Criteria," *IEEE Trans. Software Engineering*, vol. SE-14, no. 10, October 1988, pp. 1483–1498.

[FRA93] Frankl, P. G., and S. Weiss, "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow," *IEEE Trans. Software Engineering*, vol. SE-19, no. 8, August 1993, pp. 770–787.

[KAN93] Kaner, C., J. Falk, and H. Q. Nguyen, *Testing Computer Software*, 2nd ed., Van Nostrand-Reinhold, 1993.

[KAN01] Kaner, C., "Pattern: Scenario Testing," (draft), 2001, available at http://www.testing.com/test-patterns/patterns/pattern-scenario-testing-kaner.html.

[KIR94] Kirani, S., and W. T. Tsai, "Specification and Verification of Object-Oriented Programs," Technical Report TR 94-64, Computer Science Department, University of Minnesota, December 1994.

[LAN01] Lange, M., "It's Testing Time! Patterns for Testing Software, June, 2001, downloadable from http://www.testing.com/test-patterns/patterns/index.html.

[LIN94] Lindland, O. I., et al., "Understanding Quality in Conceptual Modeling," *IEEE Software*, vol. 11, no 4, July 1994, pp. 42–49.

[MAR94] Marick, B., *The Craft of Software Testing*, Prentice-Hall, 1994.

[MAR02] Marick, B., "Software Testing Patterns," 2002, http://www.testing.com/test-patterns/index.html.

[MCC76] McCabe, T., "A Software Complexity Measure," *IEEE Trans. Software Engineering*, vol. SE-2, December 1976, pp. 308–320.

[MGR94] McGregor, J. D., and T. D. Korson, "Integrated Object-Oriented Testing and Development Processes," *CACM,* vol. 37, no. 9, September 1994, pp. 59–77.

[MUS93] Musa, J., "Operational Profiles in Software Reliability Engineering," *IEEE Software,* March 1993, pp. 14–32.

[MYE79] Myers, G., *The Art of Software Testing,* Wiley, 1979.

[NTA88] Ntafos, S. C., "A Comparison of Some Structural Testing Strategies," *IEEE Trans. Software Engineering,* vol. SE-14, no. 6, June 1988, pp. 868–874.

[PHA89] Phadke, M. S., *Quality Engineering Using Robust Design,* Prentice-Hall, 1989.

[PHA97] Phadke, M. S., "Planning Efficient Software Tests," *Crosstalk,* vol. 10, no. 10, October 1997, pp. 11–15.

[TAI89] Tai, K. C., "What to Do Beyond Branch Testing," *ACM Software Engineering Notes,* vol. 14, no. 2, April 1989, pp. 58–61.

PROBLEMS AND POINTS TO PONDER

14.1. Specify, design, and implement a software tool that will compute the cyclomatic complexity for the programming language of your choice. Use the graph matrix as the operative data structure in your design.

14.2. Give at least three examples in which black-box testing might give the impression that "everything's OK," while white-box tests might uncover an error. Give at least three examples in which white-box testing might give the impression that "everything's OK," while black-box tests might uncover an error.

14.3. Read Beizer [BEI95] and determine how the program you have developed in Problem 14.1 can be extended to accommodate various link weights. Extend your tool to process execution probabilities or link processing times.

14.4. Select a software component that you have designed and implemented recently. Design a set of test cases that will ensure that all statements have been executed using basis path testing.

14.5. Why do we have to retest subclasses that are instantiated from an existing class, if the existing class has already been thoroughly tested? Can we use the test cases designed for the existing class?

14.6. Can you think of any additional testing characteristics that are not discussed in Section 14.1?

14.7. Design an automated tool that will recognize loops and categorize them as indicated in Section 14.5.3.

14.8. Myers [MYE79] uses the following program as a self-assessment of one's ability to specify adequate testing: A program reads three integer values. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral. Develop a set of test cases that you feel will adequately test this program.

14.9. Design and implement the program (with error handling where appropriate) specified in Problem 14.8. Derive a flow graph for the program and apply basis path testing to develop test cases that will guarantee that all statements in the program have been tested. Execute the cases and show your results.

14.10. Will exhaustive testing (even if it is possible for very small programs) guarantee that the program is 100 percent correct?

14.11. In your own words, describe why the class is the smallest reasonable unit for testing within an OO system.

14.12. Extend the tool described in Problem 14.7 to generate test cases for each loop category, once encountered. It will be necessary to perform this function interactively with the tester.

14.13. Apply random testing and partitioning to three classes defined in the design for the *SafeHome* system. Produce test cases that indicate the operation sequences that will be invoked.

14.14. Apply multiple class testing and tests derived from the behavioral model to the *SafeHome* design.

14.15. Test a user manual (or help facility) for an application that you use frequently. Find at least one error in the documentation.

Further Readings and Information Sources

Among dozens of books that present test case design methods are Craig and Kaskiel (*Systematic Software Testing,* Artech House, 2002), Tamres (*Introducing Software Testing,* Addison-Wesley, 2002), Whittaker (*How to Break Software,* Addison-Wesley, 2002), Jorgensen (*Software Testing: A Craftman's Approach,* CRC Press, 2002), Splaine and his colleagues (*The Web Testing Handbook,* Software Quality Engineering Publishing, 2001), Patton (*Software Testing,* Sams Publishing, 2000), Kaner and his colleagues (*Testing Computer Software,* second edition, Wiley, 1999).In addition, Hutcheson (*Software Testing Methods and Metrics: The Most Important Tests,* McGraw-Hill, 1997) and Marick (*The Craft of Software Testing: Subsystem Testing Including Object-Based and Object-Oriented Testing,* Prentice-Hall, 1995) present treatments of testing methods and strategies.

Myers [MYE79] remains a classic text, covering black-box techniques in considerable detail. Beizer [BEI90] provides comprehensive coverage of white-box techniques, introducing a level of mathematical rigor that has often been missing in other treatments of testing. His later book [BEI95] presents a concise treatment of important methods. Perry (*Effective Methods for Software Testing,* Wiley-QED, 1995) and Friedman and Voas (*Software Assessment: Reliability, Safety, Testability,* Wiley, 1995) present good introductions to testing strategies and tactics. Mosley (*The Handbook of MIS Application Software Testing,* Prentice-Hall, 1993) discusses testing issues for large information systems, and Marks (*Testing Very Big Systems,* McGraw-Hill, 1992) discusses the special issues that must be considered when testing major programming systems.

Sykes and McGregor (*Practical Guide for Testing Object-Oriented Software,* Addison-Wesley, 2001), Bashir and Goel (*Testing Object-Oriented Software,* Springer-Verlag, 2000), Binder (*Testing Object-Oriented Systems,* Addison-Wesley, 1999), Kung and his colleagues (*Testing Object-Oriented Software,* IEEE Computer Society Press, 1998), Marick (*The Craft of Software Testing,* Prentice-Hall, 1997) and Siegel and Muller (*Object-Oriented Software Testing: A Hierarchical Approach,* Wiley, 1996) present strategies and methods for testing OO systems.

Software testing is a resource-intensive activity. It is for this reason that many organizations automate parts of the testing process. Books by Dustin, Rashka, and Poston (*Automated Software Testing: Introduction, Management, and Performance,* Addison-Wesley, 1999), Graham and her colleagues (*Software Test Automation*, Addison-Wesley, 1999), and Poston (*Automating Specification-Based Software Testing,* IEEE Computer Society, 1996) discuss tools, strategies, and methods for automated testing.

A number of books consider testing methods and strategies in specialized application areas. Gardiner (*Testing Safety-Related Software: A Practical Handbook,* Springer-Verlag, 1999) has edited a book that addresses testing of safety-critical systems. Mosley (*Client/Server Software Testing on the Desk Top and the Web,* Prentice-Hall, 1999) discusses the test process for clients, servers, and network components. Rubin (*Handbook of Usability Testing,* Wiley, 1994) has written a useful guide for those who must exercise human interfaces.

Binder [BIN99] describes almost 70 testing patterns that cover testing of methods, classes/clusters, subsystems, reusable components, frameworks, and systems as well as test automation and specialized database testing. A list of these patterns can be found at www.rbsc.com/pages/TestPatternList.htm.

A wide variety of information sources on test case design methods are available on the Internet. An up-to-date list of World Wide Web references that are relevant to testing techniques can be found at the SEPA Web site:
http://www.mhhe.com/pressman.

CHAPTER 15

PRODUCT METRICS

KEY CONCEPTS

- **function points**
- **GQM paradigm**
- **indicators**
- **McCall's factors**
- **measurement**
  - **attributes**
  - **principles**
- **measures**
- **metrics**
  - **analysis model**
  - **code**
  - **design model**
  - **maintenance**
  - **object-oriented**
  - **testing**
- **quality**

A key element of any engineering process is measurement. We use measures to better understand the attributes of the models that we create and to assess the quality of the engineered products or systems that we build. But unlike other engineering disciplines, software engineering is not grounded in the basic quantitative laws of physics. Direct measures, such as voltage, mass, velocity, or temperature, are uncommon in the software world. Because software measures and metrics are often indirect, they are open to debate. Fenton [FEN91] addresses this issue when he states:

> Measurement is the process by which numbers or symbols are assigned to the attributes of entities in the real world in such a way as to define them according to clearly defined rules . . . In the physical sciences, medicine, economics, and more recently the social sciences, we are now able to measure attributes that we previously thought to be unmeasurable . . . Of course, such measurements are not as refined as many measurements in the physical sciences . . ., but they exist [and important decisions are made based on them]. We feel that the obligation to attempt to "measure the unmeasurable" in order to improve our understanding of particular entities is as powerful in software engineering as in any discipline.

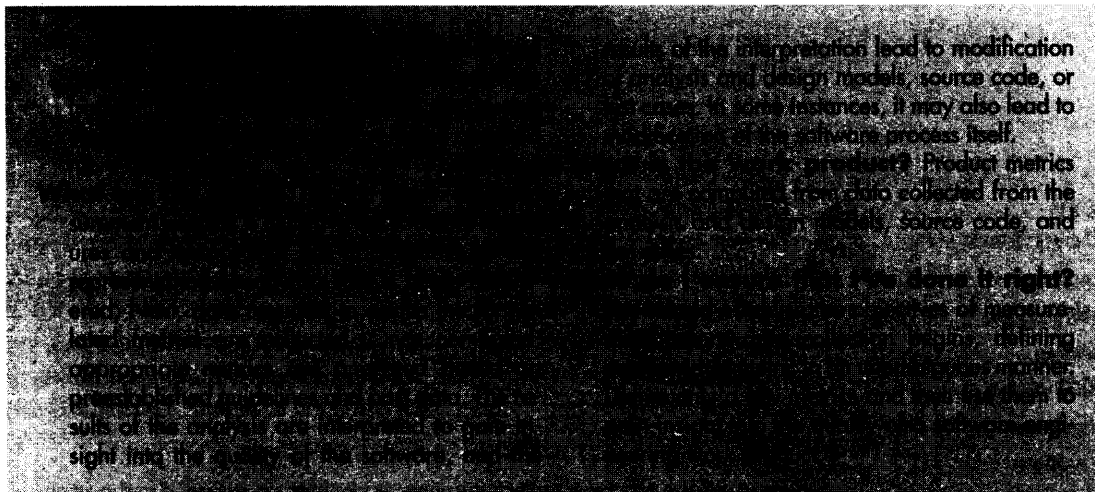
But some members of the software community continue to argue that software is "unmeasurable" or that attempts at measurement should be postponed until we better understand software and the attributes that should be used to describe it. That is a mistake.

QUICK LOOK

What is it? Software engineering is a quantitative discipline. Engineers use numbers to help them design and assess the product to be built. Until recently, software engineers had little quantitative guidance in their work—but that's changing. Product metrics help software engineers gain insight into the design and construction of the software they build. Unlike process and project metrics that apply to the project (or process) as a whole, product metrics focus on specific attributes of software engineering work products and are collected as technical tasks (analysis, design, coding, and testing) are being conducted.

Who does it? Software engineers use product metrics to help them build higher-quality software.

Why is it important? There will always be a qualitative element to the creation of computer software. The problem is that qualitative assessment may not be enough. A software engineer needs objective criteria to help guide the design of data, architecture, interfaces, and components. The tester needs quantitative guidance



Although product metrics for computer software are often not absolute, they provide us with a systematic way to assess quality based on a set of clearly defined rules. They also provide the software engineer with on-the-spot, rather than after-the-fact insight. This enables the engineer to discover and correct potential problems before they become catastrophic defects.

In this chapter, we consider measures that can be used to assess the quality of the product as it is being engineered. These measures of internal product attributes provide the software engineer with a real-time indication of the efficacy of the analysis, design, and code models; the effectiveness of test cases; and the overall quality of the software to be built.

Even the most jaded software developers will agree that high-quality software is an important goal. But how do we define quality? In the most general sense, software quality is *conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.*

There is little question that the preceding definition could be modified or extended and debated endlessly. For the purposes of this book, the definition serves to emphasize three important points:

1. Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality.[1]

1 It is important to note that quality extends to the technical characteristics of analysis and design models and the source code realization of those models. Models that exhibit high quality (in the technical sense) will lead to software that exhibits high quality from the customer's point of view.

2. Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will almost surely result.
3. There is a set of implicit requirements that often goes unmentioned (e.g., the desire for ease of use). If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect.

Software quality is a complex mix of factors that will vary across different applications and the customers who request them. In the sections that follow, software quality factors are identified and the human activities required to achieve them are described.

15.1.1 McCall's Quality Factors

The factors that affect software quality can be categorized in two broad groups: (1) factors that can be directly measured (e.g., defects uncovered during testing) and (2) factors that can be measured only indirectly (e.g., usability or maintainability). In each case measurement should occur. We must compare the software (programs, data, documents) to some datum and arrive at an indication of quality.

KEY POINT
It's interesting to note that McCall's quality factors are as valid today as they were in the 1970s. Therefore, it's reasonable to assert that the factors that affect software quality do not change with time.

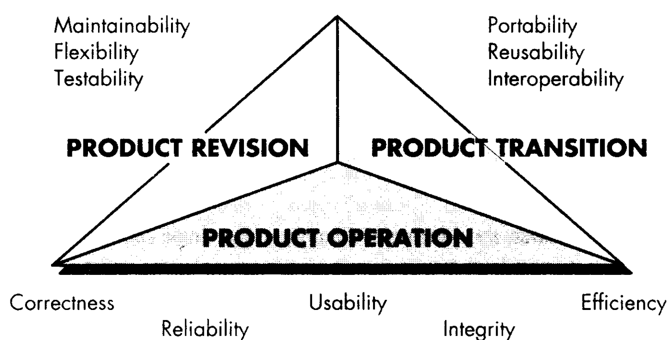
McCall, Richards, and Walters [MCC77] propose a useful categorization of factors that affect software quality. These software quality factors, shown in Figure 15.1, focus on three important aspects of a software product: its operational characteristics, its ability to undergo change, and its adaptability to new environments.

Referring to the factors noted in Figure 15.1, McCall and his colleagues provide the following descriptions:

Correctness. The extent to which a program satisfies its specification and fulfills the customer's mission objectives.

Reliability. The extent to which a program can be expected to perform its intended function with required precision. [It should be noted that other, more complete definitions of reliability have been proposed (see Chapter 26).]

FIGURE 15.1
McCall's software quality factors



Efficiency. The amount of computing resources and code required by a program to perform its function.

Integrity. The extent to which access to software or data by unauthorized persons can be controlled.

Usability. The effort required to learn, operate, prepare input for, and interpret output of a program.

Maintainability. The effort required to locate and fix an error in a program. [This is a very limited definition.]

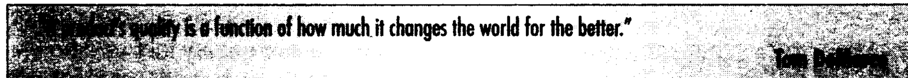
Flexibility. The effort required to modify an operational program.

Testability. The effort required to test a program to ensure that it performs its intended function.

Portability. The effort required to transfer the program from one hardware and/or software system environment to another.

Reusability. The extent to which a program [or parts of a program] can be reused in other applications—related to the packaging and scope of the functions that the program performs.

Interoperability. The effort required to couple one system to another.



Build your own checklist using these factors. First assign each a relative importance for your project. Then, grade your work products to assess the quality of the software you're building.

It is difficult, and in some cases impossible, to develop direct measures[2] of these quality factors. In fact, many of the metrics defined by McCall et al. can be measured only subjectively. The metrics may be in the form of a checklist that is used to "grade" specific attributes of the software [CAV78].

15.1.2 ISO 9126 Quality Factors

The ISO 9126 standard was developed in an attempt to identify quality attributes for computer software. The standard identifies six key quality attributes:

Functionality. The degree to which the software satisfies stated needs as indicated by the following sub-attributes: *suitability, accuracy, interoperability, compliance,* and *security.*

Reliability. The amount of time that the software is available for use as indicated by the following sub-attributes: *maturity, fault tolerance, recoverability.*

Usability. The degree to which the software is easy to use as indicated by the following sub-attributes: *understandability, learnability, operability.*

---

2 A *direct measure* implies that there is a single countable value that provides a direct indication of the attribute being examined. For example, the "size" of a program can be measured directly by counting the number of lines of code.

Efficiency. The degree to which the software makes optimal use of system resources as indicated by the following sub-attributes: *time behavior, resource behavior.*

Maintainability. The ease with which repair may be made to the software as indicated by the following sub-attributes: *analyzability, changeability, stability, testability.*

Portability. The ease with which the software can be transposed from one environment to another as indicated by the following sub-attributes: *adaptability, installability, conformance, replaceability.*

Like other software quality factors discussed in Chapter 9 and Section 15.1.1, the ISO 9126 factors do not necessarily lend themselves to direct measurement. However, they do provide a worthwhile basis for indirect measures and an excellent checklist for assessing the quality of a system.

> **"Any activity becomes creative when the doer cares about doing it right, or better."**
>
> **John Updike**

15.1.3 The Transition to a Quantitative View

In the preceding sections, a set of qualitative factors for the "measurement" of software quality was discussed. We strive to develop precise measures for software quality and are sometimes frustrated by the subjective nature of the activity. Cavano and McCall [CAV78] discuss this situation:

> The determination of quality is a key factor in every day events—wine tasting contests, sporting events [e.g., gymnastics], talent contests, etc. In these situations, quality is judged in the most fundamental and direct manner: side by side comparison of objects under identical conditions and with predetermined concepts. The wine may be judged according to clarity, color, bouquet, taste, etc. However, this type of judgment is very subjective; to have any value at all, it must be made by an expert.
>
> Subjectivity and specialization also apply to determining software quality. To help solve this problem, a more precise definition of software quality is needed as well as a way to derive quantitative measurements of software quality for objective analysis . . .

In the sections that follow, we examine a set of software metrics that can be applied to the quantitative assessment of software quality. In all cases, the metrics represent indirect measures; that is, we never really measure quality but rather some manifestation of quality. The complicating factor is the precise relationship between the variable that is measured and the quality of software.

> **"Just as temperature measurement began with an index finger . . . and grew to sophisticated scales, tools and techniques, so too is software measurement maturing."**
>
> **Shari Pfleeger**

15.2 A Framework for Product Metrics

As we noted in the introduction to this chapter, measurement assigns numbers or symbols to attributes of entities in the real word. To accomplish this, a measurement model encompassing a consistent set of rules is required. Although the theory of measurement (e.g., [KYB84]) and its application to computer software (e.g., [DEM81], [BRI96], [ZUS97]) are topics that are beyond the scope of this book, it is worthwhile to establish a fundamental framework and a set of basic principles for the measurement of product metrics for software.

15.2.1 Measures, Metrics, and Indicators

Although the terms *measure, measurement,* and *metrics* are often used interchangeably, it is important to note the subtle differences between them. Because *measure* can be used either as a noun or a verb, definitions of the term can become confusing. Within the software engineering context, a *measure* provides a quantitative indication of the e :tent, amount, dimension, capacity, or size of some attribute of a product or process. *Measurement* is the act of determining a measure. The *IEEE Standard Glossary* [IEE93] defines *metric* as "a quantitative measure of the degree to which a system, component, or process possesses a given attribute."

When a single data point has been collected (e.g., the number of errors uncovered within a single software component), a measure has been established. Measurement occurs as the result of the collection of one or more data points (e.g., a number of component reviews and unit tests are investigated to collect measures of the number of errors for each). A software metric relates the individual measures in some way (e..g, the average number of errors found per review or the average number of errors found per unit test).

A software engineer collects measures and develops metrics so that indicators will be obtained. An *indicator* is a metric or combination of metrics that provides insight into the software process, a software project, or the product itself. An indicator provides insight that enables the project manager or software engineers to adjust the process, the project, or the product to make things better.

> **"A science is as mature as its measurement tools."**
>
> **Louis Pasteur**

15.2.2 The Challenge of Product Metrics

Over the past three decades, many researchers have attempted to develop a single metric that provides a comprehensive measure of software complexity. Fenton [FEN94] characterizes this research as a search for "the impossible holy grail." Although dozens of complexity measures have been proposed [ZUS90], each takes a somewhat different view of what complexity is and what attributes of a system lead to complexity. By analogy, consider a metric for evaluating an attractive car. Some

observers might emphasize body design, others might consider mechanical characteristics, still others might tout cost, or performance, or fuel economy, or the ability to recycle when the car is junked. Since any one of these characteristics may be at odds with others, it is difficult to derive a single value for "attractiveness." The same problem occurs with computer software.

WebRef
Voluminous information on product metrics has been compiled by Horst Zuse at **irb.cs.tuberlin.de/~zuse/.**

Yet there is a need to measure and control software complexity. And if a single value of this quality metric is difficult to derive, it should be possible to develop measures of different internal program attributes (e.g., effective modularity, functional independence, and other attributes discussed in Chapters 9 through 12). These measures and the metrics derived from them can be used as independent indicators of the quality of analysis and design models. But here again, problems arise. Fenton [FEN94] notes this when he states: "The danger of attempting to find measures which characterize so many different attributes is that inevitably the measures have to satisfy conflicting aims. This is counter to the representational theory of measurement." Although Fenton's statement is correct, many people argue that product measurement conducted during the early stages of the software process provides software engineers with a consistent and objective mechanism for assessing quality.

It is fair to ask, however, just how valid product metrics are. That is, how closely aligned are product metrics to the long-term reliability and quality of a computer-based system? Fenton [FEN91] addresses this question in the following way:

> In spite of the intuitive connections between the internal structure of software products [product metrics] and its external product and process attributes, there have actually been very few scientific attempts to establish specific relationships. There are a number of reasons why this is so; the most commonly cited is the impracticality of conducting relevant experiments.

Each of the "challenges" noted here is a cause for caution, but it is no reason to dismiss product metrics.[3] Measurement is essential if quality is to be achieved.

15.2.3 Measurement Principles

Before we introduce a series of product metrics that (1) assist in the evaluation of analysis and design models, (2) provide an indication of the complexity of procedural designs and source code, and (3) facilitate the design of more effective testing, it is important to understand basic measurement principles. Roche [ROC94] suggests a measurement process that can be characterized by five activities:

What are the steps of an effective measurement process?

- *Formulation.* The derivation of software measures and metrics appropriate for the representation of the software that is being considered.

3 Although criticism of specific metrics is common in the literature, many critiques focus on esoteric issues and miss the primary objective of metrics in the real world: to help the software engineer establish a systematic and objective way to gain insight into his or her work and to improve product quality as a result.

- *Collection.* The mechanism used to accumulate data required to derive the formulated metrics.
- *Analysis.* The computation of metrics and the application of mathematical tools.
- *Interpretation.* The evaluation of metrics in an effort to gain insight into the quality of the representation.
- *Feedback.* Recommendations derived from the interpretation of product metrics transmitted to the software team.

Software metrics will be useful only if they are characterized effectively and validated so that their worth is proven. The following principles [LET03] are representative of many that can be proposed for metrics characterization and validation:

In reality, many product metrics in use today do not conform to these principles as well as they should. But that doesn't mean that they have no value—just be careful when you use them, understanding that they are intended to provide insight, not hard scientific verification.

- *A metric should have desirable mathematical properties.* That is, the metric's value should be in a meaningful range (e.g., zero to one, where zero truly means absence, one indicates the maximum value, and 0.5 represents the "half-way point"). Also, a metric that purports to be on a rational scale should not be composed of components that are only measured on an ordinal scale.
- *When a metric represents a software characteristic that increases when positive traits occur or decreases when undesirable traits are encountered, the value of the metric should increase or decrease in the same manner.*
- *Each metric should be validated empirically in a wide variety of contexts before being published or used to make decisions.* A metric should measure the factor of interest, independently of other factors. It should "scale up" to large systems and work in a variety of programming languages and system domains.

Although formulation, characterization, and validation are critical, collection and analysis are the activities that drive the measurement process. Roche [ROC94] suggests the following guidelines for these activities: (1) whenever possible, data collection and analysis should be automated; (2) valid statistical techniques should be applied to establish relationships between internal product attributes and external quality characteristics (e.g., whether the level of architectural complexity is correlated with the number of defects reported in production use); and (3) interpretative guidelines and recommendations should be established for each metric.

15.2.4 Goal-Oriented Software Measurement

WebRef
A useful discussion of GQM can be found at **www.thedacs.com/GoldPractices/practices/gqma.html.**

The *Goal/Question/Metric* (GQM) paradigm was developed by Basili and Weiss [BAS84] as a technique for identifying meaningful metrics for any part of the software process. GQM emphasizes the need to (1) establish an explicit measurement *goal* that is specific to the process activity or product characteristic that is to be assessed; (2) define a set of *questions* that must be answered in order to achieve the goal, and (3) identify well-formulated *metrics* that help to answer these questions.

A *goal definition template* [BAS94] can be used to define each measurement goal. The template takes the form:

> **Analyze** {the name of activity or attribute to be measured} **for the purpose of** {the overall objective of the analysis[4]} **with respect to** {the aspect of the activity or attribute that is considered} **from the viewpoint of** {the people who have an interest in the measurement} **in the context of** {the environment in which the measurement takes place}.

As an example, consider a goal definition template for *SafeHome:*

> **Analyze** the *SafeHome* software architecture **for the purpose of** evaluating architectural components **with respect to** the ability to make *SafeHome* more extensible **from the viewpoint of** the software engineers performing the work **in the context of** product enhancement over the next three years.

With a measurement goal explicitly defined, a set of questions is developed. Answers to these questions help the software team (or other stakeholders) to determine whether the measurement goal has been achieved. Among the questions that might be asked are:

Q_1: Are architectural components characterized in a manner that compartmentalizes function and related data?

Q_2: Is the complexity of each component within bounds that will facilitate modification and extension?

Each of these questions should be answered quantitatively, using one or more measures and metrics. For example, a metric that provides an indication of the cohesion (Chapter 9) of an architectural component might be useful in answering Q_1. Cyclomatric complexity and metrics discussed in Section 15.4.1 or 15.4.2 might provide insight for Q_2.

In actuality, there may be a number of measurement goals with related questions and metrics. In every case, the metrics that are chosen (or derived) should conform to the measurement principles discussed in Section 15.2.3 and the measurement attributes discussed in Section 15.2.5. For further information of GQM, the interested reader should see [SHE98] or [SOL99].

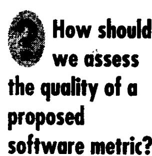
15.2.5 The Attributes of Effective Software Metrics

Hundreds of metrics have been proposed for computer software, but not all provide practical support to the software engineer. Some demand measurement that is too complex, others are so esoteric that few real world professionals have any hope of understanding them, and others violate the basic intuitive notions of what high-quality software really is.

---

4 van Solingen and Berghout [SOL99] suggest that the objective is almost always "understanding, controlling, or improving" the process activity or product attribute.

Ejiogu [EJI91] defines a set of attributes that should be encompassed by effective software metrics. The derived metric and the measures that lead to it should be:



Experience indicates that a product metric will be used only if it is intuitive and easy to compute. If dozens of "counts" have to be made, and complex computations are required, it is unlikely that the metric will be widely adopted.

- *Simple and computable.* It should be relatively easy to learn how to derive the metric, and its computation should not demand inordinate effort or time.
- *Empirically and intuitively persuasive.* The metric should satisfy the engineer's intuitive notions about the product attribute under consideration.
- *Consistent and objective.* The metric should always yield results that are unambiguous.
- *Consistent in the use of units and dimensions.* The mathematical computation of the metric should use measures that do not lead to bizarre combinations of units.
- *Programming language independent.* Metrics should be based on the analysis model, the design model, or the structure of the program itself.
- *An effective mechanism for high-quality feedback.* That is, the metric should lead to a higher-quality end product.

Although most software metrics satisfy these attributes, some commonly used metrics may fail to satisfy one or two of them. An example is the function point (discussed in Section 15.3.1)—a measure of the "functionality" delivery by the software. It can be argued[5] that the *consistent and objective* attribute fails because an independent third party may not be able to derive the same function point value as a colleague using the same information about the software. Should we therefore reject the FP measure? The answer is: Of course not! FP provides useful insight and therefore provides distinct value, even if it fails to satisfy one attribute perfectly.

15.2.6 The Product Metrics Landscape

Although a wide variety of metrics taxonomies have been proposed, the following outline addresses the most important metrics areas:

Metrics for the analysis model. These metrics address various aspects of the analysis model and include:

Functionality delivered—provides an indirect measure of the functionality that is packaged within the software.

System size—measures of the overall size of the system defined in terms of information available as part of the analysis model.

Specification quality—provides an indication of the specificity and completeness of a requirements specification.

5 An equally vigorous counter-argument can be made. Such is the nature of software metrics.

Metrics for the design model. These metrics quantify design attributes in a manner that allows a software engineer to assess design quality. Metrics include:

Architectural metrics—provide an indication of the quality of the architectural design.

Component-level metrics—measure the complexity of software components and other characteristics that have a bearing on quality.

Interface design metrics—focus primarily on usability.

Specialized OO design metrics—measure characteristics of classes and their communication and collaboration characteristics.

Metrics for source code. These metrics measure the source code and can be used to assess its complexity, maintainability, and testability, among other characteristics:

Halstead metrics—controversial but nonetheless fascinating, these metrics provide unique measures of a computer program.

Complexity metrics—measure the logical complexity of source code (can also be considered to be component-level design metrics).

Length metrics—provide an indication of the size of the software.

Metrics for testing. These metrics assist in the design of effective test cases and evaluate the efficacy of testing:

Statement and branch coverage metrics—lead to the design of test cases that provide program coverage.

Defect-related metrics—focus on bugs found, rather than on the tests themselves.

Testing effectiveness—provide a real-time indication of the effectiveness of tests that have been conducted.

In-process metrics—process related metrics that can be determined as testing is conducted.

In many cases, metrics for one model may be used in later software engineering activities. For example, design metrics may be used to estimate the effort required to generate source code. In addition, design metrics may be used in test planning and test case design.

SAFEHOME

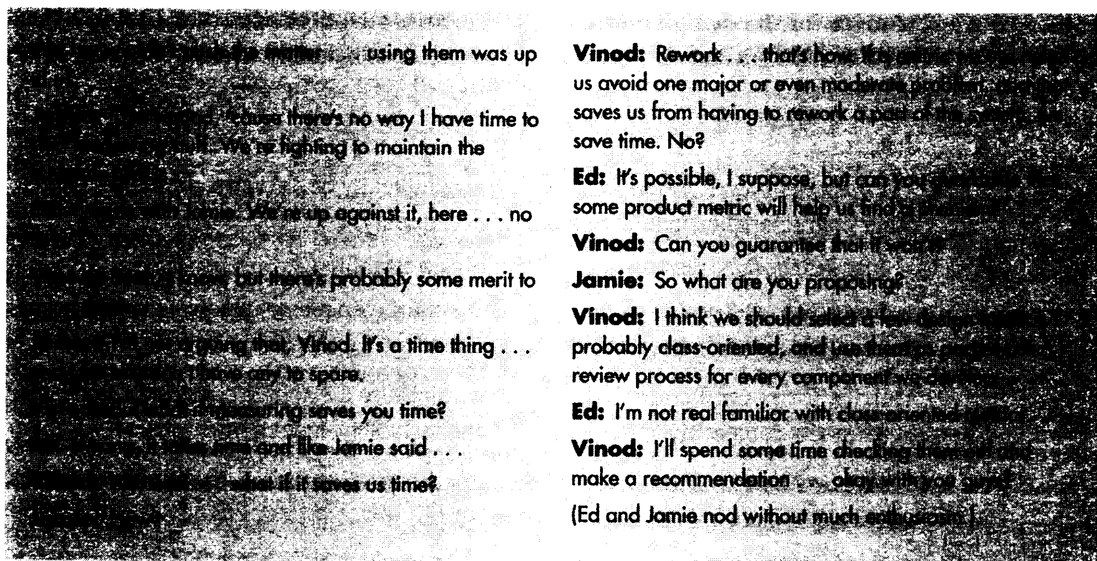
Debating Product Metrics

The scene: Vinod's cubicle.

The players: Vinod, Jamie and Ed—members of the *SafeHome* software engineering team, who are continuing work on component-level design and test case design.

The conversation:

Vinod: Doug [Doug Miller, software engineering manager] told me that we should all use product metrics, but he was kind of vague. He also said


using them was up
no way I have time to
to maintain the
against it, here . . . no
probably some merit to
It's a time thing . . .
you time?
Jamie said . . .
us time?
Vinod: Rework . . .
us avoid one major or
saves us from having
save time. No?
Ed: It's possible, I
some product metric will
Vinod: Can you
Jamie: So what are you
Vinod: I think we
probably class-oriented,
review process for every
Ed: I'm not real familiar
Vinod: I'll spend some time
make a recommendation
(Ed and Jamie nod

15.3 Metrics for the Analysis Model

Although relatively few analysis and specification metrics have appeared in the literature, it is possible to adapt metrics that are often used for project estimation and apply them in this context. These metrics examine the analysis model with the intent of predicting the "size" of the resultant system. Size is sometimes (but not always) an indicator of design complexity and is almost always an indicator of increased coding, integration, and testing effort.

15.3.1 Function-Based Metrics

WebRef
Much useful information about function points can be obtained at **www.ifpug.org** and **www.functionpoints.com.**

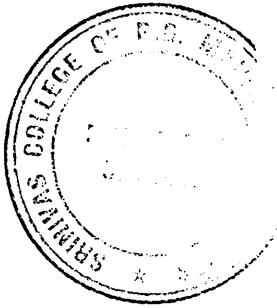
The *function point metric* (FP), first proposed by Albrecht [ALB79], can be used effectively as a means for measuring the functionality delivered by a system.[6] Using historical data, the FP can then be used to (1) estimate the cost or effort required to design, code, and test the software; (2) predict the number of errors that will be encountered during testing, and (3) forecast the number of components and/or the number of projected source lines in the implemented system.

Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity. Information domain values are defined in the following manner:[7]

6 Since Albrecht's original work, hundreds of books, papers, and articles have been written on FP. A worthwhile bibliography can be found at [IFP03].

7 In actuality, the definition of information domain values and the manner in which they are counted are a bit more complex. The interested reader should see [IFP01] for more details.

Number of external inputs (EIs). Each *external input* originates from a user or is transmitted from another application and provides distinct application-oriented data or control information. Inputs are often used to update *internal logical files* (ILFs). Inputs should be distinguished from inquiries, which are counted separately.



Number of external outputs (EOs). Each *external output* is derived within the application and provides information to the user. In this context external output refers to reports, screens, error messages, and so on. Individual data items within a report are not counted separately.

Number of external inquiries (EQs). An *external inquiry* is defined as an on-line input that results in the generation of some immediate software response in the form of an on-line output (often retrieved from an ILF).

Number of internal logical files (ILFs). Each *internal logical file* is a logical grouping of data that resides within the application's boundary and is maintained via external inputs.

Number of external interface files (EIFs). Each *external interface file* is a logical grouping of data that resides external to the application but provides data that may be of use to the application.

Once these data have been collected, the table in Figure 15.2 is completed and a complexity value is associated with each count. Organizations that use function point methods develop criteria for determining whether a particular entry is simple, average, or complex. Nonetheless, the determination of complexity is somewhat subjective.

To compute function points (FP), the following relationship is used:

$$FP = \text{count total} \times [0.65 + 0.01 \times \Sigma (F_i)] \qquad (15\text{-}1)$$

where count total is the sum of all FP entries obtained from Figure 15.2.

The F_i (i = 1 to 14) are *value adjustment factors* (VAF) based on responses to the following questions [LON02]:

1. Does the system require reliable backup and recovery?

FIGURE 15.2
Computing function points

Information Domain Value	Count		Weighting factor Simple	Average	Complex		
External Inputs (EIs)		×	3	4	6	=	
External Outputs (EOs)		×	4	5	7	=	
External Inquiries (EQs)		×	3	4	6	=	
Internal Logical Files (ILFs)		×	7	10	15	=	
External Interface Files (EIFs)		×	5	7	10	=	
Count total						→	

KEY POINT
Value adjustment factors are used to provide an indication of problem complexity.

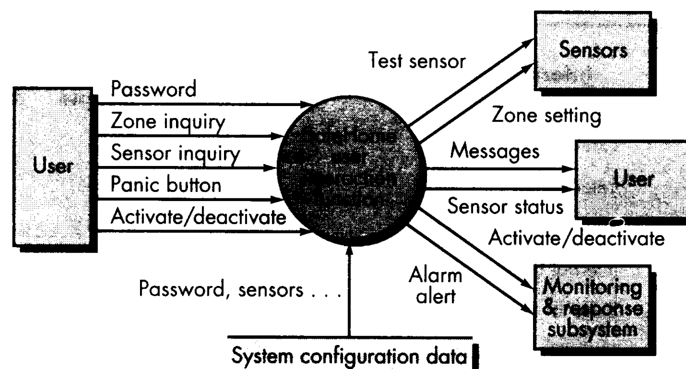
2. Are specialized data communications required to transfer information to or from the application?
3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system run in an existing, heavily utilized operational environment?
6. Does the system require on-line data entry?
7. Does the on-line data entry require the input transaction to be built over multiple screens or operations?
8. Are the ILFs updated on-line?
9. Are the inputs, outputs, files, or inquiries complex?
10. Is the internal processing complex?
11. Is the code designed to be reusable?
12. Are conversion and installation included in the design?
13. Is the system designed for multiple installations in different organizations?
14. Is the application designed to facilitate change and for ease of use by the user?

Each of these questions is answered using a scale that ranges from 0 (not important or applicable) to 5 (absolutely essential). The constant values in Equation (15-1) and the weighting factors that are applied to information domain counts are determined empirically.

WebRef
An on-line FP calculator can be found at **irb.cs.uni-magdeburg.de/sw-eng/us/java/fp/.**

To illustrate the use of the FP metric in this context, we consider a simple analysis model representation, illustrated in Figure 15.3. Referring to the figure, a data flow diagram (Chapter 8) for a function within the *SafeHome* software is represented. The function manages user interaction, accepting a user password to activate or deactivate the system, and allows inquiries on the status of security zones and various security sensors. The function displays a series of prompting messages and sends appropriate control signals to various components of the security system.

FIGURE 15.3
A data flow model for *SafeHome* software



The data flow diagram is evaluated to determine a set of key information domain measures required for computation of the function point metric. Three external inputs—**password, panic button,** and **activate/deactivate**—are shown in the figure along with two external inquires—**zone inquiry** and **sensor inquiry.** One ILF **(system configuration file)** is shown. Two external outputs **(messages** and **sensor status)** and four EIFs **(test sensor, zone setting, activate/deactivate,** and **alarm alert)** are also present. These data, along with the appropriate complexity, are shown in Figure 15.4.

The count total shown in Figure 15.4 must be adjusted using Equation (15-1):

$$\text{FP} = \text{count total} \times [0.65 + 0.01 \times \Sigma\, (F_i)]$$

where count total is the sum of all FP entries obtained from Figure 15.4 and F_i (i = 1 to 14) are value adjustment factors. For the purposes of this example, we assume that $\Sigma\, (F_i)$ is 46 (a moderately complex product). Therefore,

$$\text{FP} = 50 \times [0.65 + (0.01 \times 46)] = 56$$

Based on the projected FP value derived from the analysis model, the project team can estimate the overall implemented size of the *SafeHome* user interaction function. Assume that past data indicates that one FP translates into 60 lines of code (an object-oriented language is to be used) and that 12 FPs are produced for each person-month of effort. These historical data provide the project manager with important planning information that is based on the analysis model rather than preliminary estimates. Assume further that past projects have found an average of three errors per function point during analysis and design reviews and four errors per function point during unit and integration testing. These data can help software engineers assess the completeness of their review and testing activities.

Uemura and his colleagues [UEM99] suggest that function points can also be computed from UML class and sequence diagrams (Chapters 8 and 10). The interested reader should see [UEM99] for details.

FIGURE 15.4

Computing function points

Information Domain Value	Count		Weighting factor: Simple	Average	Complex		
External Inputs (EIs)	3	×	(3)	4	6	=	9
External Outputs (EOs)	2	×	(4)	5	7	=	8
External Inquiries (EQs)	2	×	(3)	4	6	=	6
Internal Logical Files (ILFs)	1	×	(7)	10	15	=	7
External Interface Files (EIFs)	4	×	(5)	7	10	=	20
Count total						→	50

> "Rather than just musing on what 'new metric' might apply . . . we should also be asking ourselves the more basic question, 'What will we do with metrics?' "
>
> **Michael Mah and Larry Putnam**

15.3.2 Metrics for Specification Quality

Davis and his colleagues [DAV93] propose a list of characteristics that can be used to assess the quality of the analysis model and the corresponding requirements specification: *specificity* (lack of ambiguity), *completeness, correctness, understandability, verifiability, internal and external consistency, achievability, concision, traceability, modifiability, precision,* and *reusability.* In addition, the authors [DAV93] note that high-quality specifications are electronically stored, executable or at least interpretable, annotated by relative importance, stable, versioned, organized, cross-referenced, and specified at the right level of detail.

KEY POINT
By measuring characteristics of the specification, it is possible to gain quantitative insight into specificity and completeness.

Although many of these characteristics appear to be qualitative in nature, Davis et al. [DAV93] suggest that each can be represented using one or more metrics. For example, we assume that there are n_r requirements in a specification, such that

$$n_r = n_f + n_{nf}$$

where n_f is the number of functional requirements and n_{nf} is the number of non-functional (e.g., performance) requirements.

To determine the *specificity* (lack of ambiguity) of requirements, Davis et al. suggest a metric that is based on the consistency of the reviewers' interpretation of each requirement:

$$Q_1 = n_{ui}/n_r$$

where n_{ui} is the number of requirements for which all reviewers had identical interpretations. The closer the value of Q to 1, the lower is the ambiguity of the specification.

The *completeness* of functional requirements can be determined by computing the ratio

$$Q_2 = n_u/[n_i \times n_s]$$

where n_u is the number of unique function requirements, n_i is the number of inputs (stimuli) defined or implied by the specification, and n_s is the number of states specified. The Q_2 ratio measures the percentage of necessary functions that have been specified for a system. However, it does not address nonfunctional requirements. To incorporate these into an overall metric for completeness, we must consider the degree to which requirements have been validated:

$$Q_3 = n_c/[n_c + n_{nv}]$$

where n_c is the number of requirements that have been validated as correct and n_{nv} is the number of requirements that have not yet been validated.

> **"Measure what is measurable, and what is not measurable, make measurable."**
> **Galileo**

15.4 Metrics for the Design Model

It is inconceivable that the design of a new aircraft, a new computer chip, or a new office building would be conducted without defining design measures, determining metrics for various aspects of design quality, and using them to guide the manner in which the design evolves. And yet, the design of complex software-based systems often proceeds with virtually no measurement. The irony of this is that design metrics for software are available, but the vast majority of software engineers continue to be unaware of their existence.

Design metrics for computer software, like all other software metrics, are not perfect. Debate continues over their efficacy and the manner in which they should be applied. Many experts argue that further experimentation is required before design measures can be used. And yet, design without measurement is an unacceptable alternative.

15.4.1 Architectural Design Metrics

Architectural design metrics focus on characteristics of the program architecture (Chapter 10) with an emphasis on the architectural structure and the effectiveness of modules or components within the architecture. These metrics are "black box" in the sense that they do not require any knowledge of the inner workings of a particular software component.

Card and Glass [CAR90] define three software design complexity measures: structural complexity, data complexity, and system complexity.

For hierarchical architectures (e.g., call and return architectures), *structural complexity* of a module i is defined in the following manner:

$$S(i) = f^2_{out}(i) \qquad (15\text{-}2)$$

KEY POINT
Metrics can provide insight into structural data and system complexity associated with architectural design.

where $f_{out}(i)$ is the fan-out[8] of module i.

Data complexity provides an indication of the complexity in the internal interface for a module i and is defined as

$$D(i) = v(i)/[f_{out}(i) + 1] \qquad (15\text{-}3)$$

where $v(i)$ is the number of input and output variables that are passed to and from module i.

---

8 *Fan-out* is defined as the number of modules immediately subordinate to the module i, that is, the number of modules that are directly invoked by module i. *Fan-in* is defined as the number of modules that directly invoke module i.

Finally, *system complexity* is defined as the sum of structural and data complexity, specified as

$$C(i) = S(i) + D(i) \tag{15-4}$$

As each of these complexity values increases, the overall architectural complexity of the system also increases. This leads to a greater likelihood that integration and testing effort will also increase.

Fenton [FEN91] suggests a number of simple morphology (i.e., shape) metrics that enable different program architectures to be compared using a set of straightforward dimensions. Referring to the call-and-return architecture in Figure 15.5, the following metrics can be defined:

size = $n + a$

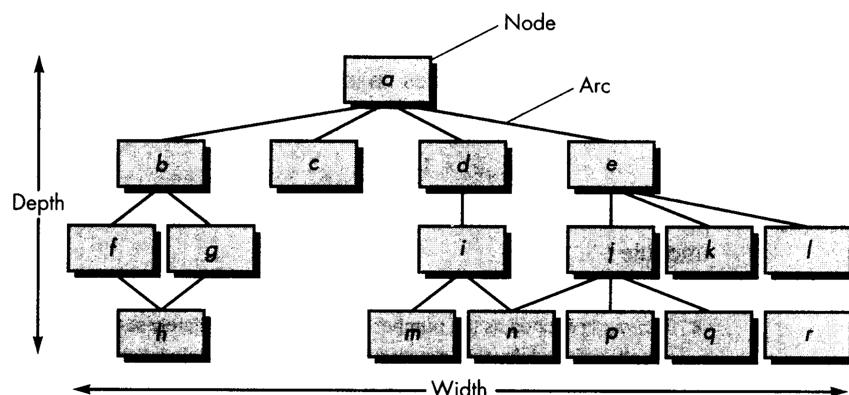
where n is the number of nodes and a is the number of arcs. For the architecture shown in Figure 15.5,

size = 17 + 18 = 35
depth = 4, the longest path from the root (top) node to a leaf node.
width = 6, maximum number of nodes at any one level of the architecture.
arc-to-node ratio, $r = a/n$,

which measures the connectivity density of the architecture and may provide a simple indication of the coupling of the architecture. For the architecture shown in Figure 15.5, r = 18/17 = 1.06.

The U.S. Air Force Systems Command [USA87] has developed a number of software quality indicators that are based on measurable design characteristics of a computer program. Using concepts similar to those proposed in IEEE Std. 982.1-1988 [IEE94], the Air Force uses information obtained from data and architectural design

FIGURE 15.5 Morphology metrics



to derive a design structure quality index (DSQI) that ranges from 0 to 1. The following values must be ascertained to compute the DSQI [CHA89]:

S_1 = the total number of modules defined in the program architecture

S_2 = the number of modules whose correct function depends on the source of data input or that produce data to be used elsewhere (in general, control modules, among others, would not be counted as part of S_2)

S_3 = the number of modules whose correct function depends on prior processing

S_4 = the number of database items (includes data objects and all attributes that define objects)

S_5 = the total number of unique database items

S_6 = the number of database segments (different records or individual objects)

S_7 = the number of modules with a single entry and exit (exception processing is not considered to be a multiple exit)

Once values S_1 through S_7 are determined for a computer program, the following intermediate values can be computed:

Program structure: D_1, where D_1 is defined as follows: If the architectural design was developed using a distinct method (e.g., data flow-oriented design or object-oriented design), then $D_1 = 1$, otherwise $D_1 = 0$.

Module independence: $D_2 = 1 - (S_2/S_1)$

Modules not dependent on prior processing: $D_3 = 1 - (S_3/S_1)$

Database size: $D_4 = 1 - (S_5/S_4)$

Database compartmentalization: $D_5 = 1 - (S_6/S_4)$

Module entrance/exit characteristic: $D_6 = 1 - (S_7/S_1)$

With these intermediate values determined, the DSQI is computed in the following manner:

$$\text{DSQI} = \Sigma\, w_i D_i \qquad (15\text{-}5)$$

where $i = 1$ to 6, w_i is the relative weighting of the importance of each of the intermediate values, and $\Sigma\, w_i = 1$ (if all D_i are weighted equally, then $w_i = 0.167$).

The value of DSQI for past designs can be determined and compared to a design that is currently under development. If the DSQI is significantly lower than average, further design work and review are indicated. Similarly, if major changes are to be made to an existing design, the effect of those changes on DSQI can be calculated.

> **"Measurement can be seen as a detour. This detour is necessary because humans mostly are not able to make clear and objective decisions [without quantitative support]."**
>
> **Horst Zuse**

15.4.2 Metrics for Object-Oriented Design

There is much about object-oriented design that is subjective—an experienced designer "knows" how to characterize an OO system so that it will effectively implement customer requirements. But, as an OO design model grows in size and complexity, a more objective view of the characteristics of the design can benefit both the experienced designer (who gains additional insight) and the novice (who obtains an indication of quality that would otherwise be unavailable).

In a detailed treatment of software metrics for OO systems, Whitmire [WHI97] describes nine distinct and measurable characteristics of an OO design:

What characteristics can be measured when we assess an OO design?

Size. Size is defined in terms of four views: population, volume, length, and functionality. *Population* is measured by taking a static count of OO entities such as classes or operations. *Volume* measures are identical to population measures but are collected dynamically—at a given instant of time. *Length* is a measure of a chain of interconnected design elements (e.g., the depth of an inheritance tree is a measure of length). *Functionality* metrics provide an indirect indication of the value delivered to the customer by an OO application.

Complexity. Like size, there are many differing views of software complexity [ZUS97]. Whitmire views complexity in terms of structural characteristics by examining how classes of an OO design are interrelated to one another.

Coupling. The physical connections between elements of the OO design (e.g., the number of collaborations between classes or the number of messages passed between objects) represent coupling within an OO system.

Sufficiency. Whitmire defines sufficiency as "the degree to which an abstraction possesses the features required of it, or the degree to which a design component possesses features in its abstraction, from the point of view of the current application." Stated another way, we ask: What properties does this abstraction (class) need to possess to be useful to me? [WHI97]. In essence, a design component (e.g., a class) is sufficient if it fully reflects all properties of the application domain object that it is modeling––that is, that the abstraction (class) possesses the features required of it.

> "Many of the decisions for which I had to rely on folklore and myth can now be made using quantitative data."
>
> Scott Whitmire

Completeness. The only difference between completeness and sufficiency is "the feature set against which we compare the abstraction or design component" [WHI97]. Sufficiency compares the abstraction from the point of view of the current application. Completeness considers multiple points of view, asking the question: What properties are required to fully represent the problem domain object? Because the criterion for completeness considers different points of view, it indirectly implies the degree to which the abstraction or design component can be reused.