SOFTWARE TESTING

Scheme and Syllabus

Subject Code: 10IS65 **I.A. Marks : 25**

Hours/Week : 04 **Exam Hours: 03**

Total Hours : 52 **Exam Marks: 100**

PART – A

UNIT 1 **6 Hours**

A Perspective on Testing, Examples: Basic definitions, Test cases, Insights from a Venn diagram, Identifying test cases, Error and fault taxonomies, Levels of testing. Examples: Generalized pseudocode, The triangle problem, The NextDate function, The commission problem, The SATM (Simple Automatic Teller Machine) problem, The currency converter, Saturn windshield wiper.

UNIT 2 **7 Hours**

Boundary Value Testing, Equivalence Class Testing, Decision Table- Based Testing: Boundary value analysis, Robustness testing, Worst-case testing, Special value testing, Examples, Random testing, Equivalence classes, Equivalence test cases for the triangle problem, NextDate function, and the commission problem, Guidelines and observations. Decision tables, Test cases for the triangle problem, NextDate function, and the commission problem, Guidelines and observations.

UNIT 3 **7 Hours**

Path Testing, Data Flow Testing: DD paths, Test coverage metrics, Basis path testing, guidelines and observations. Definition-Use testing, Slice-based testing, Guidelines and observations.

UNIT 4 **6 Hours**

Levels of Testing, Integration Testing: Traditional view of testing levels, Alternative life-cycle models, The SATM system, Separating integration and system testing. A closer look at the SATM system, Decomposition-based, call graph-based, Path-based integrations

PART – B

UNIT 5 **7 Hours**

System Testing, Interaction Testing: Threads, Basic concepts for requirements specification, Finding threads, Structural strategies and functional strategies for thread testing, SATM test threads,

System testing guidelines, ASF (Atomic System Functions) testing example. Context of interaction, A taxonomy of interactions, Interaction, composition, and determinism, Client/Server Testing,

UNIT 6 **7 Hours**

Process Framework: Validation and verification, Degrees of freedom, Varieties of software. Basic principles: Sensitivity, redundancy, restriction, partition, visibility, Feedback. The quality process, Planning and monitoring, Quality goals, Dependability properties, Analysis, Testing, Improving the process, Organizational factors.

UNIT 7 **6 Hours**

Fault-Based Testing, Test Execution: Overview, Assumptions in faultbased testing, Mutation analysis, Fault-based adequacy criteria, Variations on mutation analysis. Test Execution: Overview, from test case specifications to test cases, Scaffolding, Generic versus specific scaffolding, Test oracles, Self-checks as oracles, Capture and replay.

UNIT 8 **6 Hours**

Planning and Monitoring the Process, Documenting Analysis and Test: Quality and process, Test and analysis strategies and plans, Risk planning, Monitoring the process, Improving the process, The quality team, Organizing documents, Test strategy document, Analysis and test plan, Test design specifications documents, Test and analysis reports.

TEXT BOOKS:

1. Paul C. Jorgensen: Software Testing, A Craftsman's Approach, 3rd Edition, Auerbach Publications, 2008. (Listed topics only from Chapters 1, 2, 5, 6, 7, 9, 10, 12, 13,14, 15)

2. Mauro Pezze, Michal Young: Software Testing and Analysis – Process, Principles and Techniques, Wiley India, 2008. (Listed topics only from Chapters 2, 3, 4, 16, 17, 20, 24)

REFERENCE BOOKS:

1. Aditya P Mathur: Foundations of Software Testing, Pearson Education, 2008.

2. Srinivasan Desikan, Gopalaswamy Ramesh: Software testing Principles and Practices, 2nd Edition, Pearson Education, 2007.

3. Brian Marrick: The Craft of Software Testing, Pearson Education, 1995.

Table of contents

Sl No.	Unit Description	Page No.
1	UNIT 1 A Perspective on Testing, Examples	1-15
2	UNIT 2 Boundary Value Testing, Equivalence Class Testing, Decision Table- Based Testing	16-61
3	UNIT 3 Path Testing, Data Flow Testing	62-99
4	UNIT 4 Levels of Testing, Integration Testing	100-125
5	UNIT 5 System Testing, Interaction Testing	126-155
6	UNIT 6 Process Framework	156-177
7	UNIT 7 Fault-Based Testing, Test Execution	178-196
8	UNIT 8 Planning and Monitoring the Process, Documenting Analysis and Test	197-225

UNIT 1

A PERSPECTIVE ON TESTING

Why do we test? There are two main reasons: to make a judgment about quality or acceptability, and to discover problems. We test because we know that we are fallible — this is especially true in the domain of software and software controlled systems. The goal of this chapter is to create a perspective (or context) on software testing. We will operate within this context for the remainder of the text.

1.1 Basic Definitions

Much of testing literature is mired in confusing (and sometimes inconsistent) terminology, probably because testing technology has evolved over decades and via scores of writers. The terminology here (and throughout this book) is taken from standards developed by the Institute of Electronics and Electrical Engineers Computer Society. To get started let's look at a useful progression of terms [IEEE 83].

Error

People make errors. A good synonym is "mistake". When people make mistakes while coding, we call these mistakes "bugs". Errors tend to propagate; a requirements error may be magnified during design, and amplified still more during coding.

Fault

A fault is the result of an error. It is more precise to say that a fault is the representation of an error, where representation is the mode of expression, such as narrative text, dataflow diagrams, hierarchy charts, source code, and so on. "Defect" is a good synonym for fault; so is "bug". Faults can be elusive. When a designer makes an error of omission, the resulting fault is that something is missing that should be present in the representation. This suggests a useful refinement; to borrow from the Church, we might speak of faults of commission and faults of omission. A fault of commission occurs when we enter something into a representation that is incorrect. Faults of omission occur when we fail to enter correct information. Of these two types, faults of omission are more difficult to detect and resolve.

Failure

A failure occurs when a fault executes. Two subtleties arise here: one is that failures only occur in an executable representation, which is usually taken to be source code, or more precisely, loaded object code. The second subtlety is that this definition relates failures only to faults of commission. How can we deal with "failures" that correspond to faults of omission? We can push this still further: what about faults that never happen to execute, or maybe don't execute for a long time? The Michaelangelo virus is an example of such a fault. It doesn't execute until Michelangelo's birthday,

March 6. Reviews prevent many failures by finding faults, in fact, well done reviews can find faults of omission.

Incident

When a failure occurs, it may or may not be readily apparent to the user (or customer or tester). An incident is the symptom(s) associated with a failure that alerts the user to the occurrence of a failure.

Test

Testing is obviously concerned with errors, faults, failures, and incidents. A test is the act of exercising software with test cases. There are two distinct goals of a test: either to find failures, or to demonstrate correct execution.

Test Case

A test case has an identity, and is associated with a program behavior. A test case also has a set of inputs, a list of expected outputs.

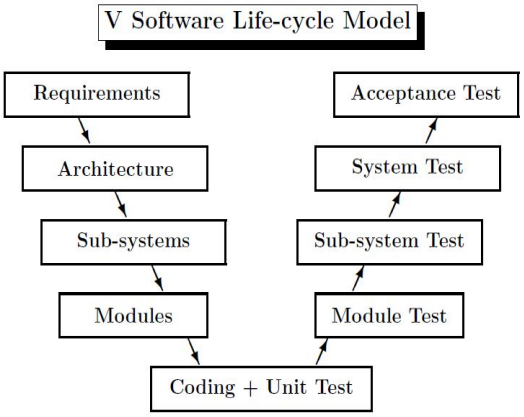


Figure 1.1 A Testing Life Cycle

Figure 1.1 portrays a life cycle model for testing. Notice that, in the development phases, there are three opportunities for errors to be made, resulting in faults that propagate through the remainder of the development. One prominent tester summarizes this life cycle as follows: the first three phases are "Putting Bugs IN", the testing phase is Finding Bugs, and the last three phases are "Getting Bugs OUT" [Poston 90]. The Fault Resolution step is another opportunity for errors (and new faults). When a "fix" causes formerly correct software to misbehave, the fix is deficient. We'll revisit this when we discuss regression testing.

From this sequence of terms, we see that test cases occupy a central position in testing. The process of testing can be subdivided into separate steps: test planning, test case development, running test cases, and evaluating test results. The focus of this book is how to identify useful sets of test cases.

1.2 Test Cases

The essence of software testing is to determine a set of test cases for the item being tested. Before going on, we need to clarify what information should be in a test case. The most obvious information is inputs; inputs are really of two types: pre-conditions (circumstances that hold prior to test case execution) and the actual inputs that were identified by some testing method. The next most obvious part of a test case is the expected outputs; again, there are two types: post conditions and actual outputs. The output portion of a test case is frequently overlooked. Unfortunate, because this is often the hard part. Suppose, for example, you were testing software that determined an optimal route for an aircraft, given certain FAA air corridor constraints and the weather data for a flight day. How would you know what the optimal route really is? There have been various responses to this problem. The academic response is to postulate the existence of an oracle, who "knows all the answers". One industrial response to this problem is known as Reference Testing, where the system is tested in the presence of expert users, and these experts make judgments as to whether or not outputs of an executed set of test case inputs are acceptable. The act of testing entails establishing the necessary pre-conditions, providing the test case inputs, observing the outputs, and then comparing these with the expected outputs to determine whether or not the test passed. The remaining information in a well-developed test case primarily supports testing management. Test cases should have an identity, and a reason for being (requirements tracing is a fine reason). It is also useful to record the execution history of a test case, including when and by whom it was run, the pass/fail result of each execution, and the version (of software) on which it was run. From all of this, it should be clear that test cases are valuable — at least as valuable as source code. Test cases need to be developed, reviewed, used, managed, and saved.

1.3 Insights from a Venn diagram

Testing is fundamentally concerned with behavior; and behavior is orthogonal to the structural view common to software (and system) developers. A quick differentiation is that the structural view focuses on "what it is" and the behavioral view considers "what it does". One of the continuing sources of difficulty for testers is that the base documents are usually written by and for developers, and therefore the emphasis is on structural, rather than behavioral, information. In this section, we develop a simple Venn diagram which clarifies several nagging questions about testing.

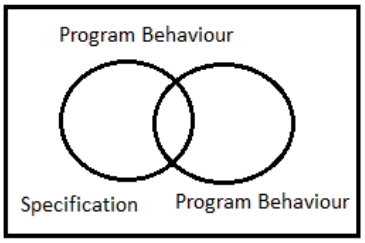


Figure 1.3 Specified and Implemented Program Behaviors

Consider a Universe of program behaviors. (Notice that we are forcing attention on the essence of testing.) Given a program and its specification, consider the set S of specified behaviors, and the set P of programmed behaviors. Figure 1.3 shows the relationship between our universe of discourse and the specified and programmed behaviors. Of all the possible program behaviors, the specified ones are in the circle labeled S; and all those behaviors actually programmed (note the slight

difference between P and U, the Universe) are in P. With this diagram, we can see more clearly the problems that confront a tester. What if there are specified behaviors that have not been programmed? In our earlier terminology, these are faults of omission. Similarly, what if there are programmed (implemented) behaviors that have not been specified? These correspond to faults of commission, and to errors which occurred after the specification was complete. The intersection of S and P (the football shaped region) is the "correct" portion, that is behaviors that are both specified and implemented. A very good view of testing is that it is the determination of the extent of program behavior that is both specified and implemented. (As a sidelight, note that "correctness" only has meaning with respect to a specification and an implementation. It is a relative term, not an absolute.)

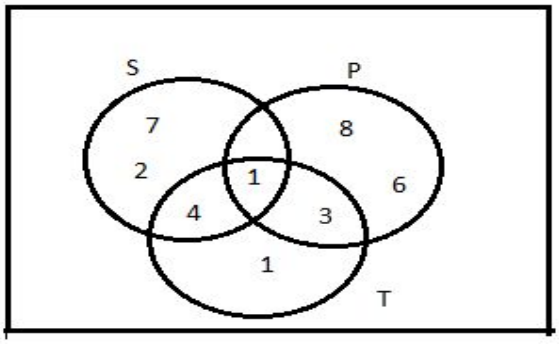


Figure 1.4 Specified, Implemented, and Tested Behaviors

The new circle in Fig. 1.4 is for Test Cases. Notice there is a slight discrepancy with our Universe of Discourse, the set of program behaviors. Since a test case causes a program behavior, the mathematicians might forgive us. Now, consider the relationships among the sets S, P, and T. There may be specified behaviors that are not tested (regions 2 and 5), specified behaviors that are tested (regions 1 and 4), and test cases that correspond to unspecified behaviors (regions 3 and 7). Similarly, there may be programmed behaviors that are not tested (regions 2 and 6), programmed behaviors that are tested (regions 1 and 3), and test cases that correspond to unprogrammed behaviors (regions 4 and 7). Each of these regions is important. If there are specified behaviors for which there are no test cases, the testing is necessarily incomplete. If there are test cases that correspond to unspecified behaviors, two possibilities arise: either such a test case is unwarranted, or the specification is deficient. (In my experience, good testers often postulate test cases of this latter type. This is a fine reason to have good testers participate in specification and design reviews.) We are already at a point where we can see some possibilities for testing as a craft: what can a tester do to make the region where these sets all intersect (region 1) be as large as possible? Another way to get at this is to ask how the test cases in the set T are identified. The short answer is that test cases are identified by a testing method. This framework gives us a way to compare the effectiveness of diverse testing methods, as we shall see in chapters 8 and 11.

1.4 Identifying Test Cases

There are two fundamental approaches to identifying test cases; these are known as functional and structural testing. Each of these approaches has several distinct test case identification methods, more commonly called testing methods.

1.4.1 Functional Testing

Functional testing is based on the view that any program can be considered to be a function that maps values from its input domain to values in its output range. (Function, domain, and range are defined in Chapter 3.) This notion is commonly used in engineering, when systems are considered to be "black boxes". This leads to the term Black Box Testing, in which the content (implementation) of a black box is not known, and the function of the black box is understood completely in terms of its inputs and outputs. In *Zen and The Art of Motorcycle Maintenance* , Pirsig refers to this as "romantic" comprehension [Pirsig 73]. Many times, we operate very effectively with black box knowledge; in fact this is central to object orientation. As an example, most people successfully operate automobiles with only black box knowledge.

Figure 1.5 An Engineer's Black Box

With the functional approach to test case identification, the only information that is used is the specification of the software. There are two distinct advantages to functional test cases: they are independent of how the software is implemented, so if the implementation changes, the test cases are still useful, and test case development can occur in parallel with the implementation, thereby reducing overall project development interval. On the negative side, functional test cases frequently suffer from two problems: there can be significant redundancies among test cases, and this is compounded by the possibility of gaps of untested software. Figure 1.6 shows the results of test cases identified by two functional methods. Method A identifies a larger set of test cases than does Method B. Notice that, for both methods, the set of test cases is completely contained within the set of specified behavior. Since functional methods are based on the specified behavior, it is hard to imagine these methods identifying behaviors that are not specified.

1.4.2 Structural Testing

Structural testing is the other fundamental approach to test case identification. To contrast it with Functional Testing, it is sometimes called White Box (or even Clear Box) Testing. The clear box metaphor is probably more appropriate, because the essential difference is that the implementation (of the Black Box) is known and used to identify test cases. Being able to "see inside" the black box allows the tester to identify test cases based on how the function is actually implemented. Structural Testing has been the subject of some fairly strong theory. To really understand structural testing, the concepts of linear graph theory (Chapter 4) are essential. With these concepts, the tester can rigorously describe exactly what is being tested. Because of its strong theoretical basis, structural testing lends itself to the definition and use of test coverage metrics. Test coverage metrics provide a way to explicitly state the extent to which a software item has been tested, and this in turn, makes testing management more meaningful.

1.4.3 The Functional Versus Structural Debate

Given two fundamentally different approaches to test case identification, the natural question is which is better? If you read much of the literature, you will find strong adherents to either choice. Referring to structural testing, Robert Poston writes: "this tool has been wasting tester's time since

the 1970s. . . [it] does not support good software testing practice and should not be in the testers tool kit" [Poston 91]. In defense of structural testing, Edward Miller [Miller 91] writes: "Branch coverage [a structural test coverage metric], if attained at the 85 percent or better level, tends to identify twice the number of defects that would have been found by 'intuitive' [functional] testing."

1.5 Error and Fault Taxonomies

Our definitions of error and fault hinge on the distinction between process and product: process refers to how we do something, and product is the end result of a process. The point at which testing and Software Quality Assurance meet is that SQA typically tries to improve the product by improving the process. In that sense, testing is clearly more product oriented. SQA is more concerned with reducing errors endemic in the development process, while testing is more concerned with discovering faults in a product. Both disciplines benefit from a clearer definition of types of faults. Faults can be classified in several ways: the development phase where the corresponding error occurred, the consequences of corresponding failures, difficulty to resolve, risk of no resolution, and so on. My favorite is based on anomaly occurrence: one time only, intermittent, recurring, or repeatable. Figure 1.9 contains a fault taxonomy [Beizer 84] that distinguishes faults by the severity of their consequences.

1.6. Levels of Testing

Thus far we have said nothing about one of the key concepts of testing — levels of abstraction. Levels of testing echo the levels of abstraction found in the Waterfall Model of the software development life cycle. While this model has its drawbacks, it is useful for testing as a means of identifying distinct levels of testing, and for clarifying the objectives that pertain to each level.

Table 1 Input/Output Faults **Type**

	Instances
Input	correct input not accepted
	incorrect input accepted
	description wrong or missing
	parameters wrong or missing
Output	wrong format
	wrong result
	correct result at wrong time (too early, too late)
	incomplete or missing result
	spurious result
	spelling/grammar
	cosmetic

Table 2 Logic Faults missing case(s)

duplicate case(s)
extreme condition neglected
misinterpretation
missing condition
extraneous condition(s)

test of wrong variable
incorrect loop iteration
wrong operator (e.g., < instead ≤)

Table 3 Computation Faults incorrect algorithm

missing computation
incorrect operand
incorrect operation
parenthesis error
insufficient precision (round-off, truncation)
wrong built-in function

There is a practical relationship between levels of testing and functional and structural testing. Most practitioners agree that structural testing is most appropriate at the unit level, while functional testing is most appropriate at the system level. While this is generally true, it is also a likely consequence of the base information produced during the requirements specification, preliminary design, and detailed design phases. The constructs defined for structural testing make the most sense at the unit level; and similar constructs are only now becoming available for the integration and system levels of testing. We develop such structures in Part IV to support structural testing at the integration and system levels for both traditional and object-oriented software.

Table 4 Interface Faults incorrect interrupt handling

I/O timing
call to wrong procedure
call to non-existent procedure
parameter mismatch (type, number)
incompatible types
superfluous inclusion

Table 5 Data Faults incorrect initialization

incorrect storage/access
wrong flag/index value
incorrect packing/unpacking
wrong variable used
wrong data reference
scaling or units error
incorrect data dimension
incorrect subscript
incorrect type
incorrect data scope
sensor data out of limits
off by one
inconsistent data

1.2 The Triangle Problem

The year of this writing marks the twentieth anniversary of publications using the Triangle Problem as an example. Some of the more notable entries in this generation of testing literature are [Gruenberger 73], [Brown 75], [Myers 79], [Pressman 82 (and the second and third editions], [Clarke 83], [Clarke 84], [Chellappa 87], and [Hetzel 88]. There are probably others, but this list should suffice.

1.2.1 Problem Statement

The Triangle Program accepts three integers as input; these are taken to be sides of a triangle. The output of the program is the type of triangle determined by the three sides: Equilateral, Isosceles, Scalene, or NotATriangle. Sometimes this problem is extended to include right triangles as a fifth type; we will use this extension in some of the exercises.

1.2.2 Discussion

Perhaps one of the reasons for the longevity of this example is that, among other things, it typifies some of the incomplete definition that impairs communication among customers, developers, and testers. This specification presumes the developers know some details about triangles, in particular the Triangle Property: the sum of any pair of sides must be strictly greater than the third side. If a, b, and c denote the three integer sides, then the triangle property is mathematically stated as three inequalities: $a < b + c$, $b < a + c$, and $c < a + b$. If any one of these fails to be true, the integers a, b, and c do not constitute sides of a triangle. If all three sides are equal, they constitute an equilateral triangle; if exactly one pair of sides is equal, they form an isosceles triangle; and if no pair of sides is equal, they constitute a scalene triangle. A good tester might further clarify the problem statement by putting limits on the lengths of the sides. What response would we expect if we presented the program with the sides -5, -4, -3? We will require that all sides be at least 1, and while we are at it, we may as well declare some upper limit, say 20,000. (Some languages, like Pascal, have an automatic limit, called MAXINT, which is the largest binary integer representable in a certain number of bits.)

1.2.3 Traditional Implementation

The "traditional" implementation of this grandfather of all examples has a rather FORTRAN-like style. The flowchart for this implementation appears in Figure 2.1. The flowchart box numbers correspond to comment numbers in the (FORTRAN-like) TurboPascal program given next. (These numbers correspond exactly to those in [Pressman 82].) I don't really like this implementation very much, so a more structured implementation is given in section 2.1.4. The variable match is used to record equality among pairs of the sides. There is a classical intricacy of the FORTRAN style connected with the variable match: notice that all three tests for the triangle property do not occur. If two sides are equal, say a and c, it is only necessary to compare a+c with b. (Since b must be greater than zero, a + b must be greater than c, because c equals a.) This observation clearly reduces the amount of comparisons that must be made. The efficiency of this version is obtained at the expense of clarity (and ease of testing!). We will find this version useful later in Part III when we discuss infeasible program execution paths. That is the best reason for retaining this version.

```
PROGRAM triangle1 (input, output);
VAR
   a, b, c, match : INTEGER;
BEGIN
     writeln ('Enter 3 integers which are sides of a triangle');
     readln (a, b, c);
     writeln ('Side A is ', a);
     writeln ('Side B is ', b);
     writeln ('Side C is ', c);
     match := 0;
     IF a = b                                    {1 }
     THEN  match := match + 1;                          {2 }
     IF a = c                                    {3 }
     THEN  match := match + 2;                          {4 }
     IF b = c                                    {5 }
     THEN  match := match + 3;                          {6 }
     IF match = 0                                  {7 }
     THEN IF (a+b) <=c                                   {8 }
          THEN writeln ('Not a Triangle')                {12.1}
          ELSE IF (b+c) <=a                             {9 }
               THEN writeln ('Not a Triangle')           {12.2}
               ELSE IF (a+c) <=b                        {10 }
                    THEN writeln ('Not a Triangle')     {12.3}
                    ELSE writeln ('Triangle is Scalene')  {11  }
     ELSE IF match=1                                   {13  }
          THEN IF (a+c) <=b                             {14  }
               THEN writeln ('Not a Triangle')          {12.4}
               ELSE writeln ('Triangle is Isosceles')    {15.1}
          ELSE IF match=2                               {16  }
               THEN IF (a+c) <=b                        {17  }
                    THEN writeln ('Not a Triangle')     {12.5}
                    ELSE writeln ('Triangle is Isosceles   {15.2}
               ELSE IF match=3                          {18  }
                    THEN IF (b+c) <=a                   {19  }
                         THEN writeln ('Not a Triangle')   {12.6}
                         ELSE writeln ('Triangle is Isosceles') {15.3}
                    ELSE writeln ('Triangle is Equilateral');   {20  }
END.
```

1.2 The Triangle Problem

The year of this writing marks the twentieth anniversary of publications using the Triangle Problem as an example. Some of the more notable entries in this generation of testing literature are [Gruenberger 73], [Brown 75], [Myers 79], [Pressman 82 (and the second and third editions], [Clarke 83], [Clarke 84], [Chellappa 87], and [Hetzel 88]. There are probably others, but this list should suffice.

1.2.1 Problem Statement

The Triangle Program accepts three integers as input; these are taken to be sides of a triangle. The output of the program is the type of triangle determined by the three sides: Equilateral, Isosceles, Scalene, or NotATriangle. Sometimes this problem is extended to include right triangles as a fifth type; we will use this extension in some of the exercises.

1.2.2 Discussion

Perhaps one of the reasons for the longevity of this example is that, among other things, it typifies some of the incomplete definition that impairs communication among customers, developers, and testers. This specification presumes the developers know some details about triangles, in particular the Triangle Property: the sum of any pair of sides must be strictly greater than the third side. If a, b, and c denote the three integer sides, then the triangle property is mathematically stated as three inequalities: a < b + c, b < a + c, and c < a + b. If any one of these fails to be true, the integers a, b, and c do not constitute sides of a triangle. If all three sides are equal, they constitute an equilateral triangle; if exactly one pair of sides is equal, they form an isosceles triangle; and if no pair of sides is equal, they constitute a scalene triangle. A good tester might further clarify the problem statement by putting limits on the lengths of the sides. What response would we expect if we presented the program with the sides -5, -4, -3? We will require that all sides be at least 1, and while we are at it, we may as well declare some upper limit, say 20,000. (Some languages, like Pascal, have an automatic limit, called MAXINT, which is the largest binary integer representable in a certain number of bits.)

1.2.3 Traditional Implementation

The "traditional" implementation of this grandfather of all examples has a rather FORTRAN-like style. The flowchart for this implementation appears in Figure 2.1. The flowchart box numbers correspond to comment numbers in the (FORTRAN-like) TurboPascal program given next. (These numbers correspond exactly to those in [Pressman 82].)

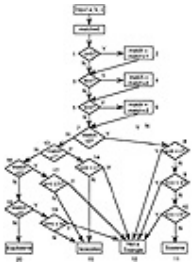


Figure 2.1 Flowchart for the Traditional Triangle Program Implementation

The variable match is used to record equality among pairs of the sides. There is a classical intricacy of the FORTRAN style connected with the variable match: notice that all three tests for the triangle property do not occur. If two sides are equal, say a and c, it is only necessary to compare a+c with b. (Since b must be greater than zero, a + b must be greater than c, because c equals a.) This observation clearly reduces the amount of comparisons that must be made. The efficiency of this version is obtained at the expense of clarity (and ease of testing!).

We will find this version useful later in Part III when we discuss infeasible program execution paths. That is the best reason for retaining this version.

```
PROGRAM triangle1 (input, output);
VAR
   a, b, c, match : INTEGER;
BEGIN
     writeln ('Enter 3 integers which are sides of a triangle');
     readln (a, b, c);
     writeln ('Side A is ', a);
```

```
    writeln ('Side B is ', b);
    writeln ('Side C is ', c);
    match := 0;
    IF a = b                                {1 }
    THEN  match := match + 1;                       {2 }
    IF a = c                                {3 }
    THEN  match := match + 2;                       {4 }
    IF b = c                                {5 }
    THEN  match := match + 3;                       {6 }
    IF match = 0                              {7 }
    THEN IF (a+b) <=c                          {8 }
         THEN writeln ('Not a Triangle')              {12.1}
        ELSE IF (b+c) <=a                       {9 }
             THEN writeln ('Not a Triangle')          {12.2}
            ELSE IF (a+c) <=b                   {10 }
                 THEN writeln ('Not a Triangle')      {12.3}
                 ELSE writeln ('Triangle is Scalene')    {11  }
    ELSE IF match=1                             {13  }
        THEN IF (a+c) <=b                       {14  }
             THEN writeln ('Not a Triangle')          {12.4}
             ELSE writeln ('Triangle is Isosceles')      {15.1}
        ELSE IF match=2                         {16  }
            THEN IF (a+c) <=b                   {17  }
                 THEN writeln ('Not a Triangle')      {12.5}
                 ELSE writeln ('Triangle is Isosceles    {15.2}
            ELSE IF match=3                     {18  }
                 THEN IF (b+c) <=a              {19  }
                      THEN writeln ('Not a Triangle')   {12.6}
                      ELSE writeln ('Triangle is Isosceles') {15.3}
                  ELSE writeln ('Triangle is Equilateral');   {20  }
END.
```

1.2.4 Structured Implementation

Figure 2.2 is a dataflow diagram description of the triangle program. We could implement it as a main program with the four indicated procedures. Since we will use this example later for unit testing, the four procedures have been merged into one TurboPascal program. Comment lines relate sections of the code to the decomposition given in Figure 1.2.

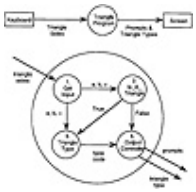


Figure 1.2 Dataflow Diagram for a Structured Triangle Program Implementation

```
PROGRAM triangle2 (input, output);
VAR
    a, b, c     : INTEGER;
    IsATriangle : BOOLEAN;
BEGIN
{Function 1: Get Input}
    writeln ('Enter 3 integers which are sides of a triangle');
    readln (a,b,c);
    writeln ('Side A is ',a);
    writeln ('Side B is ',b);
    writeln ('Side C is ',c);
{Function 2: Is A Triangle?}
```

```
    IF (a < b + c) AND (b < a + c) AND (c < a + b)
    THEN IsATriangle := TRUE
    ELSE IsATriangle := FALSE;
{Function 3: Determine Triangle Type}
    IF IsATriangle
    THEN IF (a = b) AND (b = c)
        THEN writeln ('Triangle is Equilateral');
        ELSE IF (a <> b) AND (a <> c) AND (b <> c)
            THEN writeln ('Triangle is Scalene');
            ELSE writeln ('Triangle is Isosceles')
    ELSE writeln ('Not a Triangle');
{Note: Function 4, the Output Controller, has been merged into
clauses in Function 3.}
END.
```

1.3 The NextDate Function

The complexity in the Triangle Program is due to relationships between inputs and correct outputs. We will use the NextDate function to illustrate a different kind of complexity — logical relationships among the input variables themselves.

1.3.1 Problem Statement

NextDate is a function of three variables: month, day, and year. It returns the date of the day after the input date. The month, day, and year variables have numerical values: with $1 \leq month \leq 12$, $1 \leq day \leq 31$, and $1812 \leq year \leq 2012$.

1.3.2 Discussion

There are two sources of complexity in the NextDate function: the just mentioned complexity of the input domain, and the rule that distinguishes common years from leap years. Since a year is 365.2422 days long, leap years are used for the "extra day" problem. If we declared a leap year every fourth year, there would be a slight error. The Gregorian Calendar (instituted by Pope Gregory in 1582) resolves this by adjusting leap years on century years. Thus a year is a leap year if it is divisible by 4, unless it is a century year. Century years are leap years only if they are multiples of 400 [Inglis 61], [ISO 91], so 1992, 1996, and 2000 are leap years, while the year 1900 is a common year. The NextDate function also illustrates a sidelight of software testing. Many times, we find examples of Zipf's Law, which states that 80% of the activity occurs in 20% of the space. Notice how much of the source code is devoted to leap year considerations.

1.3.3 Implementation

```
PROGRAM NextDate (INPUT, OUTPUT);
TYPE      monthType = 1..12;
        dayType   = 1..31;
        yearType  = 1812..2012;
        dateType  = record
           month : monthType;
           day   : dayType;
           year  : yearType
        end; (*dateType record *)
VAR       today, tomorrow :dateType;
BEGIN (*NextDate*)
writeln ('Enter today' 's date in the form MM DD YYYY');
readln (today.month, today.day, today.year);
```

```
tomorrow := today;
WITH today DO
   CASE month OF
   1,3,5,7,8,10:   IF day < 31 THEN tomorrow.day := day + 1
                ELSE Begin
                        tomorrow.day := 1;
                        tomorrow.month := month + 1
                     End;
   4,6,9,11 :      IF day < 30 THEN tomorrow.day := day + 1
                ELSE Begin
                        tomorrow.day := 1;
                        tomorrow.month := month + 1
                     End;
   12:   IF day < 31 THEN tomorrow.day := day + 1
         ELSE Begin
                tomorrow.day := 1;
                tomorrow.month := 1;
                IF year = 2012
                 THEN Writeln ('2012 is over')
                 ELSE tomorrow.year := year + 1
              End;
   2:    IF day < 28 THEN tomorrow.day := day + 1
         ELSE IF day = 28
            THEN IF ((year MOD 4) =0) AND ((year MOD 400) <>0)
               THEN tomorrow.day := 29 {leap year}
               ELSE Begin         {common year}
                        tomorrow.day := 1;
                        tomorrow.month := 3;
                     End
         ELSE IF   day = 29
            THEN Begin
                    tomorrow.day := 1;
                    tomorrow.month := 3;
                 End;
            ELSE writeln('Cannot have Feb.', day);
   End;(*CASE month*)
End; (*WITH today*)
Writeln ('Tomorrow''s date is', tomorrow.month:3,
     tomorrow.day:3, tomorrow.year:5);
END.  (*NextDate*)
```

1.4 The Commission Problem

Our third example is more typical of commercial computing. It contains a mix of computation and decision making, so it leads to interesting testing questions.

1.4.1 Problem Statement

Rifle salespersons in the Arizona Territory sold rifle locks, stocks, and barrels made by a gunsmith in Missouri. Locks cost $45.00, stocks cost $30.00, and barrels cost $25.00. Salespersons had to sell at least one complete rifle per month, and production limits are such that the most one salesperson could sell in a month is 70 locks, 80 stocks, and 90 barrels. Each rifle salesperson sent a telegram to the Missouri company with the total order for each town (s)he visits; salespersons visit at least one town per month, but travel difficulties made ten towns the upper limit. At the end of each month, the company computed commissions as follows: 10% on sales up to $1000, 15% on the next $800, and 20% on any sales in excess of $1800. The company had four salespersons. The telegrams from each salesperson were sorted into piles (by person) and at the end of each month a datafile is

prepared, containing the salesperson's name, followed by one line for each telegram order, showing the number of locks, stocks, and barrels in that order. At the end of the sales data lines, there is an entry of "-1" in the position where the number of locks would be to signal the end of input for that salesperson. The program produces a monthly sales report that gives the salesperson's name, the total number of locks, stocks, and barrels sold, the salesperson's total dollar sales, and finally his/her commission.

1.5 The SATM System

To better discuss the issues of integration and sytem testing, we need an example with larger scope. The automated teller machine described here is a refinement of that in [Topper 93]; it contains an interesting variety of functionality and interactions. Although it typifies real-time systems, practitioners in the commercial EDP domain are finding that even traditional COBOL systems have many of the problems usually associated with real-time systems.

1.5.1 Problem Statement

The SATM system communicates with bank customers via the fifteen screens shown in Figure 2.4. Using a terminal with features as shown in Figure 1.3, SATM customers can select any of three transaction types: deposits, withdrawals, and balance inquiries, and these can be done on two types of accounts, checking and savings. When a bank customer arrives at an SATM station, screen 1 is displayed. The bank customer accesses the SATM system with a plastic card encoded with a Personal Account Number (PAN), which is a key to an internal customer account file, containing, among other things, the customer's name and account information. If the customer's PAN matches the information in the customer account file, the system presents screen 2 to the customer. If the customer's PAN is not found, screen 4 is displayed, and the card is kept. At screen 2, the customer is prompted to enter his/her Personal Identification Number (PIN). If the PIN is correct (i.e., matches the information in the customer account file), the system displays screen 5; otherwise, screen 3 is displayed. The customer has three chances to get the PIN correct; after three failures, screen 4 is displayed, and the card is kept.

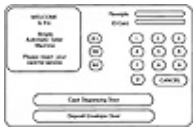


Figure 1.3 The SATM Terminal

On entry to screen 5, the system adds two pieces of information to the customer's account file: the current date, and an increment to the number of ATM sessions. The customer selects the desired transaction from the options shown on screen 5; then the system immediately displays screen 6, where the customer chooses the account to which the selected transaction will be applied. If **balance** is requested, the system checks the local ATM file for any unposted transactions, and reconciles these with the beginning balance for that day from the customer account file. Screen 14 is then displayed. If **deposit** is requested, the status of the Deposit Envelope slot is determined from a field in the Terminal Control File. If no problem is known, the system displays screen 7 to get the transaction amount. If there is a problem with the deposit envelope slot, the system displays screen 12. Once the deposit amount has been entered, the system displays screen 13, accepts the deposit envelope, and processes the deposit. The deposit amount is entered as an unposted amount in the local ATM file, and the count of deposits per month is incremented. Both of these (and other information) are processed by the Master ATM (centralized) system once per day. The system then displays screen 14. If **withdrawal** is requested, the system checks the status (jammed or free) of the withdrawal chute in the Terminal Control File. If jammed, screen 10 is displayed, otherwise, screen

7 is displayed so the customer can enter the withdrawal amount. Once the withdrawal amount is entered, the system checks the Terminal Status File to see if it has enough money to dispense. If it does not, screen 9 is displayed; otherwise the withdrawal is processed. The system checks the customer balance (as described in the Balance request transaction), and if there are insufficient funds, screen 8 is displayed. If the account balance is sufficient, screen 11 is displayed, and the money is dispensed. The withdrawal amount is written to the unposted local ATM file, and the count of withdrawals per month in incremented. The balance is printed on the transaction receipt as it is for a balance request transaction. After the cash has been removed, the system displays screen 14. When the No button is pressed in screens 10, 12, or 14, the system presents screen 15 and returns the customer's ATM card. Once the card is removed from the card slot, screen 1 is displayed. When the Yes button is pressed in screens 10, 12, or 14, the system presents screen 5 so the customer can select additional transactions.

1.5.2 Discussion

There is a surprising amount of information "buried" in the system description just given. For instance, if you read it closely, you can infer that the terminal only contains ten dollar bills (see screen 7). This textual definition is probably more precise than what is usually encountered in practice. The example is deliberately simple (hence the name).

Figure1.4 SATM Screens

A plethora of questions could be resolved by a list of assumptions. For example, is there a borrowing limit? What keeps a customer from taking out more than his actual balance if he goes to several ATM terminals? There are lots of "start up" questions: how much cash is initially in the machine? How are new customers added to the system? These, and other "real world" refinements, are eliminated to maintain simplicity.

1.6 Saturn Windshield Wiper Controller

The windshield wiper on the Saturn automobile (at least on the 1992 models) is controlled by a lever with a dial. The lever has four positions, OFF, INT (for intermittent), LOW, and HIGH, and the dial has three positions, numbered simply 1, 2, and 3. The dial positions indicate three intermittent speeds, and the dial position is relevant only when the lever is at the INT position. The decision table below shows the windshield wiper speeds (in wipes per minute) for the lever and dial positions.

Lever	OFF	INT	INT	INT	LOW	HIGH
Dial	n/a	1	2	3	n/a	n/a
Wiper	0	4	6	12	30	60

UNIT 2

BOUNDARY VALUE TESTING, EQUIVALENCE CLASS TESTING, DECISION TABLE- BASED TESTING:

2.1 Boundary Value Analysis

For the sake of comprehensible drawings, the discussion relates to a function, F, of two variables x_1 and x_2 . When the function F is implemented as a program, the input variables x_1 and x_2 will have some (possibly unstated) boundaries:

$a \leq x_1 \leq b$

$c \leq x_2 \leq d.$

Unfortunately, the intervals [a, b] and [c, d] are referred to as the ranges of x_1 and x_2 , so right away, we have an overloaded term. The intended meaning will always be clear from its context. Strongly typed languages (such as Ada and Pascal) permit explicit definition of such variable ranges. In fact, part of the historical reason for strong typing was to prevent programmers from making the kind of errors that result in faults that are easily revealed by boundary value testing. Other languages (such as COBOL, FORTRAN, and C) are not strongly typed, so boundary value testing is more appropriate for programs coded in such languages. The input space (domain) of our function F is shown in Figure 2.1. Any point within the shaded rectangle is a legitimate input to the function F.

Boundary value analysis focuses on the boundary of the input space to identify test cases. The rationale behind boundary value testing is that errors tend to occur near the extreme values of an input variable. The US. Army (CECOM) made a study of its software, and found that a surprising portion of faults turned out to be boundary value faults. Loop conditions, for example, may test for < when they should test for ≤, and counters often are "off by one". The desktop publishing program in which this manuscript was typed has an interesting boundary value problem. There are two modes of textual display: a scrolling view in which new pages are indicated by a dotted line, and a page view which displays a full page image showing where the text is placed on the page, together with headers and footers. If the cursor is at the last line of a page and new text is added, an anomaly occurs: in the first mode, the new line(s) simply appear, and the dotted line (page break) is adjusted. In the page display mode, however, the new text is lost — it doesn't appear on either page.

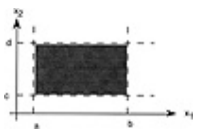


Figure 2.1 Input Domain of a Function of Two Variables

The basic idea of boundary value analysis is to use input variable values at their minimum, just above the minimum, a nominal value, just below their maximum, and at their maximum. There is a commercially available testing tool (named T) that generates such test cases for a properly specified program. This tool has been successfully integrated with two popular front-end CASE tools (Teamwork from Cadre Systems, and Software Through Pictures from Interactive Development Environments). The T tool refers to these values as min, min+, nom, max- and max; we will use this convention here. The next part of boundary value analysis is based on a critical assumption; it's known as the "single fault" assumption in reliability theory. This says that failures are only rarely the result of the simultaneous occurrence of two (or more) faults. Thus the boundary value analysis test cases are obtained by holding the values of all but one variable at their nominal values, and letting that variable assume its extreme values. The boundary value analysis test cases for our function F of two variables are:

$$\{<x_{1nom}, x_{2min}>, <x_{1nom}, x_{2min+}>, <x_{1nom}, x_{2nom}>, <x_{1nom}, x_{2max-}>,$$

$$<x_{1nom}, x_{2max}>, <x_{1min}, x_{2nom}>, <x_{1min+}, x_{2nom}>, <x_{1nom}, x_{2nom}>,$$

$$<x_{1max-}, x_{2nom}>, <x_{1max}, x_{2nom}>\}$$

These are illustrated in Figure 2.2.

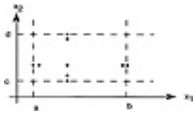


Figure 2.2 Boundary Value Analysis Test Cases for a Function of Two Variables

2.1.1 Generalizing Boundary Value Analysis

The basic boundary value analysis technique can be generalized in two ways: by the number of variables, and by the kinds of ranges. Generalizing the number of variables is easy: if we have a function of n variables, we hold all but one at the nominal values, and let the remaining variable assume the min, min+, nom, max- and max values, and repeat this for each variable. Thus for a function of n variables, boundary value analysis yields 4n + 1 test cases. Generalizing ranges depends on the nature (or more precisely, the type) of the variables themselves. In the NextDate function, for example, we have variables for the month, the day, and the year. In a FORTRAN-like language, we would most likely encode these, so that January would correspond to 1, February to 2, and so on. In a language that supports user defined types (like Pascal), we could define the variable month as an enumerated type {Jan., Feb., . . . , Dec.}. Either way, the values for min, min+, nom, max- and max are clear from the context. When a variable has discrete, bounded values, as the variables in the commission problem have, the min, min+, nom, max- and max are also easily determined. When there are no explicit bounds, as in the triangle problem, we usually have to create "artificial" bounds. The lower bound of side lengths is clearly 1 (a negative side length is silly), but what might we do for an upper bound? By default, the largest representable integer (called MAXINT in some languages) is one possibility, or we might impose an arbitrary upper limit such as 200 or 2000.

Boundary value analysis doesn't make much sense for Boolean variables; the extreme values are TRUE and FALSE, but there is no clear choice for the remaining three. We will see in Chapter 7 that Boolean variables lend themselves to decision table-based testing. Logical variables also present a problem for boundary value analysis. In the ATM example, a customer's Personal Identification Number (PIN) is a logical variable, as is the transaction type (deposit, withdrawal, or inquiry). We could "go through the motions" of boundary value analysis testing for such variables, but the exercise is not very satisfying to the "tester's intuition".

2.1.2 Limitations of Boundary Value Analysis

Boundary value analysis works well when the program to be tested is a function of several independent variables that represent bounded physical quantities. The key words here are *independent* and *physical quantities*. A quick look at the boundary value analysis test cases for NextDate (in section 5.5) shows them to be inadequate. There is very little stress on February and on leap years, for example. The real problem here is that there are interesting dependencies among the month, day, and year variables. Boundary value analysis presumes the variables to be truly independent. Even so, boundary value analysis happens to catch end-of-month and end-of-year faults. Boundary value analysis test cases are derived from the extrema of bounded, independent variables that refer to physical quantities, with no consideration of the nature of the function, nor of the semantic meaning of the variables. We see boundary value analysis test cases to be rudimentary, in the sense that they are obtained with very little insight and imagination. As with so many things, you get what you pay for.

The physical quantity criterion is equally important. When a variable refers to a physical quantity, such as temperature, pressure, air speed, angle of attack, load, and so forth, physical boundaries can be extremely important. (In an interesting example of this, Sky Harbor International Airport in Phoenix had to close on June 26, 1992 because the air temperature was 122 °F. Aircraft pilots were unable to make certain instrument settings before take-off: the instruments could only accept a maximum air temperature of 120 °F.) In another case, a medical analysis system uses stepper motors to position a carousel of samples to be analyzed. It turns out that the mechanics of moving the carousel back to the starting cell often causes the robot arm to miss the first cell. As an example of logical (versus physical) variables, we might look at PINs or telephone numbers. It's hard to imagine what faults might be revealed by PINs of 0000, 0001, 5000, 9998, and 9999.

2.2 Robustness Testing

Robustness testing is a simple extension of boundary value analysis: in addition to the five boundary value analysis values of a variable, we see what happens when the extrema are exceeded with a value slightly greater than the maximum (max+) and a value slightly less than the minimum (min-). Robustness test cases for our continuing example are shown in Figure 5.3.

Most of the discussion of boundary value analysis applies directly to robustness testing, especially the generalizations and limitations. The most interesting part of robustness testing is not with the inputs, but with the expected outputs. What happens when a physical quantity exceeds its maximum? If it is the angle of attack of an airplane wing, the aircraft might stall. If it's the load capacity of a public elevator, we hope nothing special would happen. If it's a date, like May 32, we

would expect an error message. The main value of robustness testing is that it forces attention on exception handling. With strongly typed languages, robustness testing may be very awkward. In Pascal, for example, if a variable is defined to be within a certain range, values outside that range result in run-time errors that abort normal execution. This raises an interesting question of implementation philosophy: is it better to perform explicit range checking and use exception handling to deal with "robust values", or is it better to stay with strong typing? The exception handling choice mandates robustness testing.

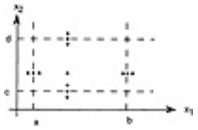


Figure 2.3 Robustness Test Cases for a Function of Two Variables

2.3 Worst Case Testing

Boundary value analysis, as we said earlier, makes the single fault assumption of reliability theory. Rejecting this assumption means that we are interested in what happens when more than one variable has an extreme value. In electronic circuit analysis, this is called "worst case analysis"; we use that idea here to generate worst case test cases. For each variable, we start with the five element set that contains the min, min+, nom, max- and max values. We then take the Cartesian product (see Chapter 3) of these sets to generate test cases. The result of the two-variable version of this is shown in Figure 2.4. Worst case testing is clearly more thorough in the sense that boundary value analysis test cases are a proper subset of worst case test cases. It also represents much more effort: worst case testing for a function of n variables generates 5^n test cases, as opposed to 4n+1 test cases for boundary value analysis.

Worst case testing follows the generalization pattern we saw for boundary value analysis. It also has the same limitations, particularly those related to independence. Probably the best application for worst case testing is where physical variables have numerous interactions, and where failure of the function is extremely costly. For really paranoid testing, we could go to robust worst case testing. This involves the Cartesian product of the seven element sets we used in robustness testing. Figure 2.5 shows the robust worst case test cases for our two variable function.

2.4 Special Value Testing

Special value testing is probably the most widely practiced form of functional testing. It also is the most intuitive and the least uniform. Special value testing occurs when a tester uses his/her domain knowledge, experience with similar programs, and information about "soft spots" to devise test cases. We might also call this "ad hoc testing" or "seat of the pants/skirt" testing. There are no guidelines, other than to use "best engineering judgment." As a result, special value testing is very dependent on the abilities of the tester.

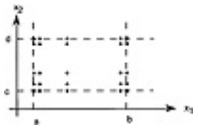


Figure 2.4 Worst Case Test Cases for a Function of Two Variables

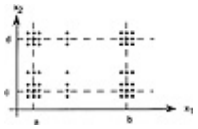


Figure 2.5 Robust Worst Case Test Cases for a Function of Two Variables

Despite all the apparent negatives, special value testing can be very useful. In the next section, you will find test cases generated by the methods we just discussed for our three unit level examples (not the ATM system). If you look carefully at these, especially for the NextDate function, you find that none is very satisfactory. If an interested tester defined special value test cases for NextDate, we would see several test cases involving February 28, February 29, and leap years. Even though special value testing is highly subjective, it often results in a set of test cases which is more effective in revealing faults than the test sets generated by the other methods we have studied—testimony to the craft of software testing.

2.5 Examples

Each of the three continuing examples is a function of three variables. Printing all the test cases from all the methods for each problem is very space consuming, so we'll just have selected examples.

2.5.1 Test Cases for the Triangle Problem

In the problem statement, there are no conditions on the triangle sides, other than being integers. Obviously, the lower bounds of the ranges are all 1. We arbitrarily take 200 as an upper bound. Table 1 contains boundary value test cases and Table 2 contains worst case test cases using these ranges.

Table 1 Boundary Value Analysis Test Cases **Case**

	a	**b**	**c**	**Expected Output**
1	100	100	1	Isosceles
2	100	100	2	Isosceles
3	100	100	100	Equilateral
4	100	100	199	Isosceles
5	100	100	200	Not a Triangle
6	100	1	100	Isosceles
7	100	2	100	Isosceles

8	100	100	100	Equilateral
9	100	199	100	Isosceles
10	100	200	100	Not a Triangle
11	1	100	100	Isosceles
12	2	100	100	Isosceles
13	100	100	100	Equilateral
14	199	100	100	Isosceles
15	200	100	100	Not a Triangle

Table 2 Worst Case Test Cases **Case**

	a	**b**	**c**	**Expected Output**
1	1	1	1	Equilateral
2	1	1	2	Not a Triangle
3	1	1	100	Not a Triangle
4	1	1	199	Not a Triangle
5	1	1	200	Not a Triangle
6	1	2	1	Not a Triangle
7	1	2	2	Isosceles
8	1	2	100	Not a Triangle
9	1	2	199	Not a Triangle
10	1	2	200	Not a Triangle
11	1	100	1	Not a Triangle
12	1	100	2	Not a Triangle
13	1	100	100	Isosceles
14	1	100	199	Not a Triangle

15	1	100	200	Not a Triangle
16	1	199	1	Not a Triangle
17	1	199	2	Not a Triangle
18	1	199	100	Not a Triangle
19	1	199	199	Isosceles
20	1	199	200	Not a Triangle
21	1	200	1	Not a Triangle
22	1	200	2	Not a Triangle
23	1	200	100	Not a Triangle
24	1	200	199	Not a Triangle
25	1	200	200	Isosceles
26	2	1	1	Not a Triangle
27	2	1	2	Isosceles
28	2	1	100	Not a Triangle
29	2	1	199	Not a Triangle
30	2	1	200	Not a Triangle
31	2	2	1	Isosceles
32	2	2	2	Equilateral
33	2	2	100	Not a Triangle
34	2	2	199	Not a Triangle
35	2	2	200	Not a Triangle
36	2	100	1	Not a Triangle
37	2	100	2	Not a Triangle
38	2	100	100	Isosceles

39	2	100	199	Not a Triangle
40	2	100	200	Not a Triangle
41	2	199	1	Not a Triangle
42	2	199	2	Not a Triangle
43	2	199	100	Not a Triangle
44	2	199	199	Isosceles
45	2	199	200	Scalene
46	2	200	1	Not a Triangle
47	2	200	2	Not a Triangle
48	2	200	100	Not a Triangle
49	2	200	199	Scalene
50	2	200	200	Isosceles
51	100	1	1	Not a Triangle
52	100	1	2	Not a Triangle
53	100	1	100	Isosceles
54	100	1	199	Not a Triangle
55	100	1	200	Not a Triangle
56	100	2	1	Not a Triangle
57	100	2	2	Not a Triangle
58	100	2	100	Isosceles
59	100	2	199	Not a Triangle
60	100	2	200	Not a Triangle
61	100	100	1	Isosceles
62	100	100	2	Isosceles

63	100	100	100	Equilateral
64	100	100	199	Isosceles
65	100	100	200	Not a Triangle
66	100	199	1	Not a Triangle
67	100	199	2	Not a Triangle
68	100	199	100	Isosceles
69	100	199	199	Isosceles
70	100	199	200	Scalene
71	100	200	1	Not a Triangle
72	100	200	2	Not a Triangle
73	100	200	100	Not a Triangle
74	100	200	199	Scalene
75	100	200	200	Isosceles
76	199	1	1	Not a Triangle
77	199	1	2	Not a Triangle
78	199	1	100	Not a Triangle
79	199	1	199	Scalene
80	199	1	200	Not a Triangle
81	199	2	1	Not a Triangle
82	199	2	2	Not a Triangle
83	199	2	100	Not a Triangle
84	199	2	199	Isosceles
85	199	2	200	Scalene
86	199	100	1	Not a Triangle

87	199	100	2	Not a Triangle
88	199	100	100	Isosceles
89	199	100	199	Isosceles
90	199	100	200	Scalene
91	199	199	1	Isosceles
92	199	199	2	Isosceles
93	199	199	100	Isosceles
94	199	199	199	Equilateral
95	199	199	200	Isosceles
96	199	200	1	Not a Triangle
97	199	200	2	Scalene
98	199	200	100	Scalene
99	199	200	199	Isosceles
100	199	200	200	Isosceles
101	200	1	1	Not a Triangle
102	200	1	2	Not a Triangle
103	200	1	100	Not a Triangle
104	200	1	199	Not a Triangle
105	200	1	200	Isosceles
106	200	2	1	Not a Triangle
107	200	2	2	Not a Triangle
108	200	2	100	Not a Triangle
109	200	2	199	Scalene
110	200	2	200	Isosceles

111	200	100	1	Not a Triangle
112	200	100	2	Not a Triangle
113	200	100	100	Not a Triangle
114	200	100	199	Scalene
115	200	100	200	Isosceles
116	200	199	1	Not a Triangle
117	200	199	2	Scalene
118	200	199	100	Scalene
119	200	199	199	Isosceles
120	200	199	200	Isosceles
121	200	200	1	Isosceles
122	200	200	2	Isosceles
123	200	200	100	Isosceles
124	200	200	199	Isosceles
125	200	200	200	Equilateral

2.5.2 Test Cases for the NextDate Problem

Table 3 Worst Case Test Cases **Case**

	month	**day**	**year**	**expected output**
1	1	1	1812	January 2, 1812
2	1	1	1813	January 2, 1813
3	1	1	1912	January 2, 1912
4	1	1	2011	January 2, 2011
5	1	1	2012	January 2, 2012

6	1	2	1812	January 3, 1812
7	1	2	1813	January 3, 1813
8	1	2	1912	January 3, 1912
9	1	2	2011	January 3, 2011
10	1	2	2012	January 3, 2012
11	1	15	1812	January 16, 1812
12	1	15	1813	January 16, 1813
13	1	15	1912	January 16, 1912
14	1	15	2011	January 16, 2011
15	1	15	2012	January 16, 2012
16	1	30	1812	January 31, 1812
17	1	30	1813	January 31, 1813
18	1	30	1912	January 31, 1912
19	1	30	2011	January 31, 2011
20	1	30	2012	January 31, 2012
21	1	31	1812	February 1, 1812
22	1	31	1813	February 1, 1813
23	1	31	1912	February 1, 1912
24	1	31	2011	February 1, 2011
25	1	31	2012	February 1, 2012
26	2	1	1812	February 2, 1812
27	2	1	1813	February 2, 1813
28	2	1	1912	February 2, 1912
29	2	1	2011	February 2, 2011

30	2	1	2012	February 2, 2012
31	2	2	1812	February 3, 1812
32	2	2	1813	February 3, 1813
33	2	2	1912	February 3, 1912
34	2	2	2011	February 3, 2011
35	2	2	2012	February 3, 2012
36	2	15	1812	February 16, 1812
37	2	15	1813	February 16, 1813
38	2	15	1912	February 16, 1912
39	2	15	2011	February 16, 2011
40	2	15	2012	February 16, 2012
41	2	30	1812	error
42	2	30	1813	error
43	2	30	1912	error
44	2	30	2011	error
45	2	30	2012	error
46	2	31	1812	error
47	2	31	1813	error
48	2	31	1912	error
49	2	31	2011	error
50	2	31	2012	error
51	6	1	1812	June 1, 1812
52	6	1	1813	June 1, 1813
53	6	1	1912	June 2, 1912

54	6	1	2011	June 2, 2011
55	6	1	2012	June 2, 2012
56	6	2	1812	June 3, 1812
57	6	2	1813	June 3, 1813
58	6	2	1912	June 3, 1912
59	6	2	2011	June 3, 2011
60	6	2	2012	June 3, 2012
61	6	15	1812	June 16, 1812
62	6	15	1813	June 16, 1813
63	6	15	1912	June 16, 1912
64	6	15	2011	June 16, 2011
65	6	15	2012	June 16, 2012
66	6	30	1812	July 31, 1812
67	6	30	1813	July 31, 1813
68	6	30	1912	July 31, 1912
69	6	30	2011	July 31, 2011
70	6	30	2012	July 31, 2012
71	6	31	1812	error
72	6	31	1813	error
73	6	31	1912	error
74	6	31	2011	error
75	6	31	2012	error
76	11	1	1812	November 2, 1812
77	11	1	1813	November 2, 1813

78	11	1	1912	November 2, 1912
79	11	1	2011	November 2, 2011
80	11	1	2012	November 2, 2012
81	11	2	1812	November 3, 1812
82	11	2	1813	November 3, 1813
83	11	2	1912	November 3, 1912
84	11	2	2011	November 3, 2011
85	11	2	2012	November 3, 2012
86	11	15	1812	November 16, 1812
87	11	15	1813	November 16, 1813
88	11	15	1912	November 16, 1912
89	11	15	2011	November 16, 2011
90	11	15	2012	November 16, 2012
91	11	30	1812	December 1, 1812
92	11	30	1813	December 1, 1813
93	11	30	1912	December 1, 1912
94	11	30	2011	December 1, 2011
95	11	30	2012	December 1, 2012
96	11	31	1812	error
97	11	31	1813	error
98	11	31	1912	error
99	11	31	2011	error
100	11	31	2012	error
101	12	1	1812	December 2, 1812

102	12	1	1813	December 2, 1813
103	12	1	1912	December 2, 1912
104	12	1	2011	December 2, 2011
105	12	1	2012	December 2, 2012
106	12	2	1812	December 3, 1812
107	12	2	1813	December 3, 1813
108	12	2	1912	December 3, 1912
109	12	2	2011	December 3, 2011
110	12	2	2012	December 3, 2012
111	12	15	1812	December 16, 1812
112	12	15	1813	December 16, 1813
113	12	15	1912	December 16, 1912
114	12	15	2011	December 16, 2011
115	12	15	2012	December 16, 2012
116	12	30	1812	December 31, 1812
117	12	30	1813	December 31, 1813
118	12	30	1912	December 31, 1912
119	12	30	2011	December 31, 2011
120	12	30	2012	December 31, 2012
121	12	31	1812	January 1, 1813
122	12	31	1813	January 1, 1814
123	12	31	1912	January 1, 1913
124	12	31	2011	January 1, 2012
125	12	31	2012	January 1, 2013

2.5.3 Test Cases for the Commission Problem

Rather than go through 125 boring test cases again, we'll look at some more interesting test cases for the commission problem. This time we'll look at boundary values for the output range, especially near the threshold points of $1000 and $1800. The output space of the commission is shown in Figure 5.6. The intercepts of these threshold planes with the axes are shown. The volume below the lower plane corresponds to sales below the $1000 threshold. The volume between the two planes is the 15% commission range. Part of the reason for using the output range to determine test cases is that cases from the input range are almost all in the 20% zone. We want to find input variable combinations that stress the boundary values: $100, $1000, $1800, and $7800. These test cases were developed with a spreadsheet, which saves a lot of calculator pecking. The minimum and maximum were easy, and the numbers happen to work out so that the border points are easy to generate. Here's where it gets interesting. Test case 9 is the $1000 border point. If we tweak the input variables we get values just below and just above the border (cases 6 - 8 and 10 - 12). If we wanted to, we could pick values near the intercepts like (22, 1, 1) and (21, 1, 1). As we continue in this way, we have a sense that we are "exercising" interesting parts of the code. We might claim that this is really a form of special value testing, because we used our mathematical insight to generate test cases.

2.6 Guidelines for Boundary value Testing

With the exception of special value testing, the test methods based on the boundary values of a function (program) are the most rudimentary of all functional testing methods. They share the common assumption that the input variables are truly independent, and when this assumption is not warranted, the methods generate unsatisfactory test cases (such as February 31, 1912 for NextDate). These methods have two other distinctions: normal versus robust values, and the single fault versus the multiple fault assumption. Just using these distinctions carefully will result in better testing. Each of these methods can be applied to the output range of a program, as we did for the commission problem.

Another useful form of output-based test cases is for systems that generate error messages. The tester should devise test cases to check that error messages are generated when they are appropriate, and are not falsely generated. Boundary value analysis can also be used for internal variables, such as loop control variables, indices, and pointers. Strictly speaking, these are not input variables, but errors in the use of these variables are quite common. Robustness testing is a good choice for testing internal variables.

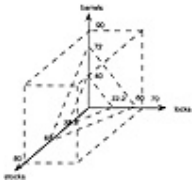


Figure 2.6 Input Space of the Commission Problem

Table 4 Output Boundary Value Analysis Test Cases **Case**

	locks	**stocks**	**barrels**	**sales**	**comm**	**Comment**
1	1	1	1	100	10	output minimum
2	1	1	2	125	12.5	output minimum +
3	1	2	1	130	13	output minimum +
4	2	1	1	145	14.5	output minimum +
5	5	5	5	500	50	midpoint
6	10	10	9	975	97.5	border point -
7	10	9	10	970	97	border point -
8	9	10	10	955	95.5	border point -
9	10	10	10	1000	100	border point
10	10	10	11	1025	103.75	border point +
11	10	11	10	1030	104.5	border point +
12	11	10	10	1045	106.75	border point +
13	14	14	14	1400	160	midpoint
14	18	18	17	1775	216.25	border point -
15	18	17	18	1770	215.5	border point -
16	17	18	18	1755	213.25	border point -
17	18	18	18	1800	220	border point
18	18	18	19	1825	225	border point +
19	18	19	18	1830	226	border point +
20	19	18	18	1845	229	border point +
21	48	48	48	4800	820	midpoint
22	70	80	89	7775	1415	output maximum -

23	70	79	90	7770	1414	output maximum -
24	69	80	90	7755	1411	output maximum -
25	70	80	90	7800	1420	output maximum

Table 5 Output Special Value Test Cases **Case**

	locks	**stocks**	**barrels**	**sales**	**comm**	**Comment**
1	10	11	9	1005	100.75	border point +
2	18	17	19	1795	219.25	border point -
3	18	19	17	1805	221	border point +

2.7 Equivalence Classes

We noted that the important aspect of equivalence classes is that they form a partition of a set, where partition refers to a collection of mutually disjoint subsets whose union is the entire set. This has two important implications for testing: the fact that the entire set is represented provides a form of completeness, and the disjointness assures a form of non-redundancy. Because the subsets are determined by an equivalence relation, the elements of a subset have something in common. The idea of equivalence class testing is to identify test cases by using one element from each equivalence class. If the equivalence classes are chosen wisely, this greatly reduces the potential redundancy among test cases. In the Triangle Problem, for example, we would certainly have a test case for an equilateral triangle, and we might pick the triple (5, 5, 5) as inputs for a test case. If we did this, we would not expect to learn much from test cases such as (6, 6, 6) and (100, 100, 100). Our intuition tells us that these would be "treated the same" as the first test case, thus they would be redundant. When we consider structural testing in Part III, we shall see that "treated the same" maps onto "traversing the same execution path".

The key (and the craft!) of equivalence class testing is the choice of the equivalence relation that determines the classes. Very often, we make this choice by "second guessing" the likely implementation, and thinking about the functional manipulations that must somehow be present in the implementation. We will illustrate this with our continuing examples, but first, we need to make a distinction between weak and strong equivalence class testing. After that, we will compare these to the traditional form of equivalence class testing.

Suppose our program is a function of three variables, a, b, and c, and the input domain consists of sets A, B, and C. Now, suppose we choose an "appropriate" equivalence relation, which induces the following partition:

$A = A1 \cup A2 \cup A3$
$B = B1 \cup B2 \cup B3 \cup B4$
$C = C1 \cup C2$

Finally, we denote elements of the partitions as follows:

$$a1 \in A1$$
$$b3 \in B3$$
$$c2 \in C2$$

2.7.1 Weak Equivalence Class Testing

With the notation as given above, weak equivalence class testing is accomplished by using one variable from each equivalence class in a test case. For the above example, we would end up with the following weak equivalence class test cases:

Test Case	**a**	**b**	**c**
WE1	a1	b1	c1
WE2	a2	b2	c2
WE3	a3	b3	c3
WE4	a1	b4	c2

This set of test cases uses one value from each equivalence class. We identify these in a systematic way, hence the apparent pattern. In fact, we will always have the same number of weak equivalence class test cases as there are classes in the partition with the largest number of subsets.

2.7.2 Strong Equivalence Class Testing

Strong equivalence class testing is based on the Cartesian Product of the partition subsets. Continuing with this example, the Cartesian Product A x B x C will have $3 \times 4 \times 2 = 24$ elements, resulting in the test cases in the table below:

Test Case	**a**	**b**	**c**
SE1	a1	b1	c1
SE2	a1	b1	c2
SE3	a1	b2	c1
SE4	a1	b2	c2
SE5	a1	b3	c1
SE6	a1	b3	c2

SE7	a1	b4	c1
SE8	a1	b4	c2
SE9	a2	b1	c1
SE10	a2	b1	c2
SE11	a2	b2	c2
SE12	a2	b2	c2
SE13	a2	b3	c1
SE14	a2	b3	c2
SE15	a2	b4	c1
SE16	a2	b4	c2
SE17	a3	b1	c1
SE18	a3	b1	c2
SE19	a3	b2	c1
SE20	a3	b2	c2
SE21	a3	b3	c1
SE22	a3	b3	c2
SE23	a3	b4	c1
SE24	a3	b4	c2

Notice the similarity between the pattern of these test cases and the construction of a truth table in propositional logic. The Cartesian Product guarantees that we have a notion of "completeness" in two senses: we cover all the equivalence classes, and we have one of each possible combination of inputs.

As we shall see from our continuing examples, the key to "good" equivalence class testing is the selection of the equivalence relation. Watch for the notion of inputs being "treated the same". Most of the time, equivalence class testing defines classes of the input domain. There is no reason why we could not define equivalence relations on the output range of the program function being tested, in fact, this is the simplest approach for the Triangle Problem.

2.7.3 Traditional Equivalence Class Testing

The traditional view of equivalence class testing defines equivalence classes in terms of validity. For each input variable, there are valid and invalid values; in the traditional approach, these are identified and numbered, and then incorporated into test cases in the weak sense. As a brief example, consider the valid ranges defined for the variables in the Commission problem:

$$1 \leq lock \leq 70$$
$$1 \leq stock \leq 80$$
$$1 \leq barrel \leq 90$$

The corresponding invalid ranges are:

$$lock < 1$$
$$lock > 70$$
$$stock < 1$$
$$stock > 80$$
$$barrel < 1$$
$$barrel > 90$$

Given these valid and invalid sets of inputs, the traditional equivalence testing strategy identifies test cases as follows:

1. For valid inputs, use one value from each valid class (as in what we have called weak equivalence class testing. (Note that each input in these test cases will be valid.)

2. For invalid inputs, a test case will have one invalid value and the remaining values will all be valid. (Thus a "single failure" should cause the test case to fail.)

If the input variables have defined ranges (as we had in the various boundary value examples), then the test cases from traditional equivalence class testing will always be a subset of those that would be generated by robustness testing.

There are two problems with traditional equivalence testing. The first is that, very often, the specification does not define what the expected output for an invalid test case should be. (We could argue that this is a deficiency of the specification, but that doesn't get us anywhere.) Thus testers spend a lot of time defining expected outputs for these cases. The second problem is that strongly typed languages eliminate the need for the consideration of invalid inputs. Traditional equivalence testing is a product of the time when languages such as FORTRAN and COBOL were dominant, hence this type of error was common. In fact, it was the high incidence of such errors that led to the implementation of strongly typed languages.

2.8 Equivalence Class Test Cases for the Triangle Problem

In the problem statement, we note that there are four possible outputs: Not a Triangle, Scalene, Isosceles, and Equilateral. We can use these to identify output (range) equivalence classes as follows.

R1 = {<a, b, c> : the triangle with sides a, b, and c is equilateral}
R2 = {<a, b, c> : the triangle with sides a, b, and c is isosceles}
R3 = {<a, b, c> : the triangle with sides a, b, and c is scalene}
R4 = {<a, b, c> : sides a, b, and c do not form a triangle}

These classes yield a simple set of test cases:

Test Case	**a**	**b**	**c**	**Expected Output**
OE1	5	5	5	Equilateral
OE2	2	2	3	Isosceles
OE3	3	4	5	Scalene
OE4	4	1	2	Not a Triangle

If we base equivalence classes on the input domain, we obtain a richer set of test cases. What are some of the possibilities for the three integers, a, b, and c? They can all be equal, exactly one pair can be equal (this can happen three ways), or none can be equal.

D1 = {<a, b, c> : a = b = c}
D2 = {<a, b, c> : a = b, a ≠ c}
D3 = {<a, b, c> : a = c, a ≠ b}
D4 = {<a, b, c> : b = c, a ≠ b}
D5 = {<a, b, c> : a ≠ b, a ≠ c, b ≠ c}

As a separate question, we can apply the triangle property to see if they even constitute a triangle. (For example, the triplet <1, 4, 1> has exactly one pair of equal sides, but these sides do not form a triangle.)

D6 = {<a, b, c> : a ≥ b + c}
D7 = {<a, b, c> : b ≥ a + c}
D8 = {<a, b, c> : c ≥ a + b}

If we wanted to be still more thorough, we could separate the “less than or equal to” into the two distinct cases, thus the set D6 would become

D6' = {<a, b, c> : a = b + c}
D6" = {<a, b, c> : a > b + c}

and similarly for D7 and D8.

We need to make one more observation. Notice that we have not taken any form of Cartesian Product. This is because all of our equivalence classes were already defined in terms of triplets <a,

b, c>. Had we followed the dictates of traditional equivalence testing, we would have equivalence classes of integers greater than zero and integers less than or equal to zero. There would lead to a boring, unsatisfactory set of test cases.

2.9 Equivalence Class Test Cases for the NextDate Function

The NextDate Function illustrates very well the craft of choosing the underlying equivalence relation. It is also a good example on which to compare traditional, weak, and strong forms of equivalence class testing. NextDate is a function of three variables, month, day, and year, and these have ranges defined as follows:

$1 \leq month \leq 12$
$1 \leq day \leq 31$
$1812 \leq year \leq 2012$

Traditional Test Cases

The valid equivalence classes are

$M1 = \{ month : 1 \leq month \leq 12 \}$
$D1 = \{ day : 1 \leq day \leq 31 \}$
$Y1 = \{ year : 1812 \leq year \leq 2012 \}$

The invalid equivalence classes are

$M2 = \{ month : month < 1 \}$
$M3 = \{ month : month > 12 \}$
$D2 = \{ day : day < 1 \}$
$D3 = \{ day : day > 31 \}$
$Y2 = \{ year : year < 1812 \}$
$Y3 = \{ year : year > 2012 \}$

These classes yield the following test cases, where the valid inputs are mechanically selected from the approximate middle of the valid range:

Case ID	Month	Day	Year	Expected Output
TE1	6	15	1912	6/16/1912
TE2	-1	15	1912	invalid input
TE3	13	15	1912	invalid input
TE4	6	-1	1912	invalid input

TE5	6	32	1912	invalid input
TE6	6	15	1811	invalid input
TE7	6	15	2013	invalid input

If we more carefully choose the equivalence relation, the resulting equivalence classes will be more useful. Recall we said earlier that the gist of the equivalence relation is that elements in a class are "treated the same way". One way to see the deficiency of the traditional approach is that the "treatment" is at the valid/invalid level. We next reduce the granularity by focusing on more specific treatment.

What must be done to an input date? If it is not the last day of a month, the NextDate function will simply increment the day value. At the end of a month, the next day is 1 and the month is incremented. At the end of a year, both the day and the month are reset to 1, and the year is incremented. Finally, the problem of leap year makes determining the last day of a month interesting. With all this in mind, we might postulate the following equivalence classes:

M1 = { month: month has 30 days}
M2 = { month: month has 31 days}
M3 = { month: month is February}
D1 = {day: $1 \leq day \leq 28$}
D2 = {day: day = 29 }
D3 = {day: day = 30 }
D4 = {day: day = 31 }
Y1 = {year: year = 1900}
Y2 = {year: $1812 \leq year \leq 2012$ AND $(year \neq 1900)$
AND (year = 0 mod 4)}
Y3 = {year : ($1812 \leq year \leq 2012$ AND $year \neq 0$ mod 4)}

By choosing separate classes for 30 and 31 day months, we simplify the last day of the month question. By taking February as a separate class, we can give more attention to leap year questions. We also give special attention to day values: days in D 1 are (nearly) always incremented, while days in D4 only have meaning for months in M2. Finally, we have three classes of years, the special case of the year 1900, leap years, and common years. This isn't a perfect set of equivalence classes, but its use will reveal many potential errors.

Weak Equivalence Class Test Cases

These classes yield the following weak equivalence class test cases. As before, the inputs are mechanically selected from the approximate middle of the corresponding class:

Case ID	Month	Day	Year	Expected Output
WE1	6	14	1900	6/15/1900
WE2	7	29	1912	7/30/1912
WE3	2	30	1913	invalid input
WE4	6	31	1900	invalid input

Notice that following a simple pattern yields test cases that are not particularly reassuring. This is worth remembering, because any tool that attempts to automate this process will rely on some fairly simple value selection criterion.

Strong Equivalence Class Test Cases

Using the same equivalence classes, we find the strong equivalence class test cases shown in the table below. The same value selection criterion is used. We still don't have a "perfect" set of test cases, but I think any tester would be a lot happier with the 36 strong equivalence class test cases. To illustrate the sensitivity to the choice of classes, notice that, among these 36 test cases, we never get a Feb. 28. If we had chosen five day classes, where D1' would be days 1 - 27, D1" would be 28, and the other three would stay the same. We would have a set of 45 test cases, and among these, there would be better coverage of Feb. 28 considerations.

Case ID	Month	Day	Year	Expected Output
SE1	6	14	1900	6/15/1900
SE2	6	14	1912	6/15/1912
SE3	6	14	1913	6/15/1913
SE4	6	29	1900	6/30/1900
SE5	6	29	1912	6/30/1912
SE6	6	29	1913	6/30/1913
SE7	6	30	1900	7/1/1900
SE8	6	30	1912	7/1/1912
SE9	6	30	1913	7/1/1913

SE10	6	31	1900	ERROR
SE11	6	31	1912	ERROR
SE12	6	31	1913	ERROR
SE13	7	14	1900	7/15/1900
SE14	7	14	1912	7/15/1912
SE15	7	14	1913	7/15/1913
SE16	7	29	1900	7/30/1900
SE17	7	29	1912	7/30/1912
SE18	7	29	1913	7/30/1913
SE19	7	30	1900	7/31/1900
SE20	7	30	1912	7/31/1912
SE21	7	30	1913	/31/1913
SE22	7	31	1900	8/1/1900
SE23	7	31	1912	8/1/1912
SE24	7	31	1913	8/1/1913
SE25	2	14	1900	2/15/1900
SE26	2	14	1912	2/15/1912
SE27	2	14	1913	2/15/1913
SE28	2	29	900	ERROR
SE29	2	29	1912	3/1/1912
SE30	2	29	1913	ERROR
SE31	2	30	1900	ERROR
SE32	2	30	1912	ERROR
SE33	2	30	1913	ERROR

SE34	2	31	1900	ERROR
SE35	2	31	1912	ERROR
SE36	2	31	1913	ERROR

We could also streamline our set of test cases by taking a closer look at the year classes. If we merge Y1 and Y3, and call the result the set of common years, our 36 test cases would drop down to 24. This change suppresses special attention to considerations in the year 1900, and it also adds some complexity to the determination of which years are leap years. Balance this against how much might be learned from the present test cases. Take a look at the test cases in which Yl is used (SE2, SE4, SE7, ...). We don't really learn much from these, so not much would be lost by skipping them. The only thing really interesting about the year 1900 is test case SE28, and a related test case that was not generated by the set of equivalence classes, Feb. 28, 1900.

2.9 Equivalence Class Test Cases for the Commission Problem

The Input domain of the Commission Problem is "naturally" partitioned by the limits on locks, stocks, and barrels. These equivalence classes are exactly those that would also be identified by traditional equivalence class testing. The first class is the valid input, the other two are invalid. The input domain equivalence classes lead to very unsatisfactory sets of test cases. We'll do a little better with equivalence classes defined on the output range of the commission function.

Variable Input Domain Equivalence Classes

Variable	Input Domain Equivalence Classes
Lock	L1 = { lock: $1 \leq lock \leq 70$ }
	L2 = { lock: $lock < 1$ }
	L3 = {lock: $lock > 70$ }
Stock	S1 = { stock: $1 \leq stock \leq 80$ }
	S2 = { stock: $stock < 1$ }
	S3 = { stock: $stock > 80$ }
Barrel	B1 = { barrel: $1 \leq barrel \leq 90$ }
	B2 = { barrel: $barrel < 1$ }
	B3 = { barrel: $barrel > 90$ }

Weak Input Domain Equivalence Class Test Cases

These classes yield the following weak equivalence class test cases. As with the other examples, the inputs are mechanically selected.

Test Case	locks	stocks	barrels	sales	commission
WE1	35	40	45	500	50
WE2	0	0	0	ERROR	ERROR
WE3	71	81	91	ERROR	ERROR

Traditional Input Domain Equivalence Class Test Cases

The same classes yield the following traditional equivalence class test cases. As with the other examples, the inputs are mechanically selected.

Test Case	locks	stocks	barrels	sales	commission
SE1	35	40	45	500	50
SE2	35	40	0	ERROR	ERROR
SE3	35	40	91	ERROR	ERROR
SE4	35	0	45	ERROR	ERROR
SE5	35	0	0	ERROR	ERROR
SE6	35	0	91	ERROR	ERROR
SE7	35	81	45	ERROR	ERROR
SE8	35	81	0	ERROR	ERROR
SE9	35	81	91	ERROR	ERROR
SE10	0	40	45	ERROR	ERROR
SE11	0	40	0	ERROR	ERROR
SE12	0	40	91	ERROR	ERROR
SE13	0	0	45	ERROR	ERROR

SE14	0	0	0	ERROR	ERROR
SE15	0	0	91	ERROR	ERROR
SE16	0	81	45	ERROR	ERROR
SE17	0	81	0	ERROR	ERROR
SE18	0	81	81	ERROR	ERROR
SE19	71	40	45	ERROR	ERROR
SE20	71	40	0	ERROR	ERROR
SE21	71	40	91	ERROR	ERROR
SE22	71	0	45	ERROR	ERROR
SE23	71	0	0	ERROR	ERROR
SE24	71	0	91	ERROR	ERROR
SE25	71	81	45	ERROR	ERROR
SE26	71	81	0	ERROR	ERROR
SE27	71	81	91	ERROR	ERROR

Output Range Equivalence Class Test Cases

Notice that, of 27 test cases, only one is a legitimate input. If we were really worried about error cases, this might be a good set of test cases. It can hardly give us a sense of confidence about the calculation portion of the problem, however. We can get some help by considering equivalence classes defined on the output range. Recall that sales is a function of the number of locks, stocks and barrels sold:

sales = 45 x locks + 30 x stocks + 25 x barrels
L1 = { <lock, stock, barrel> : sales < 1000 }
L2 = { <lock, stock, barrel> : 1000 ≤ sales ≤ 1800 }
L3 = { <lock, stock, barrel> : sales > 1800 }

Figure 5.6 helps us get a better "feel" for the input space. Elements of L1 are points with integer coordinates in the pyramid near the origin. Elements of L2 are points in the "triangular slice" between the pyramid and the rest of the input space. Finally, elements of L3 are all those points in the rectangular volume that are not in L1 or in L2. All the error cases found by the strong equivalence classes of the input domain are outside of the rectangular space shown in Figure 5.6.

As was the case with the Triangle Problem, the fact that our input is a triplet means that we no longer take test cases from a Cartesian Product.

Output Range Equivalence Class Test Cases:

Test Case	locks	stocks	barrels	sales	commision
OR1	5	5	5	500	50
Or2	15	15	15	1500	175
OR3	25	25	25	2500	360

These test cases give us some sense that we are exercising important parts of the problem.

2.10 Guidelines and Observations

Now that we have gone through three examples, we conclude with some observations about, and guidelines for equivalence class testing.

1. The traditional form of equivalence class testing is generally not as thorough as weak equivalence class testing, which in turn, is not as thorough as the strong form of equivalence class testing.

2. The only time it makes sense to use the traditional approach is when the implementation language is not strongly typed.

3. If error conditions are a high priority, we could extend strong equivalence class testing to include invalid classes.

4. Equivalence class testing is appropriate when input data is defined in terms of ranges and sets of discrete values. This is certainly the case when system malfunctions can occur for out-of-limit variable values.

5. Equivalence class testing is strengthened by a hybrid approach with boundary value testing. (We can "reuse" the effort made in defining the equivalence classes.)

6. Equivalence class testing is indicated when the program function is complex. In such cases, the complexity of the function can help identify useful equivalence classes, as in the NextDate function.

7. Strong equivalence class testing makes a presumption that the variables are independent when the Cartesian Product is taken. If there are any dependencies, these will often generate "error" test cases, as they did in the NextDate function. (The decision table technique in Chapter 7 resolves this problem.)

8. Several tries may be needed before "the right" equivalence relation is discovered, as we saw in the NextDate example. In other cases, there is an "obvious" or "natural" equivalence relation. When in doubt, the best bet is to try to second guess aspects of any reasonable implementation.

2.11 Decision Table Based Testing

Of all the functional testing methods, those based on decision tables are the most rigorous, because decision tables themselves enforce logical rigor. There are two closely related methods, Cause-Effect graphing [Elmendorf 73], {Myers 79} and the decision tableau method [Mosley 93]. These are more cumbersome to use, and are fully redundant with decision tables, so we will not discuss them here. Both are covered in [Mosley 93].

2.11.1 Decision Tables

Decision tables have been used to represent and analyze complex logical relationships since the early 1960s. They are ideal for describing situations in which a number of combinations of actions are taken under varying sets of conditions. Some of the basic decision table terms are illustrated in Figure 2.11

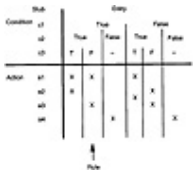


Figure 2.11 Decision Table Terminology

There are four portions of a decision table: the part to the left of the bold vertical line is the stub portion, to the right is the entry portion. The part above the bold line is the condition portion, below is the action portion. Thus we can refer to the condition stub, the condition entries, the action stub, and the action entries. A column in the entry portion is a rule. Rules indicate which actions are taken for the conditional circumstances indicated in the condition portion of the rule. In the decision table in Figure 7.1, when conditions c1, c2, and c3 are all true, actions a1 and a2 occur. When c1 and c2 are both true, and c3 is false, then actions a1 and a3 occur. The entry for c3 in the rule where c1 is true and c2 is false is called a "Don't Care" entry. The Don't Care entry has two major interpretations: the condition is irrelevant, or the condition does not apply. Sometimes people will enter the "n/a" symbol for this latter interpretation.

When we have binary conditions (True/False, Yes/No, 0/1), the condition portion of a decision table is a truth table (from propositional logic) that has been rotated 90°. This structure guarantees that we consider every possible combination of condition values. When we use decision tables for test case identification, this completeness property of a decision table will guarantee a form of complete testing. Decision tables in which all the conditions are binary are called Limited Entry Decision Tables. If conditions are allowed to have several values, the resulting tables are called Extended Entry Decision Tables. We will see examples of both types for the NextDate problem.

Decision tables are somewhat declarative (as opposed to imperative): there is no particular order implied by the conditions, and selected actions do not occur in any particular order.

2.11.2 Technique

To identify test cases with decision tables, we interpret conditions as inputs, and actions as outputs. Sometimes, conditions end up referring to equivalence classes of inputs, and actions refer to major functional processing portions of the item being tested. The rules are then interpreted as test cases. Because the decision table can mechanically be forced to be complete, we know we have a comprehensive set of test cases. There are several techniques that produce decision tables that are more useful to testers. One helpful style is to add an action to show when a rule is logically impossible.

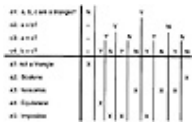


Figure 2.11.2 Decision Table for the Triangle Problem

In the decision table in Figure 2.11.2, we see examples of Don't Care entries and impossible rule usage. If the integers a, b, and c do not constitute a triangle, we don't even care about possible equalities, as indicated in the first rule. In rules 3, 4, and 6, if two pairs of integers are equal, by transitivity, the third pair must be equal, thus the negative entry makes the rule(s) impossible.

The decision table in Figure 2.11.3 illustrates another consideration related to technique: the choice of conditions can greatly expand the size of a decision table. Here, we expanded the old condition (c1: a, b, c are a triangle?) to a more detailed view of the three inequalities of the triangle property. If any one of these fails, the three integers do not constitute sides of a triangle. We could expand this still further, because there are two ways an inequality could fail: one side could equal the sum of the other two, or it could be strictly greater. The rule entry portion of this decision table is as in Figure 7.3, except that each condition entry is explicitly shown. This is a matter of style, not content. I think the style in Figure 7.2 is more readable.

Figure 2.11.3 Refined Decision Table for the Triangle Problem

When conditions refer to equivalence classes, decision tables have a characteristic appearance. Conditions in the decision table in Figure 7.4 are from the NextDate problem; they refer to the mutually exclusive possibilities for the month variable. Since a month is in exactly one equivalence class, we cannot ever have a rule in which two entries are true. The Don't Care entries (—) really mean "must be false". Some decision table aficionados use the notation F! to make this point.

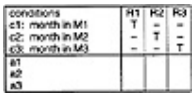


Figure 2.11.4 Decision Table with Mutually Exclusive Conditions

Use of Don't Care entries has a subtle effect on the way in which complete decision tables are recognized. For limited entry decision tables, if there are n conditions, there must be 2^n rules. When Don't Care entries really indicate that the condition is irrelevant, we can develop a rule count as follows. Rules in which no Don't Care entries occur count as one rule. Each Don't Care entry in a rule doubles the count of that rule. The rule counts for the decision table in Figure 7.3 are shown below. Notice that the sum of the rule counts is 64 (as it should be).

Conditions											
c1: a<b+c?	F	T	T	T	T	T	T	T	T	T	T
c2: b<a+c?	-	F	T	T	T	T	T	T	T	T	T
c3: c<a+b?	-	-	F	T	T	T	T	T	T	T	T
c4: a = b?	-	-	-	T	T	T	T	F	F	F	F
c5: a = c?	-	-	-	T	T	F	F	T	T	F	F
c6: b = c?	-	-	-	T	F	T	F	T	F	T	F
Rule Count	32	16	8	1	1	1	1	1	1	1	1
a1: not a triangle	X	X	X								
a2: Scalene											X
a3: Isosceles							X		X	X	
a4: Equilateral				X							
a5: Impossible					X	X		X			

If we applied this simplistic algorithm to the decision table in Figure 7.4, we get the rule counts shown below:

conditions	**R1**	**R2**	**R3**

c1: month in M1	T	--	--
c2: month in M2	--	T	--
c3: month in M3	--	--	T
Rule Count	4	4	4
a1			

Since we should only have eight rules, we clearly have a problem. To see where the problem lies, we expand each of the three rules, replacing the — entries with the T and F possibilities.

conditions	**1.1**	**1.2**	**1.3**	**1.4**	**2.1**	**2.2**	**2.3**	**2.4**	**3.1**	**3.2**	**3.3**	**3.4**
c1: mo. in M1	T	T	T	T	T	T	F	F	T	T	F	F
c2: mo. in M2	T	T	F	F	T	T	T	T	T	F	T	F
c3: mo. in M3	T	F	T	F	T	F	T	F	T	T	T	T
Rule Count	1	1	1	1	1	1	1	1	1	1	1	1
a1												

Notice that we have three rules in which all entries are T: rules 1.1, 2.1, and 3.1. We also have two rules with T, T, F entries: rules 1.2 and 2.2. Similarly, rules 1.3 and 3.2 are identical; so are rules 2.3 and 3.3. If we delete the repetitions, we end up with seven rules; the missing rule is the one in which all conditions are false. The result of this process in shown in Figure 7.5. The impossible rules are also shown.

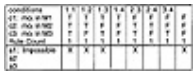


Figure 2.11.5 Mutually Exclusive Conditions with Impossible Rules

The ability to recognize (and develop) complete decision tables puts us in a powerful position with respect to redundancy and inconsistency. The decision table in Figure 7.6 is redundant — there are three conditions and nine rules. (Rule 9 is identical to rule 4.)

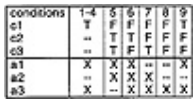


Figure 2.11.6 A Redundant Decision Table

Notice that the action entries in rule 9 are identical to those in rules 1 - 4. As long as the actions in a redundant rule are identical to the corresponding part of the decision table, there isn't much of a problem. If the action entries are different, as they are in Figure 2.11.7, we have a bigger problem.

Figure 2.11.7 An Inconsistent Decision Table

If the decision table in Figure 2.11.7 were to process a transaction in which c1 is true and both c2 and c3 are false, both rules 4 and 9 apply. We can make two observations:

1. Rules 4 and 9 are inconsistent.
2. The decision table is non-deterministic.

Rules 4 and 9 are inconsistent because the action sets are different. The whole table is non-deterministic because there is no way to decide whether to apply rule 4 or rule 9. The bottom line for testers is that care should be taken when Don't Care entries are used in a decision table.

2.11.2 Test Cases for the Triangle Problem

Using the decision table in Figure 2.11.3, we obtain eleven functional test cases: three impossible cases, three ways to fail the triangle property, one way to get an equilateral triangle, one way to get a scalene triangle, and three ways to get an isosceles triangle. If we extended the decision table to show both ways to fail an inequality, we would pick up three more test cases (where one side is exactly the sum of the other two). Some judgment is required in this because of the exponential growth of rules. In this case, we would end up with many more don't care entries, and more impossible rules.

Case ID	**a**	**b**	**c**	**Expected Output**
DT1	4	1	2	Not a Triangle
DT2	1	4	2	Not a Triangle
DT3	1	2	4	Not a Triangle
DT4	5	5	5	Equilateral
DT5	?	?	?	Impossible
DT6	?	?	?	Impossible
DT7	2	2	3	Isosceles

DT8	?	?	?	Impossible
DT9	2	3	2	Isosceles
DT10	3	2	2	Isosceles
DT11	3	4	5	Scalene

2.11.3 Test Cases for the NextDate Function

The NextDate function was chosen because it illustrates the problem of dependencies in the input domain. This makes it a perfect example for decision table based testing, because decision tables can highlight such dependencies. Recall that, in chapter 6, we identified equivalence classes in the input domain of the NextDate function. One of the limitations we found in Chapter 6 was that indiscriminate selection of input values from the equivalence classes resulted in "strange" test cases, such as finding the next date to June 31, 1812. The problem stems from the presumption that the variables are independent. If they are, a Cartesian Product of the classes makes sense. When there are logical dependencies among variables in the input domain, these dependencies are lost (suppressed is better) in a Cartesian Product. The decision table format lets us emphasize such dependencies using the notion of the "impossible action" to denote impossible combinations of conditions. In this section, we will make three tries at a decision table formulation of the NextDate function.

First Try

Identifying appropriate conditions and actions presents an opportunity for craftsmanship. Suppose we start with a set of equivalence classes close to the one we used in Chapter 6.

Variable	Equivalence Classes
Month	M1 = { month : month has 30 days} M2 = { month : month has 31 days} M3 = { month : month is February}
Day	D1 = {day : 1 ≤ day ≤ 28} D2 = {day : day = 29 } D3 = {day : day = 30 } D4 = {day : day = 31 }
Year	Y1 = {year : year is a leap year}

Y2 = {year : year is a common year}

If we wish to highlight impossible combinations, we could make a Limited Entry Decision Table with the following conditions and actions. (Note that the equivalence classes for the year variable collapse into one condition.)

conditions											
c1: month in M1?	T										
c2: month in M2?		T									
c3: month in M3?			T								
c4: day in D1?											
c5: day in D2?											
c6: day in D3?											
c7: day in D4?											
c8: year in Y1?											
a1: impossible											
a2: next date											

This decision table will have 256 rules, many of which will be impossible. If we wanted to show why these rules were impossible, we might revise our actions to the following:

Actions

a1: Too many days in a month

a2: Cannot happen in a common year

a3: Compute the next date

Second Try

If we focus on the leap year aspect of the NextDate function, we could use the set of equivalence classes as they were in Chapter 6. These classes have a Cartesian Product that contains 36 triples, with several being impossible.

To illustrate another decision table technique, this time we'll develop an Extended Entry Decision Table, and we'll take a closer look at the action stub. In making an Extended Entry Decision Table, we must ensure that the equivalence classes form a true partition of the input domain. (Recall from Chapter 3 that a partition is a set of disjoint subsets whose union is the entire set.) If there were any "overlap" among the rule entries, we would have a redundant case in which more than one rule could be satisfied. Here, Y2 is the set of years between 1812 and 2012 evenly divisible by four excluding the year 1900.

Variable	Equivalence Classes
Month	M1 = { month: month has 30 days}
	M2 = { month: month has 31 days}
	M3 = { month: month is February}
Day	D1 = {day: 1 ≤ day ≤ 28}
	D2 = {day: day = 29 }
	D3 = {day: day = 30 }
	D4 = {day: day = 31 }
Year	Y1 = {year: year = 1900}
	Y2 = {year: 1812 ≤ year ≤ 2012 AND (year ≠ 1900) AND (year = 0 mod 4)}
	Y3 = {year: (1812 ≤ year ≤ 2012 AND year ≠ 0 mod 4)}

In a sense, we could argue that we have a "gray box" technique, because we take a closer look at the NextDate function. In order to produce the next date of a given date, there are only five possible manipulations: incrementing and resetting the day and month, and incrementing the year, (we won't let time go backwards by resetting the year).

Conditions	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
c1 month in	M1	M1	M1	M1	M2	M2	M2	M2
c2: day in	D1	D2	D3	D4	D1	D2	D3	D4
c3: year in	-	-	-	-	-	-	-	-
Rule Count	3	3	3	3	3	3	3	3
actions								
a1: impossible					X			
a2: increment day	X	X			X	X	X	

a3: reset day			X					X
a4: increment month			X					?
a5: reset month								?
a6: increment year								?
conditions	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
c1: month in	M3	M3	M3	M3	M3	M3	M3	M3
c2: day in	D1	D1	D1	D2	D2	D2	D3	D3
c3: year in	Y1	Y2	Y3	Y1	Y2	Y3	-	-
Rule Count	1	1	1	1	1	1	3	3
actions								
a1: impossible				X		X	X	X
a2: increment day		X						
a3: reset day	X		X		X			
a4: increment month	X		X		X			
a5: reset month								
a6: increment year								

This decision table has 36 rules, and corresponds to the Cartesian Product of the equivalence classes. We still have the problem with logically impossible rules, but this formulation helps us identify the expected outputs of a test case. If you develop this table, you will find some cumbersome problems with December (in rule 8). We fix these next.

Third Try

We can clear up the end of year considerations with a third set of equivalence classes. This time, we are very specific about days and months, and we revert to the simpler leap-year or common year condition of the first try, so the year 1900 gets no special attention. (We could do a fourth try, showing year equivalence classes as in the second try, but by now, you get the point.)

Revised NextDate Domain Equivalence Classes

Month	M1 = {month: month has 30 days}	
	M2 = {month: month has 31 days except Dec.}	
	M3 = {month: month is December}	
	M4 = {month: month is February}	
Day	D1 = {day: 1 ≤ day ≤ 27}	
	D2 = {day: day = 28}	
	D3 = {day: day = 29}	
	D4 = {day: day = 30}	
	D5 = {day: day = 31 }	
Year	Y1 = {year: year is a leap year}	
	Y2 = {year: year is a common year}	

The Cartesian product of these contains 40 elements. The full decision table is given in Figure 7.8; it has 22 rules, compared to the 36 of the second try. Recall from Chapter 1 the question of whether a large set of test cases is necessarily better than a smaller set. Here we have a 22 rule decision table that gives a clearer picture of the NextDate function than does the 36 rule decision table. The first five rules deal with 30 day months; notice that the leap year considerations are irrelevant. The next two sets of rules (6 - 10 and 11 - 15) deal with 31 day months, where the first five deal with months other than December, and the second five deal with December. There are no impossible rules in this portion of the decision table, though there is some redundancy that an efficient tester might question. Eight of the ten rules simply increment the day. Would we really require eight separate test cases for this subfunction? Probably not, but note the insights we can get from the decision table. Finally, the last seven rules focus on February and leap year.

The decision table in Figure 2.11.8 is the basis for the source code for the NextDate function in Chapter 2. As an aside, this example shows how good testing can improve programming. All of the decision table analysis could have been done during the detailed design of the NextDate function.

Figure 2.11.8 Decision Table for the NextDate Function

Corresponding Test Cases:

Case ID	Month	Day	Year	Expected Output
1	April	15	1993	April 16, 1993
2	April	28	1993	April 29, 1993

3	April	29	1993	April 30, 1993
4	April	30	1993	May 1, 1993
5	April	31	1993	Impossible
6	Jan.	15	1993	Jan. 16, 1993
7	Jan.	28	1993	Jan. 29, 1993
8	Jan.	29	1993	Jan. 30, 1993
9	Jan.	30	1993	Jan. 31, 1993
10	Jan.	31	1993	Feb. 1, 1993
11	Dec.	15	1993	Dec. 16, 1993
12	Dec.	28	1993	Dec. 29, 1993
13	Dec.	29	1993	Dec. 30, 1993
14	Dec.	30	1993	Dec. 31, 1993
15	Dec.	31	1993	Jan. 1, 1994
16	Feb.	15	1993	Feb. 16, 1993
17	Feb.	28	1992	Feb. 29, 1992
18	Feb.	28	1993	Mar. 1, 1993
19	Feb.	29	1992	Mar. 1, 1992
20	Feb.	29	1993	Impossible
21	Feb.	30	1993	Impossible
22	Feb.	31	1993	Impossible

2.11.3 Test Cases for the Commission Problem

As we will see in this section, the Commission Problem is not well-served by a decision table analysis. Not surprising, because there is very little decisional logic in the problem. What we can do is see how decision tables can help to improve an under-specified problem (the Commission Problem is a good example of this all too common situation). To get started, recall that the Commission Problem is a portion of the larger Lock, Stock, and Barrel example, in which telegrams

representing orders are sent to a central point. The functional view of the entire problem is that the salesperson's commission is a function of the telegrams:

F(telegrams) = commission

This decomposes naturally into three subfunctions:

F1(telegrams) = (locks, stocks, barrels)

F2(locks, stocks, barrels) = sales

F3(sales) = commission

The Commission Problem is just the composition of F2 and F3. As we saw in Chapter 6, the input domain is partitioned by the limits on locks, stocks, and barrels.

Variable	Equivalence Classes for F2
Lock	L1 = {lock: lock < 1}
	L2 = {lock: 1 ≤ lock ≤ 70 }
	L3 = {lock: lock > 70 }
Stock	S1 = {stock: stock < 1 }
	S2 = {stock: 1 ≤ stock ≤ 80 }
	S3 = {stock: stock > 80 }
Barrel	B1 = {barrel: barrel < 1 }
	B2 = {barrel: 1 ≤ barrel ≤ 90 }
	B3 = {barrel: barrel > 90 }

These equivalence classes lead directly to a possible set of conditions for a decision table:

c1: lock <1

c2: 1 ≤ lock ≤ 70

c3: lock > 70

c4: stock < 1

c5: 1 ≤ stock ≤ 80

c6: stock > 80

c7: barrel < 1

c8: 1 ≤ barrel ≤ 90

c9: barrel > 90

So far, so good. Now, what actions shall we postulate? The original problem statement only addresses the cases where c2, c5, and c8 are all true. There is no information for what should happen if any of the remaining conditions are true. This is an error of omission at specification time. A tester might use decision tables as the skeleton of an interview to determine, from the user/customer, just what actions should be taken in these circumstances. (The specifier could also have done this!) We might end up with actions such as the following:

a1: Error: must sell at least one rifle lock.

a2: Error: cannot sell more than 70 rifle locks.

a3: Error: must sell at least one rifle stock.

a4: Error: cannot sell more than 80 rifle stocks.

a5: Error: must sell at least one rifle barrel.

a6: Error: cannot sell more than 90 rifle barrels.

An analysis such as this raises follow-up questions. For example, what if a negative number were entered as the value of one of the variables? Would that indicate returned items? If so, the company would likely wish to reduce the salesperson's commission. Now, what if such a reduction moved the commission to a less favorable level, such as from the 20% level to the 10% level? We get into similar questions when the sales limits are exceeded. How should excess sales be treated? Are they simply carried over to the next month? The company might prefer this, because it would likely reduce the commission level; obviously, the salesperson would prefer excess sales to contribute to the higher commission levels. This discussion ties back to the Chapter 1 discussion on specified, programmed, and tested behaviors. The examples in which limits are violated are good examples of test cases that are "outside" the sets of specified and programmed behaviors. (We might have added a fourth circle, that refers to what the customer really wants.)

We can do a little more for the F3 subfunction. Since there are three commission levels, there will clearly be three equivalence classes.

Variable	Equivalence Classes for F3
Sales	L1 = {sales: sales < 1000 }
	L2 = {sales: 1000 ≤ sales ≤ 1800 }
	L3 = {sales: sales > 1800 }

These equivalence classes lead to the simple decision table in Figure 7.9:

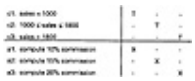


Figure 2.11.9 Decision Table for F3 of the Commission Problem

Our last step is to identify actual test cases for the Commission Problem. Note that we cannot simply provide values of sales that would give us test cases for the subfunction F3. Now you will see how arithmetically contrived this problem really is.

Test Cases for the Commission Problem:

Test Case	locks	stocks	barrels	sales	commission
DT1	5	5	5	500	50
DT2	15	15	15	1500	175
DT3	25	25	25	2500	360

2.12 Guidelines and Observations

As with the other testing techniques, decision table based testing works well for some applications (like NextDate) and is not worth the trouble for others (like the Commission Problem). Not surprisingly, the situations in which it works well are those where there is a lot of decision making (like the Triangle Problem), and those in which there are important logical relationships among input variables (like the NextDate function).

1. The decision table technique is indicated for applications characterized by any of the following:

prominent If-Then-Else logic

logical relationships among input variables

calculations involving subsets of the input variables

cause and effect relationships between inputs and outputs

high cyclomatic (McCabe) complexity (see Chapter 9)

2. Decision tables don't scale up very well (a limited entry table with n conditions has 2^n rules). There are several ways to deal with this: use extended entry decision tables, algebraically simplify tables, "factor" large tables into smaller ones, and look for repeating patterns of condition entries. For more on these techniques, see [Topper 93].

3. As with other techniques, iteration helps. The first set of conditions and actions you identify may be unsatisfactory. Use it as a stepping stone, and gradually improve on it until you are satisfied with a decision table.

EXERCISES

1. Develop a formula for the number of robustness test cases for a function of n variables.

2. Develop a formula for the number of robust worst case test cases for a function of n variables.

3. Make a Venn diagram showing the relationships among test cases form boundary value analysis, robustness testing, worst case testing, and robust worst case testing.

4. What happens if we try to do output range robustness testing? Use the commission problem as an example.

UNIT 3

PATH TESTING, DATA FLOW TESTING

3.1 Path Testing

The distinguishing characteristic of structural testing methods is that they are all based on the source code of the program being tested, and not on the definition. Because of this absolute basis, structural testing methods are very amenable to rigorous definitions, mathematical analysis, and precise measurement. In this chapter, we will examine the two most common forms of path testing. The technology behind these has been available since the mid-1970s, and the originators of these methods now have companies that market very successful tools that implement the techniques. Both techniques start with the program graph;

Definition

Given a program written in an imperative programming language, its ***program graph*** is a directed graph in which nodes are either entire statements or fragments of a statement, and edges represent flow of control. If i and j are nodes in the program graph, there is an edge from node i to node j iff the statement (fragment) corresponding to node j can be executed immediately after the statement (fragment) corresponding to node i.

Constructing a program graph from a given program is an easy process. It's illustrated here with the Pascal implementation of the Triangle program from Chapter 2. Line numbers refer to statements and statement fragments. There is an element of judgment here: sometimes it is convenient to keep a fragment (like a BEGIN) as a separate node, as at line 4. Other times is seems better to include this with another portion of a statement: the BEGIN at line 13 could really be merged with the THEN on line 12. We will see that this latitude collapses onto a unique DD-Path graph, so the differences introduced by differing judgments are moot. (A mathematician would make the point that, for a given program, there might be several distinct program graphs, all of which reduce to a unique DD-Path graph.) We also need to decide whether to associate nodes with non-executable statements such as variable and type declarations: here we do not.

A program graph of this program is given in Figure 3.1. Nodes 4 through 7 are a sequence, nodes 8 through 11 are an IF-THEN-ELSE construct (that terminates on an IF clause), and nodes 14 through 16 are an IF-THEN construct. Nodes 4 and 22 are the program source and sink nodes, corresponding to the single entry, single exit criteria.. There are no loops, so this is a directed acyclic graph. The importance of the program graph is that program executions correspond to paths from the source to the sink nodes. Since test cases force the execution of some such program path, we now have a very explicit description of the relationship between a test case and the part of the program it exercises. We also have an elegant, theoretically respectable way to deal with the potentially large number of execution paths in a program. Figure 3.2 is a graph of a simple program;

it is typical of the kind of example used to show the impossibility of completely testing even simple programs [Schach 93].

In this program, there are 5 paths from node B to node F in the interior of the loop. If the loop may have up to 18 repetitions, there are some 4.77 trillion distinct program execution paths.

```
1.  program triangle (input, output) ;
2.  VAR a, b, c        : integer;
3.     IsATriangle     : boolean;
4.  BEGIN
5.  writeln('Enter three integers which are sides of a triangle:');
6.  readln (a,b,c);
7.  writeln('Side A is ',a, 'Side B is ',b, 'side C is ',c);
8.  IF (a < b + c) AND (b < a + c) AND (c < a + b)
9.     THEN IsATriangle :=TRUE
10.    ELSE IsATriangle := FALSE ;
11.  IF IsATriangle
12.     THEN
13.     BEGIN
14.     IF (a = b) XOR (a = c) XOR (b = c) AND NOT((a=b) AND (a=c))
15.        THEN Writeln ('Triangle is Isosceles') ;
16.     IF (a = b) AND (b = c)
17.        THEN Writeln ('Triangle is Equilateral') ;
18.     IF (a <> b) AND (a <> c) AND (b <> c)
19.        THEN Writeln ('Triangle is Scalene') ;
20.     END
21.     ELSE WRITELN('Not a Triangle') ;
22.  END.
```

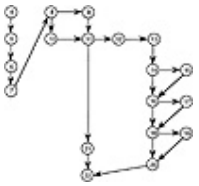


Figure 3.1 Program Graph of the Pascal Triangle Program

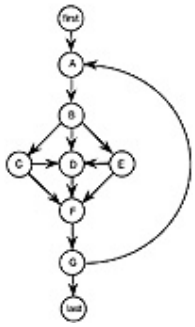


Figure 3.2 Trillions of Paths

3.2 DD-Paths

The best known form of structural testing is based on a construct known as a decision-to-decision path (DD-Path) [Miller 77]. The name refers to a sequence of statements that, in Miller's words, begins with the "outway" of a decision statement and ends with the "inway" of the next decision

statement. There are no internal branches in such a sequence, so the corresponding code is like a row of dominoes lined up so that when the first falls, all the rest in the sequence fall. Miller's original definition works well for second generation languages like FORTRAN II, because decision making statements (such as arithmetic IFs and DO loops) use statement labels to refer to target statements. With block structured languages (Pascal, Ada, C), the notion of statement fragments resolves the difficulty of applying Miller's original definition—otherwise, we end up with program graphs in which some statements are members of more than one DD-Path.

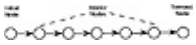


Figure 3.3 A Chain Of Nodes In A Directed Graph

We will define DD-Paths in terms of paths of nodes in a directed graph. We might call these paths chains, where a *chain* is a path in which the initial and terminal nodes are distinct, and every interior node has indegree = 1 and outdegree = 1. Notice that the initial node is 2-connected to every other node in the chain, and there are no instances of 1- or 3-connected nodes, as shown in Figure 3.3. The length (number of edges) of the chain in Figure 3.3 is 6. We can have a degenerate case of a chain that is of length 0, that is, a chain consisting of exactly one node and no edges.

Definition

A ***DD-Path*** is a chain in a program graph such that

Case 1: it consists of a single node with indeg = 0,

Case 2: it consists of a single node with outdeg = 0,

Case 3: it consists of a single node with indeg ≥ 2 or outdeg ≥ 2,

Case 4: it consists of a single node with indeg = 1 and outdeg = 1,

Case 5: it is a maximal chain of length ≥ 1.

Cases 1 and 2 establish the unique source and sink nodes of the program graph of a structured program as initial and final DD-Paths. Case 3 deals with complex nodes; it assures that no node is contained in more than one DD-Path. Case 4 is needed for "short branches"; it also preserves the one fragment, one DD-Path principle. Case 5 is the "normal case", in which a DD-Path is a single entry, single exit sequence of nodes (a chain). The "maximal" part of the case 5 definition is used to determine the final node of a normal (non-trivial) chain.

Table 1 Types of DD-Paths in Figure 9.1 **Program Graph Nodes**

	DD-Path Name	**Case of Definition**
4	first	1
5 - 8	A	5
9	B	4

10	C	4
11	D	3
12 - 14	E	5
15	F	4
16	G	3
17	H	4
18	I	3
19	J	4
20	K	3
21	L	4
22	last	2

This is a complex definition, so we'll apply it to the program graph in Figure 9.1. Node 4 is a Case 1 DD-Path, we'll call it "first"; similarly, node 22 is a Case 2 DD-Path, and we'll call it "last". Nodes 5 through 8 are a Case 5 DD-Path. We know that node 8 is the last node in this DD-Path because it is the last node that preserves the 2-connectedness property of the chain. If we went beyond node 8 to include nodes 9 and 10, these would both be 2-connected to the rest of the chain, but they are only 1-connected to each other. If we stopped at node 7, we would violate the "maximal" criterion. Node 11 is a Case 3 DD-Path, which forces nodes 9 and 10 to be individual DD-Paths by case 4. Nodes 12 through 14 are a case 5 DD-Path by the same reasoning as for nodes 5 - 8. Nodes 14 through 20 correspond to a sequence of IF-THEN statements. Nodes 16 and 18 are both Case 3 DD-Paths, and this forces nodes 15, 17, and 19 to be Case 4 DD-Paths. Node 20 is a Case 3 DD-Path, and node 21 is a Case 4 DD-Path. All of this is summarized in Table 1, where the DD-Path names correspond to the DD-Path graph in Figure 3.4. Part of the confusion with this example is that the triangle problem is logic intensive and computationally sparse. This combination yields many short DD-Paths. If the THEN and ELSE clauses contained BEGIN

. . END blocks of computational statements, we would have longer DD-Paths, as we do in the commission problem.

We can now define the DD-Path graph of a program.

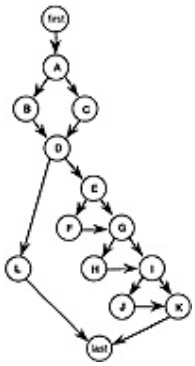


Figure 3.4 DD-Path Graph for the Triangle Program

Definition

Given a program written in an imperative language, its ***DD-Path graph*** is the directed graph in which nodes are DD-Paths of its program graph, and edges represent control flow between successor DD-Paths. In effect, the DD-Path graph is a form of condensation graph (see Chapter 4); in this condensation, 2-connected components are collapsed into individual nodes that correspond to Case 5 DD-Paths. The single node DD-Paths (corresponding to Cases 1 - 4) are required to preserve the convention that a statement (or statement fragment) is in exactly one DD-Path. Without this convention, we end up with rather clumsy DD-Path graphs, in which some statement (fragments) are in several DD-Paths.

Testers shouldn't be intimidated by this process — there are high quality commercial tools that generate the DD-Path graph of a given program. The vendors make sure that their products work for a wide variety of programming languages. In practice, it's reasonable to make DD-Path graphs for programs up to about 100 source lines. Beyond that, most testers look for a tool.

3.3 Test Coverage Metrics

The *raison d'être* of DD-Paths is that they enable very precise descriptions of test coverage. Recall (from Chapter 8) that one of the fundamental limitations of functional testing is that it is impossible to know either the extent of redundancy or the possibility of gaps corresponding to the way a set of functional test cases exercises a program. Back in Chapter 1, we had a Venn diagram showing relationships among specified, programmed, and tested behaviors. Test coverage metrics are a device to measure the extent to which a set of test cases covers (or exercises) a program.

There are several widely accepted test coverage metrics; most of those in Table 2 are due to the early work of E. F. Miller [Miller 77]. Having an organized view of the extent to which a program is tested makes it possible to sensibly manage the testing process. Most quality organizations now expect the C_1 metric (DD-Path coverage) as the minimum acceptable level of test coverage. The statement coverage metric (C_0) is still widely accepted: it is mandated by ANSI Standard 187B, and has been used successfully throughout IBM since the mid-1970s.

Table 2 Structural Test Coverage Metrics **Metric**

	Description of Coverage
C_0	Every statement
C_1	Every DD-Path (predicate outcome)
C_1^p	Every predicate to each outcome
C_2	C_1 coverage + loop coverage
C_d	C_1 coverage + every dependent pair of DD-Paths
C_{MCC}	Multiple condition coverage
C_ik	Every program path that contains up to k repetitions of a loop (usually k = 2)
C_{stat}	"Statistically significant" fraction of paths
C_∞	All possible execution paths

These coverage metrics form a lattice (see Chapter 10) in which some are equivalent, and some are implied by others. The importance of the lattice is that there are always fault types that can be revealed at one level, and can escape detection by inferior levels of testing. E. F. Miller observes that when DD-Path coverage is attained by a set of test cases, roughly 85% of all faults are revealed [Miller 91].

3.3.1 Metric Based Testing

The test coverage metrics in Table 2 tell us what to test, but not how to test it. In this section, we take a closer look at techniques that exercise source code in terms of the metrics in Table 2. We must keep an important distinction in mind: Miller's test coverage metrics are based on program graphs in which nodes are full statements, whereas our formulation allows statement fragments to be nodes. For the remainder of this section, the statement fragment formulation is "in effect".

Statement and Predicate Testing

Because our formulation allows statement fragments to be individual nodes, the statement and predicate levels (C_0 and C_1) to collapse into one consideration. In our triangle example (see Figure 3.1), nodes 8, 9, and 10 are a complete Pascal IF-THEN-ELSE statement. If we required nodes to correspond to full statements, we could execute just one of the decision alternatives and satisfy the statement coverage criterion. Because we allow statement fragments, it is "natural" to divide such a statement into three nodes. Doing so results in predicate outcome coverage. Whether or not our convention is followed, these coverage metrics require that we find a set of test cases such that, when executed, every node of the program graph is traversed at least once.

DD-Path Testing

When every DD-Path is traversed (the C_1 metric), we know that each predicate outcome has been executed; this amounts to traversing every edge in the DD-Path graph (or program graph), as opposed to just every node. For IF-THEN and IF-THEN-ELSE statements, this means that both the true and the false branches are covered (C_1p coverage). For CASE statements, each clause is covered. Beyond this, it is useful to ask what else we might do to test a DD-Path. Longer DD-Paths generally represent complex computations, which we can rightly consider as individual functions. For such DD-Paths, it may be appropriate to apply a number of functional tests, especially those for boundary and special values.

Dependent Pairs of DD-Paths

The C_d metric foreshadows the dataflow testing. The most common dependency among pairs of DD-Paths is the define/reference relationship, in which a variable is defined (receives a value) in one DD-Path and is referenced in another DD-Path. The importance of these dependencies is that they are closely related to the problem of infeasible paths. We have good examples of dependent pairs of DD-Paths: in Figure 9.4, B and D are such a pair, so are DD-Paths C and L. Simple DD-Path coverage might not exercise these dependencies, thus a deeper class of faults would not be revealed.

Multiple Condition Coverage

Look closely at the compound conditions in DD-Paths A and E. Rather than simply traversing such predicates to their TRUE and FALSE outcomes, we should investigate the different ways that each outcome can occur. One possibility is to make a truth table; a compound condition of three simple conditions would have eight rows, yielding eight test cases. Another possibility is to reprogram compound predicates into nested simple IF-THEN-ELSE logic, which will result in more DD-Paths to cover. We see an interesting trade-off: statement complexity versus path complexity. Multiple condition coverage assures that this complexity isn't swept under the DD-Path coverage rug.

Loop Coverage

The condensation graphs provide us with an elegant resolution to the problems of testing loops. Loop testing has been studied extensively, and with good reason — loops are a highly fault prone portion of source code. To start, there is an amusing taxonomy of loops in [Beizer 83]: concatenated, nested, and horrible, shown in Figure 3.5.

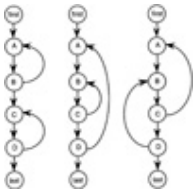


Figure 3.5 Concatenated, Nested, and Knotted Loops

Concatenated loops are simply a sequence of disjoint loops, while nested loops are such that one is contained inside another. Horrible loops cannot occur when the structured programming precepts are followed. When it is possible to branch into (or out from) the middle of a loop, and these branches are internal to other loops, the result is Beizer's horrible loop. (Other sources define this as a knot—how appropriate.) The simple view of loop testing is that every loop involves a decision, and we need to test both outcomes of the decision: one is to traverse the loop, and the other is to exit (or not enter) the loop. This is carefully proved in [Huang 79]. We can also take a modified boundary value approach, where the loop index is given its minimum, nominal, and maximum values (see Chapter 5). We can push this further, to full boundary value testing and even robustness testing. If the body of a simple loop is a DD-Path that performs a complex calculation, this should also be tested, as discussed above. Once a loop has been tested, the tester condenses it into a single node. If loops are nested, this process is repeated starting with the innermost loop and working outward. This results in the same multiplicity of test cases we found with boundary value analysis, which makes sense, because each loop index variable acts like an input variable. If loops are knotted, it will be necessary to carefully analyze them in terms of the dataflow methods discussed in Chapter 10. As a preview, consider the infinite loop that could occur if one loop tampers with the value of the other loop's index.

3.3.2. Test Coverage Analyzers

Coverage analyzers are a class of test tools that offer automated support for this approach to testing management. With a coverage analyzer, the tester runs a set of test cases on a program that has been "instrumented" by the coverage analyzer. The analyzer then uses information produced by the instrumentation code to generate a coverage report. In the common case of DD-Path coverage, for example, the instrumentation identifies and labels all DD-Paths in an original program. When the instrumented program is executed with test cases, the analyzer tabulates the DD-Paths traversed by each test case. In this way, the tester can experiment with different sets of test cases to determine the coverage of each set.

3.4 Basis Path Testing

The mathematical notion of a "basis" has attractive possibilities for structural testing. Certain sets can have a basis, and when they do, the basis has very important properties with respect to the entire set. Mathematicians usually define a basis in terms of a structure called a "vector space", which is a set of elements (called vectors) and which has operations that correspond to multiplication and addition defined for the vectors. If a half dozen other criteria apply, the structure is said to be a vector space, and all vector spaces have a basis (in fact they may have several bases). The basis of a vector space is a set of vectors such that the vectors are independent of each other and they "span" the entire vector space in the sense that any other vector in the space can be expressed in terms of the basis vectors. Thus a set of basis vectors somehow represents "the essence" of the full vector space: everything else in the space can be expressed in terms of the basis, and if one basis element is deleted, this spanning property is lost. The potential for testing is that, if we can view a program as a vector space, then the basis for such a space would be a very interesting set of elements to test. If the basis is "OK", we could hope that everything that can be expressed in terms of the basis is

also "OK". In this section, we examine the early work of Thomas McCabe, who recognized this possibility in the mid-1970s.

3.4.1 McCabe's Basis Path Method

Figure 3.6 is taken from [McCabe 82]; it is a directed graph which we might take to be the program graph (or the DD-Path graph) of some program. For the convenience of readers who have encountered this example elsewhere ([McCabe 87], [Perry 87]), the original notation for nodes and edges is repeated here. (Notice that this is not a graph derived from a structured program: nodes B and C are a loop with two exits, and the edge from B to E is a branch into the IF-THEN statement in nodes D, E, and F. The program does have a single entry (A) and a single exit (G).) McCabe based his view of testing on a major result from graph theory, which states that the cyclomatic number (see Chapter 4) of a strongly connected graph is the number of linearly independent circuits in the graph. (A circuit is similar to a chain: no internal loops or decisions, but the initial node is the terminal node. A circuit is a set of 3-connected nodes.) We can always create a strongly connected graph by adding an edge from the (every) sink node to the (every) source node. (Notice that, if the single entry, single exit precept is violated, we greatly increase the cyclomatic number, because we need to add edges from each sink node to each source node.) Figure 9.7 shows the result of doing this; it also contains edge labels that are used in the discussion that follows.

There is some confusion in the literature about the correct formula for cyclomatic complexity. Some sources give the formula as V(G) = e - n + p, while others use the formula V(G) = e - n + 2p; everyone agrees that e is the number of edges, n is the number of nodes, and p is the number of connected regions. The confusion apparently comes from the transformation of an arbitrary directed graph (such as the one in Figure 9.6) to a strongly connected directed graph obtained by adding one edge from the sink to the source node (as in Figure 3.7). Adding an edge clearly affects value computed by the formula, but it shouldn't affect the number of circuits. Here's a way to resolve the apparent inconsistency: The number of linearly independent paths from the source node to the sink node in Figure 3.6 is

$$V(G) = e - n + 2p$$
$$= 10 - 7 + 2(1) = 5,$$

and the number of linearly independent circuits in the graph in Figure 3.7 is

$$V(G) = e - n + p$$
$$= 11 - 7 + 1 = 5.$$

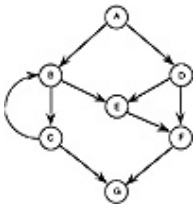


Figure 3.6 McCabe's Control Graph

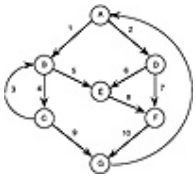


Figure 3.7 McCabe's Derived Strongly Connected Graph

The cyclomatic complexity of the strongly connected graph in Figure 3.7 is 5, thus there are five linearly independent circuits. If we now delete the added edge form node G to node A, these five circuits become five linearly independent paths from node A to node G. In small graphs, we can visually identify independent paths. Here we identify paths as sequences of nodes:

p1: A, B, C, G

p2: A, B, C, B, C, G

p3: A, B, E, F, G

p4: A, D, E, F, G

p5: A, D, F, G

We can force this beginning to look like a vector space by defining notions of addition and scalar multiplication: path addition is simply one path followed by another path, and multiplication corresponds to repetitions of a path. With this formulation, McCabe arrives at a vector space of program paths. His illustration of the basis part of this framework is that the path A, B, C, B, E, F, G is the basis sum p2 + p3 - p1, and the path A, B, C, B, C, B, C, G is the linear combination 2p2 - p1. It is easier to see this addition with an incidence matrix (see Chapter 4) in which rows correspond to paths, and columns correspond to edges, as in Table 3. The entries in this table are obtained by following a path and noting which edges are traversed. Path p1, for example, traverses edges 1, 4, and 9; while path p2 traverses the following edge sequence: 1, 4, 3, 4, 9. Since edge 4 is traversed twice by path p2, that is the entry for the edge 4 column.

Table 3 Path/Edge Traversal **path \ edges travesed**

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
p1: A, B, C, G	1	0	0	1	0	0	0	0	1	0
p2: A, B, C, B, C, G	1	0	1	2	0	0	0	0	1	0
p3: A, B, E, F, G	1	0	0	0	1	0	0	1	0	1
p4: A, D, E, F, G	0	1	0	0	0	1	0	1	0	1
p5: A, D, F, G	0	1	0	0	0	0	1	0	0	1

ex1: A, B, C, B, E, F, G	1	0	1	1	1	0	0	1	0	1
ex2: A, B, C, B, C, B, C, G	1	0	2	3	0	0	0	0	1	0

We can check the independence of paths p1 - p5 by examining the first five rows of this incidence matrix. The bold entries show edges that appear in exactly one path, so paths p2 - p5 must be independent. Path p1 is independent of all of these, because any attempt to express p1 in terms of the others introduces unwanted edges. None can be deleted, and these five paths span the set of all paths from node A to node G. At this point, you should check the linear combinations of the two example paths. The addition and multiplication are performed on the column entries.

McCabe next develops an algorithmic procedure (called the "baseline method") to determine a set of basis paths. The method begins with the selection of a "baseline" path, which should correspond to some "normal case" program execution. This can be somewhat arbitrary; McCabe advises choosing a path with as many decision nodes as possible. Next the baseline path is retraced, and in turn each decision is "flipped", that is when a node of outdegree ≥2 is reached, a different edge must be taken. Here we follow McCabe's example, in which he first postulates the path through nodes A, B, C, B, E, F, G as the baseline. (This was expressed in terms of paths p1 - p5 earlier.) The first decision node (outdegree ≥2) in this path is node A, so for the next basis path, we traverse edge 2 instead of edge 1. We get the path A, D, E, F, G, where we retrace nodes E, F, G in path 1 to be as minimally different as possible. For the next path, we can follow the second path, and take the other decision outcome of node D, which gives us the path A, D, F, G. Now only decision nodes B and C have not been flipped; doing so yields the last two basis paths, A, B, E, F, G and A, B, C, G. Notice that this set of basis paths is distinct from the one in Table 3: this is not problematic, because there is no requirement that a basis be unique.

3.4.2 Observations on McCabe's Basis Path Method

If you had trouble following some of the discussion on basis paths and sums and products of these, you may have felt a haunting skepticism, something along the lines of "Here's another academic oversimplification of a real-world problem". Rightly so, because there are two major soft spots in the McCabe view: one is that testing the set of basis paths is sufficient (it's not), and the other has to do with the yoga-like contortions we went through to make program paths look like a vector space. McCabe's example that the path A, B, C, B, C, B, C, G is the linear combination 2p2 - p1 is very unsatisfactory. What does the 2p2 part mean? Execute path p2 twice? (Yes, according to the math.) Even worse, what does the - p1 part mean? Execute path p1 backwards? Undo the most recent execution of p1? Don't do p1 next time? Mathematical sophistries like this are a real turn-off to practitioners looking for solutions to their very real problems. To get a better understanding of these problems, we'll go back to the triangle program example.

Start with the DD-Path graph of the triangle program in Figure 9.4. We begin with a baseline path that corresponds to a scalene triangle, say with sides 3, 4, 5. This test case will traverse the path p1. Now if we flip the decision at node A, we get path p2. Continuing the procedure, we flip the decision at node D, which yields the path p3. Now we continue to flip decision nodes in the baseline

path p1; the next node with outdegree = 2 is node E. When we flip node E, we get the path p4. Next we flip node G to get p5. Finally, (we know we're done, because there are only 6 basis paths) we flip node I to get p6. This procedure yields the following basis paths:

p1: A-B-D-E-G-I-J-K-Last
p2: A-C-D-E-G-I-J-K-Last
p3: A-B-D-L-Last
p4: A-B-D-E-F-G-I-J-K-Last
p5: A-B-D-E-F-G-H-I-J-K-Last
p6: A-B-D-E-F-G-H-I-K-Last

Time for a reality check: if you follow paths p2, p3, p4, p5, and p6, you find that they are all infeasible. Path p2 is infeasible, because passing through node C means the sides are not a triangle, so none of the sequel decisions can be taken. Similarly, in p3, passing through node B means the sides do form a triangle, so node L cannot be traversed. The others are all infeasible because they involve cases where a triangle is of two types (e.g., isosceles and equilateral). The problem here is that there are several inherent dependencies in the triangle problem. One is that if three integers constitute sides of a triangle, they must be one of the three possibilities: equilateral, isosceles, or scalene. A second dependency is that the three possibilities are mutually exclusive: if one is true, the other two must be false.

Recall that dependencies in the input data domain caused difficulties for boundary value testing, and that we resolved these by going to decision table based functional testing, where we addressed data dependencies in the decision table. Here we are dealing with code level dependencies, and these are absolutely incompatible with the latent assumption that basis paths are independent. McCabe's procedure successfully identifies basis paths that are topologically independent, but when these contradict semantic dependencies, topologically possible paths are seen to be logically infeasible. One solution to this problem is to always require that flipping a decision results in a semantically feasible path. Another is to reason about logical dependencies. If we think about this problem we can identify several rules:

If node B is traversed, then we must traverse nodes D and E.

If node C is traversed, then we must traverse nodes D and L.

If node E is traversed, then we must traverse one of nodes F, H, and J.

If node F is traversed, then we cannot traverse nodes H and J.

If node H is traversed, then we cannot traverse nodes F and J.

If node J is traversed, then we cannot traverse nodes F and I.

Taken together, these rules, in conjunction with McCabe's baseline method, will yield the following feasible basis path set:

fp1: A-C-D-L-Last	(Not a triangle)
fp2: A-B-D-E-F-G-I-K-Last	(Isosceles)
fp3: A-B-D-E-G-H-I-K-Last	(Equilateral)
fp4: A-B-D-E-G-I-J-K-Last	(Scalene)

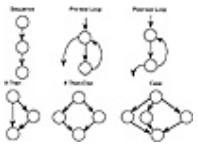


Figure 3.8 Structured Programming Constructs

The triangle problem is atypical in that there are no loops. The program has only 18 topologically possible paths, and of these, only the four basis paths listed above are feasible. Thus for this special case, we arrive at the same test cases as we did with special value testing and output range testing. For a more positive observation, basis path coverage guarantees DD-Path coverage: the process of flipping decisions guarantees that every decision outcome is traversed, which is the same as DD-Path coverage. We see this by example from the incidence matrix description of basis paths, and in our triangle program feasible basis paths. We could push this a step further and observe that the set of DD-Paths acts like a basis, because any program path can be expressed as a linear combination of DD-Paths.

3.4.3 Essential Complexity

Part of McCabe's work on cyclomatic complexity does more to improve programming than testing. In this section we take a quick look at this elegant blend of graph theory, structured programming, and the implications these have for testing. This whole package centers on the notion of essential complexity [McCabe 82], which is just the cyclomatic complexity of yet another form of condensation graph. Recall that condensation graphs are a way of simplifying an existing graph; so far our simplifications have been based on removing either strong components or DD-Paths. Here, we condense around the structured programming constructs, which are repeated as Figure 3.8.

The basic idea is to look for the graph of one of the structured programming constructs, collapse it into a single node, and repeat until no more structured programming constructs can be found. This process is followed in Figure 9.9, which starts with the DD-Path graph of the Pascal triangle program. The IF-THEN-ELSE construct involving nodes A, B, C, and D is condensed into node a, and then the three IF-THEN constructs are condensed onto nodes b, c, and d. The remaining IF-THEN-ELSE (which corresponds to the IF IsATriangle statement) is condensed into node e, resulting in a condensed graph with cyclomatic complexity V(G) = 1. In general, when a program is

well structured (i.e., is composed solely of the structured programming constructs), it can always be reduced to a graph with one path.

The graph in Figure 3.6 cannot be reduced in this way (try it!). The loop with nodes B and C cannot be condensed because of edge from B to E. Similarly, nodes D, E, and F look like an IF-THEN construct, but the edge from B to E violates the structure. McCabe went on to find elemental "unstructures" that violate the precepts of structured programming [McCabe 76]. These are shown in Figure 3.10.

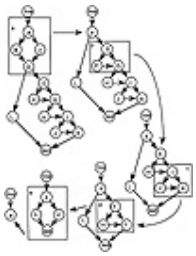


Figure 3.9 Condensing with Respect to the Structured Programming Constructs

Each of these "unstructures" contains three distinct paths, as opposed to the two paths present in the corresponding structured programming constructs, so one conclusion is that such violations increase cyclomatic complexity. The *piece d' resistance* of McCabe's analysis is that these unstructures cannot occur by themselves: if there is one in a program, there must be at least one more, so a program cannot be just slightly unstructured. Since these increase cyclomatic complexity, the minimum number of test cases is thereby increased. In the next chapter, we will see that the unstructures have interesting implications for dataflow testing.

The bottom line for testers is this: programs with high cyclomatic complexity require more testing. Of the organizations that use the cyclomatic complexity metric, most set some guideline for maximum acceptable complexity; V(G) = 10 is a common choice. What happens if a unit has a higher complexity? Two possibilities: either simplify the unit or plan to do more testing. If the unit is well structured, its essential complexity is 1, so it can be simplified easily. If the unit has an essential complexity that exceeds the guidelines, often the best choice is to eliminate the unstructures.

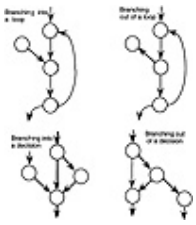


Figure 9.10 Violations of Structured Programming

3.5 Guidelines and Observations

In our study of functional testing, we observed that gaps and redundancies can both exist, and at the same time, cannot be recognized. The problem was that functional testing removes us "too far"

from the code. The path testing approaches to structural testing represent the case where the pendulum has swung too far the other way: moving from code to directed graph representations and program path formulations obscures important information that is present in the code, in particular the distinction between feasible and infeasible paths. In the next chapter, we look at dataflow based testing. These techniques move closer to the code, so the pendulum will swing back from the path analysis extreme.

McCabe was partly right when he observed: "It is important to understand that these are purely criteria that measure the quality of testing, and not a procedure to identify test cases" [McCabe 82]. He was referring to the DD-Path coverage metric (which is equivalent to the predicate outcome metric) and the cyclomatic complexity metric that requires at least the cyclomatic number of distinct program paths must be traversed. Basis path testing therefore gives us a lower bound on how much testing is necessary.

Path based testing also provides us with a set of metrics that act as cross checks on functional testing. We can use these metrics to resolve the gaps and redundancies question. When we find that the same program path is traversed by several functional test cases, we suspect that this redundancy is not revealing new faults. When we fail to attain DD-Path coverage, we know that there are gaps in the functional test cases. As an example, suppose we have a program that contains extensive error handling, and we test it with boundary value test cases (rain, mi n+, nom, max-, and max). Because these are all permissible values, DD-Paths corresponding to the error handling code will not be traversed. If we add test cases derived from robustness testing or traditional equivalence class testing, the DD-Path coverage will improve. Beyond this rather obvious use of coverage metrics, there is an opportunity for real testing craftsmanship. The coverage metrics in Table 2 can operate in two ways: as a blanket mandated standard (e.g., all units shall be tested to attain full DD-Path coverage) or as a mechanism to selectively test portions of code more rigorously than others. We might choose multiple condition coverage for modules with complex logic, while those with extensive iteration might be tested in terms of the loop coverage techniques. This is probably the best view of structural testing: use the properties of the source code to identify appropriate coverage metrics, and then use these as a cross check on functional test cases. When the desired coverage is not attained, follow interesting paths to identify additional (special value) test cases.

This is a good place to revisit the Venn diagram view of testing that we used in Chapter 1. Figure 9.11 shows the relationship between specified behaviors (set S), programmed behaviors (set P), and topologically feasible paths in a program (set T). As usual, region I is the most desirable — it contains specified behaviors that are implemented by feasible paths. By definition, every feasible path is topologically possible, so the shaded portion (regions 2 and 6) of the set P must be empty. Region 3 contains feasible paths that correspond to unspecified behaviors. Such extra functionality needs to be examined: if useful, the specification should be changed, otherwise these feasible paths should be removed. Regions 4 and 7 contain the infeasible paths; of these, region 4 is problematic. Region 4 refers to specified behaviors that have almost been implemented: topologically possible yet infeasible program paths. This region very likely corresponds to coding errors, where changes are needed to make the paths feasible. Region 5 still corresponds to specified behaviors that have not been implemented. Path based testing will never recognize this region. Finally, region 7 is a curiosity: unspecified, infeasible, yet topologically possible paths. Strictly speaking, there is no

problem here, because infeasible paths cannot execute. If the corresponding code is incorrectly changed by a maintenance action (maybe by a programmer who doesn't fully understand the code), these could become feasible paths, as in region 3.

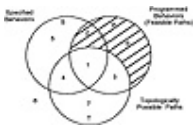


Figure 3.11 Feasible and Topologically Possible Paths

EXERCISES

1. Find the cyclomatic complexity of the graph in Figure 9.2.

2. Identify a set of basis paths for the graph in Figure 9.2.

3. Discuss McCabe's concept of "flipping" for nodes with outdegree ≥ 3.

4. Suppose we take Figure 9.2 as the DD-Path graph of some program. Develop sets of paths (which would be test cases) for the C_0, C_1, and C_2 metrics.

5. Develop multiple condition coverage test cases for the Pascal triangle program. Pay attention to the dependency between statement fragments 14 and 16 with the expression (a = b) AND (c = d). Rewrite the program segment 14 - 21 such that the compound conditions are replaced by nested IF-THEN-ELSE statements. Compare the cyclomatic complexity of your program with that of the existing version.

6. For a set V to be a vector space, two operations (addition and scalar multiplication) must be defined for elements in the set. In addition, the following criteria must hold for all vectors x, y, and z V, and for all scalars k, 1, 0, and 1:

i. if x, y V, the vector x + y V.

ii. x + y = y + x.

iii. (x + y) + z = x + (y + z).

iv. there is a vector 0 V such that x + 0 = x.

v. for any x V, there is a vector -x such that x + (-x) 0.

vi. for any X V, the vector kx V.

vii. k(x + y) kx + ky.

viii. (k + l)x kx + lx.

ix. k(lx) = (kl)x.

x. lx = x.

How many of these ten criteria hold for the "vector space" of paths in a program?

3.6 Data Flow Testing

Data flow testing is an unfortunate term, because most software developers immediately think about some connection with dataflow diagrams. Data flow testing refers to forms of structural testing that focus on the points at which variables receive values and the points at which these values are used (or referenced). We will see that data flow testing serves as a "reality check" on path testing; indeed, many of the data flow testing proponents (and researchers) see this approach as a form of path testing. We will look at two mainline forms of data flow testing: one provides a set of basic definitions and a unifying structure of test coverage metrics, while the second is based on a concept called a "program slice". Both of these formalize intuitive behaviors (and analyses) of testers, and although they both start with a program graph, both move back in the direction of functional testing.

Most programs deliver functionality in terms of data. Variables that represent data somehow receive values, and these values are used to compute values for other variables. Since the early 1960s, programmers have analyzed source code in terms of the points (statements) at which variables receive values and points at which these values are used. Many times, their analyses were based on concordances that list statement numbers in which variable names occur. Concordances were popular features of second generation language compilers (they are still popular with COBOL programmers). Early "data flow" analyses often centered on a set of faults that are now known as define/reference anomalies:

- a variable that is defined but never used (referenced)
- a variable that is used but never defined
- a variable that is defined twice before it is used

Each of these anomalies can be recognized from the concordance of a program. Since the concordance information is compiler generated, these anomalies can be discovered by what is known as "static analysis": finding faults in source code without executing it.

3.6.1 Define/Use Testing

Much of the formalization of define/use testing was done in the early 1980s [Rapps 85]; the definitions in this section are compatible with those in [Clarke 89], an article which summarizes most of define/use testing theory. This body of research is very compatible with the formulation we developed in chapters 4 and 9. It presumes a program graph in which nodes are statement fragments (a fragment may be an entire statement), and programs that follow the structured programming precepts.

The following definitions refer to a program P that has a program graph G(P), and a set of program variables V. The program graph G(P) is constructed as in Chapter 4, with statement fragments as nodes, and edges that represent node sequences. G(P) has a single entry node, and a single exit node. We also disallow edges from a node to itself. Paths, subpaths, and cycles are as they were in Chapter 4.

Definition

Node n ∈ G(P) is a ***defining node*** of the variable v ∈ V, written as DEF(v,n), iff the value of the variable v is defined at the statement fragment corresponding to node n. Input statements, assignment statements, loop control statements, and procedure calls are all examples of statements that are defining nodes. When the code corresponding to such statements executes, the contents of the memory location(s) associated with the variables are changed.

Definition

Node n ∈ G(P) is a ***usage node*** of the variable v ∈ V, written as USE(v, n), iff the value of the variable v is used at the statement fragment corresponding to node n. Output statements, assignment statements, conditional statements, loop control statements, and procedure calls are all examples of statements that are usage nodes. When the code corresponding to such statements executes, the contents of the memory location(s) associated with the variables remain unchanged.

Definition

A usage node USE(v, n) is a ***predicate use*** (denoted as P-use) iff the statement n is a predicate statement; otherwise USE(v, n) is a ***computation use*** , (denoted C-use). The nodes corresponding to predicate uses always have an outdegree ≥ 2, and nodes corresponding to computation uses always have outdegree ≤ 1.

Definition

A ***definition-use (sub)path*** with respect to a variable v (denoted du-path) is a (sub)path in PATHS(P) such that, for some v ∈ V, there are define and usage nodes DEF(v, m) and USE(v, n) such that m and n are the initial and final nodes of the (sub)path.

Definition

A ***definition-clear (sub)path*** with respect to a variable v (denoted dc-path) is a definition-use (sub)path in PATHS(P) with initial and final nodes DEF (v, m) and USE (v, n) such that no other node in the (sub)path is a defining node of v. Testers should notice how these definitions capture the essence of computing with stored data values. Du-paths and dc-paths describe the flow of data across source statements from points at which the values are defined to points at which the values are used. Du-paths that are not definition-clear are potential trouble spots.

3.6.2 Example

We will use the Commission Problem and its program graph to illustrate these definitions. The numbered source code is given next, followed by a program graph constructed according to the procedures we discussed in Chapter 4. This program computes the commission on the sales of four salespersons, hence the outer For-loop that repeats four times. During each repetition, a salesperson's name is read from the input device, and the input from that person is read to compute

the total numbers of locks, stocks, and barrels sold by the person. The While-loop is a classical sentinel controlled loop in which a value of -1 for locks signifies the end of that person's data. The totals are accumulated as the data lines are read in the While-loop. After printing this preliminary information, the sales value is computed, using the constant item prices defined at the beginning of the program. The sales value is then used to compute the commission in the conditional portion of the program.

```
 1   program lock-stock_and_barrel
 2  const
 3        lock_price = 45.0;
 4        stock_price = 30.0;
 5        barrel_price 25.0;
 6  type
 7         STRING_30 = string[30]; {Salesman's Name}
 8  var
 9         locks, stocks, barrels, num_locks, num_stocks,
10          num_barrels, salesman_index, order_index : INTEGER;
11         sales, commission : REAL;
12         salesman : STRING_30;
13
14    BEGIN  {program lock_stock_and_barrel}
15   FOR salesman_index := 1 TO 4 DO
16    BEGIN
17        READLN(salesman);
18        WRITELN ('Salesman is ', salesman);
19       num_locks := 0;
20       num_stocks := 0;
21       num_barrels := 0;
22       READ(locks);
23       WHILE locks <> -1 DO
24         BEGIN
25            READLN (stocks, barrels);
26            num_locks := num_locks + locks;
27            num_stocks := num_stocks + stocks;
28            num_barrels := num_barrels + barrels;
29           READ(locks);
30         END; (WHILE locks)
31    READLN;
32     WRITELN('Sales for ',salesman);
33     WRITELN('Locks sold: ', num_locks);
34     WRITELN('Stocks sold: ', num_stocks);
35     WRITELN('Barrels sold: ', num_barrels);
36     sales := lock_price*num_locks + stock_price*num_stocks
             + barrel_price*num_barrels;
37     WRITELN('Total sales: ', sales:8:2);
38    WRITELN;
39    IF  (sales > 1800.0) THEN
40     BEGIN
41       commission := 0.10 * 1000.0;
42        commission := commission + 0.15 * 800.0;
43         commission := commission + 0.20 * (sales-1800.0);
44     END;
45    ELSE IF  (sales > 1000.0) THEN
46     BEGIN
47        commission := 0.10 * 1000.0;
48         commission := commission + 0.15*(sales - 1000.0);
49       END
```

```
50        ELSE commission := 0.10 * sales;
51         WRITELN('Commission is $',commission:6:2);
52      END;  (FOR salesman)
53    END.  {program lock_stock_and-barrel}
```

The DD-Paths in this program are given in Table 1, and the DD-Path graph is shown in Figure 10.2. Tables 2 and 3 list the define and usage nodes for five variables in the commission problem. We use this information in conjunction with the program graph in Figure 10.1 to identify various definition-use and definition-clear paths. It's a judgment call whether or not non-executable statements such as constant (CONST) and variable (VAR) declaration statements should be considered as defining nodes. Technically, these only define memory space (the CONST declaration creates a compiler-produced initial value). Such nodes aren't very interesting when we follow what happens along their du-paths, but if there is something wrong, it's usually helpful to include them. Take your pick. We will refer to the various paths as sequences of node numbers. First, let's look at the du-paths for the variable stocks. We have DEF(stocks, 25) and USE(stocks, 27), so the path <25, 27> is a du-path wrt (with respect to) stocks. Since there are no other defining nodes for stocks, this path is also definition-clear.

Two defining and two usage nodes make the locks variable more interesting: we have DEF(locks, 22), DEF(locks, 29), USE(locks, 23), and USE(locks, 26). These yield four du-paths:

p1 = <22, 23>

p2 = <22, 23, 24, 25, 26>

p3 = <29, 30, 23>

p4 = <29, 30, 23, 24, 25, 26>

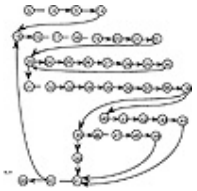


Figure 3.1 Program Graph of the Commission Program

Table 1 DD-Paths in Figure 10.1 **DD-Path**

	Nodes
1	14
2	15-22
3	23
4	24-30

5	31-39
6	40-44
7	45
8	46-49
9	50
10	51,52
11	53

Du-paths p1 and p2 refer to the priming value of locks which is read at node 22: locks has a predicate use in the While statement (node 23), and if the condition is true (as in path p2), a computation use at statement 26. The other two du-paths start near the end of the While loop and occur when the loop repeats. If we "extended" paths p1 and p3 to include node 31,

p1' = <22, 23, 31>

p3' = <29, 30, 23, 31>

then the paths p1', p2, p3', and p4 form a very complete set of test cases for the While-loop: bypass the loop, begin the loop, repeat the loop, and exit the loop. All of these du-paths are definition-clear.

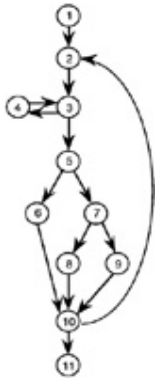


Figure 3.2 DD-Path Graph of the Commission Program

Table 2 Define/Use Information for locks, stocks, and num_locks **Variable**

	Defined at	**Used at**	**Comment**
locks	9		(to compiler)
locks	22		READ

locks		23	predicate use
locks		26	computation use
locks	29		READ
stocks	9		(to compiler)
stocks	25		READ
stocks		27	computation use
num_locks	9		(to compiler)
num_locks	19		assignment
num_locks	26		assignment
num_locks		26	computation use
num_locks		33	WRITE
num_locks		36	computation use

Table 3 Define/Use Information for Sales and Commission **Variable**

	Defined at	**Used at**	**Comment**
sales	11		(to compiler)
sales	36		assignment
sales		37	WRITE
sales		39	predicate use
sales		43	computation use
sales		45	predicate use
sales		48	computation use
sales		50	computation use
commission	11		(to compiler)
commission	41		assignment

commission	42		assignment
commission		42	computation use
commission	43		assignment
commission		43	computation use
commission	47		assignment
commission	48		assignment
commission		48	computation use
commission	50		assignment
commission		51	WRITE

The du-paths for num_locks will lead us to typical test cases for computations. With two defining nodes (DEF(num_locks, 19) and DEF(num_locks, 26)) and three usage nodes (USE(num_locks, 26), USE(num_locks, 33), USE(num_locks, 36)), we might expect six du-paths. Let's take a closer look.

Path p5 = <19, 20, 21, 22, 23, 24, 25, 26> is a du-path in which the initial value (0) has a computation use. This path is definition-clear. The next path is problematic:

p6 = <19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33>

We have ignored the possible repetition of the While-loop. We could highlight this by noting that the subpath <26, 27, 28, 29, 30, 22, 23, 24, 25> might be traversed several times. Ignoring this for now, we still have a du-path that fails to be definition-clear. If there is a problem with the value of num_locks at node 33 (the WRITE statement), we should look at the intervening DEF(num_locks, 26) node.

The next path contains p6; we can show this by using a path name in place of its corresponding node sequence:

p7 = <19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36>

p7 = < p6, 34, 35, 36>

Du-path p7 is not definition-clear because it includes node 26.

Subpaths that begin with node 26 (an assignment statement) are interesting. The first, <26, 26>, seems degenerate. If we "expanded" it into machine code, we would be able to separate the define and usage portions. We will disallow these as du-paths. Technically, the usage on the right-hand

side of the assignment refers to a value defined at node 19, (see path p5). The remaining two du-paths are both subpaths of p7:

p8 = <26, 27, 28, 29, 30, 31, 32, 33>

p9 = <26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36>

Both of these are definition-clear, and both have the loop iteration problem we discussed before.

Since there is only one defining node for sales, all the du-paths wrt sales must be definition-clear. They are interesting because they illustrate predicate and computation uses. The first three du-paths are easy:

p10 = <36, 37>

p11 = <36, 37, 38, 39>

p12 = <36, 37, 38, 39, 40, 41, 42, 43>

Notice that p12 is a definition-clear path with three usage nodes; it also contains paths p10 and p11. If we were testing with p12, we know we would also have covered the other two paths. We will revisit this toward the end of the chapter.

The IF, ELSE IF logic in statements 39 through 50 highlights an ambiguity in the original research. There are two choices for du-paths that begin with path p11: the static choice is the path <36, 37, 38, 39, 40, 41, 42, 43>, the dynamic choice is the path <36, 37, 38, 39, 45>. Here we will use the dynamic view, so the remaining du-paths for sales are

p13 = <36, 37, 38, 39, 45, 46, 47, 48>

p14 = <36, 37, 38, 39, 45, 50>

Note that the dynamic view is very compatible with the kind of thinking we used for DD-Paths. If you have followed this discussion carefully, you are probably dreading the stuff on du-paths wrt commission. You're right -- it's time for a change of pace. In statements 39 through 51, the calculation of commission is controlled by ranges of the variable sales. Statements 41 to 43 build up the value of commission by using the memory location to hold intermediate values. This is a common programming practice, and it is desirable because it shows how the final value is computed. (We could replace these lines with the statement "commission := 220 + 0.20* (sales -1800)", where 220 is the value of 0.10*1000 + 0.15*800, but this would be hard for a maintainer to understand.) The "built-up" version uses intermediate values, and these will appear as define and usage nodes in the du-path analysis. Since we decided to disallow du-paths from assignment statements like 41 and 42, we'll just consider du-paths that begin with the three "real" defining nodes: DEF(commission, 43), DEF(commission, 48), and DEF(commission, 50). There is only one usage node, USE(commission, 51).

We have another ambiguity. The static view results in one du-path:

<43, 44, 45, 48, 50, 51>

This path contains the three definitions of commission, so it is not definition-clear. The dynamic view results in three paths:

p15 = <43, 51>

p16 = <48, 51>

p17 = <50, 51>

Again, the dynamic view is preferable; it also results in definition-clear paths. (A sharp tester might ask how we would ever execute the "path" <43, 44, 45, 48, 50, 51>. We cannot.) The full set of du-paths in the problem is given in Table 4 (they are renumbered).

3.6.3 Du-path Test Coverage Metrics

The whole point of analyzing a program as in the previous section is to define a set of test coverage metrics known as the Rapps-Weyuker data flow metrics [Rapps 85]. The first three of these are equivalent to three of E. F. Miller's metrics in Chapter 9: All-Paths, All-Edges, and All-Nodes. The others presume that define and usage nodes have been identified for all program variables, and that du-paths have been identified with respect to each variable. In the following definitions, T is a set of (sub)paths in the program graph G(P) of a program P, with the set V of variables.

Definition

The set T satisfies the ***All-Defs*** criterion for the program P iff for every variable v V, T contains definition clear (sub)paths from every defining node of v to a use of v.

Definition

The set T satisfies the ***All-Uses*** criterion for the program P iff for every variable v V, T contains definition-clear (sub)paths from every defining node of v to every use of v, and to the successor node of each USE(v,n).

Definition

The set T satisfies the ***All-P-Uses /Some C-Uses*** criterion for the program P iff for every variable v V, T contains definition-clear (sub)paths from every defining node of v to every predicate use of v, and if a definition of v has no P-uses, there is a definition-clear path to at least one computation use.

Table 4 Du-Paths in Figure 10.1 **Du-Path**

	Variable	**Def Node**	**Use Node**
1	locks	22	23

2	locks	22	26
3	locks	29	23
4	locks	29	26
5	stocks	25	27
6	barrels	25	28
7	num_locks	19	26
8	num_locks	19	33
9	num_locks	19	36
10	num_locks	26	33
11	num_locks	26	36
12	num_stocks	20	27
13	num_stocks	20	34
14	num_stocks	20	36
15	num_stocks	27	34
16	num_stocks	27	36
17	num_barrels	21	28
18	num_stocks	21	35
19	num_stocks	21	36
20	num_stocks	28	35
21	num_stocks	28	36
22	sales	36	37
23	sales	36	39
24	sales	36	43
25	sales	36	45

26	sales	36	48
27	sales	36	50
28	commission	41	42
29	commission	42	43
30	commission	43	51
31	commission	47	48
32	commission	48	51
33	commission	50	51

Definition

The set T satisfies the ***All-C-Uses /Some P-Uses*** criterion for the program P iff for every variable v V, T contains definition-clear (sub)paths from every defining node of v to every computation use of v, and if a definition of v has no C-uses, there is a definition-clear path to at least one predicate use.

Definition

The set T satisfies the ***All-DU-paths*** criterion for the program P iff for every variable v V, T contains definition-clear (sub)paths from every defining node of v to every use of v, and to the successor node of each USE(v,n), and that these paths are either single loop traversals, or they are cycle free.

These test coverage metrics have several set-theory based relationships, which are referred to as "subsumption" in [Rapps 85]. When one test coverage metric subsumes another, a set of test cases that attains coverage in terms of the first metric necessarily attains coverage with respect to the subsumed metric. These relationships are shown in Figure 3.3.

We now have a more refined view of structural testing possibilities between the extremes of the (typically unattainable) All-Paths metric, and the generally accepted minimum, All-Edges.

Figure 3.3 Rapps/Weyuker Hierarchy of Data Flow Coverage Metrics

Since several du-paths are present in a full program execution path (traversed by a test case), the higher forms of coverage metrics don't always imply significantly higher numbers of test cases. In our continuing example, the three decision table functional test cases cover all the DD-Paths (see Table 5) and most of the du-paths (see Table 6). The missing du-paths (8, 9, 14, 18, 19) are all traversed by a test case in which nothing is sold (i.e., the first value of locks is -1).

3.7 Slice-Based Testing

Program slices have surfaced and submerged in software engineering literature since the early 1980s. They were originally proposed in [Weiser 85], used as an approach to software maintenance in [Gallagher 91], and most recently used to quantify functional cohesion in [Bieman 94]. Part of this versatility is due to the natural, intuitively clear intent of the program slice concept. Informally, a program slice is a set of program statements that contribute to, or affect a value for a variable at some point in the program. This notion of slice corresponds to other disciplines as well. We might study history in terms of slices: US history, European history, Russian history, Far East history, Roman history, and so on. The way such historical slices interact turns out to be very analogous to the way program slices interact.

Table 5 DD-Path Coverage of Decision Table Functional Test Cases **Case**

	locks	**stocks**	**barrels**	**sales**	**commission**	**DD-Paths**
1	5	5	5	500	50	1-5, 7, 9, 10, 11
2	15	15	15	1500	175	1-5, 7, 8, 10, 11
3	25	25	25	2500	360	1-5, 9, 10, 11

Table 6 Du-Path Coverage of Decision Table Functional Test Cases **Du-Path**

	Case 1	**Case 2**	**Case 3**
1	X	X	X
2	X	X	X
3	X	X	X
4	X	X	X
5	X	X	X
6	X	X	X
7	X	X	X

8			
9			
10	X	X	X
11	X	X	X
12	X	X	X
13	X	X	X
14			
15	X	X	X
16	X	X	X
17	X	X	X
18			
19			
20	X	X	X
21	X	X	X
22	X	X	X
23	X	X	X
24			X
25	X	X	
26		X	
27	X		
28			X
29			X
30			X
31		X	

32		X	
33	X		

We'll start by growing our working definition of a program slice. We continue with the notation we used for define-use paths: a program P that has a program graph G(P), and a set of program variables V. The first try refines the definition in [Gallagher 91] to allow nodes in P(G) to refer to statement fragments.

Definition

Given a program P, and a set V of variables in P, a ***slice on the variable set V at statement n,*** written S(V,n), is the set of all statements in P that contribute to the values of variables in V.

Listing elements of a slice S(V,n) will be cumbersome, because the elements are program statement fragments. Since it is much simpler to list fragment numbers in P(G), we make the following trivial change (it keeps the set theory purists happy):

Definition

Given a program P, and a program graph G(P) in which statements and statement fragments are numbered, and a set V of variables in P, the ***slice on the variable set V at statement fragment n,*** written S(V,n), is the set node numbers of all statement fragments in P prior to n that contribute to the values of variables in V at statement fragment n.

The idea of slices is to separate a program into components that have some useful meaning. First, we need to explain two parts of the definition. Here we mean "prior to" in the dynamic sense, so a slice captures the execution time behavior of a program with respect to the variable(s) in the slice. Eventually, we will develop a lattice (a directed, acyclic graph) of slices, in which nodes are slices, and edges correspond to the subset relationship.

The "contribute" part is more complex. In a sense, declarative statements (such as CONST and TYPE) have an effect on the value of a variable. A CONST definition sets a value that can never be changed by a definition node, and the difference between INTEGER and REAL variables can be a source of trouble. One resolution might be to simply exclude all non-executable statements. We will include CONST declarations in slices. The notion of contribution is partially clarified by the predicate (P-use) and computation (C-use) usage distinction of [Rapps 85], but we need to refine these forms of variable usage. Specifically, the USE relationship pertains to five forms of usage:

P-use used in a predicate (decision)

C-use used in computation

O-use used for output

L-use	used for location (pointers, subscripts)
I-use	iteration (internal counters, loop indices)

While we're at it, we identify two forms of definition nodes:

I-def	defined by input
A-def	defined by assignment

For now, presume that the slice S(V, n) is a slice on one variable, that is, the set V consists of a single variable, v. If statement fragment n is a defining node for v, then n is included in the slice. If statement fragment n is a usage node for v, then n is not included in the slice. P-uses and C-uses of other variables (not the v in the slice set V) are included to the extent that their execution affects the value of the variable v. As a guideline, if the value of v is the same whether a statement fragment is included or excluded, exclude the statement fragment. L-use and I-use variables are typically invisible outside their modules, but this hardly precludes the problems such variables often create. Another judgment call: here (with some peril) we choose to exclude these from the intent of "contribute". Thus O-use, L-use, and I-use nodes are excluded from slices..

Example

The commission problem is used in this book because it contains interesting data flow properties, and these are not present in the Triangle problem (or in NextDate). Follow these examples while looking at the source code for the commission problem that we used to analyze in terms of define-use paths.

S_1: S(salesman, 17) = {17}

S_2: S(salesman, 18) = {17}

S_3: S(salesman, 32) = {17}

The salesman variable is the simplest case in the program. It has one defining node (an I-def at node 17). It also occurs at nodes 18 and 22, both times as an output variable (O-use), hence it is not included in slices S_2 and S_3. Both <17, 18> and <17, 32> are definition-clear du-paths.

Slices on the locks variable show why it is potentially fault-prone. It has a P-use at node 23 and a C-use at node 26, and has two definitions, the I-defs at nodes 22 and 24.

S_4: S(locks, 22) = {22}

S_5: S(locks, 23) = {22, 23, 24, 29, 30}

S_6: S(locks, 26) = {22, 23, 24, 29, 30}

S_7: S(locks, 29) = {29}

The slices for stocks and barrels are boring. Both are short, definition-clear paths contained entirely within a loop, so they are not affected by iterations of the loop. (Think of the loop body as a DD-Path.)

S_8: S(stocks, 25) = {22, 23, 24, 25, 29, 30}

S_9: S(stocks, 27) = {22, 23, 24, 25, 29, 30}

S_{10}: S(barrels, 25) = {22, 23, 24, 25, 29, 30}

S_{11}: S(barrels, 28) = {22, 23, 24, 25, 29, 30}

The next four slices illustrate how repetition appears in slices. Node 19 is an A-def for num_locks, and node 26 contains both an A-def and a C-use. The remaining nodes in S_{13} (22, 23, 24, 29, and 30) pertain to the While-loop controlled by locks. Slices S_{13}, S_{14}, and S_{15} are equal because nodes 33 and 36 are, respectively, an O-use and a C-use of num_locks.

S_{12}: S(num_locks, 19) = {19}

S_{13}: S(num_locks, 26) = {19, 22, 23, 24, 26, 29, 30}

S_{14}: S(num_locks, 33) = {19, 22, 23, 24, 26, 29, 30}

S_{15}: S(num_locks, 36) = {19, 22, 23, 24, 26, 29, 30}

The slices on num_stocks and num_barrels are quite similar. They are initialized by A-defs at nodes 20 and 21, and then are redefined by A-defs at nodes 27 and 28. Again, the remaining nodes (22, 23, 24, 29, and 30) pertain to the While-loop controlled by locks.

S_{16}: S(num_stocks, 20) = {20}

S_{17}: S(num_stocks, 27) = {20, 22, 23, 24, 25, 27, 29, 30}

S_{18}: S(num_stocks, 34) = {20, 22, 23, 24, 25, 27, 29, 30}

S_{19}: S(num_stocks, 36) = {20, 22, 23, 24, 25, 27, 29, 30}

S_{20}: S(num_barrels, 21) = {21}

S_{21}: S(num_barrels, 28) = {21, 22, 23, 24, 25, 28, 29, 30}

S_{22}: S(num_barrels, 35) = {21, 22, 23, 24, 25, 28, 29, 30}

S_{23}: S(num_barrels, 36) = {21, 22, 23, 24, 25, 28, 29, 30}

The next three slices demonstrate our convention regarding compiler-defined values.

S_{24}: S(lock_price, 36) = {3}

S_{25}: S(stock_price, 36) = {4}

S_{26}: S(barrel_price, 36) = {5}

The slices on sales and commission are the interesting ones. There is only one defining node for sales, the A-def at node 36. The remaining slices on sales show the P-uses, C-uses, and the O-use in definition-clear paths.

S_{27}: S(sales, 36) = {3, 4, 5, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 36}

S_{28}: S(sales, 37) = {3, 4, 5, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 36}

S_{29}: S(sales, 39) = {3, 4, 5, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 36}

S_{30}: S(sales, 43) = {3, 4, 5, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 36}

S_{31}: S(sales, 45) = {3, 4, 5, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 36}

S_{32}: S(sales, 48) = {3, 4, 5, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 36}

S_{33}: S(sales, 50) = {3, 4, 5, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 36}

Think about slice S27 in terms of its "components", the slices on the C-use variables. We can write $S_{27} = S_{24} \cup S_{25} \cup S_{26} \cup S_{13} \cup S_{17} \cup S_{21}$, where the values of the six C-use variables at node 36 are defined by the six slices joined together by the union operation. Notice how the formalism corresponds to our intuition: if the value of sales is wrong, we first look at how it is computed, and if this is OK, we check how the components are computed.

Everything comes together (literally) with the slices on commission. There are six A-def nodes for commission (corresponding to the six du-paths we identified earlier). Three computations of commission are controlled by P-uses of sales in the IF, ELSE IF logic. This yields three "paths" of slices that compute commission. (See Figure 10.4.)

S_{34}: S(commission, 41) = {41}

S_{35}: S(commission, 42) = {41, 42}

S_{36}: S(commission, 43) = {3, 4, 5, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 36, 41, 42, 43}

S_{37}: S(commission, 47) = {47}

S_{38}: S(commission, 48) = {3, 4, 5, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 36, 47, 48}

S_{39}: S(commission, 50) = {3, 4, 5, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 36, 50}

Whichever computation is taken, all come together in the last slice.

S_{40}: S(commission, 51) = {3, 4, 5, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 36, 41, 42, 43, 47, 48, 50}

The slice information improves our insight. Look at the lattice in Figure 10.4; it is a directed acyclic graph in which slices are nodes, and an edge represents the proper subset relationship.

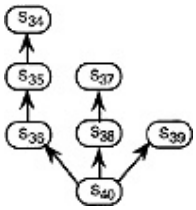


Figure 3.4 Lattice of Slices on Commission

This lattice is drawn so that the position of the slice nodes roughly corresponds with their position in the source code. The definition-clear paths <43, 51>, <48, 51>, and <50,51> correspond to the edges that show slices S_{36}, S_{38}, and S_{39} are subsets of slice S_{40}. Figure 3.5 shows a lattice of slices for the entire program. Some slices (those that are identical to others) have been deleted for clarity. All are listed in Table 7, along with slice-specific test objectives that are appropriate.

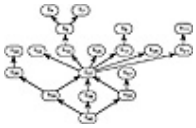


Figure 3.5 Lattice of Slices in the Commission Program

There is a natural hybrid between slice-based testing and functional testing. Since slices are defined with respect to variables, slices correspond to a functional decomposition of a program. Because the subfunctions are smaller, corresponding sets of functional test cases will be more reasonable.

10.2.2 Style and Technique

When we analyze a program in terms of "interesting" slices, we can focus on parts of interest while disregarding unrelated parts. We couldn't do this with du-paths — they are sequences that include statements and variables that may not be of interest. Before discussing some analytic techniques, we'll first look at "good style". We could have built these stylistic precepts into the definitions, but then the definitions become even more cumbersome.

Table 7 Test Objectives for Slices in Figure 10.5 **Equivalent Slices**

	Test Objective
1, 2, 3	salesman read and written correctly
4	locks read correctly
5, 6	locks sentinel correct
7	additional locks read correctly
8, 9 (10, 11)	iterative read of stocks correct

(8, 9) 10, 11	iterative read of barrels correct
12	num_locks initialized correctly
13, 14, 15	num_locks computed correctly
16	num_stocks initialized correctly
17, 18, 19	num_stocks computed correctly
20	num_barrels initialized correctly
21, 22, 23	num_barrels computed correctly
24	lock_price constant definition correct
25	stock_price constant definition correct
26	barrel_price constant definition correct
27, 28, 29, 30, 31, 32, 33	sales computed correctly
34	commission on first 1000 correct for sales > 1800
35	commission on next 800 correct for sales > 1800
36	commission on excess over 1800 correct for sales > 1800
37	commission on first 1000 correct for 1000 <sales <1800
38	commission on excess over 1000 correct for 1000 <sales <1800
39	commission on sales < 1000 correct
40	commission written correctly

1. Never make a slice S(V, n) for which variables v of V do not appear in statement fragment n. This possibility is permitted by the definition of a slice, but it is bad practice. As an example, suppose we defined a slice on the locks variable at node 27. Defining such slices necessitates tracking the values of all variables at all points in the program.

2. Make slices on one variable. The set V in slice S(V,n) can contain several variables, and sometimes such slices are useful. The slice S(V, 36) where

V= { num_locks, num_stocks, num_barrels }

contains all the elements of the slice S({sales}, 36) except the CONST declarations and statement 36. Since these two slices are so similar, why define the one in terms of C-uses?

3. Make slices for all A-def nodes. When a variable is computed by an assignment statement, a slice on the variable at that statement will include (portions of) all du-paths of the variables used in the computation. Slice S({sales}, 36) is a good example of an A-def slice.

4. Make slices for P-use nodes. When a variable is used in a predicate, the slice on that variable at the decision statement shows how the predicate variable got its value. This is very useful in decision-intensive programs like the Triangle program and NextDate.

5. Slices on non-P-use usage nodes aren't very interesting. We discussed C-use slices in point 2, where we saw they were very redundant with the A-def slice. Slices on O-use variables can always be expressed as unions of slices on all the A-defs (and I-defs) of the O-use variable. Slices on I-use and O-use variables are useful during debugging, but if they are mandated for all testing, the test effort is dramatically increased.

6. Consider making slices compilable. Nothing in the definition of a slice requires that the set of statements is compilable, but if we make this choice, it means that a set of compiler directive and declarative statements is a subset of every slice. As an example, the slice S_5, which is S(locks, 23) = {22, 23, 24, 29, 30}, contains the statements

```
22  READ(locks);
23  WHILE locks <> -1 DO
24    BEGIN
29      READ(locks);
30    END; {WHILE locks}
```

If we add statements 1-14 and 53, we have the compilable slice shown here:

```
 1  program lock_stock_and_barrel
 2  const
 3     lock_price = 45.0;
 4     stock_price = 30.0;
 5     barrel_price = 25.0;
 6  type
 7     STRING_30 = string[30]; {Salesman's Name}
 8  var
 9     locks, stocks, barrels, num_locks, num_stocks,
10      num_barrels, salesman_index, order_index : INTEGER;
11      sales, commission : REAL;
12      salesman : STRING_30;
13
14  BEGIN {program lock_stock_and_barrel}
22      READ(locks);
23      WHILE locks <> -1 DO
24        BEGIN
29          READ(locks);
30        END; {WHILE locks}
53  END. {program lock_stock_and_barrel}
```

If we added this same set of statements to all the slices we made for the commission program, our lattices remain undisturbed, but each slice is separately compilable (and therefore executable). In

the first chapter, we suggested that good testing practices lead to better programming practices. Here we have a good example. Think about developing programs in terms of compilable slices. If we did this, we could code a slice and immediately test it. We can then code and test other slices, and them merge them (Gallagher calls this "slice splicing") into a pretty solid program. Try coding the commission program this way.

Guidelines and Observations

Dataflow testing is clearly indicated for programs that are computationally intensive. As a corollary, in control intensive programs, if control variables are computed (P-uses), dataflow testing is also indicated. The definitions we made for define/use paths and slices give us very precise ways to describe parts of a program that we would like to test. There are academic tools that support these definitions, but they haven't migrated to the commercial marketplace. Some pieces are there; you can find programming language compilers that provide on-screen highlighting of slices, and most debugging tools let you "watch" certain variables as you step through a program execution. Here are some tidbits that may prove helpful to you, particularly when you have a difficult module to test.

1. Slices don't map nicely into test cases (because the other, non-related code is still in an executable path). On the other hand, they are a handy way to eliminate interaction among variables. Use the slice composition approach to re-develop difficult sections of code, and these slices before you splice (compose) them with other slices.

2. Relative complements of slices yield a "diagnostic" capability. The relative complement of a set B with respect to another set A is the set of all elements of A that are not elements of B. It is denoted as A -B. Consider the relative complement set S(commission, 48) - S(sales, 35):

S(commission, 48) = {3, 4, 5,36,18,19, 20, 23, 24, 25, 26, 27, 34, 38, 39, 40, 44,45,47}

S(sales, 35) = {3, 4, 5, 36, 18, 19, 20, 23, 24, 25, 26, 27}

S(commission, 48) - S(sales, 35) = {34, 38, 39, 40, 44,45,47}

If there is a problem with commission at line 48, we can divide the program into two parts, the computation of sales at line 34, and the computation of commission between lines 35 and 48. If sales is OK at line 34, the problem must lie in the relative complement; if not, the problem may be in either portion.

3. There is a many-to-many relationship between slices and DD-Paths: statements in one slice may be in several DD-Paths, and statements in one DD-Path may be in several slices. Well-chosen relative complements of slices can be identical to DD-Paths. For example, consider S(commission, 40) -S(commission, 37).

4. If you develop a lattice of slices, it's convenient to postulate a slice on the very first statement. This way, the lattice of slices always terminates in one root node. Show equal slices with a two-way arrow.

5. Slices exhibit define/reference information. Consider the following slices on num_locks:

S(num_locks, 17) = φ

S(num_locks, 24) = {17, 20, 27?}

S(num_locks, 31) = {17, 20, 24, 27}

S(num_locks, 34) = {17, 20, 24, 27}

S(num_locks, 17) is the first definition of num_locks.

S(num_locks, 24) - S(num_locks, 17) is a definition-clear, define reference path.

When slices are equal, the corresponding paths are definition-clear.

EXERCISES

1. Think about the static versus dynamic ambiguity of du-paths in terms of DD-Paths. As a start, what DD-Paths are found in the du-paths p12, p13, and p14 for sales?

2. Try to merge some of the DD-Path based test coverage metrics into the Rapps/Weyuker hierarchy shown in Figure 10.2.

3. Express slice S_{40} as the union of other pertinent slices.

4. Find the following program slices:

a. S(commission, 48)

b. S(sales, 35)

c. S(commission, 40), S(commission, 39), S(commission, 38)

d. S(num_locks, 34)

e. S(num_stocks, 34)

f. S(num_barrels, 34)

5. Find the definition-clear paths (with respect to SALES) from line 35 to: 36, 40, 42, 45, 47.

6. Make a lattice of "interesting" slices. As a minimum, include the ones from question 4.

UNIT 4

LEVELS OF TESTING, INTEGRATION TESTING

4.1 Traditional View of Testing Levels

The traditional model of software development is the Waterfall model, which is drawn as a V in Figure 4.1 to emphasize the basic levels of testing. In this view, information produced in one of the development phases constitutes the basis for test case identification at that level. Nothing controversial here: we certainly would hope that system test cases are somehow correlated with the requirements specification, and that unit test cases are derived from the detailed design of the unit. Two observations: there is a clear presumption of functional testing here, and there is an implied "bottom-up" testing order.

Figure 4.1 The Waterfall Life Cycle

Of the three traditional levels of testing (unit, integration, and system), unit testing is best understood. The testing theory and techniques we worked through in Parts I and II are directly applicable to unit testing. System testing is understood better than integration testing, but both need clarification. The bottom-up approach sheds some insight: test the individual components, and then integrate these into subsystems until the entire system is tested. System testing should be something that the customer (or user) understands, and it often borders on customer acceptance testing. Generally, system testing is functional rather than structural; this is mostly due to the absence of a structural basis for system test cases. In the traditional view, integration testing is what's left over: it's not unit testing, and it's not system testing. Most of the usual discussions on integration testing center on the order in which units are integrated: top-down, bottom-up, or the "big bang" (everything at once). Of the three levels, integration is the least well understood; we'll do something about that in this chapter and the next. The waterfall model is closely associated with top-down development and design by functional decomposition. The end result of preliminary design is a functional decomposition of the entire system into a treelike structure of functional components. Figure 4.2 contains a partial functional decomposition of our ATM system. With this decomposition. top-down integration would begin with the main program, checking the calls to the three next level procedures (Terminal I/O, ManageSessions, and ConductTransactions). Following the tree, the ManageSessions procedure would be tested, and then the CardEntry, PIN Entry, and SelectTransaction procedures. In each case, the actual code for lower level units is replaced by a stub, which is a throw-away piece of code that takes the place of the actual code. Bottom-up integration would be the opposite sequence, starting with the CardEntry, PIN Entry, and SelectTransaction procedures, and working up toward the main program. In bottom-up integration, units at higher levels are replaced by drivers (another form of throw-away code) that emulate the procedure calls. The big bang approach simply puts all the units together at once, with no stubs or drivers. Whichever approach is taken, the goal of traditional integration testing is to integrate

previously tested units with respect to the functional decomposition tree. While this describes integration testing as a process, discussions of this type offer little information about the goals or techniques. Before addressing these (real) issues, we need to understand the consequences of the alternative life cycle models.

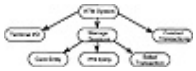


Figure 4.2 Partial Functional Decomposition of the ATM System

4.2 Alternative Life Cycle Models

Since the early 1980s, practitioners have devised alternatives in response to shortcomings of the traditional waterfall model of software development [Agresti 86]. Common to all of these alternatives is the shift away from the functional decomposition to an emphasis on composition. Decomposition is a perfect fit both to the top-down progression of the waterfall model and to the bottom-up testing order. One of the major weaknesses of waterfall development cited by [Agresti 86] is the over-reliance on this whole paradigm. Functional decomposition can only be well done when the system is completely understood, and it promotes analysis to the near exclusion of synthesis. The result is a very long separation between requirements specification and a completed system, and during this interval, there is no opportunity for feedback from the customer. Composition, on the other hand, is closer the way people work: start with something known and understood, then add to it gradually, and maybe remove undesired portions. There is a very nice analogy with positive and negative sculpture. In negative sculpture, work proceeds by removing unwanted material, as in the mathematician's view of sculpting Michelangelo's David: start with a piece of marble, and simply chip away all non-David. Positive sculpture is often done with a medium like wax. The central shape is approximated, and then wax is either added or removed until the desired shape is attained. Think about the consequences of a mistake: with negative sculpture, the whole work must be thrown away, and restarted. (There is a museum in Florence, Italy that contains half a dozen such false starts to The David.) With positive sculpture, the erroneous part is simply removed and replaced. The centrality of composition in the alternative models has a major implication for integration testing.

4.2.1 Waterfall Spin-offs

There are three mainline derivatives of the waterfall model: incremental development, evolutionary development, and the Spiral model [Boehm 88]. Each of these involves a series of increments or builds, as shown in Figure 4.3. Within a build, the normal waterfall phases from detailed design through testing occur, with one important difference: system testing is split into two steps, regression and progression testing.

Figure 4.3 Life Cycle with a Build Sequence

It is important to keep preliminary design as an integral phase, rather than to try to amortize such high level design across a series of builds. (To do so usually results in unfortunate consequences of design choices made during the early builds that are regrettable in later builds.) Since preliminary design remains a separate step, we are tempted to conclude that integration testing is unaffected in the spin-off models. To some extent this is true: the main impact of the series of builds is that regression testing becomes necessary. The goal of regression testing is to assure that things that worked correctly in the previous build still work with the newly added code. Progression testing assumes that regression testing was successful, and that the new functionality can be tested. (We like to think that the addition of new code represents progress, not a regression.) Regression testing is an absolute necessity in a series of builds because of the well-known "ripple effect" of changes to an existing system. (The industrial average is that one change in five introduces a new fault.)

The differences among the three spin-off models are due to how the builds are identified. In incremental development, the motivation for separate builds is usually to level off the staff profile. With pure waterfall development, there can be a huge bulge of personnel for the phases from detailed design through unit testing. Most organizations cannot support such rapid staff fluctuations, so the system is divided into builds that can be supported by existing personnel. In evolutionary development, there is still the presumption of a build sequence, but only the first build is defined. Based on it, later builds are identified, usually in response to priorities set by the customer/user, so the system evolves to meet the changing needs of the user. The spiral model is a combination of rapid prototyping and evolutionary development, in which a build is defined first in terms of rapid prototyping, and then is subjected to a go/no go decision based on technology-related risk factors. From this we see that keeping preliminary design as an integral step is difficult for the evolutionary and spiral models. To the extent that this cannot be maintained as an integral activity, integration testing is negatively affected.

Because a build is a set of deliverable end-user functionality, one advantage of these spin-off models is that all three yield earlier synthesis. This also results in earlier customer feedback, so two of the deficiencies of waterfall development are mitigated.

4.2.2 Specification Based Models

Two other variations are responses to the "complete understanding" problem. (Recall that functional decomposition is successful only when the system is completely understood.) When systems are not fully understood (by either the customer or the developer), functional decomposition is perilous at best. The rapid prototyping life cycle (Figure 12.4) deals with this by drastically reducing the specification-to-customer feedback loop to produce very early synthesis. Rather than build a final system, a "quick and dirty" prototype is built and then used to elicit customer feedback. Depending on the feedback, more prototyping cycles may occur. Once the developer and the customer agree that a prototype represents the desired system, the developer goes ahead and builds to a correct specification. At this point, any of the waterfall spin-offs might also be used.

Figure 4.4 Rapid Prototyping Life Cycle

Rapid prototyping has interesting implications for system testing. Where are the requirements? Is the last prototype the specification? How are system test cases traced back to the prototype? One good answer to questions such as these is to use the prototyping cycle(s) as information gathering activities, and then produce a requirements specification in a more traditional manner. Another possibility is to capture what the customer does with the prototype(s), define these as scenarios that are important to the customer, and then use these as system test cases. The main contribution of rapid prototyping is that it brings the operational (or behavioral) viewpoint to the requirements specification phase. Usually, requirements specification techniques emphasize the structure of a system, not its behavior. This is unfortunate, because most customers don't care about the structure, and they do care about the behavior.

Executable specifications (Figure 4.5) are an extension of the rapid prototyping concept. With this approach, the requirements are specified in an executable format (such as finite state machines or Petri nets). The customer then executes the specification to observe the intended system behavior, and provides feedback as in the rapid prototyping model.

Figure 4.5 Executable Specification

One big difference is that the requirements specification document is explicit, as opposed to a prototype. More important, it is often a mechanical process to derive system test cases from an executable specification. Although more work is required to develop an executable specification, this is partially offset by the reduced effort to generate system test cases. Another important distinction: when system testing is based on an executable specification, we have a form of structural testing at the system level.

4.2.3 An Object-Oriented Life Cycle Model

When software is developed with an object orientation, none of our life cycle models fit very well. The main reasons: the object orientation is highly compositional in nature, and there is dense interaction among the construction phases of object-oriented analysis, object-oriented design, and object-oriented programming. We could show this with pronounced feedback loops among waterfall phases, but the fountain model [Henderson-Sellers 90] is a much more appropriate

metaphor. In the fountain model, (see Figure 12.6) the foundation is the requirements analysis of real world systems.

Figure 4.6 Fountain Model of Object-Oriented Software Development

As the object-oriented paradigm proceeds, details "bubble up" through specification, design, and coding phases, but at each stage, some of the "flow" drops back to the previous phase(s). This model captures the reality of the way people actually work (even with the traditional approaches).

4.3 Formulations of the SATM System

In this and the next three chapters, we will relate our discussion to a higher level example, the Simple Automatic Teller Machine (SATM) system. The version developed here is a revision of that found in [Topper 93]; it is built around the fifteen screens shown in Figure 4.7. This is a greatly reduced system; commercial ATM systems have hundreds of screens and numerous time-outs.

The SATM terminal is sketched in Figure 12.8; in addition to the display screen, there are function buttons B1, B2, and B3, a digit keypad with a cancel key, slots for printer receipts and ATM cards, and doors for deposits and cash withdrawals. The SATM system is described here in two ways: with a structured analysis approach, and with an object-oriented approach. These descriptions are not complete, but they contain detail sufficient to illustrate the testing techniques under discussion.

4.3.1 SATM with Structured Analysis

The structured analysis approach to requirements specification is the most widely used method in the world. It enjoys extensive CASE tool support as well as commercial training, and is described in numerous texts. The technique is based on three complementary models: function, data, and control. Here we use data flow diagrams for the functional models, entity/relationship models for data, and finite state machine models for the control aspect of the SATM system. The functional and data models were drawn with the Deft CASE tool from Sybase Inc. That tool identifies external devices (such as the terminal doors) with lower case letters, and elements of the functional decomposition with numbers (such as 1.5 for the Validate Card function). The open and filled arrowheads on flow arrows signify whether the flow item is simple or compound. The portions of the SATM system shown here pertain generally to the personal identification number (PIN) verification portion of the system.

Figure 4.7 Screens for the SATM System

The Deft CASE tool distinguishes between simple and compound flows, where compound flows may be decomposed into other flows, which may themselves be compound. The graphic appearance of this choice is that simple flows have filled arrowheads, while compound flows have open arrowheads. As an example, the compound flow "screen" has the following decomposition:

screen is comprised of

screen1	welcome
screen2	enter PIN
screen3	wrong PIN
screen4	PIN failed, card retained
screen5	select trans type
screen6	select account type
screen7	enter amount
screen8	insufficient funds
screen9	cannot dispense that amount
screen10	cannot process withdrawals
screen11	take your cash
screen12	cannot process deposits
screen13	put dep envelop in slot
screen14	another transaction?
screen15	Thanks; take card and receipt

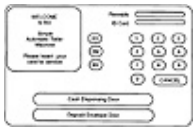


Figure 4.8 The SATM Terminal

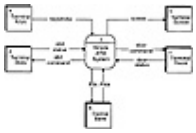


Figure 4.9 Context Diagram of the SATM System

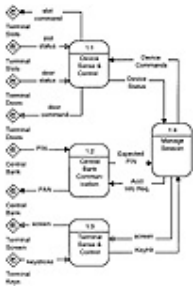


Figure 4.10 Level 1 Dataflow Diagram of the SATM System

Figure 4.11 is an (incomplete) Entity/Relationship diagram of the major data structures in the SATM system: Customers, Accounts, Terminals, and Transactions. Good data modeling practice dictates postulating an entity for each portion of the system that is described by data that is retained (and used by functional components). Among the data the system would need for each customer are the customer's identification and personal account number (PAN); these are encoded into the magnetic strip on the customer's ATM card. We would also want to know information about a customer's account(s), including the account numbers, the balances, the type of account (savings or checking), and the Personal Identification Number (PIN) of the account. At this point, we might ask why the PIN is not associated with the customer, and the PAN with an account. Some design has crept into the specification at this point: if the data were as questioned, a person's ATM card could be used by anyone; as it is, the present separation predisposes a security checking procedure. Part of the E/R model describes relationships among the entities: a customer HAS account(s), a customer conducts transaction(s) in a SESSION, and, independent of customer information, transaction(s) OCCUR at an ATM terminal. The single and double arrowheads signify the singularity or plurality of these relationships: one customer may have several accounts, and may conduct none or several transactions. Many transactions may occur at a terminal, but one transaction never occurs at a multiplicity of terminals.

Figure 4.11 Entity/Relationship Model of the SATM System

The dataflow diagrams and the entity/relationship model contain information that is primarily structural. This is problematic for testers, because test cases are concerned with behavior, not with structure. As a supplement, the functional and data information are linked by a control model; here we use a finite state machine. Control models represent the point at which structure and behavior intersect; as such, they are of special utility to testers.

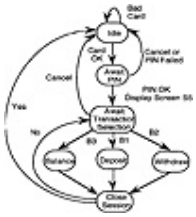


Figure 4.12 Upper Level SATM Finite State Machine

The upper level finite state machine in Figure 4.12 divides the system into states that correspond to stages of customer usage. Other choices are possible, for instance, we might choose states to be screens being displayed (this turns out to be a poor choice). Finite state machines can be hierarchically decomposed in much the same way as dataflow diagrams. The decomposition of the Await PIN state is shown in Figure 4.13. In both of these figures, state transitions are caused either by events at the ATM terminal (such as a keystroke) or by data conditions (such as the recognition that a PIN is correct). When a transition occurs, a corresponding action may also occur. We choose to use screen displays as such actions; this choice will prove to be very handy when we develop system level test cases.

The function, data, and control models are the basis for design activities in the waterfall model (and its spin-offs). During design, some of the original decisions may be revised based on additional insights and more detailed requirements (for example, performance or reliability goals). The end result is a functional decomposition such as the partial one shown in the structure chart in Figure 4.14. Notice that the original first level decomposition into four subsystems is continued: the functionality has beendecomposed to lower levels of detail. Choices such as these are the essence of design, and design is beyond the scope of this book. In practice, testers often have to live with the results of poor design choices.

Figure 4.13 PIN Entry Finite State Machine

If we only use a structure chart to guide integration testing, we miss the fact that some (typically lower level) functions are used in more than one place. Here, for example, the ScreenDriver function is used by several other modules, but it only appears once in the functional decomposition. In the next chapter, we will see that a “call graph” is a much better basis for integration test case identification. We can develop the beginnings of such a call graph from a more detailed view of

portions of the system. To support this, we need a numbered decomposition, and a more detailed view of two of the components.

Here is the functional decomposition carried further in outline form: the numbering scheme preserves the levels of the components in Figure 12.14.

Figure 4.14 A Decomposition Tree for the SATM System

1 SATM System

1.1 Device Sense & Control

1.1.1 Door Sense & Control

1.1.1.1 Get Door Status

1.1.1.2 Control Door

1.1.1.3 Dispense Cash

1.1.2 Slot Sense & Control

1.1.2.1 WatchCardSlot

1.1.2.2 Get Deposit Slot Status

1.1.2.3 Control Card Roller

1.1.2.3 Control Envelope Roller

1.1.2.5 Read Card Strip

1.2 Central Bank Comm.

1.2.1 Get PIN for PAN

1.2.2 Get Account Status

1.2.3 Post Daily Transactions

1.3 Terminal Sense & Control

1.3.1 Screen Driver

1.3.2 Key Sensor

1.4 Manage Session

1.4.1 Validate Card

1.4.2 Validate PIN

1.4.2.1 GetPIN

1.4.3 Close Session

1.4.3.1 New Transaction Request

1.4.3.2 Print Receipt

1.4.3.3 Post Transaction Local

1.4.4 Manage Transaction

1.4.4.1 Get Transaction Type

1.4.4.2 Get Account Type

1.4.4.3 Report Balance

1.4.4.4 Process Deposit

1.4.4.5 Process Withdrawal

As part of the specification and design process, each functional component is normally expanded to show its inputs, outputs, and mechanism. We do this here with pseudo-code (or PDL, for program design language) for three modules. This particular PDL is loosely based on Pascal; the point of any PDL is to communicate, not to develop something that can be compiled. The main program description follows the finite state machine description given in Figure 4.12. States in that diagram are "implemented" with a Case statement.

```
Main Program
State = AwaitCard
CASE State OF
AwaitCard:          ScreenDriver(1, null)
                    WatchCardSlot(CardSlotStatus)
                    WHILE CardSlotStatus is Idle DO
                          WatchCardSlot(CardSlotStatus)
                    ControlCardRoller(accept)
                    ValidateCard(CardOK, PAN)
                    IF CardOK    THEN  State = AwaitPIN
                                 ELSE  ControlCardRoller(eject)
```

```
                                    State = AwaitCard
AwaitPIN:          ValidatePIN(PINok, PAN)
                   IF PINok     THEN  ScreenDriver(2, null)
                                      State = AwaitTrans
                                ELSE  ScreenDriver(4, null)
                                      State = AwaitCard
AwaitTrans:        ManageTransaction
                   State = CloseSession
CloseSession:      IF NewTransactionRequest
                         THEN  State = AwaitTrans
                         ELSE  PrintReceipt
                               PostTransactionLocal
                               CloseSession
                               ControlCardRoller(eject)
                               State = AwaitCard
End, (CASE State)
END. (Main program SATM)
```

The ValidatePIN procedure is based on another finite state machine shown in Figure 4.13, in which states refer to the number of PIN entry attempts.

```
Procedure ValidatePIN(PINok, PAN)
GetPINforPAN(PAN, ExpectedPIN)
Try = First
CASE Try OF
First:       ScreenDriver(2, null)
             GetPIN(EnteredPIN)
             IF EnteredPIN = ExpectedPIN
                   THEN  PINok = TRUE
                         RETURN
                   ELSE  ScreenDriver(3, null)
                         Try = Second
Second:      ScreenDriver(2, null)
             GetPIN(EnteredPIN)
             IF EnteredPIN = ExpectedPIN
                   THEN  PINok = TRUE
                         RETURN
                   ELSE  ScreenDriver(3, null)
                         Try = Third
Third:       ScreenDriver(2, null)
             GetPIN(EnteredPIN)
             IF EnteredPIN = ExpectedPIN
                   THEN  PINok = TRUE
                         RETURN
                   ELSE  ScreenDriver(4, null)
                         PINok = FALSE
END, (CASE Try)
END.  (Procedure ValidatePIN)
```

The GetPIN procedure is based on another finite state machine in which states refer to the number of digits received, and in any state, either another digit key can be touched, or the cancel key can be

touched. Rather than another CASE statement implementation, the "states" are collapsed into iterations of a WHILE loop.

```
Procedure GetPIN(EnteredPIN, CancelHit)
Local Data: DigitKeys =  {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
BEGIN
CancelHit = FALSE
EnteredPIN = null string
DigitsRcvd=0
WHILE NOT(DigitsRcvd=4 OR CancelHit) DO
    BEGIN
        KeySensor(KeyHit)
        IF KeyHit IN DigitKeys
        THEN BEGIN
              EnteredPIN = EnteredPIN + KeyHit
              INCREMENT(DigitsRcvd)
              IF DigitsRcvd=1 THEN ScreenDriver(2, 'X-')
              IF DigitsRcvd=2 THEN ScreenDriver(2, 'XX-')
              IF DigitsRcvd=3 THEN ScreenDriver(2, 'XXX-')
              IF DigitsRcvd=4 THEN ScreenDriver(2, 'XXXX')
              END
    END   {WHILE}
END.   (Procedure GetPIN)
```

If we follow the pseudocode in these three modules, we can identify the "uses" relationship among the modules in the functional decomposition.

Module	**Uses Modules**
SATM Main	WatchCardSlot
	Control Card Roller
	Screen Driver
	Validate Card
	Validate PIN
	Manage Transaction
	New Transaction Request
ValidatePIN	GetPINforPAN
	GetPIN

	Screen Driver
GetPIN	KeySensor
	Screen Driver

Notice that the "uses" information is not readily apparent in the functional decomposition. This information is developed (and extensively revised) during the more detailed phases of the design process. We will revisit this in Chapter 13.

4.4 Separating Integration and System Testing

We are almost in a position to make a clear distinction between integration and system testing. We need this distinction to avoid gaps and redundancies across levels of testing, to clarify appropriate goals for these levels, and to understand how to identify test cases at different levels. This whole discussion is facilitated by a concept essential to all levels of testing: the notion of a "thread". A thread is a construct that refers to execution time behavior; when we test a system, we use test cases to select (and execute) threads. We can speak of levels of threads: system threads describe system level behavior, integration threads correspond to integration level behavior, and unit threads correspond to unit level behavior. Many authors use the term, but few define it, and of those that do, the offered definitions aren't very helpful. For now, we take "thread" to be a primitive term, much like function and data. In the next two chapters, we shall see that threads are most often recognized in terms of the way systems are described and developed. For example, we might think of a thread as a path through a finite state machine description of a system, or we might think of a thread as something that is determined by a data context and a sequence of port level input events, such as those in the context diagram of the SATM system. We could also think of a thread as a sequence of source statements, or as a sequence of machine instructions. The point is, threads are a generic concept, and they exist independently of how a system is described and developed.

We have already observed the structural versus behavioral dichotomy; here we shall find that both of these views help us separate integration and system testing. The structural view reflects both the process by which a system is built and the techniques used to build it. We certainly expect that test cases at various levels can be traced back to developmental information. While this is necessary, it fails to be sufficient: we will finally make our desired separation in terms of behavioral constructs.

Figure 4.15 SATM Class Hierarchy

4.4.1 Structural Insights

Everyone agrees that there must be some distinction, and that integration testing is at a more detailed level than system testing. There is also general agreement that integration testing can safely assume that the units have been separately tested, and that, taken by themselves, the units function correctly. One common view, therefore, is that integration testing is concerned with the interfaces among the units. One possibility is to fall back on the symmetries in the waterfall life cycle model, and say that integration testing is concerned with preliminary design information, while system testing is at the level of the requirements specification. This is a popular academic view, but it begs an important question: how do we discriminate between specification and preliminary design? The pat academic answer to this is the what vs. how dichotomy: the requirements specification defines what, and the preliminary design describes how. While this sounds good at first, it doesn't stand up well in practice. Some scholars argue that just the choice of a requirements specification technique is a design choice

The life cycle approach is echoed by designers who often take a "Don't Tread On Me" view of a requirements specification: a requirements specification should neither predispose nor preclude a design option. With this view, when information in a specification is so detailed that it "steps on the designer's toes", the specification is too detailed. This sounds good, but it still doesn't yield an operational way to separate integration and system testing.

The models used in the development process provide some clues. If we follow the definition of the SATM system, we could first postulate that system testing should make sure that all fifteen display screens have been generated. (An output domain based, functional view of system testing.) The entity/relationship model also helps: the one-to-one and one-to-many relationships help us understand how much testing must be done. The control model (in this case, a hierarchy of finite state machines) is the most helpful. We can postulate system test cases in terms of paths through the finite state machine(s); doing this yields a system level analog of structural testing. The functional models (dataflow diagrams and structure charts) move in the direction of levels because both express a functional decomposition. Even with this, we cannot look at a structure chart and identify where system testing ends and integration testing starts. The best we can do with structural information is identify the extremes. For instance, the following threads are all clearly at the system level:

1. Insertion of an invalid card. (this is probably the "shortest" system thread)
2. Insertion of a valid card, followed by three failed PIN entry attempts.
3. Insertion of a valid card, a correct PIN entry attempt, followed by a balance inquiry.
4. Insertion of a valid card, a correct PIN entry attempt, followed by a deposit.
5. Insertion of a valid card, a correct PIN entry attempt, followed by a withdrawal.
6. Insertion of a valid card, a correct PIN entry attempt, followed by an attempt to withdraw more cash than the account balance.

We can also identify some integration level threads. Go back to the PDL descriptions of ValidatePIN and GetPIN. ValidatePIN calls GetPIN, and GetPIN waits for KeySensor to report when a key is touched. If a digit is touched, GetPIN echoes an "X" to the display screen, but if the cancel key is touched, GetPIN terminates, and ValidatePIN considers another PIN entry attempt. We could push still lower, and consider keystroke sequences such as two or three digits followed by cancel keystroke.

4.4.2 Behavioral Insights

Here is a pragmatic, explicit distinction that has worked well in industrial applications. Think about a system in terms of its port boundary, which is the location of system level inputs and outputs. Every system has a port boundary; the port boundary of the SATM system includes the digit keypad, the function buttons, the screen, the deposit and withdrawal doors, the card and receipt slots, and so on. Each of these devices can be thought of as a "port", and events occur at system ports. The port input and output events are visible to the customer, and the customer very often understands system behavior in terms of sequences of port events. Given this, we mandate that system port events are the "primitives" of a system test case, that is, a system test case (or equivalently, a system thread) is expressed as an interleaved sequence of port input and port output events. This fits our understanding of a test case, in which we specify pre-conditions, inputs, outputs, and post-conditions. With this mandate we can always recognize a level violation: if a test case (thread) ever requires an input (or an output) that is not visible at the port boundary, the test case cannot be a system level test case (thread). Notice that this is clear, recognizable, and enforceable. We will refine this in Chapter 14 when we discuss threads of system behavior.

Integration Testing

Craftspersons are recognized by two essential characteristics: they have a deep knowledge of the tools of their trade, and they have a similar knowledge of the medium in which they work, so that they understand their tools in terms of how they "work" with the medium. In Parts II and III, we focused on the tools (techniques) available to the testing craftsperson. Our goal there was to understand testing techniques in terms of their advantages and limitations with respect to particular types of faults. Here we shift our emphasis to the medium, with the goal that a better understanding of the medium will improve the testing craftsperson's judgment.

4.5 A Closer Look at the SATM System

we described the SATM system in terms of its output screens (Figure 4.7), the terminal itself (Figure 4.8), its context and partial dataflow (Figures 4.9 and 4.10), an entity/relationship model of its data (Figure 4.11), finite state machines describing some of its behavior (Figures 4.12 and 4.13), and a partial functional decomposition (Figure 4.14). We also developed a PDL description of the main program and two units, ValidatePIN and GetPIN.

We begin here by expanding the functional decomposition that was started in Figure 4.12; the numbering scheme preserves the levels of the components in that figure. For easier reference, each

component that appears in our analysis is given a new (shorter) number; these numbers are given in Table 1. (The only reason for this is to make the figures and spreadsheet more readable.) If you look closely at the units that are designated by letters, you see that they are packaging levels in the decomposition; they are never called as procedures. The decomposition in Table 1 is pictured as a decomposition tree in Figure 13.1. This decomposition is the basis for the usual view of integration testing. It is important to remember that such a decomposition is primarily a packaging partition of the system. As software design moves into more detail, the added information lets us refine the functional decomposition tree into a unit calling graph. The unit calling graph is the directed graph in which nodes are program units and edges correspond to program calls; that is, if unit A calls unit B, there is a directed edge from node A to node B. We began the development of the call graph for the SATM system in Chapter 12 when we examined the calls made by the main program and the ValidatePIN and GetPIN modules. That information is captured in the adjacency matrix given below in Table 2. This matrix was created with a spreadsheet; this turns out to be a handy tool for testers.

Table 1 SATM Units and Abbreviated Names **Unit Number**

	Level Number	**Unit Name**
1	1	SATM System
A	1.1	Device Sense & Control
D	1.1.1	Door Sense & Control
2	1.1.1.1	Get Door Status
3	1.1.1.2	Control Door
4	1.1.1.3	Dispense Cash
E	1.1.2	Slot Sense & Control
5	1.1.2.1	WatchCardSlot
6	1.1.2.2	Get Deposit Slot Status
7	1.1.2.3	Control Card Roller
8	1.1.2.3	Control Envelope Roller
9	1.1.2.5	Read Card Strip
10	1.2	Central Bank Comm.

11	1.2.1	Get PIN for PAN
12	1.2.2	Get Account Status
13	1.2.3	Post Daily Transactions
B	1.3	Terminal Sense & Control
14	1.3.1	Screen Driver
15	1.3.2	Key Sensor
C	1.4	Manage Session
16	1.4.1	Validate Card
17	1.4.2	Validate PIN
18	1.4.2.1	GetPIN
F	1.4.3	Close Session
19	1.4.3.1	New Transaction Request
20	1.4.3.2	Print Receipt
21	1.4.3.3	Post Transaction Local
22	1.4.4	Manage Transaction
23	1.4.4.1	Get Transaction Type
24	1.4.4.2	Get Account Type
25	1.4.4.3	Report Balance
26	1.4.4.4	Process Deposit
27	1.4.4.5	Process Withdrawal

The SATM call graph is shown in Figure 4.5.2 Some of the hierarchy is obscured to reduce the confusion in the drawing. One thing should be quite obvious: drawings of call graphs do not scale up well. Both the drawings and the adjacency matrix provide insights to the tester. Nodes with high degree will be important to integration testing, and paths from the main program (node 1) to the sink nodes can be used to identify contents of builds for an incremental development.

4.5.2 Decomposition Based Integration

Most textbook discussions of integration testing only consider integration testing based on the functional decomposition of the system being tested. These approaches are all based on the functional decomposition, expressed either as a tree (Figure 4.5.1) or in textual form. These discussions inevitably center on the order in which modules are to be integrated. There are four choices: from the top of the tree downward (top down), from the bottom of the tree upward (bottom up), some combination of these (sandwich), or most graphically, none of these (the big bang). All of these integration orders presume that the units have been separately tested, thus the goal of decomposition based integration is to test the interfaces among separately tested units.

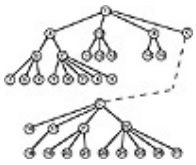
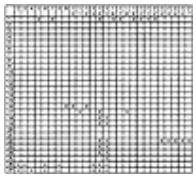


Figure 4.5.1 SATM Functional Decomposition Tree

Table 2 Adjacency Matrix for the SATM Call Graph



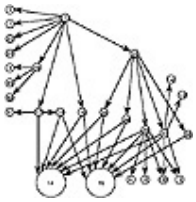


Figure 4.5.2 SATM Call Graph

We can dispense with the big bang approach most easily: in this view of integration, all the units are compiled together and tested at once. The drawback to this is that when (not if!) a failure is observed, there are few clues to help isolate the location(s) of the fault. (Recall the distinction we made in Chapter 1 between faults and failures.)

4.5.2 Top-Down Integration

Top-down integration begins with the main program (the root of the tree). Any lower level unit that is called by the main program appears as a "stub", where stubs are pieces of throw-away code that emulate a called unit. If we performed top-down integration testing for the SATM system, the first step would be to develop stubs for all the units called by the main program: WatchCardSlot, Control Card Roller, Screen Driver, Validate Card, Validate PIN, Manage Transaction, and New

Transaction Request. Generally, testers have to develop the stubs, and some imagination is required. Here are two examples of stubs.

```
Procedure GetPINforPAN (PAN, ExpectedPIN)  STUB
IF PAN = '1123' THEN PIN := '8876';
IF PAN = '1234' THEN PIN := '8765';
IF PAN = '8746' THEN PIN := '1253';
End,

Procedure KeySensor (KeyHit) STUB
data: KeyStrokes STACK OF ' 8 ' . ' 8 ' , ' 7 ' , ' cancel '
KeyHit = POP (KeyStrokes)
End,
```

In the stub for GetPINforPAN, the tester replicates a table look-up with just a few values that will appear in test cases. In the stub for KeySensor, the tester must devise a sequence of port events that can occur once each time the KeySensor procedure is called. (Here, we provided the keystrokes to partially enter the PIN '8876', but the user hit the cancel button before the fourth digit.) In practice, the effort to develop stubs is usually quite significant. There is good reason to consider stub code as part of the software development, and maintain it under configuration management.

Once all the stubs for SATM main have been provided, we test the main program as if it were a stand-alone unit. We could apply any of the appropriate functional and structural techniques, and look for faults. When we are convinced that the main program logic is correct, we gradually replace stubs with the actual code. Even this can be problematic. Would we replace all the stubs at once? If we did, we would have a "small bang" for units with a high outdegree. If we replace one stub at a time, we retest the main program once for each replaced stub. This means that, for the SATM main program example here, we would repeat its integration test eight times (once for each replaced stub, and once with all the stubs).

4.5.2 Bottom-up Integration

Bottom-up integration is a "mirror image" to the top-down order, with the difference that stubs are replaced by driver modules that emulate units at the next level up in the tree. In bottom-up integration, we start with the leaves of the decomposition tree (units like ControlDoor and DispenseCash), and test them with specially coded drivers. There is probably less throw-away code in drivers than there is in stubs. Recall we had one stub for each child node in the decomposition tree. Most systems have a fairly high fan-out near at the leaves, so in the bottom-up integration order, we won't have as many drivers. This is partially offset by the fact that the driver modules will be more complicated.

4.5.3 Sandwich Integration

Sandwich integration is a combination of top-down and bottom-up integration. If we think about it in terms of the decomposition tree, we are really just doing big bang integration on a sub-tree. There will be less stub and driver development effort, but this will be offset to some extent by the added

difficulty of fault isolation that is a consequence of big bang integration. (We could probably discuss the size of a sandwich, from dainty finger sandwiches to Dagwood-style sandwiches, but not now.)

4.6 Call Graph Based Integration

One of the drawbacks of decomposition based integration is that the basis is the functional decomposition tree. If we use the call graph instead, we mitigate this deficiency; we also move in the direction of behavioral testing. We are in a position to enjoy the investment we made in the discussion of graph theory. Since the call graph is a directed graph, why not use it the way we used program graphs? This leads us to two new approaches to integration testing: we'll refer to them as pair-wise integration and neighborhood integration.

4.6.1 Pair-wise Integration

The idea behind pair-wise integration is to eliminate the stub/driver development effort. Rather than develop stubs and/or drivers, why not use the actual code? At first, this sounds like big bang integration, but we restrict a session to just a pair of units in the call graph. The end result is that we have one integration test session for each edge in the call graph (40 for the SATM call graph in Figure 4.2). This is not much of a reduction in sessions from either top-down or bottom-up (42 sessions), but it is a drastic reduction in stub/driver development.

4.6.2 Neighborhood Integration

We can let the mathematics carry us still further by borrowing the notion of a "neighborhood" from topology. (This isn't too much of a stretch — graph theory is a branch of topology.) We (informally) define the neighborhood of a node in a graph to be the set of nodes that are one edge away from the given node. In a directed graph, this means all the immediate predecessor nodes and all the immediate successor nodes (notice that these correspond to the set of stubs and drivers of the node). The eleven neighborhoods for the SATM example (based on the call graph in Figure 4.2) are given in Table 3.

Table 3 SATM Neighborhoods **Node**

	Predecessors	**Successors**
16	1	9, 10, 12
17	1	11, 14, 18
18	17	14, 15
19	1	14, 15

23	22	14, 15
24	22	14, 15
26	22	14, 15, 6, 8, 2, 3
27	22	14, 15, 2, 3, 4, 13
25	22	15
22	1	23, 24, 26, 27, 25
1	n/a	5, 7, 2, 21, 16, 17, 19, 22

We can always compute the number of neighborhoods for a given call graph. There will be one neighborhood for each interior node, plus one extra in case there are leaf nodes connected directly to the root node. (An interior node has a non-zero indegree and a non-zero outdegree.) We have

```
Interior nodes = nodes - (source nodes + sink nodes)
   Neighborhoods = interior nodes + source nodes
```

which combine to

```
Neighborhoods = nodes -sink nodes
```

Neighborhood integration yields a drastic reduction in the number of integration test sessions (down to 11 from 40), and it avoids stub and driver development. The end result is that neighborhoods are essentially the sandwiches that we slipped past in the previous section. (There is a slight difference, because the base information for neighborhoods is the call graph, not the decomposition tree.) What they share with sandwich integration is more significant: neighborhood integration testing has the fault isolation difficulties of "medium bang" integration.

4.7 Path Based Integration

Much of the progress in the development of mathematics comes from an elegant pattern: have a clear idea of where you want to go, and then define the concepts that take you there. We do this here for path based integration testing, but first we need to motivate the definitions.

We already know that the combination of structural and functional testing is highly desirable at the unit level; it would be nice to have a similar capability for integration (and system) testing. We also know that we want to express system testing in terms of behavioral threads. Lastly, we revise our goal for integration testing: rather than test interfaces among separately developed and tested units, we focus on interactions among these units. ("Co-functioning" might be a good term.) Interfaces are structural; interaction is behavioral.

When a unit executes, some path of source statements is traversed. Suppose that there is a call to another unit along such a path: at that point, control is passed from the calling unit to the called unit, where some other path of source statements is traversed. We cleverly ignored this situation in Part III, because this is a better place to address the question. There are two possibilities: abandon the single-entry, single exit precept and treat such calls as an exit followed by an entry, or "suppress" the call statement because control eventually returns to the calling unit anyway. The suppression choice works well for unit testing, but it is antithetical to integration testing.

We can finally make the definitions for path based integration testing. Our goal is to have an integration testing analog of DD-Paths.

Definition

An ***MM-Path*** is an interleaved sequence of module execution paths and messages.

The basic idea of an MM-Path is that we can now describe sequences of module execution paths that include transfers of control among separate units. Since these transfers are by messages, MM-Paths always represent feasible execution paths, and these paths cross unit boundaries. We can find MM-Paths in an extended program graph in which nodes are module execution paths and edges are messages. The hypothetical example in Figure 4.7.3 shows an MM-Path (the dark line) in which module A calls module B, which in turn calls module C.

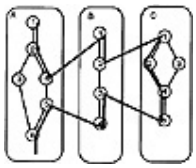


Figure 4.7..3 MM-Path Across Three Units

In module A, nodes 1 and 5 are source nodes, and nodes 4 and 6 are sink nodes. Similarly in module B, nodes 1 and 3 are source nodes, and nodes 2 and 4 are sink nodes. Module C has a single source node, 1, and a single sink node, 4. There are seven module execution paths in Figure 4.7.3:

MEP(A,1) = <1, 2, 3,5>

MEP(A,2) = <1, 2, 4>

MEP(A,3) = <5, 6>

MEP(B,1) = <1, 2>

MEP(B,1) = <3, 4>

MEP(C,1) = <1, 2, 4, 5>

MEP(C,2) = <1, 3, 4, 5>

We can now define an integration testing analog of the DD-Path graph that serves unit testing so effectively.

Definition

Given a set of units, their ***MM-Path graph*** is the directed graph in which nodes are module execution paths and edges correspond to messages and returns from one unit to another.

Notice that MM-Path graphs are defined with respect to a set of units. This directly supports composition of units and composition based integration testing. We can even compose down to the level of individual module execution paths, but that is probably more detailed than necessary.

Figure 4.7.4 MM-Path Graph Derived from Figure 4.7.4

Figure 4.7.4 shows the MM-Path graph for the example in Figure 4.7.3. The solid arrows indicate messages; the corresponding returns are indicated by dotted arrows. We should consider the relationships among module execution paths, program path, DD-Paths, and MM-Paths. A program path is a sequence of DD-Paths, and an MM-Path is a sequence of module execution paths. Unfortunately, there is no simple relationship between DD-Paths and module execution paths. Either might be contained in the other, but more likely, they partially overlap. Since MM-Paths implement a function that transcends unit boundaries, we do have one relationship: consider the "intersection" of an MM-Path with a unit. The module execution paths in such an intersection are an analog of a slice with respect to the (MM-Path) function. Stated another way, the module execution paths in such an intersection are the restriction of the function to the unit in which they occur.

The MM-Path definition needs some practical guidelines. How long is an MM-Path? Nothing in the definition prohibits an MM-Path to cover an entire ATM session. (This extreme loses the forest because of the trees.) There are three observable behavioral criteria that put endpoints on MM-Paths. The first is "event quiescence", which occurs when a system is (nearly) idle, waiting for a port input event to trigger further processing. The SATM system exhibits event quiescence in several places: one is the tight loop at the beginning of SATM Main where the system has displayed the welcome screen and is waiting for a card to be entered into the card slot. Event quiescence is a system level property; there is an analog at the integration level: message quiescence. Message quiescence occurs when a unit that sends no messages is reached (like module C in Figure 13.3).

There is a still subtler form: data quiescence. This occurs when a sequence of processing culminates in the creation of stored data that is not immediately used. In the ValidateCard unit, the account balance is obtained, but it is not used until after a successful PIN entry. Figure 13.5 shows how data quiescence appears in a traditional dataflow diagram.

The first guideline for MM-Paths: points of quiescence are "natural" endpoints for an MM-Path. Our second guideline also serves to distinguish integration from system testing.

Definition

An ***atomic system function (ASF)*** is an action that is observable at the system level in terms of port input and output events.

An atomic system function begins with a port input event, traverses one or more MM-Paths, and terminates with a port output event. When viewed from the system level, there is no compelling reason to decompose an ASF into lower levels of detail (hence the atomicity). In the SATM system, digit entry is a good example of an ASF, so are card entry, cash dispensing, and session closing. PIN entry is probably too big, it might be called a molecular system function.

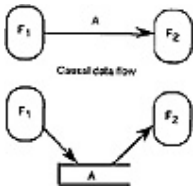


Figure 4.7.5 Data Quiescence

Our second guideline: atomic system functions are an upper limit for MM-Paths: we don't want MM-Paths to cross ASF boundaries. This means that ASFs represent the seam between integration and system testing. They are the largest item to be tested by integration testing, and the smallest item for system testing. We can test an ASF at both levels. Again, the digit entry ASF is a good example. During system testing, the port input event is a physical key press that is detected by KeySensor and sent to GetPIN as a string variable. (Notice that KeySensor performs the physical to logical transition.) GetPIN determines whether a digit key or the cancel key was pressed, and responds accordingly. (Notice that button presses are ignored.) The ASF terminates with either screen 2 or 4 being displayed. Rather than require system keystrokes and visible screen displays, we could use a driver to provide these, and test the digit entry ASF via integration testing. We can see this using our continuing example.

4.7.2 MM-Paths and ASFS in the SATM System

The PDL descriptions developed in Chapter 12 are repeated for convenient reference; statement fragments are numbered as we did to construct program graphs.

```
1.  Main Program
2.  State = AwaitCard
3.  CASE State OF
4.  AwaitCard:        ScreenDriver (1, null)
5.                    WatchCardSlot (CardSlotStatus)
6.                    WHILE CardSlotStatus is Idle DO
7.                        WatchCardSlot (CardSlotStatus)
8.                    ControlCardRoller (accept)
```

```
 9.                    ValidateCard (CardOK, PAN)
10.                    IF CardOK THEN State = AwaitPIN
11.                              ELSE ControlCardRoller (eject)
12.                                   State = AwaitCard
13.  AwaitPIN:         ValidatePIN (PINok, PAN)
14.                    IF PINok THEN ScreenDriver (5, null)
15.                                  State = AwaitTrans
16.                             ELSE ScreenDriver (4, null)
17.                                  State = AwaitCard
18.  AwaitTrans:       ManageTransaction
19.                    State = CloseSession
20.  CloseSession:     IF NewTransactionRequest
21.                        THEN State = AwaitTrans
22.                        ELSE PrintReceipt
23.                        PostTransactionLocal
24.                        CloseSession
25.                        ControlCardRoller (eject)
26.                        State = AwaitCard
27.  End,  (CASE State)
28.  END.   (Main program SATM)

29.  Procedure ValidatePIN (PINok, PAN)
30.  GetPINforPAN (PAN, ExpectedPIN)
31.  Try = First
32.  CASE Try OF
33.  First:   ScreenDriver ( 2, null )
34.           GetPIN (EnteredPIN)
35.           IF EnteredPIN = ExpectedPIN
36.               THEN PINok = TRUE
37.                    RETURN
38.               ELSE ScreenDriver ( 3, null )
39.                    Try = Second
40.  Second:  ScreenDriver ( 2, null )
41.           GetPIN (EnteredPIN)
42.           IF EnteredPIN ExpectedPIN
43.               THEN PINok = TRUE
44.                    RETURN
45.               ELSE ScreenDriver ( 3, null )
46.                    Try = Third
47.  Third:   ScreenDriver (2, null )
48.           GetPIN (EnteredPIN)
49.           IF EnteredPIN = ExpectedPIN
50.               THEN PINok = TRUE
51.                    RETURN
52.               ELSE ScreenDriver ( 4, null )
53.                    PINok = FALSE
54.  END,  (CASE Try)
55.  END.     (Procedure ValidatePIN)
56.  Procedure GetPIN (EnteredPIN, CancelHit)
57.  Local Data: DigitKeys = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
58.  BEGIN
59.  CancelHit FALSE
60.  EnteredPIN = null string
61.  DigitsRcvd=0
```

```
62.  WHILE NOT (DigitsRcvd=4 OR CancelHit) DO
63.     BEGIN
64.         KeySensor (KeyHit)
65.         IF KeyHit IN DigitKeys
66.         THEN BEGIN
67.              EnteredPIN = EnteredPIN + KeyHit
68.         INCREMENT (DigitsRcvd)
69.         IF DigitsRcvd=1 THEN ScreenDriver (2, ' X- ' )
70.         IF DigitsRcvd=2 THEN ScreenDriver (2, ' XX- ' )
71.         IF DigitsRcvd=3 THEN ScreenDriver (2, ' XXX- '  )
72.         IF DigitsRcvd=4 THEN ScreenDriver (2, ' XXXX ' )
73.         END
74.     END {WHILE}
75.  END. (Procedure GetPIN)
```

There are 20 source nodes in SATM Main: 1, 5, 6, 8, 9, 10, 12, 14, 15, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27. ValidatePIN has 11 source nodes: 29, 31, 34, 35, 39, 41, 46, 47, 48, 53; and in GetPIN there are 6 source nodes: 56, 65, 70, 71, 72, 73.

SATM Main contains 16 sink nodes: 4, 5, 7, 8, 9, 11, 13, 14, 16, 18, 20, 22, 23, 24, 25, 28. There are 14 sink nodes in ValidatePIN : 30, 33, 34, 37, 38, 40, 41, 44, 47, 48, 51, 52, 55; and 5 sink nodes in GetPIN: 64, 69, 70, 71, 72.

Most of the module execution paths in SATM Main are very short; this pattern is due to the high density of messages to other units. Here are the first two module execution paths in SATM Main: <1, 2, 3, 4>, <5> and <6, 7>, <8> . The module execution paths in ValidatePIN are slightly longer: <29, 30>, <31, 32, 33>, <34>, <35, 36, 37>, and so on. The beginning portion of GetPIN is a good example of a module execution path: the sequence < 58, 59, 60, 61, 62, 63, 64> begins with a source node (58) and ends with a sink node (64) which is a call to the KeyHit procedure. This is also a point of "event quiescence", where nothing will happen until the customer touches a key.

There are four MM-Paths in statements 64 through 72: each begins with KeySensor observing a port input event (a keystroke) and ends with a closely knit family of port output events (the calls to ScreenDriver with different PIN echoes). We could name these four MM-Paths GetDigit1, GetDigit2, GetDigit3, and GetDigit4. They are slightly different because the later ones include the earlier IF statements. (If the tester was the designer, this module might be reworked so that the WHILE loop repeated a single MM-Path.) Technically, each of these is also an atomic system function since they begin and end with port events.

There are interesting ASFs in ValidatePIN. This unit controls all screen displays relevant to the PIN entry process. It begins with the display of screen 2 (which asks the customer to enter his/her PIN). Next, GetPIN is called, and the system is event quiescent until a keystroke occurs. These keystrokes initiate the GetDigit ASFs we just discussed. Here we find a curious integration fault. Notice that screen 2 is displayed in two places: by the THEN clauses in the WHILE loop in GetPIN and by the first statements in each CASE clause in ValidatePIN. We could fix this by removing the screen displays from GetPIN and simply returning the string (e.g., 'X—') to be displayed.

UNIT 5
SYSTEM TESTING, INTERACTION TESTING

Of the three levels of testing, the system level is closest to everyday experience. We test many things: a used car before we buy it, an on-line network service before we subscribe, and so on. A common pattern in these familiar forms is that we evaluate a product in terms of our expectations; not with respect to a specification or a standard. Consequently, the goal is not to find faults, but to demonstrate performance. Because of this, we tend to approach system testing from a functional standpoint rather than from a structural one. Since it is so intuitively familiar, system testing in practice tends to be less formal than it might be, and this is compounded by the reduced testing interval that usually remains before a delivery deadline. We begin with further elaboration on the thread concept, highlighting some of the practical problems of thread-based system testing. Since system testing is closely coupled with requirements specification, we will discuss how to find threads in common notations. All of this leads to an orderly thread-based system testing strategy that exploits the symbiosis between functional and structural testing; we will apply the strategy to our SATM system.

5.1 Threads

Threads are hard to define, in fact some published definitions are counter-productive, misleading, and/or wrong. It's possible to simply treat threads as a primitive concept which needs no formal definition. For now we will use examples to develop a "shared vision". Here are several views of a thread:

- a scenario of normal usage
- a system level test case
- a stimulus/response pair
- behavior that results from a sequence of system level inputs
- an interleaved sequence of port input and output events
- a sequence of transitions in a state machine description of the system
- an interleaved sequence of object messages and method executions
- a sequence of machine instructions
- a sequence of source instructions
- a sequence of atomic system functions

Threads have distinct levels. A unit level thread is usefully understood as an execution-time path of source instructions, or alternatively as a path of DD-Paths. An integration level thread is a sequence of MM-Paths that implements an atomic system function. We might also speak of an integration

level thread as an alternating sequence of module executions and messages. If we continue this pattern, a system level thread is a sequence of atomic system functions. Because atomic system functions have port events as their inputs and outputs, the sequence of atomic system functions implies an interleaved sequence of port input and output events. The end result is that threads provide a unifying view of our three levels of testing. Unit testing tests individual functions, integration testing examines interactions among units, and system testing examines interactions among atomic system functions. In this chapter, we focus on system level threads and we answer some fundamental questions: How big is a thread? Where do we find them? How do we test them?

5.1.1 Thread Possibilities

Defining the endpoints of a system level thread is a little awkward. We motivate a tidy, graph theory based definition by working backwards from where we want to go with threads. Here are three candidate threads:

- Entry of a digit
- Entry of a Personal Identification Number (PIN)
- A simple transaction: ATM Card Entry, PIN entry, select transaction type (deposit, withdraw), present account details (checking or savings, amount), conduct the operation, and report the results.
- An ATM session, containing two or more simple transactions.

Digit entry is a good example of a minimal atomic system function that is implemented with a single MM-Path. It begins with a port input event (the digit keystroke) and ends with a port output event (the screen digit echo), so it qualifies as a stimulus/response pair. This level of granularity is too fine for the purposes of system testing. We saw this to be an appropriate level for integration testing.

The second candidate, PIN Entry, is a good example of an upper limit to integration testing, and at the same time, a starting point of system testing. PIN Entry is a good example of an atomic system function. It is also a good example of a family of stimulus/response pairs (system level behavior that is initiated by a port input event, traverses some programmed logic, and terminates in one of several possible responses [port output events]). As we saw in Chapter 13, PIN Entry entails a sequence of system level inputs and outputs:

1. A screen requesting PIN digits
2. An interleaved sequence of digit keystrokes and screen responses
3. The possibility of cancellation by the customer before the full PIN is entered
4. A system disposition: (A customer has three chances to enter the correct PIN. Once a correct PIN has been entered, the user sees a screen requesting the transaction type; otherwise a screen

advises the customer that the ATM card will not be returned, and no access to ATM functions is provided.)

5.1.2 Thread Definitions

Definition

A ***unit thread*** is a path in the program graph of a unit.

There are two levels of threads used in integration testing: MM-Paths and atomic system functions. The definitions from Chapter 13 are repeated here so the coherence across the levels is more evident. Recall that MM-Paths are defined as paths in the directed graph in which module execution paths are nodes, and edges show execution time sequence.

Definition

An ***MM-Path*** is a path in the MM-Path graph of a set of units.

Definition

Given a system defined in terms of atomic system functions, the ***ASF Graph*** of the system is the directed graph in which nodes are atomic system functions and edges represent sequential flow.

Definition

A ***source ASF*** is an atomic system function that appears as a source node in the ASF graph of a system; similarly, a ***sink ASF*** is an atomic system function that appears as a sink node in the ASF graph.

In the SATM system, the Card Entry ASF is a source ASF, and the session termination ASF is a sink ASF. Notice that intermediary ASFs could never be tested at the system level by themselves — they need the predecessor ASFs to "get there".

Definition

A ***system thread*** is a path from a source ASF to a sink ASF in the ASF graph of a system.

Definition

Given a system defined in terms of system threads, the ***Thread Graph*** of the system is the directed graph in which nodes are system threads and edges represent sequential execution of individual threads.

This set of definitions provides a coherent set of increasingly broader views of threads, starting with threads within a unit and ending with interactions among system level threads. We can use these views as the ocular on a microscope, switching views to get to different levels of granularity.

Having these concepts is only part of the problem, supporting them is another. We next take a tester's view of requirements specification to see how to identify threads.

5.2 Basis Concepts for Requirements Specification

Recall the notion of a basis of a vector space: a set of independent elements from which all the elements in the space can be generated. Rather than anticipate all the variations in scores of requirements specification methods, notations, and techniques, we will discuss system testing with respect to a basis set of requirements specification constructs: data, actions, ports, events, and threads. Every system can be expressed in terms of these five fundamental concepts (and every requirements specification technique is some combination of these). We examine these fundamental concepts here to see how they support the tester's process of thread identification.

5.2.1 Data

When a system is described in terms of its data, the focus is on the information used and created by the system. We describe data in terms of variables, data structures, fields, records, data stores, and files. Entity/relationship models are the most common choice at the highest level, and some form of a regular expression (e.g., Jackson diagrams or data structure diagrams) is used at a more detailed level. The data-centered view is also the starting point for several flavors of object-oriented analysis. Data refers to information that is either initialized, stored, updated, or (possibly) destroyed. In the SATM system, initial data describe the various accounts (PANs) and their PINs, and each account has a data structure with information such as the account balance. As ATM transactions occur, the results are kept as created data and used in the daily posting of terminal data to the central bank. For many systems, the data centered view dominates. These systems are often developed in terms of CRUD actions (Create, Retrieve, Update, Delete). We could describe the transaction portion of the SATM system in this way, but it wouldn't work well for the user interface portion.

Sometimes threads can be identified directly from the data model. Relationships between data entities can be one-to-one, one-to-many, many-to-one, or many-to-many; these distinctions all have implications for threads that process the data. For example, if bank customers can have several accounts, each account will need a unique PIN. If several people can access the same account, they will need ATM cards with identical PANs. We can also find initial data (such as PAN, ExpectedPIN pairs) that are read but never written. Such read-only data must be part of the system initialization process. If not, there must be threads that create such data. Read-only data is therefore an indicator of source ASFs.

5.2.2 Actions

Action-centered modeling is by far the most common requirements specification form. This is a historical outgrowth of the action-centered nature of imperative programming languages. Actions have inputs and outputs, and these can be either data or port events. Here are some methodology-specific synonyms for actions: transform, data transform, control transform, process, activity, task, method, and service. Actions can also be decomposed into lower level actions, as we saw with the dataflow diagrams in Chapter 12. The input/output view of actions is exactly the basis of functional

testing, and the decomposition (and eventual implementation) of actions is the basis of structural testing.

5.2.3 Ports

Every system has ports (and port devices); these are the sources and destinations of system level inputs and outputs (port events). The slight distinction between ports and port devices is sometimes helpful to testers. Technically, a port is the point at which an I/O device is attached to a system, as in serial and parallel ports, network ports, and telephone ports. Physical actions (keystrokes and light emissions from a screen) occur on port devices, and these are translated from physical to logical (or logical to physical). In the absence of actual port devices, much of system testing can be accomplished by "moving the port boundary inward" to the logical instances of port events. From now on, we will just use the term "port" to refer to port devices. The ports in the SATM system include the digit and cancel keys, the function keys, the display screen, the deposit and withdrawal doors, the card and receipt slots, and several less obvious devices, such as the rollers that move cards and deposit envelopes into the machine, the cash dispenser, the receipt printer, and so on. Thinking about the ports helps the tester define both the input space that functional system testing needs; and similarly, the output devices provide output-based functional test information. (For example, we would like to have enough threads to generate all 15 SATM screens.)

5.2.4 Events

Events are somewhat schizophrenic: they have some characteristics of data and some of actions. An event is a system level input (or output) that occurs at a port. Like data, events can be inputs to or outputs of actions. Events can be discrete (such as SATM keystrokes) or they can be continuous (such as temperature, altitude, or pressure). Discrete events necessarily have a time duration, and this can be a critical factor in real-time systems. We might picture input events as destructive read-out data, but it's a stretch to imagine output events as destructive write operations.

Events are like actions in the sense that they are the translation point between real-world physical events and internal logical manifestations of these. Port input events are physical-to-logical translations, and symmetrically, port output events are logical-to-physical translations. System testers should focus on the physical side of events, not the logical side (the focus of integration testers). There are situations where the context of present data values changes the logical meaning of physical events. In the SATM system, for example, the port input event of depressing button B1 means "Balance" when screen 5 is being displayed, "checking" when screen 6 is being displayed, and "yes" when screens 10, 11, and 14 are being displayed. We refer to such situations as "context sensitive port events", and we would expect to test such events in each context.

5.2.5 Threads

Unfortunately for testers, threads are the least frequently used of the five fundamental constructs. Since we test threads, it usually falls to the tester to find them in the interactions among the data, events, and actions. About the only place that threads appear *per se* in a requirements specification is when rapid prototyping is used in conjunction with a scenario recorder. It's easy to find threads in

control models, as we will soon see. The problem with this is that control models are just that — they are models, not the reality of a system.

5.2.6 Relationships Among Basis Concepts

Figure 5.1 is an entity/relationship model of our basis concepts. Notice that all relationships are many-to-many: Data and Events are generalized into an entity; the two relationships to the Action entity are for inputs and outputs. The same event can occur on several ports, and typically many events occur on a single port. Finally, an action can occur in several threads, and a thread is composed of several actions. This diagram demonstrates some of the difficulty of system testing. Testers must use events and threads to ensure that all the many-to-many relationships among the five basis concepts are correct.

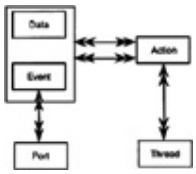


Figure 5.1 E/R Model of Basis Concepts

5.2.7 Modeling with the Basis Concepts

All flavors of requirements specification develop models of a system in terms of the basis concepts. Figure 5.2 shows three fundamental forms of requirements specification models: structural, contextual, and behavioral. Structural models are used for development; these express the functional decomposition and data decomposition, and the interfaces among components. Contextual models are often the starting point of structural modeling. They emphasize system ports and, to a lesser extent, actions, and threads very indirectly. The models of behavior (also called control models) are where four of the five basis constructs come together. Selection of an appropriate control model is the essence of requirements specification: models that are too weak cannot express important system behaviors, while models that are too powerful typically obscure interesting behaviors. As a general rule, decision tables are a good choice only for computational systems, finite state machines are good for menu-driven systems, and Petri nets are the model of choice for concurrent systems. Here we use finite state machines for the SATM system, and in Chapter 16, we will use Petri nets to analyze thread interaction.

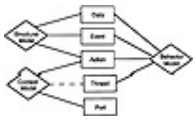


Figure 5.2 Modeling Relationships Among Basis Constructs

We must make an important distinction between a system itself (reality) and models of a system. Consider a system in which some function F cannot occur until two prerequisite events E1 and E2 have occurred, and that they can occur in either order. We could use the notion of event partitioning to model this situation. The result would be a diagram like that in Figure 5.3.

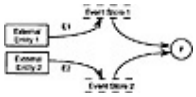


Figure 5.3 Event Partitioning View of Function F

In the event partitioning view, events E1 and E2 occur from their respective external entities. When they occur, they are held in their respective event stores. (An event store acts like a destructive read operation.) When both events have occurred, function F gets its prerequisite information from the event stores. Notice we cannot tell from the model which event occurs first; we only know that both must occur. We could also model the system as a finite state machine (Figure 5.4), in which states record which event has occurred. The state machine view explicitly shows the two orders of the events.

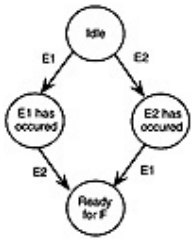


Figure 5.4 FSM for Function F

Both of these models express the same prerequisites for the function F, and neither is the reality of the system. Of these two models, the state machine is more useful to the tester, because paths are instantly convertible to threads.

5.3 Finding Threads

The finite state machine models of the SATM system are the best place to look for system testing threads. We'll start with a hierarchy of state machines; the upper level is shown in Figure 5.5. At this level, states correspond to stages of processing, and transitions are caused by logical (rather than port) events. The Card Entry "state" for example, would be decomposed into lower levels that deal with details like jammed cards, cards that are upside-down, stuck card rollers, and checking the card against the list of cards for which service is offered. Once the details of a macro-state are tested, we use an easy thread to get to the next macro-state.

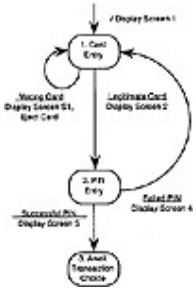


Figure 5.5 Top Level SATM State Machine

The PIN Entry state is decomposed into the more detailed view in Figure 5.6. The adjacent states are shown because they are sources and destinations of transitions from the PIN Entry portion. At this level, we focus on the PIN retry mechanism; all of the output events are true port events, but the input events are still logical events. The states and edges are numbered for reference later when we discuss test coverage. To start the thread identification process, we first list the port events shown on the state transitions; they appear in Table 1. We skipped the eject card event because it isn't really part of the PIN Entry component.

Table 1 Events in the PIN Entry Finite State Machine **Port Input Events**

	Port Output Events
Legitimate Card	Display screen 1
Wrong Card	Display screen 2
Correct PIN	Display screen 3
Incorrect PIN	Display screen 4
Canceled	Display screen 5

Notice that Correct PIN and Incorrect PIN are really compound port input events. We can't actually enter an entire PIN, we enter digits, and at any point, we might hit the cancel key. These more detailed possibilities are shown in Figure 5.7. A truly paranoid tester might decompose the digit port input event into the actual choices (0-pressed, 1-pressed, …,9-pressed), but this should have been tested at a lower level. The port events in the PIN Try finite state machine are in Table 2.

Figure 5.6 PIN Entry Finite State Machine

The "x" in the state names in the PIN Try machine refers to which try (first, second, or third) is passing through the machine.

Table 2 Port Events in the PIN Try Finite State Machine **Port Input Events**

	Port Output Events
digit	echo 'X---'

cancel	echo 'XX--'
	echo 'XXX-'
	echo 'XXXX'

In addition to the true port events in the PIN Try finite state machine, there are three logical output events (Correct PIN, Incorrect PIN, and Canceled); these correspond exactly to the higher level events in Figure 5.6. The hierarchy of finite state machines multiplies the number of threads. There are 156 distinct paths form the First PIN Try state to the Await Transaction Choice or Card Entry states in Figure 5.6. Of these, 31 correspond to eventually correct PIN entries (1 on the first try, 5 on the second try, and 25 on the third try); the other 125 paths correspond to those with incorrect digits or with cancel keystrokes. This is a fairly typical ratio. The input portion of systems, especially interactive systems, usually has a large number of threads to deal with input errors and exceptions. It is "good form" to reach a state machine in which transitions are caused by actual port input events, and the actions on transitions are port output events. If we have such a finite state machine, generating system test cases for these threads is a mechanical process — just follow a path of transitions, and note the port inputs and outputs as they occur along the path. This interleaved sequence is performed by the test executor (person or program). Tables 3 and 4 follow two paths through the hierarchic state machines.

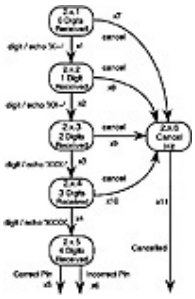


Figure 5.7 PIN Try Finite State Machine

Table 3 corresponds to a thread in which a PIN is correctly entered on the first try. Table 4 corresponds to a thread in which a PIN is incorrectly entered on the first try, cancels after the third digit on the second try, and gets it right on the third try. To make the test case explicit, we assume a pre-condition that the expected PIN is '1234'.

Table 3 Port Event Sequence for Correct PIN on First Try **Port Input Event**

	Port Output Event
	Screen 2 displayed with '----'
1 pressed	

	Screen 2 displayed with 'X---'
2 pressed	
	Screen 2 displayed with 'XX--'
3 pressed	
	Screen 2 displayed with 'XXX-'
4 pressed	
	Screen 2 displayed with 'XXXX'
(Correct PIN)	Screen 5 displayed

The event in parentheses in the last row of Table 3 is the logical event that "bumps up" to the parent state machine and causes a transition there to the Await Transaction Choice state.

Table 4 Port Event Sequence for Correct PIN on Third Try **Port Input Event**

	Port Output Event
	Screen 2 displayed with '---'
1 pressed	
	Screen 2 displayed with 'X---'
2 pressed	
	Screen 2 displayed with 'XX--'
3 pressed	
	Screen 2 displayed with 'XXX-'
5 pressed	
	Screen 2 displayed with 'XXXX'
(Incorrect PIN)	Screen 3 displayed
(second try)	Screen 2 displayed with '----'

1 pressed	
	Screen 2 displayed with 'X---'
2 pressed	
	Screen 2 displayed with 'XX--'
3 pressed	
	Screen 2 displayed with 'XXX-'
cancel key pressed	
(end of second try)	Screen 3 displayed
	Screen 2 displayed with '----'
1 pressed	
	Screen 2 displayed with 'X---'
2 pressed	
	Screen 2 displayed with 'XX--'
3 pressed	
	Screen 2 displayed with 'XXX-'
4 pressed	
	Screen 2 displayed with 'XXXX'
(Correct PIN)	Screen 5 displayed

If you look closely at Tables 3 and 4, you will see that the bottom third of Table 4 is exactly Table 3; thus a thread can be a subset of another thread.

5.4 Structural Strategies for Thread Testing

While generating thread test cases is easy, deciding which ones to actually use is more complex. (If you have an automatic test executor, this is not a problem.) We have the same path explosion problem at the system level that we had at the unit level. Just as we did there, we can use the directed graph insights to make an intelligent choice of threads to test.

5.4.1 Bottom-up Threads

When we organize state machines in a hierarchy, we can work from the bottom up. There are six paths in the PIN Try state machine. If we traverse these six, we test for three things: correct recognition and echo of entered digits, response to the cancel keystroke, and matching expected and entered PINs. These paths are described in Table 5 as sequences of the transitions in Figure 14.7. A thread that traverses the path is described in terms of its input keystrokes, thus the input sequence 1234 corresponds to the thread described in more detail in Table 3 (the cancel keystroke is indicated with a 'C'). Once this portion is tested, we can go up a level to the PIN Entry machine, where there are four paths. These four are concerned with the three try mechanism and the sequence of screens presented to the user. In Table 6, the paths in the PIN Entry state machine (Figure 5.6) are named as transition sequences.

Table 5 Thread Paths in the PIN Try FSM **Input Event Sequence**

	Path of Transitions
1234	x1, x2, x3, x4, x5
1235	x1, x2, x3, x4, x6
C	x7, x11
1C	x1, x8, x11
12C	x1, x2, x9, x11
123C	x1, x2, x3, x10, x11

Table 6 Thread Paths in the PIN Entry FSM **Input Event Sequence**

	Path of Transitions
1234	1
12351234	2, 3
1235C1234	2,4,5
CCC	2, 4, 6

These threads were identified with the goal of path traversal in mind. Recall from our discussion of structural testing that these goals can be misleading. The assumption is that path traversal uncovers faults, and traversing a variety of paths reduces redundancy. The last path in Table 6 illustrates how structural goals can be counter-productive. Hitting the cancel key three times does indeed cause the

three try mechanism to fail, and returns the system to the Card Entry state, but it seems like a degenerate thread. There is a more serious flaw with these threads: we could not really execute them "by themselves", because of the hierarchic state machines. What really happens with the '1235' input sequence in Table 5? It traverses an interesting path in the PIN Try machine, and then it "returns" to the PIN Entry machine where it is seen as a logical event (incorrect PIN), which causes a transition to state 2.2 (Second PIN Try). If no additional keystrokes occur, this machine would remain in state 2.2. We show how to overcome such situations next.

5.4.2 Node and Edge Coverage Metrics

Because the finite state machines are directed graphs, we can use the same test coverage metrics that we applied at the unit level. The hierarchic relationship means that the upper level machine must treat the lower machine as a procedure that is entered and returned. (Actually, we need to do this for one more level to get to true threads that begin with the Card Entry state.) The two obvious choices are node coverage and edge coverage. Table 7 is extended from Table 4 to show the node and edge coverage of the three-try thread. Node (state) coverage is analogous to statement coverage at the unit level — it is the bare minimum. In the PIN Entry example, we can attain node coverage without ever executing a thread with a correct PIN. If you examine Table 8, you will see that two threads (initiated by C1234 and 123C1C1C) traverse all the states in both machines. Edge (state transition) coverage is a more acceptable standard. If the state machines are "well formed" (transitions in terms of port events), edge coverage also guarantees port event coverage. The threads in Table 9 were picked in a structural way, to guarantee that the less traveled edges (those caused by cancel keystrokes) are traversed.

5.5 Functional Strategies for Thread Testing

The finite state machine based approaches to thread identification are clearly useful, but what if no behavioral model exists for a system to be tested? The testing craftsperson has two choices: develop a behavioral model, or resort to the system level analogs of functional testing. Recall that when functional test cases are identified, we use information from the input and output spaces as well as the function itself. We describe functional threads here in terms of coverage metrics that are derived from three of the basis concepts (events, ports, and data).

Table 7 Node and Edge Traversal of a Thread **Port Input Event**

	Port Output Event	**Nodes**	**Edges**
	Screen 2 displayed with '----'	2.1	a
1 pressed		2.1.1	
	Screen 2 displayed with 'X---'		x1
2 pressed		2.1.2	

	Screen 2 displayed with 'XX--'		x2
3 pressed		2.1.3	
	Screen 2 displayed with 'XXX-'		x3
5 pressed		2.1.4	
	Screen 2 displayed with 'XXXX'		x4
(Incorrect PIN)	Screen 3 displayed	2.1.5, 3	x6, 2
(second try)	Screen 2 displayed with '----'	2.2	
1 pressed		2.2.1	
	Screen 2 displayed with 'X---'		x1
2 pressed		2.2.2	
	Screen 2 displayed with 'XX--'		x2
3 pressed		2.2.3	
	Screen 2 displayed with 'XXX-'		x3
cancel pressed		2.2.4	x10
(end of 2nd try)	Screen 3 displayed	2.2.6	x11
	Screen 2 displayed with '----'	2.3	4
1 pressed		2.3.1	
	Screen 2 displayed with 'X---'		x1
2 pressed		2.3.2	
	Screen 2 displayed with 'XX--'		x2
3 pressed		2.3.3	
	Screen 2 displayed with 'XXX-'		x3
4 pressed		2.3.4	

	Screen 2 displayed with 'XXXX'		x4
(Correct PIN)	Screen 5 displayed	2.3.5, 3	x5, 5

Table 8 Thread/State Incidence **Input Events**

	2.1	**2.x.1**	**2.x.2**	**2.x.3**	**2.x.4**	**2.x.5**	**2.2.6**	**2.2**	**2.3**	**3**	**1**
1234	x	x	x	x	x	x				x	
12351234	x	x	x	x	x	x		x		x	
C1234	x	x	x	x	x	x	x	x		x	
1C12C1234	x	x	x	x			x	x	x	x	
123C1C1C	x	x	x	x	x		x	x	x		x

Table 9 Thread/Transition Incidence **Input Events**

	x1	**x2**	**x3**	**x4**	**x5**	**x6**	**x7**	**x8**	**x9**	**x10**	**x11**	**1**	**2**	**3**	**4**	**5**	**6**
1234	x	x	x	x	x							x					
12351234	x	x	x	x	x	x							x	x			
C1234	x	x	x	x	x		x				x		x	x			
1C12C1234	x	x	x	x	x			x	x		x		x		x	x	
123C1C1C	x	x	x					x		x	x		x		x		x

5.5.1 Event-Based Thread Testing

Consider the space of port input events. There are five port input thread coverage metrics of interest. Attaining these levels of system test coverage requires a set of threads such that:

- PI1: each port input event occurs
- PI2: common sequences of port input events occur
- PI3: each port input event occurs in every "relevant" data context
- PI4: for a given context, all "inappropriate" input events occur
- PI5: for a given context, all possible input events occur

The PI1 metric is a bare minimum, and is inadequate for most systems. PI2 coverage is the most common, and it corresponds to the intuitive view of system testing because it deals with "normal use". It is difficult to quantify, however. What is a "common" sequence of input events? What is an uncommon one?

We can also define two coverage metrics based on port output events:

- PO1: each port output event occurs
- PO2: each port output event occurs for each cause

PO1 coverage is an acceptable minimum. It is particularly effective when a system has a rich variety of output messages for error conditions. (The SATM system does not.) PO2 coverage is a good goal, but it is hard to quantify; we will revisit this in Chapter 16 when we examine thread interaction. For now, note that PO2 coverage refers to threads that interact with respect to a port output event.. Usually a given output event only has a small number of causes. In the SATM system, screen 10 might be displayed for three reasons: the terminal might be out of cash, it may be impossible to make a connection with the central bank to get the account balance, or the withdrawal door might be jammed. In practice, some of the most difficult faults found in field trouble reports are those in which an output occurs for an unsuspected cause. One example: my local ATM system (not the SATM) has a screen that informs me that "Your daily withdrawal limit has been reached". This screen should occur when I attempt to withdraw more than $300 in one day. When I see this screen, I used to assume that my wife has made a major withdrawal (thread interaction), so I request a lesser amount. I found out that the ATM also produces this screen when the amount of cash in the dispenser is low. Rather than provide a lot of cash to the first users, the central bank prefers to provide less cash to more users.

5.5.2 Port-Based Thread Testing

Port-based testing is a useful complement to event-based testing. With port-based testing, we ask, for each port, what events can occur at that port. We then seek threads that exercise input ports and output ports with respect to the event lists for each port. (This presumes such event lists have been specified; some requirements specification techniques mandate such lists.) Port-based testing is particularly useful for systems in which the port devices come from external suppliers. The main reason for port-based testing can be seen in the entity/relationship model of the basis constructs (Figure 14.1). The many-to-many relationship between ports and events should be exercised in both directions. Event based testing covers the one-to-many relationship from events to ports, and conversely, port-based testing covers the one-to many relationship from ports to events. The SATM system fails us at this point — there is no SATM event that occurs at more than one port.

5.5.3 Data-Based Thread Testing

Port and event based testing work well for systems that are primarily event driven. Such systems are sometimes called "reactive" systems because they react to stimuli (port input events), and often the reaction is in the from of port output events. Reactive systems have two important characteristics:

they are "long-running" (as opposed to the short burst of computation we see in a payroll program) and they maintain a relationship with their environment. Typically, event driven, reactive systems do not have a very interesting data model (as we see with the SATM system), so data model based threads aren't particularly useful. But what about conventional systems which are data driven? These systems, described as "static" in [Topper 93], are transformational (rather than reactive); they support transactions on a database. When these systems are specified, the entity/relationship model is dominant, and is therefore a fertile source of system testing threads. To attach our discussion to something familiar, we use the entity/relationship model of a simple library system (see Figure 5.8) from [Topper 93].

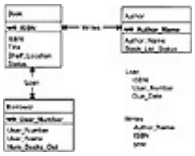


Figure 5.8 E/R Model of a Library

Here are some typical transactions in the library system:

1. Add a book to the library.
2. Delete a book from the library.
3. Add a borrower to the library.
4. Delete a borrower from the library.
5. Loan a book to a borrower.
6. Process the return of a book from a borrower.

These transactions are all mainline threads; in fact, they represent families of threads. For example, suppose the book loan transaction is attempted for a borrower whose current number of checked out books is at the lending limit (a nice boundary value example). We might also try to return a book that was never owned by the library. One more: suppose we delete a borrower that has some unreturned books. All of these are interesting threads to test, and all are at the system level. We can identify each of these examples, and many more, by close attention to the information in the entity/relationship model. As we did with event-based testing, we describe sets of threads in terms of data-based coverage metrics. These refer to relationships for an important reason. Information in relationships is generally populated by system level threads, whereas that in the entities is usually handled at the unit level. (When entity/relationship modeling is the starting point of object-oriented analysis, this is enforced by encapsulation.)

- DM1: Exercise the cardinality of every relationship.
- DM2: Exercise the participation of every relationship.
- DM3: Exercise the functional dependencies among relationships.

Cardinality refers of the four possibilities of relationship that we discussed in Chapter 3: one-to-one, one-to-many, many-to-one, and many-to-many. In the library example, both the loan and the writes relationships are many-to-many, meaning that one author can write many books, and one book can have many authors; and that one book can be loaned to many borrowers (in sequence) and one borrower can borrow many books. Each of these possibilities results in a useful system testing thread.

Participation refers to whether or not every instance of an entity participates in a relationship. In the writes relationship, both the Book and the Author entities have mandatory participation (we cannot have a book with no authors, or an author of no books). In some modeling techniques, participation is expressed in terms of numerical limits; the Author entity, for example, might be expressed as "at least 1 and at most 12". When such information is available, it leads directly to obvious boundary value system test threads.

Sometimes transactions determine explicit logical connections among relationships; these are known as functional dependencies. For example, we cannot loan a book that is not possessed by the library, and we would not delete a book that is out on loan. Also, we would not delete a borrower who still has some books checked out. These kinds of dependencies are reduced when the database is normalized, but they still exist, and they lead to interesting system test threads.

5.6 SATM Test Threads

If we apply the discussion of this chapter to the SATM system, we get a set of threads that constitutes a thorough system level test. We develop such a set of threads here in terms of an overall state model in which states correspond to key atomic system functions. The macro-level states are: Card Entry, PIN Entry, Transaction Request, (and processing), and Session Management. The stated order is the testing order, because these stages are in prerequisite order. (We cannot enter a PIN until successful card entry, we cannot request a transaction until successful PIN entry, and so on.) We also need some pre-condition data that define some actual accounts with PANs, Expected PINs, and account balances. These are given in Table 10. Two less obvious pre-conditions are that the ATM terminal is initially displaying screen 1 and the total cash available to the withdrawal dispenser is $500 (in $10 notes).

Table 10 SATM Test Data **PAN**

	Expected PIN	**Checking Balance**	**Savings Balance**
100	1234	$1000.00	$800.00
200	4567	$100.00	$90.00
300	6789	$25.00	$20.00

We will express threads in tables in which pairs of rows correspond to port inputs and expected port outputs at each of the four major stages. We start with three basic threads, one for each transaction type (balance inquiry, deposit, and withdrawal).

Thread 1 (balance)	**Card Entry (PAN)**	**PIN Entry**	**Transaction Request**	**Session Management**
Port Inputs	100	1234	B1, B1	B2
Port Outputs	screen 2	screen 5	screen 6, screen 14 $1000.00	screen 15, eject card, screen 1

In thread 1, a valid card with PAN = 100 is entered, which causes screen 2 to be displayed. The PIN digits '1234' are entered, and since they match the expected PIN for the PAN, screen 5 inviting a transaction selection is displayed. When button B1 is touched the first time (requesting a balance inquiry), screen 6 asking which account is displayed. When B1 is pressed the second time (checking), screen 14 is displayed and the checking account balance ($1000.00) is printed on the receipt. When B2 is pushed, screen 15 is displayed, the receipt is printed, the ATM card is ejected, and then screen 1 is displayed.

Thread 2 is a deposit to checking: Same PAN and PIN, but B2 is touched when screen 5 is displayed, and B1 is touched when screen 6 is displayed. The amount 25.00 is entered when screen 7 is displayed and then screen 13 is displayed. The deposit door opens and the deposit envelope is placed in the deposit slot. Screen 14 is displayed, and when B2 is pushed, screen 15 is displayed, the receipt showing the new checking account balance of $1025.00 is printed, the ATM card is ejected, and then screen 1 is displayed.

Thread 2 (deposit)	**Card Entry (PAN)**	**PIN Entry**	**Transaction Request**	**Session Management**
Port Inputs	100	1234	B2, B1, 25.00 insert env.	B2
Port Outputs	screen 2	screen 5	screen 6, screen 7, screen 13, dep. door opens, screen 14, $1025.00	screen 15, eject card, screen 1

Thread 3 is a withdrawal from savings: Again the same PAN and PIN, but B3 is touched when screen 5 is displayed, and B2 is touched when screen 6 is displayed. The amount 30.00 is entered when screen 7 is displayed and then screen 11 is displayed. The withdrawal door opens and three $10 notes are dispensed. Screen 14 is displayed, and when B2 is pushed, screen 15 is displayed, the

receipt showing the new savings account balance of $770.00 is printed, the ATM card is ejected, and then screen 1 is displayed.

Thread 3 (withdrawal)	**Card Entry (PAN)**	**PIN Entry**	**Transaction Request**	**Session Management**
Port Inputs	100	1234	B3, B2, 30.00	B2
Port Outputs	screen 2	screen 5	screen 6, screen 7, screen 11, withdrawal door opens, 3 $10 notes, screen 14, $770.00	screen 15, eject card, screen 1

A few of these detailed descriptions are needed to show the pattern; the remaining threads are described in terms of input and output events that are the objective of the test thread.

Thread 4 is the shortest thread in the SATM system, it consists of an invalid card, which is immediately rejected.

Thread 4	**Card Entry (PAN)**	**PIN Entry**	**Transaction Request**	**Session Management**
Port Inputs	400			
Port Outputs	eject card screen 1			

Following the macro-states along thread 1, we next perform variations on PIN Entry. We get four new threads from Table 9, which yield edge coverage in the PIN Entry finite state machines.

Thread 5 (balance)	**Card Entry (PAN)**	**PIN Entry**	**Transaction Request**	**Session Management**
Port Inputs	100	12351234	as in thread 1	
Port Outputs	screen 2	screens 3,2,5		
Thread 6 (balance)	**Card Entry (PAN)**	**PIN Entry**	**Transaction Request**	**Session Management**
Port Inputs	100	C1234	as in thread 1	
Port Outputs	screen 2	screens 3,2,5		

Thread 7 (balance)	**Card Entry (PAN)**	**PIN Entry**	**Transaction Request**	**Session Management**
Port Inputs	100	1C12C1234	as in thread 1	
Port Outputs	screen 2	screens 3,2, 3,2,5		
Thread 8 (balance)	**Card Entry (PAN)**	**PIN Entry**	**Transaction Request**	**Session Management**
Port Inputs	100	123C1C1C		
Port Outputs	screen 2	screens 3,2, 3,2,4,1		

Moving to the Transaction Request stage, there are variations with respect to the type of transaction (balance, deposit, or withdraw), the account (checking or savings) and several that deal with the amount requested. Threads 1, 2, and 3 cover the type and account variations, so we focus on the amount-driven threads. Thread 9 rejects the attempt to withdraw an amount not in $10 increments, Thread 10 rejects the attempt to withdraw more than the account balance, and Thread 11 rejects the attempt to withdraw more cash than the dispenser contains.

Thread 9 (withdrawal)	**Card Entry (PAN)**	**PIN Entry**	**Transaction Request**	**Session Management**
Port Inputs	100	1234	B3, B2, 15.00 Cancel	B2
Port Outputs	screen 2	screen 5	screens 6,7, 9, 7	screen 15, eject card, screen 1
Thread 10 (withdrawal)	**Card Entry (PAN)**	**PIN Entry**	**Transaction Request**	**Session Management**
Port Inputs	300	6789	B3, B2, 50.00 Cancel	B2
Port Outputs	screen 2	screen 5	screens 6,7,8	screen 15, eject card, screen 1
Thread 11 (withdrawal)	**Card Entry (PAN)**	**PIN Entry**	**Transaction Request**	**Session Management**
Port Inputs	100	1234	B3, B2, 510.00 Cancel	B2

Port Outputs	screen 2	screen 5	screens 6,7, 10	screen 15, eject card, screen 1

Having exercised the transaction processing portion, we proceed to the session management stage, where we test the multiple transaction option.

Thread 12 (balance)	**Card Entry (PAN)**	**PIN Entry**	**Transaction Request**	**Session Management**
Port Inputs	100	1234	B1, B1	B1, Cancel
Port Outputs	screen 2	screen 5	screen 6, screen 14 $1000.00	screen 15, screen 5, screen 15, eject card, screen 1

At this point, the threads provide coverage of all output screens except for screen 12, which informs the user that deposits cannot be processed. Causing this condition is problematic (maybe we should place a fish sandwich in the deposit envelope slot). This is an example of a thread selected by a pre-condition that is a hardware failure. We just give it a thread name here, it's thread 13. Next, we develop threads 14 through 22 to exercise context sensitive input events. They are shown in Table 11; notice that some of the first 13 threads exercise context sensitivity.

Table 11 Threads for Context Sensitive Input Events **Thread**

	Keystroke	**Screen**	**Logical Meaning**
6	cancel	2	PIN Entry error
14	cancel	5	transaction selection error
15	cancel	6	account selection error
16	cancel	7	amount selection error
17	cancel	8	amount selection error
18	cancel	13	deposit envelope not ready
1	B1	5	balance
1	B1	6	checking
19	B1	10	yes (a non-withdrawal transaction)

20	B1	12	yes (a non-deposit transaction)
12	B1	14	yes (another transaction)
2	B2	5	deposit
3	B2	6	savings
21	B2	10	no (no additional transaction)
22	B2	12	no (no additional transaction)
1	B2	14	no (no additional transaction)

These 22 threads comprise a reasonable test of the portion of the SATM system that we have specified. Of course there are untested aspects; one good example involves the balance of an account. Consider two threads, one that deposits $40 to an account, and a second that withdraws $80, and suppose that the balance obtained form the central bank at the Card Entry stage is $50. There are two possibilities: one is to use the central bank balance, record all transactions, and then resolve these when the daily posting occurs. The other is to maintain a running local balance, which is what would be shown on a balance inquiry transaction. If the central bank balance is used, the withdrawal transaction is rejected, but if the local balance is used, it is processed.

Another prominent untested portion of the SATM system is the Amount Entry process that occurs in screens 7 and 8. The possibility of a cancel keystroke at any point during amount entry produces a multiplicity greater than that of PIN Entry. There is a more subtle (and therefore more interesting) test for Amount Entry. What actually happens when we enter an amount? To be specific, suppose we wish to enter $40.00. We expect an echo after each digit keystroke, but in which position does the echo occur? Two obvious solutions: always require six digits to be entered (so we would enter '004000') or use the high order digits first and shift left as successive digits are entered, as shown in Figure 14.10. Most ATM systems use the shift approach, and this raises the subtle point: how does the ATM system know when all amount digits have been entered? The ATM system clearly cannot predict that the deposit amount is $40.00 instead of $400.00 or $4000.000 because there is no "enter" key to signify when the last digit has been entered. The reason for this digression is that this is a good example of the kind of detail discovered by testers that is often missing from a requirements specification. (Such details would likely be found with either Rapid Prototyping or using an executable specification.)

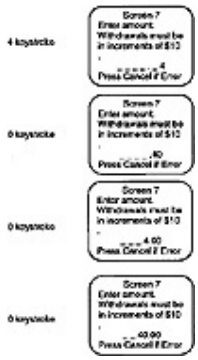


Figure 5.10 Digit Echoes with Left Shifts

5.7 System Testing Guidelines

If we disallow compound sessions (more than one transaction) and if we disregard the multiplicity due to Amount Entry possibilities, there are 435 distinct threads per valid account in the SATM system. Factor in the effects of compound sessions and the Amount Entry possibilities and there are tens of thousands of possible threads for the SATM system. We end this chapter with three strategies to deal with the thread explosion problem.

5.7.1 Pseudo-Structural System Testing

When we studied unit testing, we saw that the combination of functional and structural testing yields a desirable cross-check. We can only claim pseudo-structural testing [Jorgensen 94], because the node and edge coverage metrics are defined in terms of a control model of a system, and are not derived directly from the system implementation. (Recall we started out with a concern over the distinction between reality and models of reality.) In general, behavioral models are only approximations of a system's reality, which is why we could decompose our models down to several levels of detail. If we made a true structural model, its size and complexity would make it too cumbersome to use. The big weakness of pseudo-structural metrics is that the underlying model may be a poor choice. The three most common behavioral models (decision tables, finite state machines, and Petri nets) are appropriate, respectively, to transformational, interactive, and concurrent systems. Decision tables and finite state machines are good choices for ASF testing. If an ASF is described using a decision table, conditions typically include port input events, and actions are port output events. We can then devise test cases that cover every condition, every action, or most completely, every rule. As we saw for finite state machine models, test cases can cover every state, every transition, or every path.

Thread testing based on decision tables is cumbersome. We might describe threads as sequences of rules from different decision tables, but this becomes very messy to track in terms of coverage. We need finite state machines as a minimum, and if there is any form of interaction, Petri nets are a better choice. There we can devise thread tests that cover every place, every transition, and every sequence of transitions.

EXERCISES

1. One of the problems of system testing, particularly with interactive systems, is to anticipate all the strange things the user might do. What happens in the SATM system if a customer enters three digits of a PIN and then walks away?

2. To remain "in control" of abnormal user behavior (the behavior is abnormal, not the user), the SATM system might introduce a timer with a 30 second time-out. When no port input event occurs for 30 seconds, the SATM system asks if the user needs more time. The user can answer yes or no. Devise a new screen and identify port events that would implement such a time-out event.

3. Suppose you add this time-out feature to the SATM system. What regression testing would you perform?

4. Make an additional refinement to the PIN Try finite state machine (Figure 14.6) to implement your timeout mechanism, then revise the thread test case in Table 3.

5. The text asserts that "the B1 function button occurs in five separate contexts (screens being displayed) and has three different meanings". Examine the fifteen screens (points of event quiescence) and decide whether there are three or five different logical meanings to a B1 keystroke.

6. Does it make sense to use test coverage metrics in conjunction with operational profiles? Discuss this.

7. Develop an operational profile for the NextDate problem. Use the decision table formulation, and provide individual condition probabilities. Since a rule is the conjunction of its conditions, the product of the condition entry probabilities is the rule probability.

5.8 Interaction Testing

Faults and failures due to interaction are the bane of testers. Their subtlety makes them difficult to recognize and even more difficult to reveal by testing. These are deep faults, ones that remain in a system even after extensive thread testing. Unfortunately, faults of interaction most frequently occur as failures in delivered systems that have been in use for some time. Typically they have a very low probability of execution, and they occur only after a large number of threads have been executed. Most of this chapter is devoted to describing forms of interaction, not to testing them. As such, it is really more concerned with requirements specification that with testing. The connection is important: knowing how to specify interactions is the first step in detecting and testing for them. This chapter is also a somewhat philosophical and mildly mathematical discussion of faults and failures of interaction; we cannot hope to test something if we don't understand it. We begin with an important addition to our five basis constructs, and use this to develop a taxonomy of types of interaction. Next, we develop a simple extension to conventional Petri nets that reflects the basis constructs, and then we illustrate the whole discussion with the SATM and Saturn Windshield Wiper systems, and sometimes with examples from telephone systems. We conclude by applying the taxonomy to an important application type: client-server systems.

5.8.1 Context of Interaction

Part of the difficulty of specifying and testing interactions is that they are so common. Think of all the things that interact in everyday life: people, automobile drivers, regulations, chemical compounds, and abstractions, to name just a few. We are concerned with interactions in software controlled systems (particularly the unexpected ones), so we start by restricting our discussion to interactions among our basis system constructs: actions, data, events ports, and threads.

One way to establish a context for interaction is to view it as a relationship among the five constructs. If we did this, we would find that the relation Interacts With is a reflexive relationship on each entity (data interacts with data, actions with other actions, and so on). It also is a binary relationship between data and events, data and threads, and events and threads. The data modeling approach isn't a dead-end, however. Whenever a data model contains such pervasive relationships, that is a clue that an important entity is missing. If we add some tangible reality to our fairly abstract constructs, we get a more useful framework for our study of interaction. The missing element is location, and location has two components: time and position. Data modeling provides another choice: we can treat location as sixth basic entity, or as an attribute of the other five. We choose the attribute approach here.

What does it mean for location (time and position) to be an attribute of any of the five basis constructs? This is really a short-coming of nearly all requirements specification notations and techniques. (This is probably also the reason that interactions are seldom recognized and tested.) Information about location is usually created when a system is implemented. Sometimes location is mandated as a requirement — when this happens, the requirement is really a forced implementation choice. We first clarify the meaning of the components of location: time and position.

We can take two views of time: as an instant or as a duration. The instantaneous view lets us describe when something happens — it is a point when time is an axis. The duration view is an interval on the time axis. When we think about durations, we usually are interested in the length of the time interval, not the endpoints (the start and finish times). Both views are useful. Because threads execute, they have a duration; they also have points in time when they execute. Similar observations apply to events. Often events have very short durations, and this problematic if the duration is so short that the event isn't recognized by the system.

The position aspect is easier. We could take a very tangible, physical view of position and describe it in terms of some coordinate system. Position can be a three dimensional Cartesian coordinate system with respect to some origin, or it could be a longitude-latitude-elevation geographic point. For most systems, it is more helpful to slightly abstract position into processor residence. Taken together, time and position tell the tester when and where something happens, and this is essential to understand interactions.

Before we develop our taxonomy, we need some ground rules about threads and processors. For now, a processor is something that executes threads, or a device where events occur.

1. Since threads execute, they have a strictly positive time duration. We usually speak of the execution time of a thread, but we might also be interested in when a thread occurs (executes). Since actions are degenerate cases of threads, actions also have durations.

2. In a single processor, two threads cannot execute simultaneously. This resembles a fundamental precept of physics: no two bodies may occupy the same space at the same time. Sometimes threads appear to be simultaneous, as in time sharing on a single processor; in fact, time shared threads are interleaved. Even though threads cannot execute simultaneously on a single processor, events can be simultaneous. (This is really problematic for testers.)

3. Events have a strictly positive time duration. When we consider events to be actions that execute on port devices, this reduces to the first ground rule.

4. Two (or more) input events can occur simultaneously, but an event cannot occur simultaneously in two (or more) processors. This is immediately clear if we consider port devices to be separate processors.

5. In a single processor, two output events cannot begin simultaneously. This is a direct consequence of output events being caused by thread executions. We need both the instantaneous and duration views of time to fully explain this ground rule. Suppose two output events are such that the duration of one is much greater than the duration of the other. The durations may overlap (because they occur on separate devices), but the start times cannot be identical, as shown in Figure 16.1. Thee is an example of this in the SATM system, when a thread causes screen 15 to be displayed and the ejects that ATM card. The screen is still being displayed when the card eject event occurs. (This may be a fine distinction; we could also say that port devices are separate processors, and that port output events are really a form of inter-processor communication.)

Figure 5.8.1 Overlapping Events

6. A thread cannot span more than one processor. This convention helps in the definition of threads. by confining a thread to a single processor, we create a natural endpoint for threads; this also results in more simple threads rather than fewer complex threads. In a multi-processing setting, this choice also results in another form of quiescence — trans-processor quiescence.

Taken together, these six ground rules force what we might call “same behavior” onto the interactions in the taxonomy we define in section 16.3.

5.9 Interaction, Composition, and Determinism

The question of non-determinism looms as a backdrop to deep questions in science and philosophy. Einstein didn’t believe in non-determinism; he once commented that he doubted that God would play dice with the universe. Non-determinism generally refers to consequences of random events, asking in effect, if there are truly random events (inputs), can we ever predict their consequences? The logical extreme of this debate ends in the philosophical/theological question of free will versus pre-destination. Fortunately, for testers, the software version of non-determinism is less severe. You might want to consider this section to be a technical editorial. It is based on my experience and analysis using the OSD framework. I find it yields reasonable answers to the problem of non-determinism; you may too.

Let's start with a working definition if determinism; here are two possibilities:

1. A system is deterministic if, given its inputs, we can always predict its outputs.
2. A system is deterministic if it always produces the same outputs for a given set of inputs.

Since the second view (repeatable outputs) is less stringent than the first (predictable outputs), we'll use it as our working definition. Then a non-deterministic system is one in which there is at least one set of inputs that results in two distinct sets of outputs. It's easy to devise a non-deterministic finite state machine; Figure 16.11 is one example.

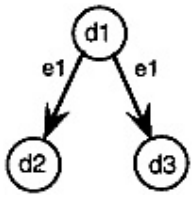


Figure 5.11 A Non-deterministic Finite State Machine

When the machine in Figure 5.11 is in state d1, if event e1 occurs, there is a transition either to state d2 or to d3.

If it is so easy to create a non-deterministic finite state machine, why all the fuss about determinism in the first place? (It turns out that we can always find a deterministic equivalent to any non-deterministic finite state machine anyway.) Finite state machines are models of reality; they only approximate the behavior of a real system. This is why it is so important to choose an appropriate model — we would like to use the best approximation. Roughly speaking, decision tables are the mode I of choice for static interactions, finite state machines suffice for dynamic interactions in a single processor, and some form of Petri net is needed for dynamic interactions in multiple processors. Before going on, we should indicate instances of non-determinism in the other two models. A multiple hit decision table is one in which the inputs (variables in the condition stub) are such that more than one rule is selected. In Petri nets, non-determinism occurs when more than one transition is enabled. The choice of which rule executes or which transition fires is made by an external agent. (Notice that the choice is actually an input!)

Our question of non-determinism reduces to threads in an OSD net, and this is where interactions, composition, and determinism come together. To ground our discussion in something "real", consider the SATM threads we used earlier:

T1: withdraw $40.00

T2: withdraw $60.00

T3: deposit $30.00

Threads T1, T2, and T3 interact via a data place for the account balance, and they may be executed in different processors. The initial balance is $50.00.

Begin with thread TI; if no other thread executes, it will execute correctly, leaving a balance of $10.00. Suppose we began with thread T2; we should really call it "attempt to withdraw $60.00", because, if no other thread executes, it will result in the insufficient funds screen. We should really separate T2 into two threads, T2.1 which is a successful withdrawal that ends with the display of screen 11 (take cash), and T2.2 which is a failed withdrawal that ends with the display of screen 8 (insufficient funds). Now let's add some interaction with thread T3. Threads T2 and T3 are 2-connected via the balance data place. If T3 executes before T2 reads the balance data, then T2.1 occurs, otherwise T2.2 occurs. The difference between the two views of determinism is visible here: When the OSD net of T2 begins to execute, we cannot predict the outcome (T2.1 or T2.2), so by the first definition, this is non-deterministic. By the second definition, however, we can recreate the interaction (including times) between T2 and T2. If we do, and we capture the behavior as a marking of the composite OSD net, we will satisfy the repeatable definition of determinism.

5.11 Client-Server Testing

Client-server systems are difficult to test because they exhibit the most difficult form of interactions, the dynamic ones across multiple processors. Here we can enjoy the benefits of our strong theoretical development. Client-server systems always entail at least two processors, one where the server software exists and executes, and one (usually several) where the client software executes. The main components are usually a database management system, application programs that use the database, and presentation programs that produce user-defined output. The position of these components results in the fat server vs. fat client distinction [Lewis 94] (see Figure 5.12).

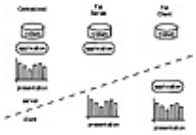


Figure 5.12 Fat Clients and Servers

Client-server systems also include a network to connect the clients with the server, the network software, and a graphical user interface (GUI) for the clients. To make matters worse, we can differentiate homogeneous and heterogeneous CS systems in terms of client processors that are identical or diverse. The multiple terminal version of the SATM system would be a fat client system, since the central bank does very little of the transaction processing.

EXERCISES

1. Figure 16.4 has a nice connection with set theory. Take the propositions p, q, r, and s to be the following set theory statements about some sets S and P:

p: $S \subseteq P$

q: $S \cap P = \emptyset$

r: $S \cap P \neq \emptyset$

s: $S \not\subset P$

Convince yourself that the relationships in the Square of Opposition apply to these set theory propositions.

2. Find and discuss examples of n-connectivity for the events in Table 4.

3. The Central ATM system (CATM) is the "other side" of the SATM system; it supports the following activities:

a. Open and Close bank accounts.

b. Maintain daily balances of accounts to reflect the twice daily postings of SATM transactions. These occur at 9:00 am and 3:00 pm.

c. Provide Expected PIN and account balance information to an SATM terminal,

d. Apply a service charge ($1.00) to any account that shows more than three ATM transactions in a given day.

Consider the CATM and SATM functions, and allocate these to Fat Server and Fat Client formulations.

4. List examples of the various kinds of interactions you can find in the combined CATM/SATM system. Decide whether these interactions are affected by the fat server vs. the fat client formulations.

UNIT 6

PROCESS FRAMEWORK

6.1 Validation and Verification

While software products and processes may be judged on several properties ranging from time-to-market to performance to usability, the software test and analysis techniques we consider are focused more narrowly on improving or assessing dependability. Assessing the degree to which a software system actually fulfills its requirements, in the sense of meeting the user's real needs, is called *validation.* Fulfilling requirements is not the same as conforming to a requirements specification. A specification is a statement about a particular proposed solution to a problem, and that proposed solution may or may not achieve its goals. Moreover, specifications are written by people, and therefore contain mistakes. A system that meets its actual goals is *useful*, while a system that is consistent with its specification is *dependable*.

"Verification" is checking the consistency of an implementation with a specification. Here, "specification" and "implementation" are roles, not particular artifacts. For example, an overall design could play the role of "specification" and a more detailed design could play the role of "implementation"; checking whether the detailed design is consistent with the overall design would then be verification of the detailed design. Later, the same detailed design could play the role of "specification" with respect to source code, which would be verified against the design. In every case, though, verification is a check of consistency between two descriptions, in contrast to validation which compares a description (whether a requirements specification, a design, or a running system) against actual needs. Figure 5.1 sketches the relation of verification and validation activities with respect to artifacts produced in a software development project. The figure should not be interpreted as prescribing a sequential process, since the goal of a consistent set of artifacts and user satisfaction are the same whether the software artifacts (specifications, design, code, etc.) are developed sequentially, iteratively, or in parallel.

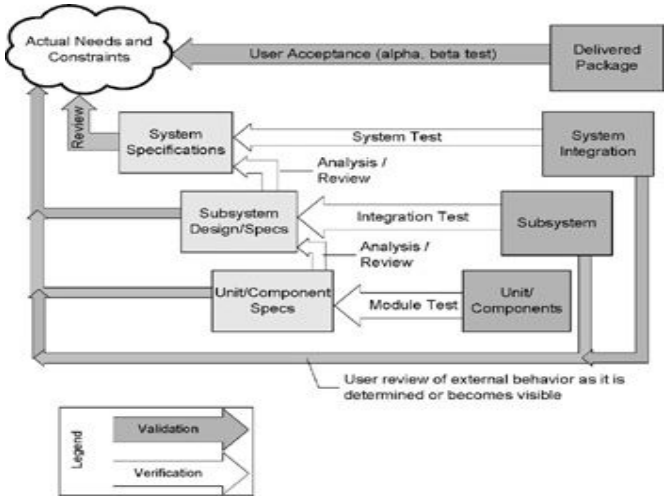


Figure 2.1: Validation activities check work products against actual user requirements, while verification activities check consistency of work products.

Validation activities refer primarily to the overall system specification and the final code. With respect to overall system specification, validation checks for discrepancies between actual needs and the system specification as laid out by the analysts, to ensure that the specification is an adequate guide to building a product that will fulfill its goals. With respect to final code, validation aims at checking discrepancies between actual need and the final product, to reveal possible failures of the development process and to make sure the product meets end-user expectations. Validation checks between the specification and final product are primarily checks of decisions that were left open in the specification (e.g., details of the user interface or product features). We have omitted one important set of verification checks from Figure 2.1 to avoid clutter. In addition to checks that compare two or more artifacts, verification includes checks for self-consistency and well-formedness. For example, while we cannot judge that a program is "correct" except in reference to a specification of what it should do, we can certainly determine that some programs are "incorrect" because they are ill- formed. We may likewise determine that a specification itself is ill-formed because it is inconsistent (requires two properties that cannot both be true) or ambiguous (can be interpreted to require some property or not), or because it does not satisfy some other well-formedness constraint that we impose, such as adherence to a standard imposed by a regulatory agency.

Validation against actual requirements necessarily involves human judgment and the potential for ambiguity, misunderstanding, and disagreement. In contrast, a specification should be sufficiently precise and unambiguous that there can be no disagreement about whether a particular system behavior is acceptable. While the term *testing* is often used informally both for gauging usefulness and verifying the product, the activities differ in both goals and approach.

Dependability properties include correctness, reliability, robustness, and safety. Correctness is absolute consistency with a specification, always and in all circumstances. Correctness with respect to nontrivial specifications is almost never achieved. Reliability is a statistical approximation to correctness, expressed as the likelihood of correct behavior in expected use. Robustness, unlike correctness and reliability, weighs properties as more and less critical, and distinguishes which properties should be maintained even under exceptional circumstances in which full functionality cannot be maintained. A good requirements document, or set of documents, should include both a requirements analysis and a requirements specification, and should clearly distinguish between the two. The requirements analysis describes the problem. The specification describes a proposed solution. This is not a book about requirements engineering, but we note in passing that confounding requirements analysis with requirements specification will inevitably have negative impacts on both validation and verification.

6.2 Degrees of Freedom

Given a precise specification and a program, it seems that one ought to be able to arrive at some logically sound argument or proof that a program satisfies the specified properties. After all, if a civil engineer can perform mathematical calculations to show that a bridge will carry a specified amount of traffic, shouldn't we be able to similarly apply mathematical logic to verification of programs?

For some properties and some very simple programs, it is in fact possible to obtain a logical correctness argument, albeit at high cost. In a few domains, logical correctness arguments may even be cost-effective for a few isolated, critical components (e.g., a safety interlock in a medical device). In general, though, one cannot produce a complete logical "proof" for the full specification of practical programs in full detail. This is not just a sign that technology for verification is immature. It is, rather, a consequence of one of the most fundamental properties of computation.

Suppose we do make use of the fact that programs are executed on real machines with finite representations of memory values. Consider the following trivial Java class:

```
1 class Trivial{
2    static int sum(int a, int b) { return a+b; }
3 }
```

The Java language definition states that the representation of an int is 32 binary digits, and thus there are only $2^{32} \times 2^{32} = 264 \approx 10^{21}$ different inputs on which the method Trivial.sum() need be tested to obtain a proof of its correctness. At one nanosecond (10^{-9} seconds) per test case, this will take approximately 10^{12} seconds, or about 30,000 years.

A technique for verifying a property can be inaccurate in one of two directions (Figure 6.2). It may be *pessimistic*, meaning that it is not guaranteed to accept a program even if the program does possess the property being analyzed, or it can be *optimistic* if it may accept some programs that do not possess the property (i.e., it may not detect all violations). Some analysis techniques may give a third possible answer, "don't know." We can consider these techniques to be either optimistic or pessimistic depending on how we interpret the "don't know" result. Perfection is unobtainable, but one can choose techniques that err in only a particular direction.

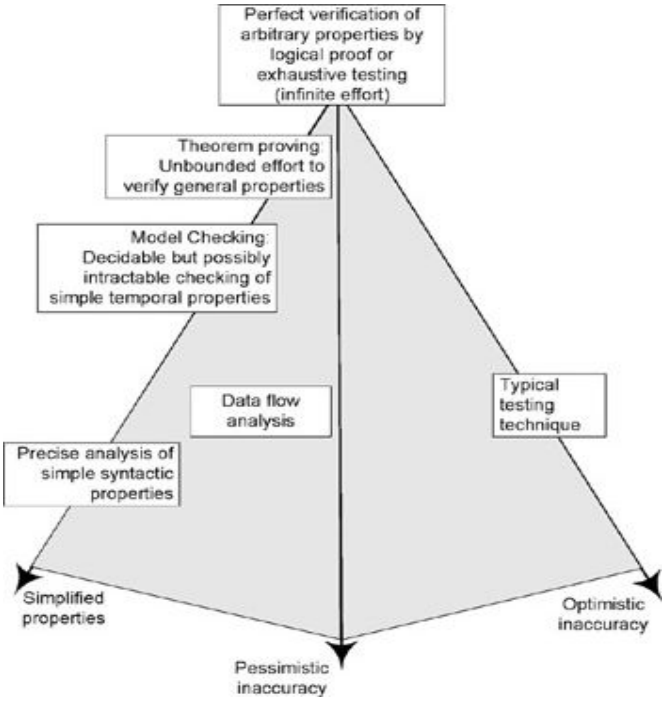


Figure 6.2: Verification trade-off dimensions

A software verification technique that errs only in the pessimistic direction is called a *conservative* analysis. It might seem that a conservative analysis would always be preferable to one that could accept a faulty program. However, a conservative analysis will often produce a very large number of spurious error reports, in addition to a few accurate reports. A human may, with some effort, distinguish real faults from a few spurious reports, but cannot cope effectively with a long list of purported faults of which most are false alarms. Often only a careful choice of complementary optimistic and pessimistic techniques can help in mutually reducing the different problems of the techniques and produce acceptable results.

Many different terms related to *pessimistic* and *optimistic* inaccuracy appear in the literature on program analysis. We have chosen these particular terms because it is fairly easy to remember which is which. Other terms a reader is likely to encounter include:

Safe A *safe* analysis has no optimistic inaccuracy; that is, it accepts only correct programs. In other kinds of program analysis, safety is related to the goal of the analysis. For example, a safe analysis related to a program optimization is one that allows that optimization only when the result of the optimization will be correct.

Sound Soundness is a term to describe evaluation of formulas. An analysis of a program *P* with respect to a formula *F* is *sound* if the analysis returns *True* only when the program actually does satisfy the formula. If satisfaction of a formula *F* is taken as an indication of correctness, then a *sound* analysis is the same as a *safe* or *conservative* analysis.

If the sense of *F* is reversed (i.e., if the truth of *F* indicates a fault rather than correctness) then a *sound* analysis is not necessarily *conservative*. In that case it is allowed optimistic inaccuracy but must not have pessimistic inaccuracy. (Note, however, that use of the term *sound* has not always been consistent in the software engineering literature. Some writers use the term *unsound* as we use the term *optimistic.*)

Complete Completeness, like soundness, is a term to describe evaluation of formulas. An analysis of a program *P* with respect to a formula *F* is *complete* if the analysis always returns *True* when the program actually does satisfy the formula. If satisfaction of a formula *F* is taken as an indication of correctness, then a *complete* analysis is one that admits only optimistic inaccuracy. An analysis that is sound but incomplete is a conservative analysis.

Many examples of substituting simple, checkable properties for actual properties of interest can be found in the design of modern programming languages. Consider, for example, the property that each variable should be initialized with a value before its value is used in an expression. In the C language, a compiler cannot provide a precise static check for this property, because of the possibility of code like the following:

```
1   int i, sum;
2   int first=1;
3   for (i=0; i<10; ++i) {
4           if (first) {
5               sum=0; first=0;
```

```
6       }
7       sum += i;
8   }
```

It is impossible in general to determine whether each control flow path can be executed, and while a human will quickly recognize that the variable sum is initialized on the first iteration of the loop, a compiler or other static analysis tool will typically not be able to rule out an execution in which the initialization is skipped on the first iteration. Java neatly solves this problem by making code like this illegal; that is, the rule is that a variable must be initialized on *all* program control paths, whether or not those paths can ever be executed.

Software developers are seldom at liberty to design new restrictions into the programming languages and compilers they use, but the same principle can be applied through external tools, not only for programs but also for other software artifacts. Consider, for example, the following condition that we might wish to impose on requirements documents:

1. Each significant domain term shall appear with a definition in the glossary of the document.

This property is nearly impossible to check automatically, since determining whether a particular word or phrase is a "significant domain term" is a matter of human judgment. Moreover, human inspection of the requirements document to check this requirement will be extremely tedious and error-prone. What can we do? One approach is to separate the decision that requires human judgment (identifying words and phrases as "significant") from the tedious check for presence in the glossary.

1.a Each significant domain term shall be set off in the requirements document by the use of a standard style *term*. The default visual representation of the *term* style is a single underline in printed documents and purple text in on-line displays.

1.b Each word or phrase in the *term* style shall appear with a definition in the glossary of the document.

Property (1a) still requires human judgment, but it is now in a form that is much more amenable to inspection. Property (1b) can be easily automated in a way that will be completely precise (except that the task of determining whether definitions appearing in the glossary are clear and correct must also be left to humans). As a second example, consider a Web-based service in which user sessions need not directly interact, but they do read and modify a shared collection of data on the server. In this case a critical property is maintaining integrity of the shared data. Testing for this property is notoriously difficult, because a "race condition" (interference between writing data in one process and reading or writing related data in another process) may cause an observable failure only very rarely.

Fortunately, there is a rich body of applicable research results on concurrency control that can be exploited for this application. It would be foolish to rely primarily on direct testing for the desired integrity properties. Instead, one would choose a (well- known, formally verified) concurrency control protocol, such as the two-phase locking protocol, and rely on some combination of static analysis and program testing to check conformance to that protocol. Imposing a particular concurrency control protocol substitutes a much simpler, *sufficient* property (two-phase locking) for the complex property of interest (serializability), at some cost in generality; that is, there are

programs that violate two-phase locking and yet, by design or dumb luck, satisfy serializability of data access.

It is a common practice to further impose a global order on lock accesses, which again simplifies testing and analysis. Testing would identify execution sequences in which data is accessed without proper locks, or in which locks are obtained and relinquished in an order that does not respect the two-phase protocol or the global lock order, even if data integrity is not violated on that particular execution, because the locking protocol failure indicates the potential for a dangerous race condition in some other execution that might occur only rarely or under extreme load.

With the adoption of coding conventions that make locking and unlocking actions easy to recognize, it may be possible to rely primarily on flow analysis to determine conformance with the locking protocol, with the role of dynamic testing reduced to a "back-up" to raise confidence in the soundness of the static analysis. Note that the critical decision to impose a particular locking protocol is *not* a post-hoc decision that can be made in a testing "phase" at the end of development. Rather, the plan for verification activities with a suitable balance of cost and assurance is part of system design.

6.3 Varieties of Software

The software testing and analysis techniques presented in the main parts of this book were developed primarily for procedural and object-oriented software. While these "generic" techniques are at least partly applicable to most varieties of software, particular application domains (e.g., real-time and safety-critical software) and construction methods (e.g., concurrency and physical distribution, graphical user interfaces) call for particular properties to be verified, or the relative importance of different properties, as well as imposing constraints on applicable techniques. Typically a software system does not fall neatly into one category but rather has a number of relevant characteristics that must be considered when planning verification.

Exercises

2.1 The Chipmunk marketing division is worried about the start-up time of the new version of the RodentOS operating system (an (imaginary) operating system of Chipmunk). The marketing division representative suggests a software requirement stating that the start-up time shall not be annoying to users.

Explain why this simple requirement is not verifiable and try to reformulate the requirement to make it verifiable.

2.2 Consider a simple specification language *SL* that describes systems diagrammatically in terms of *functions*, which represent data transformations and correspond to nodes of the diagram, and *flows*, which represent data flows and correspond to arcs of the diagram.[4] Diagrams can be hierarchically refined by associating a function *F* (a node of the diagram) with an *SL* specification that details function *F*. Flows are labeled to indicate the type of data.

Suggest some checks for self-consistency for *SL*.

2.3 A calendar program should provide *timely* reminders; for example, it should remind the user of an upcoming event early enough for the user to take action, but not too early. Unfortunately, "early enough" and "too early" are qualities that can only be validated with actual users. How might you derive verifiable dependability properties from the timeliness requirement?

2.4 It is sometimes important in multi-threaded applications to ensure that a sequence of accesses by one thread to an aggregate data structure (e.g., some kind of table) appears to other threads as an atomic transaction. When the shared data structure is maintained by a database system, the database system typically uses concurrency control protocols to ensure the atomicity of the transactions it manages. No such automatic support is typically available for data structures maintained by a program in main memory.

Among the options available to programmers to ensure serializability (the illusion of atomic access) are the following:

- The programmer could maintain very coarse-grain locking, preventing any interleaving of accesses to the shared data structure, even when such interleaving would be harmless. (For example, each transaction could be encapsulated in an single synchronized Java method.) This approach can cause a great deal of unnecessary blocking between threads, hurting performance, but it is almost trivial to verify either automatically or manually.
- Automated static analysis techniques can sometimes verify serializability with finer-grain locking, even when some methods do not use locks at all. This approach can still reject some sets of methods that would ensure serializability.
- The programmer could be required to use a particular concurrency control protocol in his or her code, and we could build a static analysis tool that checks for conformance with that protocol. For example, adherence to the common two-phase-locking protocol, with a few restrictions, can be checked in this way.
- We might augment the data accesses to build a *serializability graph* structure representing the "happens before" relation among transactions in testing. It can be shown that the transactions executed in serializable manner if and only if the serializability graph is acyclic.

Compare the relative positions of these approaches on the three axes of verification techniques: pessimistic inaccuracy, optimistic inaccuracy, and simplified properties.

2.5 When updating a program (e.g., for removing a fault, changing or adding a functionality), programmers may introduce new faults or expose previously hidden faults. To be sure that the updated version maintains the functionality provided by the previous version, it is common practice to reexecute the test cases designed for the former versions of the program. Reexecuting test cases designed for previous versions is called regression testing.

When testing large complex programs, the number of regression test cases may be large. If updated software must be expedited (e.g., to repair a security vulnerability before it is exploited), test designers may need to select a subset of regression test cases to be reexecuted. Subsets of test cases can be selected according to any of several different criteria. An interesting property of some regression test selection criteria is that they do not to exclude any test case that could possibly reveal a fault.

6.3 Basic principles

Analysis and testing (A&T) has been common practice since the earliest software projects. A&T activities were for a long time based on common sense and individual skills. It has emerged as a distinct discipline only in the last three decades.

This chapter advocates six principles that characterize various approaches and techniques for analysis and testing: sensitivity, redundancy, restriction, partition, visibility, and feedback. Some of these principles, such as partition, visibility, and feedback, are quite general in engineering. Others, notably sensitivity, redundancy, and restriction, are specific to A&T and contribute to characterizing A&T as a discipline.

6.3.1 Sensitivity

Human developers make errors, producing faults in software. Faults may lead to failures, but faulty software may not fail on every execution. The sensitivity principle states that it is better to fail every time than sometimes. Consider the cost of detecting and repairing a software fault. If it is detected immediately (e.g., by an on-the-fly syntactic check in a design editor), then the cost of correction is very small, and in fact the line between fault prevention and fault detection is blurred. If a fault is detected in inspection or unit testing, the cost is still relatively small. If a fault survives initial detection efforts at the unit level, but triggers a failure detected in integration testing, the cost of correction is much greater. If the first failure is detected in system or acceptance testing, the cost is very high indeed, and the most costly faults are those detected by customers in the field.

A fault that triggers a failure on every execution is unlikely to survive past unit testing. A characteristic of faults that escape detection until much later is that they trigger failures only rarely, or in combination with circumstances that seem unrelated or are difficult to control. For example, a fault that results in a failure only for some unusual configurations of customer equipment may be difficult and expensive to detect. A fault that results in a failure randomly but very rarely - for example, a race condition that only occasionally causes data corruption - may likewise escape detection until the software is in use by thousands of customers, and even then be difficult to diagnose and correct.

The small C program in Figure 6.1 has three faulty calls to string copy procedures. The call to strcpy, strncpy, and stringCopy all pass a source string "Muddled," which is too long to fit in the array middle. The vulnerability of strcpy is well known, and is the culprit in the by-now-standard buffer overflow attacks on many network services. Unfortunately, the fault may or may not cause an observable failure depending on the arrangement of memory (in this case, it depends on what

appears in the position that would be middle[7], which will be overwritten with a newline character). The standard recommendation is to use strncpy in place of strcpy. While strncpy avoids overwriting other memory, it truncates the input without warning, and sometimes without properly null-terminating the output. The replacement function stringCopy, on the other hand, uses an assertion to ensure that, if the target string is too long, the program always fails in an observable manner.

```
1 /**
2 * Worse than broken: Are you feeling lucky?
3 */
4
5 #include <assert.h>
6
7  char before[ ] = "=Before=";
8  char middle[ ] = "Middle";
9  char after[ ] = "=After=";
10
11 void show() {
12   printf("%s\n%s\n%s\n", before, middle, after);
13 }
14
15 void stringCopy(char *target, const char *source, int howBig);
16
17 int main(int argc, char *argv) {
18   show();
19   strcpy(middle, "Muddled"); /* Fault, but may not fail */
20   show();
21   strncpy(middle, "Muddled", sizeof(middle)); /* Fault, may not fail */
22   show();
23   stringCopy(middle, "Muddled",sizeof(middle)); /* Guaranteed to fail */
24   show();
25 }
26
27 /* Sensitive version of strncpy; can be counted on to fail
28 * in an observable way EVERY time the source is too large
29 * for the target, unlike the standard strncpy or strcpy.
30 */
31 void stringCopy(char *target, const char *source, int howBig) {
32   assert(strlen(source) < howBig);
33   strcpy(target, source);
34 }
```

Figure 6.1: Standard C functions *strcpy* and strncpy may or may not fail when the source string is too long. The procedure *stringCopy* is *sensitive*: It is guaranteed to fail in an observable way if the source string is too long. The sensitivity principle says that we should try to make these faults

easier to detect by making them cause failure more often. It can be applied in three main ways: at the design level, changing the way in which the program fails; at the analysis and testing level, choosing a technique more reliable with respect to the property of interest; and at the environment level, choosing a technique that reduces the impact of external factors on the results.

Replacing strcpy and strncpy with stringCopy in the program of Figure 6.1 is a simple example of application of the sensitivity principle in design. Run-time array bounds checking in many programming languages (including Java but not C or C++) is an example of the sensitivity principle applied at the language level. A variety of tools and replacements for the standard memory management library are available to enhance sensitivity to memory allocation and reference faults in C and C++. The fail-fast property of Java iterators is another application of the sensitivity principle. A Java iterator provides a way of accessing each item in a collection data structure. Without the fail-fast property, modifying the collection while iterating over it could lead to unexpected and arbitrary results, and failure might occur rarely and be hard to detect and diagnose. A fail-fast iterator has the property that an immediate and observable failure (throwing ConcurrentModificationException) occurs when the illegal modification occurs. Although fail-fast behavior is not guaranteed if the update occurs in a different thread, a fail-fast iterator is far more sensitive than an iterator without the fail-fast property.

Redundancy

Redundancy is the opposite of independence. If one part of a software artifact (program, design document, etc.) constrains the content of another, then they are not entirely independent, and it is possible to check them for consistency. The concept and definition of redundancy are taken from information theory. In communication, redundancy can be introduced into messages in the form of error- detecting and error-correcting codes to guard against transmission errors. In software test and analysis, we wish to detect faults that could lead to differences between intended behavior and actual behavior, so the most valuable form of redundancy is in the form of an explicit, redundant statement of intent.

Where redundancy can be introduced or exploited with an automatic, algorithmic check for consistency, it has the advantage of being much cheaper and more thorough than dynamic testing or manual inspection. Static type checking is a classic application of this principle: The type declaration is a statement of intent that is at least partly redundant with the use of a variable in the source code. The type declaration constrains other parts of the code, so a consistency check (type check) can be applied. An important trend in the evolution of programming languages is introduction of additional ways to declare intent and automatically check for consistency. For example, Java enforces rules about explicitly declaring each exception that can be thrown by a method.

Checkable redundancy is not limited to program source code, nor is it something that can be introduced only by programming language designers. For example, software design tools typically provide ways to check consistency between different design views or artifacts. One can also intentionally introduce redundancy in other software artifacts, even those that are not entirely formal. For example, one might introduce rules quite analogous to type declarations for

semistructured requirements specification documents, and thereby enable automatic checks for consistency and some limited kinds of completeness. When redundancy is already present - as between a software specification document and source code - then the remaining challenge is to make sure the information is represented in a way that facilitates cheap, thorough consistency checks. Checks that can be implemented by automatic tools are usually preferable, but there is value even in organizing information to make inconsistency easier to spot in manual inspection. Of course, one cannot always obtain cheap, thorough checks of source code and other documents. Sometimes redundancy is exploited instead with run-time checks. Defensive programming, explicit run-time checks for conditions that should always be true if the program is executing correctly, is another application of redundancy in programming.

Restriction

When there are no acceptably cheap and effective ways to check a property, sometimes one can change the problem by checking a different, more restrictive property or by limiting the check to a smaller, more restrictive class of programs.

Consider the problem of ensuring that each variable is initialized before it is used, on every execution. Simple as the property is, it is not possible for a compiler or analysis tool to precisely determine whether it holds. See the program in Figure 6.2 for an illustration. Can the variable k ever be uninitialized the first time i is added to it? If someCondition(0) always returns true, then k will be initialized to zero on the first time through the loop, before k is incremented, so perhaps there is no potential for a run-time error - but method someCondition could be arbitrarily complex and might even depend on some condition in the environment. Java's solution to this problem is to enforce a stricter, simpler condition: A program is not permitted to have any syntactic control paths on which an uninitialized reference could occur, regardless of whether those paths could actually be executed. The program in Figure 6.2 has such a path, so the Java compiler rejects it.

```
1   /** A trivial method with a potentially uninitialized variable.
2     * Maybe someCondition(0) is always true, and therefore k is
3     * always initialized before use ... but it's impossible, in
4     * general, to know for sure. Java rejects the method.
5     */
6   static void questionable() {
7         int k;
8         for (int i=0; i < 10; ++i) {
9            if (someCondition(i)) {
10               k=0;
11            } else {
12               k+=i;
13            }
14        }
15        System.out.println(k);
16  }
17 }
```

Figure 3.2: Can the variable *k* ever be uninitialized the first time *i* is added to it? The property is undecidable, so Java enforces a simpler, stricter property.

Java's rule for initialization before use is a program source code restriction that enables precise, efficient checking of a simple but important property by the compiler. The choice of programming language(s) for a project may entail a number of such restrictions that impact test and analysis. Additional restrictions may be imposed in the form of programming standards (e.g., restricting the use of type casts or pointer arithmetic in C), or by tools in a development environment. Other forms of restriction can apply to architectural and detailed design. Consider, for example, the problem of ensuring that a transaction consisting of a sequence of accesses to a complex data structure by one process appears to the outside world as if it had occurred atomically, rather than interleaved with transactions of other processes. This property is called *serializability*: The end result of a set of such transactions should appear as if they were applied in some serial order, even if they didn't.

Partition

Partition, often also known as "divide and conquer," is a general engineering principle. Dividing a complex problem into subproblems to be attacked and solved independently is probably the most common human problem-solving strategy. Software engineering in particular applies this principle in many different forms and at almost all development levels, from early requirements specifications to code and maintenance. Analysis and testing are no exception: the partition principle is widely used and exploited. Partitioning can be applied both at process and technique levels. At the process level, we divide complex activities into sets of simple activities that can be attacked independently. For example, testing is usually divided into unit, integration, subsystem, and system testing. In this way, we can focus on different sources of faults at different steps, and at each step, we can take advantage of the results of the former steps. For instance, we can use units that have been tested as stubs for integration testing. Some static analysis techniques likewise follow the modular structure of the software system to divide an analysis problem into smaller steps.

Visibility

Visibility means the ability to measure progress or status against goals. In software engineering, one encounters the visibility principle mainly in the form of process visibility, and then mainly in the form of schedule visibility: ability to judge the state of development against a project schedule. Quality process visibility also applies to measuring achieved (or predicted) quality against quality goals. The principle of visibility involves setting goals that can be assessed as well as devising methods to assess their realization.

Visibility is closely related to observability, the ability to extract useful information from a software artifact. The architectural design and build plan of a system determines what will be observable at each stage of development, which in turn largely determines the visibility of progress against goals at that stage.

A variety of simple techniques can be used to improve observability. For example, it is no accident that important Internet protocols like HTTP and SMTP (Simple Mail Transport Protocol, used by Internet mail servers) are based on the exchange of simple textual commands. The choice of simple, human-readable text rather than a more compact binary encoding has a small cost in performance and a large payoff in observability, including making construction of test drivers and oracles much simpler. Use of human-readable and human-editable files is likewise advisable wherever the performance cost is acceptable.

Feedback

Feedback is another classic engineering principle that applies to analysis and testing. Feedback applies both to the process itself (process improvement) and to individual techniques (e.g., using test histories to prioritize regression testing).

Systematic inspection and walkthrough derive part of their success from feedback. Participants in inspection are guided by checklists, and checklists are revised and refined based on experience. New checklist items may be derived from root cause analysis, analyzing previously observed failures to identify the initial errors that lead to them.

Exercises

3.1 Indicate which principles guided the following choices:

1. Use an externally readable format also for internal files, when possible.
2. Collect and analyze data about faults revealed and removed from the code.
3. Separate test and debugging activities; that is, separate the design and execution of test cases to reveal failures (test) from the localization and removal of the corresponding faults (debugging).
4. Distinguish test case design from execution.
5. Produce complete fault reports.
6. Use information from test case design to improve requirements and design specifications.
7. Provide interfaces for fully inspecting the internal state of a class.

3.2 A simple mechanism for augmenting fault tolerance consists of replicating computation and comparing the obtained results. Can we consider redundancy for fault tolerance an application of the redundancy principle?

3.3 A system safety specification describes prohibited behaviors (what the system must never do). Explain how specified safety properties can be viewed as an implementation of the redundancy principle.

3.4 Process visibility can be increased by extracting information about the progress of the process. Indicate some information that can be easily produced to increase process visibility.

6.2 The Quality Process

One can identify particular activities and responsibilities in a software development process that are focused primarily on ensuring adequate dependability of the software product, much as one can identify other activities and responsibilities concerned primarily with project schedule or with product usability. It is convenient to group these quality assurance activities under the rubric "quality process," although we must also recognize that quality is intertwined with and inseparable from other facets of the overall process. Like other parts of an overall software process, the quality process provides a framework for selecting and arranging activities aimed at a particular goal, while also considering interactions and trade-offs with other important goals. All software development activities reflect constraints and trade-offs, and quality activities are no exception. For example, high dependability is usually in tension with time to market, and in most cases it is better to achieve a reasonably high degree of dependability on a tight schedule than to achieve ultra-high dependability on a much longer schedule, although the opposite is true in some domains (e.g., certain medical devices).

6.2.1 Planning and Monitoring

Process visibility is a key factor in software process in general, and software quality processes in particular. A process is visible to the extent that one can answer the question, "How does our progress compare to our plan?" Typically, schedule visibility is a main emphasis in process design ("Are we on schedule? How far ahead or behind?"), but in software quality process an equal emphasis is needed on progress against quality goals. If one cannot gain confidence in the quality of the software system long before it reaches final testing, the quality process has not achieved adequate visibility. A well-designed quality process balances several activities across the whole development process, selecting and arranging them to be as cost-effective as possible, and to improve early visibility. Visibility is particularly challenging and is one reason (among several) that quality activities are usually placed as early in a software process as possible. For example, one designs test cases at the earliest opportunity (not "just in time") and uses both automated and manual static analysis techniques on software artifacts that are produced before actual code.

6.2.2 Quality Goals

Process visibility requires a clear specification of goals, and in the case of quality process visibility this includes a careful distinction among dependability qualities. A team that does not have a clear idea of the difference between reliability and robustness, for example, or of their relative importance in a project, has little chance of attaining either. Goals must be further refined into a clear and reasonable set of objectives. If an organization claims that nothing less than 100% reliability will suffice, it is not setting an ambitious objective. Rather, it is setting no objective at all, and choosing not to make reasoned trade-off decisions or to balance limited resources across various activities. It is, in effect, abrogating responsibility for effective quality planning, and leaving trade-offs among cost, schedule, and quality to an arbitrary, ad hoc decision based on deadline and budget alone. The relative importance of qualities and their relation to other project objectives varies. Time-to-market may be the most important property for a mass market product,

usability may be more prominent for a Web based application, and safety may be the overriding requirement for a life-critical system.

The external properties of software can ultimately be divided into dependability (does the software do what it is intended to do?) and usefulness. There is no precise dependability way to distinguish these, but a rule of thumb is that when software is not dependable, we say it has a fault, or a defect, or (most often) a bug, resulting in an undesirable behavior or failure. It is quite possible to build systems that are very reliable, relatively free from usefulness hazards, and completely useless. They may be unbearably slow, or have terrible user interfaces and unfathomable documentation, or they may be missing several crucial features. How should these properties be considered in software quality? One answer is that they are not part of quality at all unless they have been explicitly specified, since quality is the presence of specified properties. However, a company whose products are rejected by its customers will take little comfort in knowing that, by some definitions, they were high-quality products.

We can do better by considering quality as fulfillment of required and desired properties, as distinguished from specified properties. For example, even if a client does not explicitly specify the required performance of a system, there is always *some* level of performance that is required to be useful. One of the most critical tasks in software quality analysis is making desired properties explicit, since properties that remain unspecified (even informally) are very likely to surface unpleasantly when it is discovered that they are not met. In many cases these implicit requirements can not only be made explicit, but also made sufficiently precise that they can be made part of dependability or reliability. For example, while it is better to explicitly recognize usability as a requirement than to leave it implicit, it is better yet to augment[1] usability requirements with specific interface standards, so that a deviation from the standards is recognized as a defect.

Interface standards augment, rather than replace, usability requirements because conformance to the standards is not sufficient assurance that the requirement is met. This is the same relation that other specifications have to the user requirements they are intended to fulfill. In general, verifying conformance to specifications does not replace validating satisfaction of requirements.

6.2.3 Dependability Properties

The simplest of the dependability properties is correctness: A program or system is correct if it is consistent with its specification. By definition, a specification divides all possible system behaviours into two classes, *successes* (or correct executions) and *failures*. All of the possible behaviors of a correct system are successes.

A program cannot be mostly correct or somewhat correct or 30% correct. It is absolutely correct on all possible behaviors, or else it is not correct. It is very easy to achieve correctness, since every program is correct with respect to some (very bad) specification. Achieving correctness with respect to a useful specification, on the other hand, is seldom practical for nontrivial systems. Therefore, while correctness may be a noble goal, we are often interested in assessing some more achievable level of dependability. Reliability is a statistical approximation to correctness, in the sense that 100% reliability is indistinguishable from correctness. Roughly speaking, reliability is a measure of

the likelihood of correct function for some "unit" of behavior, which could be a single use or program execution or a period of time. Like correctness, reliability is relative to a specification (which determines whether a unit of behavior is counted as a success or failure). Unlike correctness, reliability is also relative to a particular usage profile. The same program can be more or less reliable depending on how it is used.

Particular measures of reliability can be used for different units of execution and different ways of counting success and failure. *Availability* is an appropriate measure when a failure has some duration in time. For example, a failure of a network router may make it impossible to use some functions of a local area network until the service is restored; between initial failure and restoration we say the router is "down" or "unavailable." The availability of the router is the time in which the system is "up" (providing normal service) as a fraction of total time. Thus, a network router that averages 1 hour of down time in each 24-hour period would have an availability of 2324, or 95.8%.

Mean time between failures (MTBF) is yet another measure of reliability, also using time as the unit of execution. The hypothetical network switch that typically fails once in a 24-hour period and takes about an hour to recover has a mean time between failures of 23 hours. Note that availability does not distinguish between two failures of 30 minutes each and one failure lasting an hour, while MTBF does. The definitions of correctness and reliability have (at least) two major weaknesses. First, since the success or failure of an execution is relative to a specification, they are only as strong as the specification. Second, they make no distinction between a failure that is a minor annoyance and a failure that results in catastrophe. These are simplifying assumptions that we accept for the sake of precision, but in some circumstances - particularly, but not only, for critical systems - it is important to consider dependability properties that are less dependent on specification and that do distinguish among failures depending on severity.

Software safety is an extension of the well-established field of system safety into software. Safety is concerned with preventing certain undesirable behaviors, called *hazards.* It is quite explicitly not concerned with achieving any useful behavior apart from whatever functionality is needed to prevent hazards. Software safety is typically a concern in "critical" systems such as avionics and medical systems, but the basic principles apply to any system in which particularly undesirable behaviors can be distinguished from run-of-the-mill failure. For example, while it is annoying when a word processor crashes, it is much more annoying if it irrecoverably corrupts document files. The developers of a word processor might consider safety with respect to the hazard of file corruption separately from reliability with respect to the complete functional requirements for the word processor. Just as correctness is meaningless without a specification of allowed behaviors, safety is meaningless without a specification of hazards to be prevented, and in practice the first step of safety analysis is always finding and classifying hazards. Typically, hazards are associated with some system in which the software is embedded (e.g., the medical device), rather than the software alone. The distinguishing feature of safety is that it is concerned *only* with these hazards, and not with other aspects of correct functioning.

The concept of safety is perhaps easier to grasp with familiar physical systems. For example, lawn-mowers in the United States are equipped with an interlock device, sometimes called a "dead-man switch." If this switch is not actively held by the operator, the engine shuts off. The dead-man

switch does not contribute in any way to cutting grass; its sole purpose is to prevent the operator from reaching into the mower blades while the engine runs.

One is tempted to say that safety is an aspect of correctness, because a good system specification would rule out hazards. However, safety is best considered as a quality distinct from correctness and reliability for two reasons. First, by focusing on a few hazards and ignoring other functionality, a separate safety specification can be much simpler than a complete system specification, and therefore easier to verify. To put it another way, while a good system specification *should* rule out hazards, we cannot be confident that either specifications or our attempts to verify systems are good enough to provide the degree of assurance we require for hazard avoidance. Second, even if the safety specification were redundant with regard to the full system specification, it is important because (by definition) we regard avoidance of hazards as more crucial than satisfying other parts of the system specification.

Correctness and reliability are contingent upon normal operating conditions. It is not reasonable to expect a word processing program to save changes normally when the file does not fit in storage, or to expect a database to continue to operate normally when the computer loses power, or to expect a Web site to provide completely satisfactory service to all visitors when the load is 100 times greater than the maximum for which it was designed. Software that fails under these conditions, which violate the premises of its design, may still be "correct" in the strict sense, yet the manner in which the software fails is important. It is acceptable that the word processor fails to write the new file that does not fit on disk, but unacceptable to also corrupt the previous version of the file in the attempt. It is acceptable for the database system to cease to function when the power is cut, but unacceptable for it to leave the database in a corrupt state. And it is usually preferable for the Web system to turn away some arriving users rather than becoming too slow for all, or crashing. Software that gracefully degrades or fails "softly" outside its normal operating parameters is *robust*.

Software safety is a kind of robustness, but robustness is a more general notion that concerns not only avoidance of hazards (e.g., data corruption) but also partial functionality under unusual situations. Robustness, like safety, begins with explicit consideration of unusual and undesirable situations, and should include augmenting software specifications with appropriate responses to undesirable events.

Figure 6.1 illustrates the relation among dependability properties.

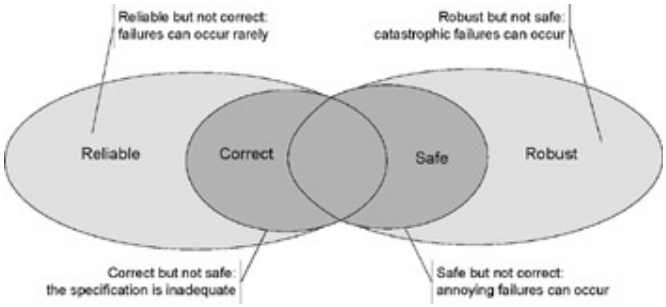


Figure 6.1: Relation among dependability properties

We are simplifying matters somewhat by considering only specifications of behaviors. A specification may also deal with other properties, such as the disk space required to install the application. A system may thus also be "incorrect" if it violates one of these static properties.

Analysis

Analysis techniques that do not involve actual execution of program source code play a prominent role in overall software quality processes. Manual inspection techniques and automated analyses can be applied at any development stage. They are particularly well suited at the early stages of specifications and design, where the lack of executability of many intermediate artifacts reduces the efficacy of testing.

.

Excerpt of Web Presence Feasibility Study

Purpose of this document

This document was prepared for the Chipmunk IT management team. It describes the results of a feasibility study undertaken to advise Chipmunk corporate management whether to embark on a substantial redevelopment effort to add online shopping functionality to the Chipmunk Computers' Web presence.

Goals

The primary goal of a Web presence redevelopment is to add online shopping facilities. Marketing estimates an increase of 15% over current direct sales within 24 months, and an additional 8% savings in direct sales support costs from shifting telephone price inquiries to online price inquiries. […]

Architectural Requirements

The logical architecture will be divided into three distinct subsystems: human interface, business logic, and supporting infrastructure. Each major subsystem must be structured for phased development, with initial features delivered 6 months from inception, full features at 12 months, and a planned revision at 18 months from project inception. […]

Quality Requirements

Dependability With the introduction of direct sales and customer relationship management functions, dependability of Chipmunk's Web services becomes businesscritical. A critical core of functionality will be identified, isolated from less critical functionality in design and implementation, and subjected to the highest level of scrutiny. We estimate that this will be approximately 20% of new development and revisions, and that the V&V costs for those portions will be approximately triple the cost of V&V for noncritical development.

Usability The new Web presence will be, to a much greater extent than before, the public face of Chipmunk Computers. […]

Security Introduction of online direct ordering and billing raises a number of security issues. Some of these can be avoided initially by contracting with one of several service companies that provide secure credit card transaction services. Nonetheless, order tracking, customer relationship management, returns, and a number of other functions that cannot be effectively outsourced raise significant security and privacy issues. Identifying and isolating security concerns will add a significant but manageable cost to design validation. […]

Sometimes the best aspects of manual inspection and automated static analysis can be obtained by carefully decomposing properties to be checked. For example, suppose a desired property of requirements documents is that each special term in the application domain appear in a glossary of terms. This property is not directly amenable to an automated static analysis, since current tools cannot distinguish meaningful domain terms from other terms that have their ordinary meanings. The property can be checked with manual inspection, but the process is tedious, expensive, and error-prone. A hybrid approach can be applied if each domain term is marked in the text. Manually checking that domain terms are marked is much faster and therefore less expensive than manually looking each term up in the glossary, and marking the terms permits effective automation of cross-checking with the glossary.

Testing

Despite the attractiveness of automated static analyses when they are applicable, and despite the usefulness of manual inspections for a variety of documents including but not limited to program source code, dynamic testing remains a dominant technique. A closer look, though, shows that dynamic testing is really divided into several distinct activities that may occur at different points in a project. Tests are executed when the corresponding code is available, but testing activities start earlier, as soon as the artifacts required for designing test case specifications are available. Thus, acceptance and system test suites should be generated before integration and unit test suites, even if executed in the opposite order.

Early test design has several advantages. Tests are specified independently from code and when the corresponding software specifications are fresh in the mind of analysts and developers, facilitating review of test design. Moreover, test cases may highlight inconsistencies and incompleteness in the corresponding software specifications. Early design of test cases also allows for early repair of software specifications, preventing specification faults from propagating to later stages in development. Finally, programmers may use test cases to illustrate and clarify the software specifications, especially for errors and unexpected conditions.

No engineer would build a complex structure from parts that have not themselves been subjected to quality control. Just as the "earlier is better" rule dictates using inspection to reveal flaws in requirements and design before they are propagated to program code, the same rule dictates module testing to uncover as many program faults as possible before they are incorporated in larger subsystems of the product. At Chip- munk, developers are expected to perform functional and structural module testing before a work assignment is considered complete and added to the project baseline. The test driver and auxiliary files are part of the work product and are expected to make

reexecution of test cases, including result checking, as simple and automatic as possible, since the same test cases will be used over and over again as the product evolves.

Improving the Process

While the assembly-line, mass production industrial model is inappropriate for software, which is at least partly custom-built, there is almost always some commonality among projects undertaken by an organization over time. Confronted by similar problems, developers tend to make the same kinds of errors over and over, and consequently the same kinds of software faults are often encountered project after project. The quality process, as well as the software development process as a whole, can be improved by gathering, analyzing, and acting on data regarding faults and failures.

The first part of a process improvement feedback loop, and often the most difficult to implement, is gathering sufficiently complete and accurate raw data about faults and failures. A main obstacle is that data gathered in one project goes mainly to benefit other projects in the future and may seem to have little direct benefit for the current project, much less to the persons asked to provide the raw data. It is therefore helpful to integrate data collection as well as possible with other, normal development activities, such as version and configuration control, project management, and bug tracking. It is also essential to minimize extra effort. For example, if revision logs in the revision control database can be associated with bug tracking records, then the time between checking out a module and checking it back in might be taken as a rough guide to cost of repair. Raw data on faults and failures must be aggregated into categories and prioritized. Faults may be categorized along several dimensions, none of them perfect. Fortunately, a flawless categorization is not necessary; all that is needed is some categorization scheme that is sufficiently fine-grained and tends to aggregate faults with similar causes and possible remedies, and that can be associated with at least rough estimates of relative frequency and cost. A small number of categories - maybe just one or two - are chosen for further study.

The analysis step consists of tracing several instances of an observed fault or failure back to the human error from which it resulted, or even further to the factors that led to that human error. The analysis also involves the reasons the fault was not detected and eliminated earlier (e.g., how it slipped through various inspections and levels of testing). This process is known as "root cause analysis," but the ultimate aim is for the most cost-effective countermeasure, which is sometimes but not always the ultimate root cause. For example, the persistence of security vulnerabilities through buffer overflow errors in network applications may be attributed at least partly to widespread use of programming languages with unconstrained pointers and without array bounds checking, which may in turn be attributed to performance concerns and a requirement for interoperability with a large body of legacy code. The countermeasure could involve differences in programming methods (e.g., requiring use of certified "safe" libraries for buffer management), or improvements to quality assurance activities (e.g., additions to inspection checklists), or sometimes changes in management practices.

Organizational Factors

The quality process includes a wide variety of activities that require specific skills and attitudes and may be performed by quality specialists or by software developers. Planning the quality process involves not only resource management but also identification and allocation of responsibilities. A poor allocation of responsibilities can lead to major problems in which pursuit of individual goals conflicts with overall project success. For example, splitting responsibilities of development and quality-control between a development and a quality team, and rewarding high productivity in terms of lines of code per person-month during development may produce undesired results. The development team, not rewarded to produce high-quality software, may attempt to maximize productivity to the detriment of quality. The resources initially planned for quality assurance may not suffice if the initial quality of code from the "very productive" development team is low. On the other hand, combining development and quality control responsibilities in one undifferentiated team, while avoiding the perverse incentive of divided responsibilities, can also have unintended effects: As deadlines near, resources may be shifted from quality assurance to coding, at the expense of product quality.

Conflicting considerations support both the separation of roles (e.g., recruiting quality specialists), and the mobility of people and roles (e.g, rotating engineers between development and testing tasks). At Chipmunk, responsibility for delivery of the new Web presence is distributed among a development team and a quality assurance team. Both teams are further articulated into groups. The quality assurance team is divided into the analysis and testing group, responsible for the dependability of the system, and the usability testing group, responsible for usability. Responsibility for security issues is assigned to the infrastructure development group, which relies partly on external consultants for final tests based on external attack attempts.

Having distinct teams does not imply a simple division of all tasks between teams by category. At Chipmunk, for example, specifications, design, and code are inspected by mixed teams; scaffolding and oracles are designed by analysts and developers; integration, system, acceptance, and regression tests are assigned to the test and analysis team; unit tests are generated and executed by the developers; and coverage is checked by the testing team before starting integration and system testing. A specialist has been hired for analyzing faults and improving the process. The process improvement specialist works incrementally while developing the system and proposes improvements at each release.

Exercises

4.1 We have stated that 100% reliability is indistinguishable from correctness, but they are not quite identical. Under what circumstance might an incorrect program be 100% reliable? *Hint:* Recall that a program may be more or less reliable depending on how it is used, but a program is either correct or incorrect regardless of usage.

4.2 We might measure the reliability of a network router as the fraction of all packets that are correctly routed, or as the fraction of total service time in which packets are correctly routed. When might these two measures be different?

4.3 If I am downloading a very large file over a slow modem, do I care more about the availability of my internet service provider or its mean time between failures?

4.4 Can a system be correct and yet unsafe?

4.5 Under what circumstances can making a system more safe make it less reliable?

4.6 Software application domains can be characterized by the relative importance of schedule (calendar time), total cost, and dependability. For example, while all three are important for game software, schedule (shipping product in September to be available for holiday purchases) has particular weight, while dependability can be somewhat relaxed. Characterize a domain you are familiar with in these terms.

4.7 Consider responsiveness as a desirable property of an Internet chat program. The informal requirement is that messages typed by each member of a chat session appear instantaneously on the displays of other users. Refine this informal requirement into a concrete specification that can be verified. Is anything lost in the refinement?

4.8 Identify some correctness, robustness and safety properties of a word processor.

UNIT 7

FAULT-BASED TESTING, TEST EXECUTION

A model of potential program faults is a valuable source of information for evaluating and designing test suites. Some fault knowledge is commonly used in functional and structural testing, for example when identifying singleton and error values for parameter characteristics in category-partition testing or when populating catalogs with erroneous values, but a fault model can also be used more directly. Fault-based testing uses a fault model directly to hypothesize potential faults in a program under test, as well as to create or evaluate test suites based on its efficacy in detecting those hypothetical faults.

Overview

Engineers study failures to understand how to prevent similar failures in the future. For example, failure of the Tacoma Narrows Bridge in 1940 led to new understanding of oscillation in high wind and to the introduction of analyses to predict and prevent such destructive oscillation in subsequent bridge design. The causes of an airline crash are likewise extensively studied, and when traced to a structural failure they frequently result in a directive to apply diagnostic tests to all aircraft considered potentially vulnerable to similar failures.

Experience with common software faults sometimes leads to improvements in design methods and programming languages. For example, the main purpose of automatic memory management in Java is not to spare the programmer the trouble of releasing unused memory, but to prevent the programmer from making the kind of memory management errors (dangling pointers, redundant deallocations, and memory leaks) that frequently occur in C and C++ programs. Automatic array bounds checking cannot prevent a programmer from using an index expression outside array bounds, but can make it much less likely that the fault escapes detection in testing, as well as limiting the damage incurred if it does lead to operational failure (eliminating, in particular, the buffer overflow attack as a means of subverting privileged programs). Type checking reliably detects many other faults during program translation.

7.1 Assumptions in Fault-Based Testing

The effectiveness of fault-based testing depends on the quality of the fault model and on some basic assumptions about the relation of the seeded faults to faults that might actually be present. In practice, the seeded faults are small syntactic changes, like replacing one variable reference by another in an expression, or changing a comparison from < to <=. We may hypothesize that these are representative of faults actually present in the program.

Put another way, if the program under test has an actual fault, we may hypothesize that it differs from another, corrected program by only a small textual change. If so, then we need merely distinguish the program from all such small variants (by selecting test cases for which either the original or the variant program fails) to ensure detection of all such faults. This is known as the

competent programmer hypothesis,an assumption that the program under test is "close to" (in the sense of textual difference) a correct program.

Fault-Based Testing: Terminology

Original program The program unit (e.g., C function or Java class) to be tested.

Program location A region in the source code. The precise definition is defined relative to the syntax of a particular programming language. Typical locations are statements, arithmetic and Boolean expressions, and procedure calls.

Alternate expression Source code text that can be legally substituted for the text at a program location. A substitution is legal if the resulting program is syntactically correct (i.e., it compiles without errors).

Alternate program A program obtained from the original program by substituting an alternate expression for the text at some program location.

Distinct behavior of an alternate program *R* for a test *t* The behavior of an alternate program *R* is distinct from the behavior of the original program *P* for a test *t*,if *R* and *P* produce a different result for *t*, or if the output of *R* is not defined for *t*.

Distinguished set of alternate programs for a test suite *T* A set of alternate programs are distinct if each alternate program in the set can be distinguished from the original program by at least one test in *T*.

Fault-based testing can guarantee fault detection only if the competent programmer hypothesis and the coupling effect hypothesis hold. But guarantees are more than we expect from other approaches to designing or evaluating test suites, including the structural and functional test adequacy criteria discussed in earlier chapters. Fault-based testing techniques can be useful even if we decline to take the leap of faith required to fully accept their underlying assumptions. What is essential is to recognize the dependence of these techniques, and any inferences about software quality based on fault-based testing, on the quality of the fault model. This also implies that developing better fault models, based on hard data about real faults rather than guesses, is a good investment of effort.

7.2 Mutation Analysis

Mutation analysis is the most common form of software fault-based testing. A fault model is used to produce hypothetical faulty programs by creating variants of the program under test. Variants are created by "seeding" faults, that is, by making a small change to the program under test following a pattern in the fault model. The patterns for changing program text are called *mutation operators*, and each variant program is called a *mutant*.

Mutation Analysis: Terminology

Original program under test The program or procedure (function) to be tested.

Mutant A program that differs from the original program for one syntactic element (e.g., a statement, a condition, a variable, a label).

Distinguished mutant A mutant that can be distinguished for the original program by executing at least one test case.

Equivalent mutant A mutant that cannot be distinguished from the original program.

Mutation operator A rule for producing a mutant program by syntactically modifying the original program.

Mutants should be plausible as faulty programs. Mutant programs that are rejected by a compiler, or that fail almost all tests, are not good models of the faults we seek to uncover with systematic testing.

We say a mutant is valid if it is syntactically correct. A mutant obtained from the program of Figure 7.1 by substituting while for switch in the statement at line 13 would not be valid, since it would result in a compile-time error. We say a mutant is useful if, in addition to being valid, its behavior differs from the behavior of the original program for no more than a small subset of program test cases. A mutant obtained by substituting 0 for 1000 in the statement at line 4 would be valid, but not useful, since the mutant would be distinguished from the program under test by all inputs and thus would not give any useful information on the effectiveness of a test suite. Defining mutation operators that produce valid and useful mutations is a nontrivial task.

```
 1
 2  /** Convert each line from standard input */
 3  void transduce() {
 4     #define BUFLEN 1000
 5     char buf[BUFLEN]; /* Accumulate line into this buffer */
 6     int pos=0; /* Index for next character in buffer */
 7
 8     char inChar; /* Next character from input */
 9
10     int atCR = 0; /* 0="within line", 1="optional DOS LF" */
11
12     while ((inChar = getchar()) != EOF ) {
13       switch (inChar) {
14       case LF:
15         if (atCR) { /* Optional DOS LF */
16           atCR = 0;
17         } else {     /* Encountered CR within line */
18           emit(buf, pos);
19           pos=0;
20         }
21         break;
22       case CR:
23         emit(buf, pos);
24         pos=0;
25         atCR = 1;
```

```
26        break;
27      default:
28        if (pos >= BUFLEN-2) fail("Buffer overflow");
29        buf[pos++] = inChar;
30      }/* switch */
31    }
32    if (pos > 0) {
33        emit(buf, pos);
34    }
35  }
```

Figure 7.1: Program transduce converts line endings among Unix, DOS, and Macintosh conventions. The main procedure, which selects the output line end convention, and the output procedure emit are not shown.

Since mutants must be valid, mutation operators are syntactic patterns defined relative to particular programming languages. Figure 7.2 shows some mutation operators for the C language. Constraints are associated with mutation operators to guide selection of test cases likely to distinguish mutants from the original program. For example, the mutation operator *svr* (scalar variable replacement) can be applied only to variables of compatible type (to be valid), and a test case that distinguishes the mutant from the original program must execute the modified statement in a state in which the original variable and its substitute have different values.

⇨Open table as spreadsheet **ID**	**Operator**	**Description**	**Constraint**
Operand Modifications			
crp	constant for constant replacement	replace constant $C1$ with constant $C2$	$C1 \neq C2$
scr	scalar for constant replacement	replace constant C with scalar variable X	$C \neq X$
acr	array for constant replacement	replace constant C with array reference $A[I]$	$C \neq A[I]$
scr	struct for constant replacement	replace constant C with struct field S	$C \neq S$
svr	scalar variable replacement	replace scalar variable X with a scalar variable Y	$X \neq Y$
csr	constant for scalar variable replacement	replace scalar variable X with a constant C	$X \neq C$
asr	array for scalar variable replacement	replace scalar variable X with an array reference $A[I]$	$X \neq A[I]$

➡Open table as spreadsheet **ID**	**Operator**	**Description**	**Constraint**
ssr	struct for scalar replacement	replace scalar variable X with struct field S	$X \neq S$
vie	scalar variable initialization elimination	remove initialization of a scalar variable	
car	constant for array replacement	replace array reference $A[I]$ with constant C	$A[I] \neq C$
sar	scalar for array replacement	replace array reference $A[I]$ with scalar variable X	$A[I] \neq C$
cnr	comparable array replacement	replace array reference with a comparable array reference	
sar	struct for array reference replacement	replace array reference $A[I]$ with a struct field S	$A[I] \neq S$
Expression Modifications			
abs	absolute value insertion	replace e by abs(e)	$e < 0$
aor	arithmetic operator replacement	replace arithmetic operator ψ with arithmetic operator φ	$e_1 \psi e_2 \neq e_1 \varphi e_2$
lcr	logical connector replacement	replace logical connector ψ with logical connector φ	$e_1 \psi e_2 \neq e_1 \varphi e_2$
ror	relational operator replacement	replace relational operator ψ with relational operator φ	$e_1 \psi e_2 \neq e_1 \varphi e_2$
uoi	unary operator insertion	insert unary operator	
cpr	constant for predicate replacement	replace predicate with a constant value	
Statement Modifications			
sdl	statement deletion	delete a statement	
sca	switch case replacement	replace the label of one case with another	
ses	end block shift	move } one statement earlier and later	

Figure 7.2: A sample set of mutation operators for the C language, with associated constraints to select test cases that distinguish generated mutants from the original program.

Many of the mutants of Figure 7.2 can be applied equally well to other procedural languages, but in general a mutation operator that produces valid and useful mutants for a given language may not apply to a different language or may produce invalid or useless mutants for another language. For example, a mutation operator that removes the "friend" keyword from the declaration of a C++ class would not be applicable to Java, which does not include friend classes.

7.3 Fault-Based Adequacy Criteria

Given a program and a test suite *T*, mutation analysis consists of the following steps:

Select mutation operators If we are interested in specific classes of faults, we may select a set of mutation operators relevant to those faults.

Generate mutants Mutants are generated mechanically by applying mutation operators to the original program.

Distinguish mutants Execute the original program and each generated mutant with the test cases in *T* . A mutant is *killed* when it can be distinguished from the original program.

Figure 7.3 shows a sample of mutants for program Transduce, obtained by applying the mutant operators in Figure 16.2(See 7.8.3). Test suite *TS*

$$TS = \{1U, 1D, 2U, 2D, 2M, End, Long\}$$

➪Open table as spreadsheet **ID**	**Operator**	**line**	**Original/Mutant**	**1U**	**1D**	**2U**	**2D**	**2M**	**End**	**Long**	**Mixed**
M_i	ror	28	(pos >= BUFLEN–2) (pos == BUFLEN–2)	-	-	-	-	-	-	-	-
M_j	ror	32	(pos > 0) (pos >= 0)	-	x	x	x	x	-	-	-
M_k	sdl	16	atCR = 0 *nothing*	-	-	-	-	-	-	-	-
M_l	ssr	16	atCR = 0 pos = 0	-	-	-	-	-	-	-	x

➪Open table as spreadsheet **Test case**	**Description**	**Test case**	**Description**

➪Open table as spreadsheet **Test case**	**Description**	**Test case**	**Description**
1U	One line, Unix line-end	2M	Two lines, Mac line-end
1D	One line, DOS line-end	End	Last line not terminated with line-end sequence
2U	Two lines, Unix line-end	Long	Very long line (greater than buffer length)
2D	Two lines, DOS line-end	Mixed	Mix of DOS and Unix line ends in the same file

Figure 7.3: A sample set of mutants for program *Transduce* generated with mutation operators from Figure 7.2 x indicates the mutant is killed by the test case in the column head.

kills M_j, which can be distinguished from the original program by test cases 1*D*,2*U*, 2*D*, and 2*M*. Mutants M_i, M_k, and M_l are not distinguished from the original program by any test in *TS*. We say that mutants not killed by a test suite are *live*.

A mutant can remain *live* for two reasons:

- The mutant can be distinguished from the original program, but the test suite *T* does not contain a test case that distinguishes them (i.e., the test suite is not adequate with respect to the mutant).
- The mutant cannot be distinguished from the original program by any test case (i.e., the mutant is equivalent to the original program).

Given a set of mutants *SM* and a test suite *T*, the fraction of nonequivalent mutants killed by *T* measures the adequacy of *T* with respect to *SM*. Unfortunately, the problem of identifying equivalent mutants is undecidable in general, and we could err either by claiming that a mutant is equivalent to the program under test when it is not or by counting some equivalent mutants among the remaining live mutants.

The adequacy of the test suite *TS* evaluated with respect to the four mutants of Figure 16.3 is 25%. However, we can easily observe that mutant *Mi* is equivalent to the original program (i.e., no input would distinguish it). Conversely, mutants M_k and M_l seem to be nonequivalent to the original program: There should be at least one test case that distinguishes each of them from the original program. Thus the adequacy of *TS*, measured after eliminating the equivalent mutant M_i, is 33%.

Mutant M_l is killed by test case *Mixed*, which represents the unusual case of an input file containing both DOS- and Unix-terminated lines. We would expect that *Mixed* would also kill M_k, but this does not actually happen: Both *Mk* and the original program produce the same result for *Mixed*. This happens because both the mutant and the original program fail in the same way.[1] The use of a simple oracle for checking the correctness of the outputs (e.g., checking each output against an

expected output) would reveal the fault. The test suite *TS*2 obtained by adding test case *Mixed* to *TS* would be 100% adequate (relative to this set of mutants) after removing the fault.

Mutation Analysis vs. Structural Testing

For typical sets of syntactic mutants, a mutation-adequate test suite will also be adequate with respect to simple structural criteria such as statement or branch coverage. Mutation adequacy can simulate and subsume a structural coverage criterion if the set of mutants can be killed only by satisfying the corresponding test coverage obligations. Statement coverage can be simulated by applying the mutation operator sdl (statement deletion) to each statement of a program. To kill a mutant whose only difference from the program under test is the absence of statement *S* requires executing the mutant and the program under test with a test case that executes *S* in the original program. Thus to kill all mutants generated by applying the operator sdl to statements of the program under test, we need a test suite that causes the execution of each statement in the original program. Branch coverage can be simulated by applying the operator cpr (constant for predicate replacement) to all predicates of the program under test with constants *True* and *False*. To kill a mutant that differs from the program under test for a predicate *P* set to the constant value *False*, we need to execute the mutant and the program under test with a test case that causes the execution of the *True* branch of *P*. To kill a mutant that differs from the program under test for a predicate *P* set to the constant value *True*,we need to execute the mutant and the program under test with a test case that causes the execution of the *False* branch of *P*.

7.4 Variations on Mutation Analysis

The mutation analysis process described in the preceding sections, which kills mutants based on the outputs produced by execution of test cases, is known as strong mutation. It can generate a number of mutants quadratic in the size of the program. Each mutant must be compiled and executed with each test case until it is killed. The time and space required for compiling all mutants and for executing all test cases for each mutant may be impractical. The computational effort required for mutation analysis can be reduced by decreasing the number of mutants generated and the number of test cases to be executed. Weak mutation analysis decreases the number of tests to be executed by killing mutants when they produce a different intermediate state, rather than waiting for a difference in the final result or observable program behavior. weak mutation analysis

With weak mutation, a single program can be seeded with many faults. A "metamutant" program is divided into segments containing original and mutated source code, with a mechanism to select which segments to execute. Two copies of the meta-mutant are executed in tandem, one with only original program code selected and the other with a set of live mutants selected. Execution is paused after each segment to compare the program state of the two versions. If the state is equivalent, execution resumes with the next segment of original and mutated code. If the state differs, the mutant is marked as dead, and execution of original and mutated code is restarted with a new selection of live mutants.

Estimating Population Sizes

Counting fish Lake Winnemunchie is inhabited by two kinds of fish, a native trout and an introduced species of chub. The Fish and Wildlife Service wishes to estimate the populations to evaluate their efforts to eradicate the chub without harming the population of native trout.

The population of chub can be estimated statistically as follows. 1000 chub are netted, their dorsal fins are marked by attaching a tag, then they are released back into the lake. Over the next weeks, fishermen are asked to report the number of tagged and untagged chub caught. If 50 tagged chub and 300 untagged chub are caught, we can calculate

$$\frac{1000}{\text{untagged chub population}} = \frac{50}{300}$$

and thus there are about 6000 untagged chub remaining in the lake.

It may be tempting to also ask fishermen to report the number of trout caught and to perform a similar calculation to estimate the ratio between chub and trout. However, this is valid only if trout and chub are equally easy to catch, or if one can adjust the ratio using a known model of trout and chub vulnerability to fishing.

Counting residual faults A similar procedure can be used to estimate the number of faults in a program: Seed a given number S of faults in the program. Test the program with some test suite and count the number of revealed faults. Measure the number of seeded faults detected, D_S, and also the number of natural faults D_N detected. Estimate the total number of faults remaining in the program, assuming the test suite is as effective at finding natural faults as it is at finding seeded faults, using the formula

$$\frac{S}{\text{total natural faults}} = \frac{D_S}{D_N}$$

If we estimate the number of faults remaining in a program by determining the proportion of seeded faults detected, we must be wary of the pitfall of estimating trout population by counting chub. The seeded faults are chub, the real faults are trout, and we must either have good reason for believing the seeded faults are no easier to detect than real remaining faults, or else make adequate allowances for uncertainty. The difference is that we cannot avoid the problem by repeating the process with trout - once a fault has been detected, our knowledge of its presence cannot be erased. We depend, therefore, on a very good fault model, so that the chub are as representative as possible of trout. Of course, if we use special bait for chub, or design test cases to detect particular seeded faults, then statistical estimation of the total population of fish or errors cannot be justified.

Hardware Fault-based Testing

Fault-based testing is widely used for semiconductor and hardware system validation and evaluation both for evaluating the quality of test suites and for evaluating fault tolerance.

Semiconductor testing has conventionally been aimed at detecting random errors in fabrication, rather than design faults. Relatively simple fault models have been developed for testing

semiconductor memory devices, the prototypical faults being "stuck- at-0" and "stuck-at-1" (a gate, cell, or pin that produces the same logical value regardless of inputs). A number of more complex fault models have been developed for particular kinds of semiconductor devices (e.g., failures of simultaneous access in dualport memories). A test vector (analogous to a test suite for software) can be judged by the number of hypothetical faults it can detect, as a fraction of all possible faults under the model.

In evaluation of fault tolerance in hardware, the usual approach is to modify the state or behavior rather than the system under test. Due to a difference in terminology between hardware and software testing, the corruption of state or modification of behavior is called a "fault," and artificially introducing it is called "fault injection." Pin-level fault injection consists of forcing a stuck-at-0, a stuck-at-1, or an intermediate voltage level (a level that is neither a logical 0 nor a logical 1) on a pin of a semiconductor device. Heavy ion radiation is also used to inject random faults in a running system. A third approach, growing in importance as hardware complexity increases, uses software to modify the state of a running system or to simulate faults in a running simulation of hardware logic design. Fault seeding can be used statistically in another way: To estimate the number of faults remaining in a program. Usually we know only the number of faults that have been detected, and not the number that remains. However, again to the extent that the fault model is a valid statistical model of actual fault occurrence, we can estimate that the ratio of actual faults found to those still remaining should be similar to the ratio of seeded faults found to those still remaining.

Once again, the necessary assumptions are troubling, and one would be unwise to place too much confidence in an estimate of remaining faults. Nonetheless, a prediction with known weaknesses is better than a seat-of-the-pants guess, and a set of estimates derived in different ways is probably the best one can hope for. While the focus of this chapter is on fault-based testing of software, related techniques can be applied to whole systems (hardware and software together) to evaluate fault tolerance. Some aspects of fault-based testing of hardware are discussed in the sidebar on page 323.

Exercises

7.1 Consider the C function in Figure 16.4, used to determine whether a misspelled word differs from a dictionary word by at most one character, which may be a deletion, an insertion, or a substitution (e.g., "text" is edit distance 1 from "test" by a substitution, and edit distance 1 from "tests" by deletion of "s").

```
1
2  /* edit1( s1, s2 ) returns TRUE iff s1 can be transformed to s2
3   * by inserting, deleting, or substituting a single character, or
4   * by a no-op (i.e., if they are already equal).
5   */ 6 int edit1( char *s1, char *s2) {
7    if (*s1 == 0) {
8      if (*s2 == 0) return TRUE;
9      /* Try inserting a character in s1 or deleting in s2 */
10      if (*(s2+1)==0) return TRUE;
11      return FALSE;
```

```
12  }
13  if (*s2 == 0) { /* Only match is by deleting last char from s1 */
14    if (*(s1 + 1) == 0) return TRUE;
15    return FALSE;
16  }
17  /* Now we know that neither string is empty */
18  if (*s1 == *s2) {
19    return edit1(s1 +1, s2 +1);
20  }
21
22  /* Mismatch; only dist 1 possibilities are identical strings after
23   * inserting, deleting, or substituting character
24   */
25
26  /* Substitution: We "look past" the mismatched character */
27  if (strcmp(s1+1, s2+1) == 0) return TRUE;
28  /* Deletion: look past character in s1 */
29  if (strcmp(s1+1, s2) == 0) return TRUE;
30  /* Insertion: look past character in s2 */
31  if (strcmp(s1, s2+1) == 0) return TRUE;
32  return FALSE;
33 }
```

Figure 16.4: C function to determine whether one string is within edit distance 1 of another.

Suppose we seed a fault in line 27, replacing s1 +1 by s1 + 0. Is there a test case that will kill this mutant using weak mutation, but not using strong mutation? Display such a test case if there is one, or explain why there is none.

7.2 We have described weak mutation as continuing execution up to the point that a mutant is killed, then restarting execution of the original and mutated program from the beginning. Why doesn't execution just continue after killing a mutant? What would be necessary to make continued execution possible?

7.3 Motivate the need for the competent programmer and the coupling effect hypotheses. Would mutation analysis still make sense if these hypotheses did not hold? Why?

7.4 Generate some invalid, valid-but-not-useful, useful, equivalent and nonequivalent mutants for the program in Figure 16.1(See 7.8.3) using mutant operators from Figure

7.4 Test Execution

Whereas test design, even when supported by tools, requires insight and ingenuity in similar measure to other facets of software design, test execution must be sufficiently automated for frequent reexecution without little human involvement. This chapter describes approaches for creating the run-time support for generating and managing test data, creating scaffolding for test execution, and automatically distinguishing between correct and incorrect test case executions.

From Test Case Specifications to Test Cases

If the test case specifications produced in test design already include concrete input values and expected results, as for example in the category-partition method, then producing a complete test case may be as simple as filling a template with those values. A more general test case specification (e.g., one that calls for "a sorted sequence, length greater than 2, with items in ascending order with no duplicates") may designate many possible concrete test cases, and it may be desirable to generate just one instance or many. There is no clear, sharp line between test case design and test case generation. A rule of thumb is that, while test case design involves judgment and creativity, test case generation should be a mechanical step. Automatic generation of concrete test cases from more abstract test case specifications reduces the impact of small interface changes in the course of development. Corresponding changes to the test suite are still required with each program change, but changes to test case specifications are likely to be smaller and more localized than changes to the concrete test cases.

Scaffolding

During much of development, only a portion of the full system is available for testing. In modern development methodologies, the partially developed system is likely to consist of one or more runnable programs and may even be considered a version or prototype of the final system from very early in construction, so it is possible at least to execute each new portion of the software as it is constructed, but the external interfaces of the evolving system may not be ideal for testing; often additional code must be added. For example, even if the actual subsystem for placing an order with a supplier is available and fully operational, it is probably not desirable to place a thousand supply orders each night as part of an automatic test run. More likely a portion of the order placement software will be "stubbed out" for most test executions. Code developed to facilitate testing is called *scaffolding*, by analogy to the temporary structures erected around a building during construction or maintenance. Scaffolding may include test drivers (substituting for a main or calling program), test harnesses (substituting for parts of the deployment environment), and stubs (substituting for functionality called or used by the software under test), in addition to program instrumentation and support for recording and managing test execution. A common estimate is that half of the code developed in a software project is scaffolding of some kind, but the amount of scaffolding that must be constructed with a software project can vary widely, and depends both on the application domain and the architectural design and build plan, which can reduce cost by exposing appropriate interfaces and providing necessary functionality in a rational order. The purposes of scaffolding are to provide controllability to execute test cases and observability to judge the outcome of test execution. Sometimes scaffolding is required to simply make a module executable, but even in incremental development with immediate integration of each module, scaffolding for controllability and observability may be required because the external interfaces of the system may not provide sufficient control to drive the module under test through test cases, or sufficient observability of the effect. It may be desirable to substitute a separate test "driver" program for the full system, in order to provide more direct control of an interface or to remove dependence on other subsystems.

Generic versus Specific Scaffolding

The simplest form of scaffolding is a driver program that runs a single, specific test case. If, for example, a test case specification calls for executing method calls in a particular sequence, this is easy to accomplish by writing the code to make the method calls in that sequence. Writing hundreds or thousands of such test-specific drivers, on the other hand, may be cumbersome and a disincentive to thorough testing. At the very least one will want to factor out some of the common driver code into reusable modules. Sometimes it is worthwhile to write more generic test drivers that essentially interpret test case specifications.

At least some level of generic scaffolding support can be used across a fairly wide class of applications. Such support typically includes, in addition to a standard interface for executing a set of test cases, basic support for logging test execution and results. Figure 7.1 illustrates use of generic test scaffolding in the JFlex lexical analyzer generator.

```
  1  public final class IntCharSet {
 75  ...
 76    public void add(Interval intervall) {
186  ...
187    }
```

```
  1  package JFlex.tests;
  2
  3  import JFlex.IntCharSet;
  4  import JFlex.Interval;
  5  import junit.framework.TestCase;
 11  ...
 12  public class CharClassesTest extends TestCase {
 25  ...
 26    public void testAdd1() {
 27      IntCharSet set = new IntCharSet(new Interval('a','h'));
 28      set.add(new Interval('o','z'));
 29      set.add(new Interval('A','Z'));
 30      set.add(new Interval('h','o'));
 31      assertEquals("{ ['A'-'Z']['a'-'z'] }", set.toString());
 32    }
 33
 34    public void testAdd2() {
 35      IntCharSet set = new IntCharSet(new Interval('a','h'));
 36      set.add(new Interval('o','z'));
 37      set.add(new Interval('A','Z'));
 38      set.add(new Interval('i','n'));
 39      assertEquals("{ ['A'-'Z']['a'-'z'] }", set.toString());
 40    }
 99  ...
100  }
```

Figure 7.1: Excerpt of JFlex 1.4.1 source code (a widely used open-source scanner generator) and accompanying JUnit test cases. JUnit is typical of basic test scaffolding libraries, providing support for test execution, logging, and simple result checking (*assertEquals* in the example). The illustrated version of JUnit uses Java reflection to find and execute test case methods; later versions of JUnit use Java annotation (metadata) facilities, and other tools use source code preprocessors or generators.

Test Oracles

It is little use to execute a test suite automatically if execution results must be manually inspected to apply a pass/fail criterion. Relying on human intervention to judge test outcomes is not merely expensive, but also unreliable. Even the most conscientious and hard-working person cannot maintain the level of attention required to identify one failure in a hundred program executions, little more one or ten thousand. That is a job for a computer. Software that applies a pass/fail criterion to a program execution is called a *test oracle*, often shortened to *oracle*. In addition to rapidly classifying a large number of test case executions, automated test oracles make it possible to classify behaviors that exceed human capacity in other ways, such as checking real-time response against latency requirements or dealing with voluminous output data in a machine-readable rather than human-readable form.

Capture-replay testing, a special case of this in which the predicted output or behavior is preserved from an earlier execution, is discussed in this chapter. A related approach is to capture the output of a trusted alternate version of the program under test. For example, one may produce output from a trusted implementation that is for some reason unsuited for production use; it may too slow or may depend on a component that is not available in the production environment. It is not even necessary that the alternative implementation be *more* reliable than the program under test, as long as it is sufficiently different that the failures of the real and alternate version are likely to be independent, and both are sufficiently reliable that not too much time is wasted determining which one has failed a particular test case on which they disagree.

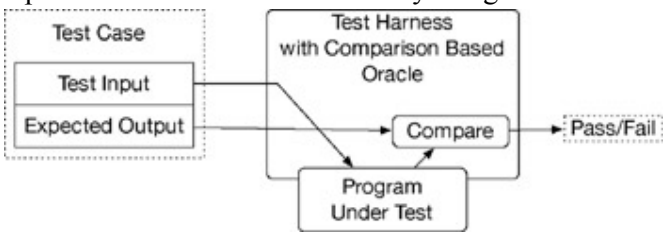


Figure 7.2: A test harness with a comparison-based test oracle processes test cases consisting of (program input, predicted output) pairs.

A third approach to producing complex (input, output) pairs is sometimes possible: It may be easier to produce program input corresponding to a given output than vice versa. For example, it is simpler to scramble a sorted array than to sort a scrambled array. A common misperception is that a test oracle always requires predicted program output to compare to the output produced in a test execution. In fact, it is often possible to judge output or behavior without predicting it. For example, if a program is required to find a bus route from station *A* to station *B*, a test oracle need not

independently compute the route to ascertain that it is in fact a valid route that starts at *A* and ends at *B*. Oracles that check results without reference to a predicted output are often partial, in the sense that they can detect some violations of the actual specification but not others. They check necessary but not sufficient conditions for correctness. For example, if the specification calls for finding the optimum bus route according to some metric, partial oracle a validity check is only a partial oracle because it does not check optimality. Similarly, checking that a sort routine produces sorted output is simple and cheap, but it is only a partial oracle because the output is also required to be a permutation of the input. A cheap partial oracle that can be used for a large number of test cases is often combined with a more expensive comparison-based oracle that can be used with a smaller set of test cases for which predicted output has been obtained.

Ideally, a single expression of a specification would serve both as a work assignment and as a source from which useful test oracles were automatically derived. Specifications are often incomplete, and their informality typically makes automatic derivation of test oracles impossible. The idea is nonetheless a powerful one, and wherever formal or semiformal specifications (including design models) are available, it is worth- while to consider whether test oracles can be derived from them.

Self-Checks as Oracles

A program or module specification describes *all* correct program behaviors, so an oracle based on a specification need not be paired with a particular test case.

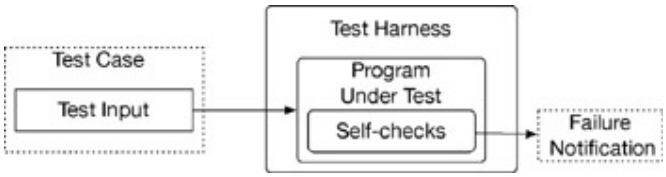


Figure 7.3: When self-checks are embedded in the program, test cases need not include predicted outputs.

Self-check assertions may be left in the production version of a system, where they provide much better diagnostic information than the uncontrolled application crash the customer may otherwise report. If this is not acceptable - for instance, if the cost of a runtime assertion check is too high - most tools for assertion processing also provide controls for activating and deactivating assertions. It is generally considered good design practice to make assertions and self-checks be free of side-effects on program state. Side-effect free assertions are essential when assertions may be deactivated, because otherwise suppressing assertion checking can introduce program failures that appear only when one is *not* testing. Self-checks in the form of assertions embedded in program code are useful primarily for checking module and subsystem-level specifications, rather than overall program behavior. Devising program assertions that correspond in a natural way to specifications (formal or informal) poses two main challenges: bridging the gap between concrete execution values and abstractions used in specification, and dealing in a reasonable way with quantification over collections of values.

$$(|\langle k, v\rangle \in \phi(dict)|)$$
$$o = dict.get(k)$$
$$(|o = v|)$$

φ is an abstraction function that constructs the abstract model type (sets of key, value pairs) from the concrete data structure. φ is a logical association that need not be implemented when reasoning about program correctness. To create a test oracle, it is useful to have an actual implementation of φ. For this example, we might implement a special observer method that creates a simple textual representation of the set of (key, value) pairs. Assertions used as test oracles can then correspond directly to the specification. Besides simplifying implementation of oracles by implementing this mapping once and using it in several assertions, structuring test oracles to mirror a correctness argument is rewarded when a later change to the program invalidates some part of that argument (e.g., by changing the treatment of duplicates or using a different data structure in the implementation).

In addition to an abstraction function, reasoning about the correctness of internal structures usually involves structural invariants, that is, properties of the data structure that are preserved by all operations. Structural invariants are good candidates for self checks implemented as assertions. They pertain directly to the concrete data structure implementation, and can be implemented within the module that encapsulates that data structure. For example, if a dictionary structure is implemented as a red-black tree or an AVL tree, the balance property is an invariant of the structure that can be checked by an assertion within the module. Figure 7.4 illustrates an invariant check found in the source code of the Eclipse programming invariant.

```
 1  package org.eclipse.jdt.internal.ui.text;
 2  import java.text.CharacterIterator;
 3  import org.eclipse.jface.text.Assert;
 4  /**
 5  *A <code>CharSequence</code> based implementation of
 6  * <code>CharacterIterator</code>.
 7  * @since 3.0
 8  */
 9  public class SequenceCharacterIterator implements CharacterIterator {
13  ...
14      private void invariant() {
15          Assert.isTrue(fIndex >= fFirst);
16          Assert.isTrue(fIndex <= fLast);
17      }
49  ...
50      public SequenceCharacterIterator(CharSequence sequence, int first, int last)
51        throws IllegalArgumentException {
52          if (sequence == null)
53              throw new NullPointerException();
```

```
 54              if (first < 0 || first > last)
 55                  throw new IllegalArgumentException();
 56              if (last > sequence.length())
 57                  throw new IllegalArgumentException();
 58              fSequence= sequence;
 59              fFirst= first;
 60              fLast= last;
 61              fIndex= first;
 62              invariant();
 63          }
143  ...
144          public char setIndex(int position) {
145              if (position >= getBeginIndex() && position <= getEndIndex())
146                  fIndex= position;
147              else
148                  throw new IllegalArgumentException();
149
150              invariant();
151              return current();
152          }
263  ...
264  }
```

Figure 7.4: A structural invariant checked by run-time assertions. Excerpted from the Eclipse programming environment, version 3. © 2000, 2005 IBM Corporation; used under terms of the Eclipse Public License v1.0.

There is a natural tension between expressiveness that makes it easier to write and understand specifications, and limits on expressiveness to obtain efficient implementations. It is not much of a stretch to say that programming languages are just formal specification languages in which expressiveness has been purposely limited to ensure that specifications can be executed with predictable and satisfactory performance. An important way in which specifications used for human communication and reasoning about programs are more expressive and less constrained than programming languages is that they freely quantify over collections of values. For example, a specification of database consistency might state that account identifiers are unique; that is, *for all* account records in the database, there *does not exist* another account record with the same identifier.

The problem of quantification over large sets of values is a variation on the basic problem of program testing, which is that we cannot exhaustively check all program behaviors. Instead, we select a tiny fraction of possible program behaviors or inputs as representatives. The same tactic is applicable to quantification in specifications. If we cannot fully evaluate the specified property, we can at least select some elements to check (though at present we know of no program assertion packages that support sampling of quantifiers). For example, although we cannot afford to enumerate all possible paths between two points in a large map, we may be able to compare to a sample of other paths found by the same procedure. As with test design, good samples require some

insight into the problem, such as recognizing that if the shortest path from *A* to *C* passes through *B*, it should be the concatenation of the shortest path from *A* to *B* and the shortest path from *B* to *C*.

A final implementation problem for self-checks is that asserted properties sometimes involve values that are either not kept in the program at all (so-called ghost variables) or values that have been replaced ("before" values). A specification of noninterference between threads in a concurrent program may use ghost variables to track entry and exit of threads from a critical section. The postcondition of an in-place sort operation will state that the new value is sorted and a permutation of the input value. This permutation relation refers to both the "before" and "after" values of the object to be sorted. A run-time assertion system must manage ghost variables and retained "before" values and must ensure that they have no side-effects outside assertion checking.

It may seem unreasonable for a program specification to quantify over an infinite collection, but in fact it can arise quite naturally when quantifiers are combined with negation. If we say "there is no integer greater than 1 that divides *k* evenly," we have combined negation with "there exists" to form a statement logically equivalent to universal ("for all") quantification over the integers. We may be clever enough to realize that it suffices to check integers between 2 and $\sqrt{k}$, but that is no longer a direct translation of the specification statement.

Capture and Replay

Sometimes it is difficult to either devise a precise description of expected behavior or adequately characterize correct behavior for effective self-checks. For example, while many properties of a program with a graphical interface may be specified in a manner suitable for comparison-based or self-check oracles, some properties are likely to require a person to interact with the program and judge its behavior. If one cannot completely avoid human involvement in test case execution, one can at least avoid unnecessary repetition of this cost and opportunity for error. The principle is simple. The first time such a test case is executed, the oracle function is carried out by a human, and the interaction sequence is captured. Provided the execution was judged (by the human tester) to be correct, the captured log now forms an (input, predicted output) pair for subsequent automated retesting. The savings from automated retesting with a captured log depends on how many build-and-test cycles we can continue to use it in, before it is invalidated by some change to the program. Distinguishing between significant and insignificant variations from predicted behavior, in order to prolong the effective lifetime of a captured log, is a major challenge for capture/replay testing. Capturing events at a more abstract level suppresses insignificant changes. For example, if we log only the actual pixels of windows and menus, then changing even a typeface or background color can invalidate an entire suite of execution logs.

Mapping from concrete state to an abstract model of interaction sequences is sometimes possible but is generally quite limited. A more fruitful approach is capturing input and output behavior at multiple levels of abstraction within the implementation. We have noted the usefulness of a layer in which abstract input events (e.g., selection of an object) are captured in place of concrete events (left mouse button depressed with mouse positioned at 235, 718). Typically, there is a similar abstract layer in graphical output, and much of the capture/replay testing can work at this level. Small changes to a program can still invalidate a large number of execution logs, but it is much

more likely that an insignificant detail can either be ignored in comparisons or, even better, the abstract input and output can be systematically transformed to reflect the intended change.

Exercises

7.1 Voluminous output can be a barrier to naive implementations of comparison- based oracles. For example, sometimes we wish to show that some abstraction of program behavior is preserved by a software change. The naive approach is to store a detailed execution log of the original version as predicted output, and compare that to a detailed execution log of the modified version. Unfortunately, a detailed log of a single execution is quite lengthy, and maintaining detailed logs of many test case executions may be impractical. Suggest more efficient approaches to implementing comparison-based test oracles when it is not possible to store the whole output.

7.2 We have described as an ideal but usually unachievable goal that test oracles could be derived automatically from the same specification statement used to record and communicate the intended behavior of a program or module. To what extent does the "test first" approach of extreme programming (XP) achieve this goal? Discuss advantages and limitations of using test cases as a specification statement.

7.3 Often we can choose between on-line self-checks (recognizing failures as they occur) and producing a log of events or states for off-line checking. What considerations might motivate one choice or the other?

UNIT 8

PLANNING AND MONITORING THE PROCESS, DOCUMENTING ANALYSIS AND TEST

Any complex process requires planning and monitoring. The quality process requires coordination of many different activities over a period that spans a full development cycle and beyond. Planning is necessary to order, provision, and coordinate all the activities that support a quality goal, and monitoring of actual status against a plan is required to steer and adjust the process.

Overview

Planning involves scheduling activities, allocating resources, and devising observable, unambiguous milestones against which progress and performance can be monitored. Monitoring means answering the question, "How are we doing?" Quality planning is one aspect of project planning, and quality processes must be closely coordinated with other development processes. Coordination among quality and development tasks may constrain ordering (e.g., unit tests are executed after creation of program units). It may shape tasks to facilitate coordination; for example, delivery may be broken into smaller increments to allow early testing. Some aspects of the project plan, such as feedback and design for testability, may belong equally to the quality plan and other aspects of the project plan.

Quality planning begins at the inception of a project and is developed with the overall project plan, instantiating and building on a quality strategy that spans several projects. Like the overall project plan, the quality plan is developed incrementally, beginning with the feasibility study and continuing through development and delivery. Formulation of the plan involves risk analysis and contingency planning. Execution of the plan involves monitoring, corrective action, and planning for subsequent releases and projects. Allocating responsibility among team members is a crucial and difficult part of planning. When one person plays multiple roles, explicitly identifying each responsibility is still essential for ensuring that none are neglected.

Quality and Process

A software plan involves many intertwined concerns, from schedule to cost to usability and dependability. Despite the intertwining, it is useful to distinguish individual concerns and objectives to lessen the likelihood that they will be neglected, to allocate responsibilities, and to make the overall planning process more manageable.

An appropriate quality process follows a form similar to the overall software process in which it is embedded. In a strict (and unrealistic) waterfall software process, one would follow the "V model" (Figure 8.1 in a sequential manner, beginning unit testing only as implementation commenced following completion of the detailed design phase, and finishing unit testing before integration testing commenced. In the XP "test first" method, unit testing is conflated with subsystem and system testing. A cycle of test design and test execution is wrapped around each small-grain incremental development step. The role that inspection and peer reviews would play in other

processes is filled in XP largely by pair programming. A typical spiral process model lies somewhere between, with distinct planning, design, and implementation steps in several increments coupled with a similar unfolding of analysis and test activities. A general principle, across all software processes, is that the cost of detecting and repairing a fault increases as a function of time between committing an error and detecting the resultant faults. Thus, whatever the intermediate work products in a software plan, an efficient quality plan will include a matched set of intermediate validation and verification activities that detect most faults within a short period of their introduction. Any step in a software process that is not paired with a validation or verification step is an opportunity for defects to fester, and any milestone in a project plan that does not include a quality check is an opportunity for a misleading assessment of progress.

The particular verification or validation step at each stage depends on the nature of the intermediate work product and on the anticipated defects. For example, anticipated defects in a requirements statement might include incompleteness, ambiguity, inconsistency, and overambition relative to project goals and resources. A review step might address some of these, and automated analyses might help with completeness and consistency checking.

The evolving collection of work products can be viewed as a set of descriptions of different parts and aspects of the software system, at different levels of detail. Portions of the implementation have the useful property of being executable in a conventional sense, and are the traditional subject of testing, but every level of specification and design can be both the subject of verification activities and a source of information for verifying other artifacts. A typical intermediate artifact - say, a subsystem interface definition or a database schema - will be subject to the following steps:

Internal consistency check Check the artifact for compliance with structuring rules that define "well-formed" artifacts of that type. An important point of leverage is defining the syntactic and semantic rules thoroughly and precisely enough that many common errors result in detectable violations. This is analogous to syntax and strong-typing rules in programming languages, which are not enough to guarantee program correctness but effectively guard against many simple errors.

External consistency check Check the artifact for consistency with related artifacts. Often this means checking for conformance to a "prior" or "higher-level" specification, but consistency checking does not depend on sequential, top-down development - all that is required is that the related information from two or more artifacts be defined precisely enough to support detection of discrepancies. Consistency usually proceeds from broad, syntactic checks to more detailed and expensive semantic checks, and a variety of automated and manual verification techniques may be applied.

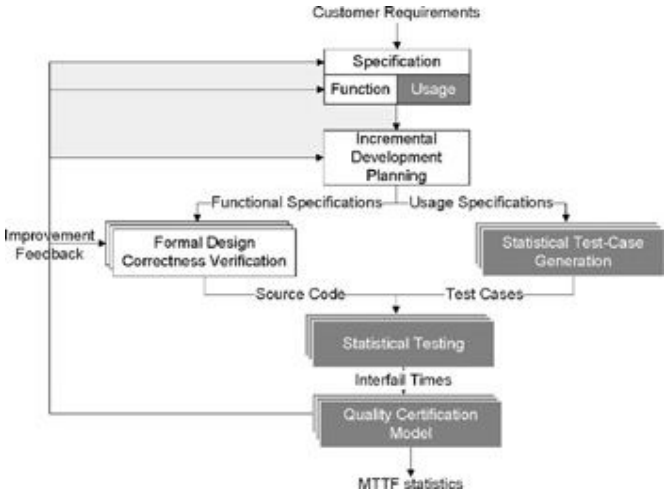
Generation of correctness conjectures Correctness conjectures, which can be test outcomes or other objective criteria, lay the groundwork for external consistency checks of other work products, particularly those that are yet to be developed or revised. Generating correctness conjectures for other work products will frequently motivate refinement of the current product. For example, an interface definition may be elaborated and made more precise so that implementations can be effectively tested.

Test and Analysis Strategies

Lessons of past experience are an important asset of organizations that rely heavily on technical skills. A body of explicit knowledge, shared and refined by the group, is more valuable than islands of individual competence. Organizational knowledge in a shared and systematic form is more amenable to improvement and less vulnerable to organizational change, including the loss of key individuals. Capturing the lessons of experience in a consistent and repeatable form is essential for avoiding errors, maintaining consistency of the process, and increasing development efficiency.

Cleanroom

The Cleanroom process model, introduced by IBM in the late 1980s, pairs development with V&V activities and stresses analysis over testing in the early phases. Testing is left for system certification. The Cleanroom process involves two cooperating teams, the development and the quality teams, and five major activities: specification, planning, design and verification, quality certification, and feedback.



In the *specification* activity, the development team defines the required behavior of the system, while the quality team defines usage scenarios that are later used for deriving system test suites. The *planning* activity identifies incremental development and certification phases.

After planning, all activities are iterated to produce incremental releases of the system. Each system increment is fully deployed and certified before the following step. Design and code undergo formal inspection ("*Correctness verification*") before release. One of the key premises underpinning the Cleanroom process model is that rigorous design and formal inspection produce "nearly fault-free software."

The quality strategy is an intellectual asset of an individual organization prescribing a set of solutions to problems specific to that organization. Among the factors that particularize the strategy are:

Structure and size Large organizations typically have sharper distinctions between development and quality groups, even if testing personnel are assigned to development teams. In smaller organizations, it is more common for a single person to serve multiple roles. Where responsibility is distributed among more individuals, the quality strategy will require more elaborate attention to

coordination and communication, and in general there will be much greater reliance on documents to carry the collective memory.

In a smaller organization, or an organization that has devolved responsibility to small, semi-autonomous teams, there is typically less emphasis on formal communication and documents but a greater emphasis on managing and balancing the multiple roles played by each team member.

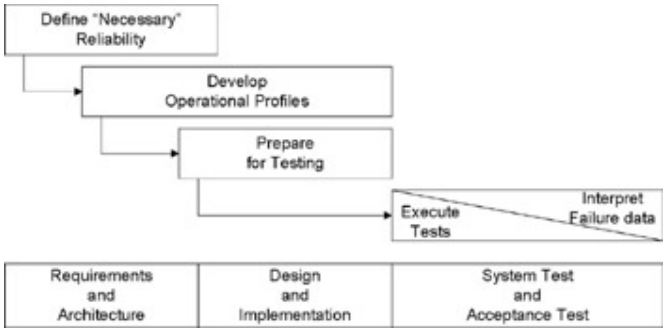
Overall process We have already noted the intertwining of quality process with other aspects of an overall software process, and this is of course reflected in the quality strategy. For example, if an organization follows the Cleanroom methodology, then inspections will be required but unit testing forbidden. An organization that adopts the XP methodology is likely to follow the "test first" and pair programming elements of that approach, and in fact would find a more document-heavy approach a difficult fit.

Notations, standard process steps, and even tools can be reflected in the quality strategy to the extent they are consistent from project to project. For example, if an organization consistently uses a particular combination of UML diagram notations to document subsystem interfaces, then the quality strategy might include derivation of test designs from those notations, as well as review and analysis steps tailored to detect the most common and important design flaws at that point. If a particular version and configuration control system is woven into process management, the quality strategy will likewise exploit it to support and enforce quality process steps.

Application domain The domain may impose both particular quality objectives (e.g., privacy and security in medical records processing), and in some cases particular steps and documentation required to obtain certification from an external authority. For example, the RTCA/DO-178B standard for avionics software requires testing to the modified condition/decision coverage (MC/DC) criterion.

SRET

The software reliability engineered testing (SRET) approach, developed at AT&T in the early 1990s, assumes a spiral development process and augments each coil of the spiral with rigorous testing activities. SRET identifies two main types of testing: *development testing*, used to find and remove faults in software at least partially developed in-house, and *certification testing*, used to either accept or reject outsourced software. The SRET approach includes seven main steps. Two initial, quick decision-making steps determine which systems require separate testing and which type of testing is needed for each system to be tested. The five core steps are executed in parallel with each coil of a spiral development process.



The five core steps of SRET are:

Define "Necessary" Reliability Determine operational models, that is, distinct patterns of system usage that require separate testing, classify failures according to their severity, and engineer the reliability strategy with fault prevention, fault removal, and fault tolerance activities.

Develop Operational Profiles Develop both overall profiles that span operational models and operational profiles within single operational models.

Prepare for Testing Specify test cases and procedures.

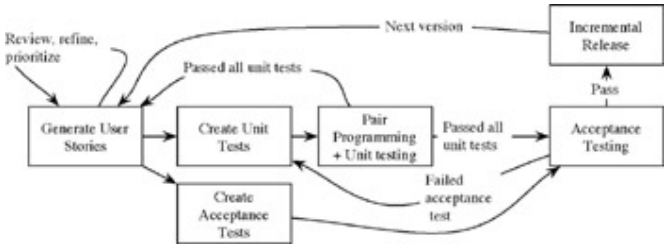
Execute Tests

Interpret Failure Data Interpretation of failure data depends on the type of testing. In development testing, the goal is to track progress and compare present failure intensities with objectives. In certification testing, the goal is to determine if a software component or system should be accepted or rejected.

Extreme Programming (XP)

The extreme programming methodology (XP) emphasizes simplicity over generality, global vision and communication over structured organization, frequent changes over big releases, continuous testing and analysis over separation of roles and responsibilities, and continuous feedback over traditional planning.

Customer involvement in an XP project includes requirements analysis (development, refinement, and prioritization of *user stories*) and acceptance testing of very frequent iterative releases. Planning is based on prioritization of user stories, which are implemented in short iterations. Test cases corresponding to scenarios in user stories serve as partial specifications.



Test cases suitable for batch execution are part of the system code base and are implemented prior to the implementation of features they check ("test-first"). Developers work in pairs, incrementally developing and testing a module. Pair programming effectively conflates a review activity with coding. Each release is checked by running all the tests devised up to that point of development, thus essentially merging unit testing with integration and system testing. A failed acceptance test is viewed as an indication that additional unit tests are needed.

Although there are no standard templates for analysis and test strategies, we can identify a few elements that should be part of almost any good strategy. A strategy should specify common quality requirements that apply to all or most products, promoting conventions for unambiguously stating and measuring them, and reducing the likelihood that they will be overlooked in the quality plan for a particular project. A strategy should indicate a set of documents that is normally produced during

the quality process, and their contents and relationships. It should indicate the activities that are prescribed by the overall process organization. Often a set of standard tools and practices will be prescribed, such as the interplay of a version and configuration control tool with review and testing procedures. In addition, a strategy includes guidelines for project staffing and assignment of roles and responsibilities. An excerpt of a sample strategy document is presented in Chapter 24(See 8.5).

Test and Analysis Plans

An analysis and test plan details the steps to be taken in a particular project. A plan should answer the following questions:

- What quality activities will be carried out?
- What are the dependencies among the quality activities and between quality and development activities?
- What resources are needed and how will they be allocated?
- How will both the process and the evolving product be monitored to maintain an adequate assessment of quality and early warning of quality and schedule problems?

Each of these issues is addressed to some extent in the quality strategy, but must be elaborated and particularized. This is typically the responsibility of a quality manager, who should participate in the initial feasibility study to identify quality goals and estimate the contribution of test and analysis tasks on project cost and schedule.

To produce a quality plan that adequately addresses the questions above, the quality manager must identify the items and features to be verified, the resources and activities that are required, the approaches that should be followed, and criteria for evaluating the results. Items and features to be verified circumscribe the target of the quality plan. While there is an obvious correspondence between items to be developed or modified and those to undergo testing, they may differ somewhat in detail. For example, overall evaluation of the user interface may be the purview of a separate human factors group. The items to be verified, moreover, include many intermediate artifacts such as requirements specifications and design documents, in addition to portions of the delivered system. Approaches to be taken in verification and validation may vary among items. For example, the plan may prescribe inspection and testing for all items and additional static analyses for multi-threaded subsystems. Quality goals must be expressed in terms of properties satisfied by the product and must be further elaborated with metrics that can be monitored during the course of the project. For example, if known failure scenarios are classified as critical, severe, moderate, and minor, then we might decide in advance that a product version may enter end-user acceptance testing only when it has undergone system testing with no outstanding critical or severe failures.

Defining quality objectives and process organization in detail requires information that is not all available in the early stages of development. Test items depend on design decisions; detailed approaches to evaluation can be defined only after examining requirements and design specifications; tasks and schedule can be completed only after the design; new risks and contingencies may be introduced by decisions taken during development. On the other hand, an early plan is necessary for estimating and controlling cost and schedule. The quality manager must

start with an initial plan based on incomplete and tentative information, and incrementally refine the plan as more and better information becomes available during the project.

The primary tactic available for reducing the schedule risk of a critical dependence is to decompose a task on the critical path, factoring out subtasks that can be performed earlier. For example, an acceptance test phase late in a project is likely to have a critical dependence on development and system integration. One cannot entirely remove this dependence, but its potential to delay project completion is reduced by factoring test design from test execution.

Figure 8.1 shows alternative schedules for a simple project that starts at the beginning of January and must be completed by the end of May. In the top schedule, indicated as *CRITICAL SCHEDULE*, the tasks *Analysis and design*, *Code and Integration*, *Design and execute subsystem tests*, and *Design and execute system tests* form a critical path that spans the duration of the entire project. A delay in any of the activities will result in late delivery. In this schedule, only the *Produce user documentation* task does not belong to the critical path, and thus only delays of this task can be tolerated.

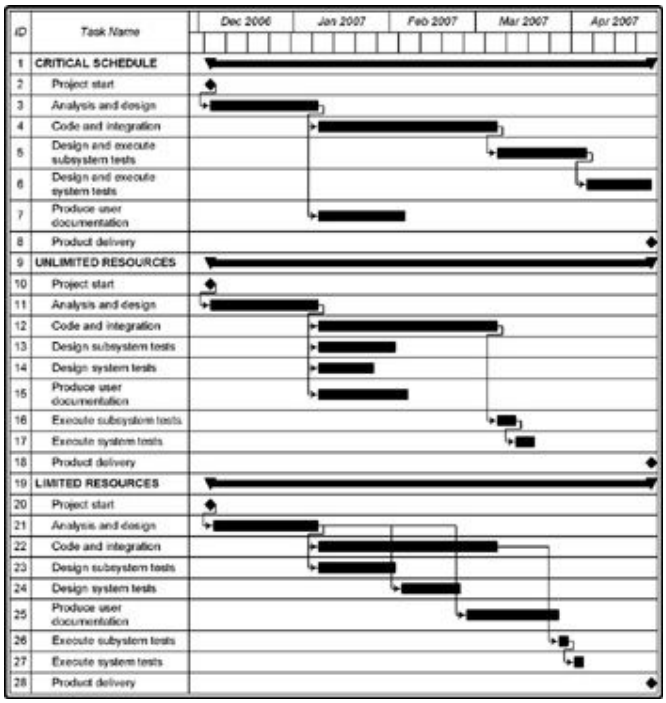


Figure 8.1: Three possible simple schedules with different risks and resource allocation. The bars indicate the duration of the tasks. Diamonds indicate milestones, and arrows between bars indicate precedence between tasks.

In the middle schedule, marked as *UNLIMITED RESOURCES*, the test design and execution activities are separated into distinct tasks. Test design tasks are scheduled early, right after *analysis and design*, and only test execution is scheduled after *Code and integration*. In this way the tasks *Design subsystem tests* and *Design system tests* are removed from the critical path, which now spans 16 weeks with a tolerance of 5 weeks with respect to the expected termination of the project. This schedule assumes enough resources for running *Code and integration*, *Production of user documentation*, *Design of subsystem tests*, and *Design of system tests*.

The *LIMITED RESOURCES* schedule at the bottom of Figure 20.1 rearranges tasks to meet resource constraints. In this case we assume that test design and execution, and production of user documentation share the same resources and thus cannot be executed in parallel. We can see that, despite the limited parallelism, decomposing testing activities and scheduling test design earlier results in a critical path of 17 weeks, 4 weeks earlier than the expected termination of the project. Notice that in the example, the critical path is formed by the tasks *Analysis and design*, *Design subsystem tests*, *Design system tests*, *Produce user documentation*, *Execute subsystem tests*, and *Execute system tests*. In fact, the limited availability of resources results in dependencies among *Design subsystem tests*, *Design system tests* and *Produce user documentation* that last longer than the parallel task *Code and integration*.

The completed plan must include frequent milestones for assessing progress. A rule of thumb is that, for projects of a year or more, milestones for assessing progress should occur at least every three months. For shorter projects, a reasonable maximum interval for assessment is one quarter of project duration.

Figure 8.2 shows a possible schedule for the initial analysis and test plan for the business logic of the Chipmunk Web presence in the form of a GANTT diagram. In the initial plan, the manager has allocated time and effort to inspections of all major artifacts, as well as test design as early as practical and ongoing test execution during development. Division of the project into major parts is reflected in the plan, but further elaboration of tasks associated with units and smaller subsystems must await corresponding elaboration of the architectural design. Thus, for example, inspection of the shopping facilities code and the unit test suites is shown as a single aggregate task. Even this initial plan does reflect the usual Chipmunk development strategy of regular "synch and stabilize" periods punctuating development, and the initial quality plan reflects the Chipmunk strategy of assigning responsibility for producing unit test suites to developers, with review by a member of the quality team.

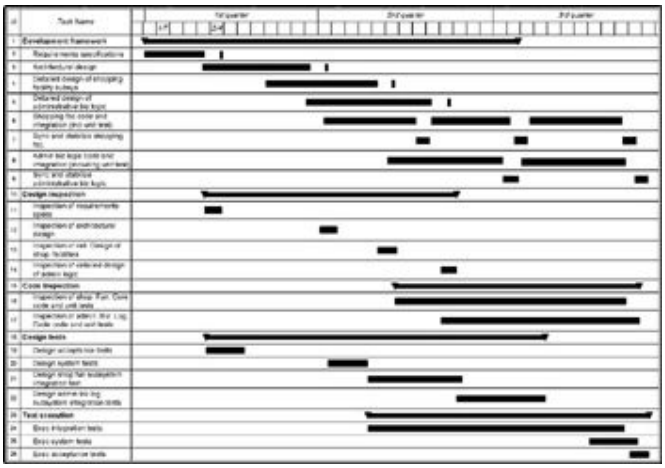


Figure 20.2: Initial schedule for quality activities in development of the business logic subsystem of the Chipmunk Web presence, presented as a GANTT diagram.

The GANTT diagram shows four main groups of analysis and test activities: *design inspection*, *code inspection*, *test design*, and *test execution*. The distribution of activities over time is constrained by resources and dependence among activities. For example, *system test execution* starts

after completion of *system test design* and cannot finish before system integration (the *sync and stablize* elements of *development framework*) is complete. Inspection activities are constrained by specification and design activities. Test design activities are constrained by limited resources. Late scheduling of the design of integration tests for the administrative business logic subsystem is necessary to avoid overlap with design of tests for the shopping functionality subsystem.

The GANTT diagram does not highlight intermediate milestones, but we can easily identify two in April and July, thus dividing the development into three main phases. The first phase (January to April) corresponds to requirements analysis and architectural design activities and terminates with the architectural design baseline. In this phase, the quality team focuses on design inspection and on the design of acceptance and system tests. The second phase (May to July) corresponds to subsystem design and to the implementation of the first complete version of the system. It terminates with the first stabilization of the administrative business logic subsystem. In this phase, the quality team completes the design inspection and the design of test cases. In the final stage, the development team produces the final version, while the quality team focuses on code inspection and test execution.

Absence of test design activities in the last phase results from careful identification of activities that allowed early planning of critical tasks.

Risk Planning

Risk is an inevitable part of every project, and so risk planning must be a part of every plan. Risks cannot be eliminated, but they can be assessed, controlled, and monitored.

The duration of integration, system, and acceptance test execution depends to a large extent on the quality of software under test. Software that is sloppily constructed or that undergoes inadequate analysis and test before commitment to the code base will slow testing progress. Even if responsibility for diagnosing test failures lies with developers and not with the testing group, a test execution session that results in many failures and generates many failure reports is inherently more time consuming than executing a suite of tests with few or no failures. This schedule vulnerability is yet another reason to emphasize earlier activities, in particular those that provide early indications of quality problems. Inspection of design and code (with quality team participation) can help control this risk, and also serves to communicate quality standards and best practices among the team. If unit testing is the responsibility of developers, test suites are part of the unit deliverable and should undergo inspection for correctness, thoroughness, and automation. While functional and structural coverage criteria are no panacea for measuring test thoroughness, it is reasonable to require that deviations from basic coverage criteria be justified on a case-by-case basis. A substantial deviation from the structural coverage observed in similar products may be due to many causes, including inadequate testing, incomplete specifications, unusual design, or implementation decisions. The modules that present unusually low structural coverage should be inspected to identify the cause.

The cost of analysis and test is multiplied when some requirements demand a very high level of assurance. For example, if a system that has previously been used in biological research is modified or redeveloped for clinical use, one should anticipate that all development costs, and particularly

costs of analysis and test, will be an order of magnitude higher. In addition to the risk of underestimating the cost and schedule impact of stringent quality requirements, the risk of failing to achieve the required dependability increases. One important tactic for controlling this risk is isolating critical properties as far as possible in small, simple components. Of course these aspects of system specification and architectural design are not entirely within control of the quality team; it is crucial that at least the quality manager, and possibly other members of the quality team, participate in specification and design activities to assess and communicate the impact of design alternatives on cost and schedule.

Monitoring the Process

The quality manager monitors progress of quality activities, including results as well as schedule, to identify deviations from the quality plan as early as possible and take corrective action. Effective monitoring, naturally, depends on a plan that is realistic, well organized, and sufficiently detailed with clear, unambiguous milestones and criteria. We say a process is *visible* to the extent that it can be effectively monitored.

Successful completion of a planned activity must be distinguished from mere termination, as otherwise it is too tempting to meet an impending deadline by omitting some planned work. Skipping planned verification activities or addressing them superficially can seem to accelerate a late project, but the boost is only apparent; the real effect is to postpone detection of more faults to later stages in development, where their detection and removal will be far more threatening to project success.

Risk Management in the Quality Plan: Risks Generic to Process Management

The quality plan must identify potential risks and define appropriate control tactics. Some risks and control tactics are generic to process management, while others are specific to the quality process. Here we provide a brief overview of some risks generic to process management. Risks specific to the quality process are summarized in the sidebar on page 391.

Personnel Risks	**Example Control Tactics**
A staff member is lost (becomes ill, changes employer, etc.) or is underqualified for task (the project plan assumed a level of skill or familiarity that the assigned member did not have).	Cross train to avoid overdependence on individuals; encourage and schedule continuous education; provide open communication with opportunities for staff self-assessment and identification of skills gaps early in the project; provide competitive compensation and promotion policies and a rewarding work environment to retain staff; include training time in the project schedule.
Technology Risks	**Example Control Tactics**
Many faults are introduced interfacing to an unfamiliar commercial off-the-shelf (COTS) component.	Anticipate and schedule extra time for testing unfamiliar interfaces; invest training time for COTS components and for training with new tools; monitor, document, and publicize common errors and correct idioms; introduce new tools in lower-risk pilot projects or prototyping

	exercises.
Test and analysis automation tools do not meet expectations.	Introduce new tools in lower-risk pilot projects or prototyping exercises; anticipate and schedule time for training with new tools.
COTS components do not meet quality expectations.	Include COTS component qualification testing early in project plan; introduce new COTS components in lower-risk pilot projects or prototyping exercises.
Schedule Risks	**Example Control Tactics**
Inadequate unit testing leads to unanticipated expense and delays in integration testing.	Track and reward quality unit testing as evidenced by low-fault densities in integration.
Difficulty of scheduling meetings makes inspection a bottleneck in development.	Set aside times in a weekly schedule in which inspections take precedence over other meetings and other work; try distributed and asynchronous inspection techniques, with a lower frequency of face-to-face inspection meetings.

Risk Management in the Quality Plan: Risks Specific to Quality Management

Here we provide a brief overview of some risks specific to the quality process. Risks generic to process management are summarized in the sidebar at page 390.

Development Risks	**Example Control Tactics**
Poor quality software delivered to testing group or inadequate unit test and analysis before committing to the code base.	Provide early warning and feedback; schedule inspection of design, code and test suites; connect development and inspection to the reward system; increase training through inspection; require coverage or other criteria at unit test level.
Executions Risks	**Example Control Tactics**
Execution costs higher than planned; scarce resources available for testing (testing requires expensive or complex machines or systems not easily available.)	Minimize parts that require full system to be executed; inspect architecture to assess and improve testability; increase intermediate feedback; invest in scaffolding.
Requirements Risks	**Example Control Tactics**
High assurance critical requirements.	Compare planned testing effort with former projects with similar criticality level to avoid underestimating testing effort; balance test and analysis; isolate critical parts, concerns and properties.

One key aggregate measure is the number of faults that have been revealed and removed, which can be compared to data obtained from similar past projects. Fault detection and removal can be tracked against time and will typically follow a characteristic distribution similar to that shown in Figure 8.3. The number of faults detected per time unit tends to grow across several system builds, then to decrease at a much lower rate (usually half the growth rate) until it stabilizes.

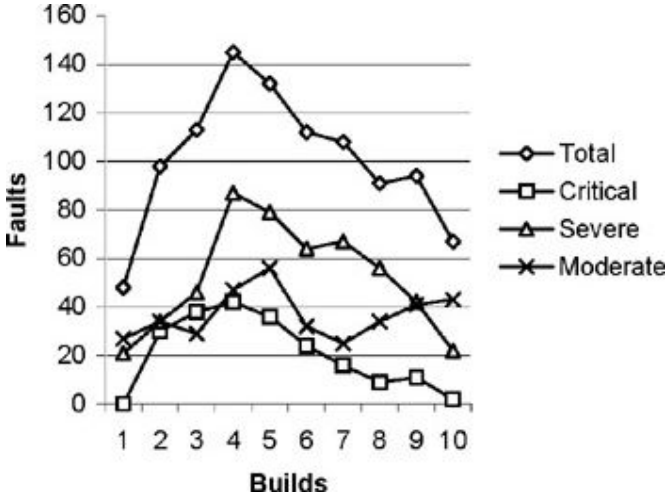


Figure 8.3: A typical distribution of faults for system builds through time.

An unexpected pattern in fault detection may be a symptom of problems. If detected faults stop growing earlier than expected, one might hope it indicates exceptionally high quality, but it would be wise to consider the alternative hypothesis that fault detection efforts are ineffective. A growth rate that remains high through more than half the planned system builds is a warning that quality goals may be met late or not at all, and may indicate weaknesses in fault removal or lack of discipline in development (e.g., a rush to add features before delivery, with a consequent deemphasis on quality control).

A second indicator of problems in the quality process is faults that remain open longer than expected. Quality problems are confirmed when the number of open faults does not stabilize at a level acceptable to stakeholders.

The accuracy with which we can predict fault data and diagnose deviations from expectation depends on the stability of the software development and quality processes, and on availability of data from similar projects. Differences between organizations and across application domains are wide, so by far the most valuable data is from similar projects in one's own organization.

The faultiness data in Figure 8.3 are aggregated by severity levels. This helps in better understanding the process. Growth in the number of *moderate* faults late in the development process may be a symptom of good use of limited resources concentrated in removing *critical* and *severe* faults, not spent solving *moderate* problems. Accurate classification schemata can improve monitoring and may be used in very large projects, where the amount of detailed information cannot be summarized in overall data. The orthogonal defect classification (ODC) approach has two main steps: (1) fault classification and (2) fault analysis.

ODC fault classification is done in two phases: when faults are detected and when they are fixed. At detection time, we record the *activity* executed when the fault is revealed, the *trigger* that exposed the fault, and the perceived or actual *impact* of the fault on the customer. A possible taxonomy for activities and triggers is illustrated in the sidebar at page 395. Notice that triggers depend on the activity. The sidebar at page 396 illustrates a possible taxonomy of customer impacts.

The detailed information on faults allows for many analyses that can provide information on the development and the quality process. As in the case of analysis of simple faultiness data, the interpretation depends on the process and the product, and should be based on past experience. The taxonomy of faults, as well as the analysis of faultiness data, should be refined while applying the method. When we first apply the ODC method, we can perform some preliminary analysis using only part of the collected information:

Distribution of fault types versus activities Different quality activities target different classes of faults. For example, algorithmic (that is, local) faults are targeted primarily by unit testing, and we expect a high proportion of faults detected by unit testing to be in this class. If the proportion of algorithmic faults found during unit testing is unusually small, or a larger than normal proportion of algorithmic faults are found during integration testing, then one may reasonably suspect that unit tests have not been well designed. If the mix of faults found during integration testing contains an unusually high proportion of algorithmic faults, it is also possible that integration testing has not focused strongly enough on interface faults.

Distribution of triggers over time during field test Faults corresponding to simple usage should arise early during field test, while faults corresponding to complex usage should arise late. In both cases, the rate of disclosure of new faults should asymptotically decrease. Unexpected distributions of triggers over time may indicate poor system or acceptance test. If triggers that correspond to simple usage reveal many faults late in acceptance testing, we may have chosen a sample that is not representative of the user population. If faults continue growing during acceptance test, system testing may have failed, and we may decide to resume it before continuing with acceptance testing.

Age distribution over target code Most faults should be located in new and rewritten code, while few faults should be found in base or re-fixed code, since base and re-fixed code has already been tested and corrected. Moreover, the proportion of faults in new and rewritten code with respect to base and re-fixed code should gradually increase. Different patterns may indicate holes in the fault tracking and removal process or may be a symptom of inadequate test and analysis that failed in revealing faults early (in previous tests of base or re-fixed code). For example, an increase of faults located in base code after porting to a new platform may indicate inadequate tests for portability.

Distribution of fault classes over time The proportion of missing code faults should gradually decrease, while the percentage of extraneous faults may slowly increase, because missing functionality should be revealed with use and repaired, while extraneous code or documentation may be produced by updates. An increasing number of missing faults may be a symptom of instability of the product, while a sudden sharp increase in extraneous faults may indicate maintenance problems.

Improving the Process

Many classes of faults that occur frequently are rooted in process and development flaws. For example, a shallow architectural design that does not take into account resource allocation can lead to resource allocation faults. Lack of experience with the development environment, which leads to misunderstandings between analysts and programmers on rare and exceptional cases, can result in faults in exception handling. A performance assessment system that rewards faster coding without regard to quality is likely to promote low quality code.

The occurrence of many such faults can be reduced by modifying the process and environment. For example, resource allocation faults resulting from shallow architectural design can be reduced by introducing specific inspection tasks. Faults attributable to inexperience with the development environment can be reduced with focused training sessions. Persistently poor programming practices may require modification of the reward system.

Often, focused changes in the process can lead to product improvement and significant cost reduction. Unfortunately, identifying the weak aspects of a process can be extremely difficult, and often the results of process analysis surprise even expert managers. The analysis of the fault history can help software engineers build a feedback mechanism to track relevant faults to their root causes, thus providing vital information for improving the process. In some cases, information can be fed back directly into the current product development, but more often it helps software engineers improve the development of future products. For example, if analysis of faults reveals frequent occurrence of severe memory management faults in C programs, we might revise inspection checklists and introduce dynamic analysis tools, but it may be too late to change early design decisions or select a different programming language in the project underway. More fundamental changes may be made in future projects.

Root cause analysis (RCA) is a technique for identifying and eliminating process faults. RCA was first developed in the nuclear power industry and later extended to software analysis. It consists of four main steps to select significant classes of faults and track them back to their original causes: *What*, *When*, *Why*, and *How*.

What are the faults? The goal of this first step is to identify a class of important faults. Faults are categorized by severity and kind. The severity of faults characterizes the impact of the fault on the product. Although different methodologies use slightly different scales and terms, all of them identify a few standard levels, described in Table 8.1.

Table 8.1: Standard severity levels for root cause analysis (RCA).
➪Open table as spreadsheet

Level	Description	Example
Critical	The product is unusable.	The fault causes the program to crash.
Severe	Some product features cannot be used, and there is no workaround.	The fault inhibits importing files saved with a previous version of the program, and there is no way to convert files saved in the old format to the new one.

Table 8.1: Standard severity levels for root cause analysis (RCA).
Open table as spreadsheet

Level	Description	Example
Moderate	Some product features require workarounds to use, and reduce efficiency, reliability, or convenience and usability.	The fault inhibits exporting in Postscript format. Postscript can be produced using the printing facility, but the process is not obvious or documented (loss of usability) and requires extra steps (loss of efficiency).
Cosmetic	Minor inconvenience.	The fault limits the choice of colors for customizing the graphical interface, violating the specification but causing only minor inconvenience.

The RCA approach to categorizing faults, in contrast to ODC, does not use a predefined set of categories. The objective of RCA is not to compare different classes of faults over time, or to analyze and eliminate all possible faults, but rather to identify the few most important classes of faults and remove their causes. Successful application of RCA progressively eliminates the causes of the currently most important faults, which lose importance over time, so applying a static predefined classification would be useless. Moreover, the precision with which we identify faults depends on the specific project and process and varies over time.

ODC Classification of Triggers Listed by Activity

Design Review and Code Inspection

Design Conformance A discrepancy between the reviewed artifact and a prior-stage artifact that serves as its specification.

Logic/Flow An algorithmic or logic flaw.

Backward Compatibility A difference between the current and earlier versions of an artifact that could be perceived by the customer as a failure.

Internal Document An internal inconsistency in the artifact (e.g., inconsistency between code and comments).

Lateral Compatibility An incompatibility between the artifact and some other system or module with which it should interoperate.

Concurrency A fault in interaction of concurrent processes or threads.

Language Dependency A violation of language-specific rules, standards, or best practices.

Side Effects A potential undesired interaction between the reviewed artifact and some other part of the system.

Rare Situation An inappropriate response to a situation that is not anticipated in the artifact. (Error handling as specified in a prior artifact *design conformance*, not *rare situation*.)

Structural (White-Box) Test

Simple Path The fault is detected by a test case derived to cover a single program element.

Complex Path The fault is detected by a test case derived to cover a combination of program elements.

Functional (Black-Box) Test

Coverage The fault is detected by a test case derived for testing a single procedure (e.g., C function or Java method), without considering combination of values for possible parameters.

Variation The fault is detected by a test case derived to exercise a particular combination of parameters for a single procedure.

Sequencing The fault is detected by a test case derived for testing a sequence of procedure calls.

Interaction The fault is detected by a test case derived for testing procedure interactions.

System Test

Workload/Stress The fault is detected during workload or stress testing.

Recovery/Exception The fault is detected while testing exceptions and recovery procedures.

Startup/Restart The fault is detected while testing initialization conditions during start up or after possibly faulty shutdowns.

Hardware Configuration The fault is detected while testing specific hardware configurations.

Software Configuration The fault is detected while testing specific software configurations.

Blocked Test Failure occurred in setting up the test scenario.

ODC Classification of Customer Impact

Installability Ability of the customer to place the software into actual use. (Usability of the installed software is not included.)

Integrity/Security Protection of programs and data from either accidental or malicious destruction or alteration, and from unauthorized disclosure.

Performance The perceived and actual impact of the software on the time required for the customer and customer end users to complete their tasks.

Maintenance The ability to correct, adapt, or enhance the software system quickly and at minimal cost.

Serviceability Timely detection and diagnosis of failures, with minimal customer impact.

Migration Ease of upgrading to a new system release with minimal disruption to existing customer data and operations.

Documentation Degree to which provided documents (in all forms, including electronic) completely and correctly describe the structure and intended uses of the software.

Usability The degree to which the software and accompanying documents can be understood and effectively employed by the end user.

Standards The degree to which the software complies with applicable standards.

Reliability The ability of the software to perform its intended function without unplanned interruption or failure.

Accessibility The degree to which persons with disabilities can obtain the full benefit of the software system.

Capability The degree to which the software performs its intended functions consistently with documented system requirements.

Requirements The degree to which the system, in complying with document requirements, actually meets customer expectations

ODC Classification of Defect Types for Targets *Design* and *Code*

Assignment/Initialization A variable was not assigned the correct initial value or was not assigned any initial value.

Checking Procedure parameters or variables were not properly validated before use.

Algorithm/Method A correctness or efficiency problem that can be fixed by reimplementing a single procedure or local data structure, without a design change.

Function/Class/Object A change to the documented design is required to conform to product requirements or interface specifications.

Timing/Synchronization The implementation omits necessary synchronization of shared resources, or violates the prescribed synchronization protocol.

Interface/Object-Oriented Messages Module interfaces are incompatible; this can include syntactically compatible interfaces that differ in semantic interpretation of communicated data.

Relationship Potentially problematic interactions among procedures, possibly involving different assumptions but not involving interface incompatibility.

A good RCA classification should follow the uneven distribution of faults across categories. If, for example, the current process and the programming style and environment result in many interface faults, we may adopt a finer classification for interface faults and a coarse-grain classification of

other kinds of faults. We may alter the classification scheme in future projects as a result of having identified and removed the causes of many interface faults.

Classification of faults should be sufficiently precise to allow identifying one or two most significant classes of faults considering severity, frequency, and cost of repair. It is important to keep in mind that severity and repair cost are not directly related. We may have cosmetic faults that are very expensive to repair, and critical faults that can be easily repaired. When selecting the target class of faults, we need to consider all the factors. We might, for example, decide to focus on a class of moderately severe faults that occur very frequently and are very expensive to remove, investing fewer resources in preventing a more severe class of faults that occur rarely and are easily repaired.

When did faults occur, and when were they found? It is typical of mature software processes to collect fault data sufficient to determine when each fault was detected (e.g., in integration test or in a design inspection). In addition, for the class of faults identified in the first step, we attempt to determine when those faults were introduced (e.g., was a particular fault introduced in coding, or did it result from an error in architectural design?).

Why did faults occur? In this core RCA step, we attempt to trace representative faults back to causes, with the objective of identifying a "root" cause associated with many faults in the class. Analysis proceeds iteratively by attempting to explain the error that led to the fault, then the cause of that error, the cause of that cause, and so on. The rule of thumb "ask why six times" does not provide a precise stopping rule for the analysis, but suggests that several steps may be needed to find a cause in common among a large fraction of the fault class under consideration.

The *80/20* or *Pareto* Rule

Fault classification in root cause analysis is justified by the so-called *80/20* or *Pareto* rule. The Pareto rule is named for the Italian economist Vilfredo Pareto, who in the early nineteenth century proposed a mathematical power law formula to describe the unequal distribution of wealth in his country, observing that 20% of the people owned 80% of the wealth.

Pareto observed that in many populations, a few (20%) are vital and many (80%) are trivial. In fault analysis, the Pareto rule postulates that 20% of the code is responsible for 80% of the faults. Although proportions may vary, the rule captures two important facts:

1. Faults tend to accumulate in a few modules, so identifying potentially faulty modules can improve the cost effectiveness of fault detection.
2. Some classes of faults predominate, so removing the causes of a predominant class of faults can have a major impact on the quality of the process and of the resulting product.

The predominance of a few classes of faults justifies focusing on one class at a time.

Tracing the causes of faults requires experience, judgment, and knowledge of the development process. We illustrate with a simple example. Imagine that the first RCA step identified *memory leaks* as the most significant class of faults, combining a moderate frequency of occurrence with severe impact and high cost to diagnose and repair. The group carrying out RCA will try to identify the cause of memory leaks and may conclude that many of them result from *forgetting to release memory in exception handlers*. The RCA group may trace this problem in exception handling to

lack of information: *Programmers can't easily determine what needs to be cleaned up in exception handlers*. The RCA group will ask *why* once more and may go back to a design error: *The resource management scheme assumes normal flow of control* and thus does not provide enough information to guide implementation of exception handlers. Finally, the RCA group may identify the root problem in an early design problem: *Exceptional conditions were an afterthought dealt with late in design*.

Each step requires information about the class of faults and about the development process that can be acquired through inspection of the documentation and interviews with developers and testers, but the key to success is curious probing through several levels of cause and effect.

How could faults be prevented? The final step of RCA is improving the process by removing root causes or making early detection likely. The measures taken may have a minor impact on the development process (e.g., adding consideration of exceptional conditions to a design inspection checklist), or may involve a substantial modification of the process (e.g., making explicit consideration of exceptional conditions a part of all requirements analysis and design steps). As in tracing causes, prescribing preventative or detection measures requires judgment, keeping in mind that the goal is not perfection but cost-effective improvement. ODC and RCA are two examples of feedback and improvement, which are an important dimension of most good software processes.

The Quality Team

The quality plan must assign roles and responsibilities to people. As with other aspects of planning, assignment of responsibility occurs at a strategic level and a tactical level. The tactical level, represented directly in the project plan, assigns responsibility to individuals in accordance with the general strategy. It involves balancing level of effort across time and carefully managing personal interactions. The strategic level of organization is represented not only in the quality strategy document, but in the structure of the organization itself.

The strategy for assigning responsibility may be partly driven by external requirements. For example, independent quality teams may be required by certification agencies or by a client organization. Additional objectives include ensuring sufficient accountability that quality tasks are not easily overlooked; encouraging objective judgment of quality and preventing it from being subverted by schedule pressure; fostering shared commitment to quality among all team members; and developing and communicating shared knowledge and values regarding quality.

Each of the possible organizations of quality roles makes some objectives easier to achieve and some more challenging. Conflict of one kind or another is inevitable, and therefore in organizing the team it is important to recognize the conflicts and take measures to control adverse consequences. If an individual plays two roles in potential conflict (e.g., a developer responsible for delivering a unit on schedule is also responsible for integration testing that could reveal faults that delay delivery), there must be countermeasures to control the risks inherent in that conflict. If roles are assigned to different individuals, then the corresponding risk is conflict between the individuals (e.g., if a developer and a tester do not adequately share motivation to deliver a quality product on schedule).

An independent and autonomous testing team lies at one end of the spectrum of possible team organizations. One can make that team organizationally independent so that, for example, a project manager with schedule pressures can neither bypass quality activities or standards, nor reallocate people from testing to development, nor postpone quality activities until too late in the project. Separating quality roles from development roles minimizes the risk of conflict between roles played by an individual, and thus makes most sense for roles in which independence is paramount, such as final system and acceptance testing. An independent team devoted to quality activities also has an advantage in building specific expertise, such as test design. The primary risk arising from separation is in conflict between goals of the independent quality team and the developers.

When quality tasks are distributed among groups or organizations, the plan should include specific checks to ensure successful completion of quality activities. For example, when module testing is performed by developers and integration and system testing is performed by an independent quality team, the quality team should check the completeness of module tests performed by developers, for example, by requiring satisfaction of coverage criteria or inspecting module test suites. If testing is performed by an independent organization under contract, the contract should carefully describe the testing process and its results and documentation, and the client organization should verify satisfactory completion of the contracted tasks.

It may be logistically impossible to maintain an independent quality group, especially in small projects and organizations, where flexibility in assignments is essential for resource management. Aside from the logistical issues, division of responsibility creates additional work in communication and coordination. Finally, quality activities often demand deep knowledge of the project, particularly at detailed levels (e.g., unit and early integration test). An outsider will have less insight into how and what to test, and may be unable to effectively carry out the crucial earlier activities, such as establishing acceptance criteria and reviewing architectural design for testability. For all these reasons, even organizations that rely on an independent verification and validation (IV&V) group for final product qualification allocate other responsibilities to developers and to quality professionals working more closely with the development team.

The more development and quality roles are combined and intermixed, the more important it is to build into the plan checks and balances to be certain that quality activities and objective assessment are not easily tossed aside as deadlines loom. For example, XP practices like "test first" together with pair programming (sidebar on page 381) guard against some of the inherent risks of mixing roles. Separate roles do not necessarily imply segregation of quality activities to distinct individuals. It is possible to assign both development and quality responsibility to developers, but assign two individuals distinct responsibilities for each development work product. Peer review is an example of mixing roles while maintaining independence on an item-by-item basis. It is also possible for developers and testers to participate together in some activities.

Many variations and hybrid models of organization can be designed. Some organizations have obtained a good balance of benefits by rotating responsibilities. For example, a developer may move into a role primarily responsible for quality in one project and move back into a regular development role in the next. In organizations large enough to have a distinct quality or testing group, an appropriate balance between independence and integration typically varies across levels

of project organization. At some levels, an appropriate balance can be struck by giving responsibility for an activity (e.g., unit testing) to developers who know the code best, but with a separate oversight responsibility shared by members of the quality team. For example, unit tests may be designed and implemented by developers, but reviewed by a member of the quality team for effective automation (particularly, suitability for automated regression test execution as the product evolves) as well as thoroughness. The balance tips further toward independence at higher levels of granularity, such as in system and acceptance testing, where at least some tests should be designed independently by members of the quality team.

Outsourcing test and analysis activities is sometimes motivated by the perception that testing is less technically demanding than development and can be carried out by lower-paid and lower-skilled individuals. This confuses test execution, which should in fact be straightforward, with analysis and test design, which are as demanding as design and programming tasks in development. Of course, less skilled individuals *can* design and carry out tests, just as less skilled individuals *can* design and write programs, but in both cases the results are unlikely to be satisfactory.

Outsourcing can be a reasonable approach when its objectives are not merely minimizing cost, but maximizing independence. For example, an independent judgment of quality may be particularly valuable for final system and acceptance testing, and may be essential for measuring a product against an independent quality standard (e.g., qualifying a product for medical or avionic use). Just as an organization with mixed roles requires special attention to avoid the conflicts between roles played by an individual, radical separation of responsibility requires special attention to control conflicts between the quality assessment team and the development team.

The plan must clearly define milestones and delivery for outsourced activities, as well as checks on the quality of delivery in both directions: Test organizations usually perform quick checks to verify the consistency of the software to be tested with respect to some minimal "testability" requirements; clients usually check the completeness and consistency of test results. For example, test organizations may ask for the results of inspections on the delivered artifact before they start testing, and may include some quick tests to verify the installability and testability of the artifact. Clients may check that tests satisfy specified functional and structural coverage criteria, and may inspect the test documentation to check its quality. Although the contract should detail the relation between the development and the testing groups, ultimately, outsourcing relies on mutual trust between organizations.

Documenting Analysis and Test

Mature software processes include documentation standards for all the activities of the software process, including test and analysis activities. Documentation can be inspected to verify progress against schedule and quality goals and to identify problems, supporting process visibility, monitoring, and replicability.

Overview

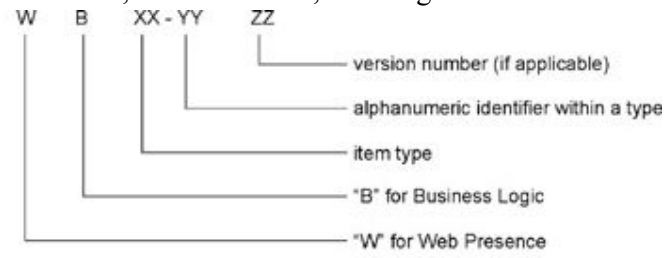
Documentation is an important element of the software development process, including the quality process. Complete and well-structured documents increase the reusability of test suites within and across projects. Documents are essential for maintaining a body of knowledge that can be reused across projects. Consistent documents provide a basis for monitoring and assessing the process, both internally and for external authorities where certification is desired. Finally, documentation includes summarizing and presenting data that forms the basis for process improvement. Test and analysis documentation includes summary documents designed primarily for human comprehension and details accessible to the human reviewer but designed primarily for automated analysis.

Documents are divided into three main categories: planning, specification, and reporting. *Planning documents* describe the organization of the quality process and include strategies and plans for the division or the company, and plans for individual projects. *Specification documents* describe test suites and test cases. A complete set of analysis and test specification documents include test design specifications, test case specification, checklists, and analysis procedure specifications. *Reporting documents* include details and summary of analysis and test results.

Organizing Documents

In a small project with a sufficiently small set of documents, the arrangement of other project artifacts (e.g., requirements and design documents) together with standard content (e.g., mapping of subsystem test suites to the build schedule) provides sufficient organization to navigate through the collection of test and analysis documentation. In larger projects, it is common practice to produce and regularly update a global guide for navigating among individual documents.

Naming conventions help in quickly identifying documents. A typical standard for document names would include keywords indicating the general scope of the document, its nature, the specific document, and its version, as in Figure 8.4.



analysis and test documentation

WB05-YYZZ	analysis and test strategy
WB06-YYZZ	analysis and test plan
WB07-YYZZ	test design specifications
WB08-YYZZ	test case specification
WB09-YYZZ	checklists
WB10-YYZZ	analysis and test logs
WB11-YYZZ	analysis and test summary reports
WB12-YYZZ	other analysis and test documents

Figure 8.4: Sample document naming conventions, compliant with IEEE standards.

Chipmunk Document Template *Document Title*

Approvals

issued by	*name*	*signature*	*date*
approved by	*name*	*signature*	*date*
distribution status	(*internal use only, restricted*, …)		
distribution list	*(people to whom the document must be sent)*		

History

version	description

Table of Contents

- *List of sections.*

Summary

- *Summarize the contents of the document. The summary should clearly explain the relevance of the document to its possible uses.*

Goals of the document

- *Describe the purpose of this document: Who should read it, and why?*

Required documents and references

- *Provide a reference to other documents and artifacts needed for understanding and exploiting this document. Provide a rationale for the provided references.*

Glossary

- *Provide a glossary of terms required to understand this document.*

Section 1

- …

Section N

- …

Test Strategy Document

Analysis and Test Plan

While the format of an analysis and test strategy vary from company to company, the structure of an analysis and test plan is more standardized

The overall quality plan usually comprises several individual plans of limited scope. Each test and analysis plan should indicate the items to be verified through analysis or testing. They may include specifications or documents to be inspected, code to be analyzed or tested, and interface specifications to undergo consistency analysis. They may refer to the whole system or part of it - like a subsystem or a set of units. Where the project plan includes planned development increments, the analysis and test plan indicates the applicable versions of items to be verified.

For each item, the plan should indicate any special hardware or external software required for testing. For example, the plan might indicate that one suite of subsystem tests for a security package can be executed with a software simulation of a smart card reader, while another suite requires access to the physical device. Finally, for each item, the plan should reference related documentation, such as requirements and design specifications, and user, installation, and operations guides.

An Excerpt of the Chipmunk Analysis and Test Strategy

Document CP05-14.03: Analysis and Test Strategy

…

Applicable Standards and Procedures

Artifact	*Applicable Standards and Guidelines*
Web application	Accessibility: W3C-WAI …
Reusable component (internally developed)	Inspection procedure: [WB12-03.12]
External component	Qualification procedure: [WB12-22.04]

Documentation Standards

Project documents must be archived according to the standard Chipmunk archive procedure [WB02-01.02]. Standard required documents include

Document	*Content & Organization Standard*
Quality plan	[WB06-01.03]
Test design specifications	[WB07-01.01] (per test suite)
Test case specifications	[WB08-01.07] (per test suite)
Test logs	[WB10-02.13]
Test summary reports	[WB11-01.11]

Inspection reports	[WB12-09.01]

Analysis and Test Activities

Tools

The following tools are approved and should be used in all development projects. Exceptions require configuration committee approval and must be documented in the project plan.

Fault logging	Chipmunk BgT [WB10-23.01]
…	

Staff and Roles

A development work unit consists of unit source code, including unit test cases, stubs, and harnesses, and unit test documentation. A unit may be committed to the project baseline when the source code, test cases, and test results have passed peer review. A test and analysis plan may not address all aspects of software quality and testing activities. It should indicate the features to be verified and those that are excluded from consideration (usually because responsibility for them is placed elsewhere). For example, if the item to be verified includes a graphical user interface, the test and analysis plan might state that it deals only with functional properties and not with usability, which is to be verified separately by a usability and human interface design team. Explicit indication of features *not* to be tested, as well as those included in an analysis and test plan, is important for assessing completeness of the overall set of analysis and test activities. Assumption that a feature not considered in the current plan is covered at another point is a major cause of missing verification in large projects.

The quality plan must clearly indicate criteria for deciding the success or failure of each planned activity, as well as the conditions for suspending and resuming analysis and test. The core of an analysis and test plan is a detailed schedule of tasks. The schedule is usually illustrated with GANTT and PERT diagrams showing the relation among tasks as well as their relation to other project milestones. The schedule includes the allocation of limited resources (particularly staff) and indicates responsibility for reresources and responsibilities sults.

A quality plan document should also include an explicit risk plan with contingencies. As far as possible, contingencies should include unambiguous triggers (e.g., a date on which a contingency is activated if a particular task has not be completed) as well as recovery procedures. Finally, the test and analysis plan should indicate scaffolding, oracles, and any other software or hardware support required for test and analysis activities.

Test Design Specification Documents

Design documentation for test suites and test cases serve essentially the same purpose as other software design documentation, guiding further development and preparing for maintenance. Test

suite design must include all the information needed for initial selection of test cases and maintenance of the test suite over time, including rationale and anticipated evolution. Specification of individual test cases includes purpose, usage, and anticipated changes.

A Standard Organization of an Analysis and Test Plan

Analysis and test items:

- The items to be tested or analyzed. The description of each item indicates version and installation procedures that may be required.

Features to be tested:

- The features considered in the plan.

Features not to be tested:

- Features not considered in the current plan.

Approach:

- The overall analysis and test approach, sufficiently detailed to permit identification of the major test and analysis tasks and estimation of time and resources.

Pass/Fail criteria:

- Rules that determine the status of an artifact subjected to analysis and test.

Suspension and resumption criteria:

- Conditions to trigger suspension of test and analysis activities (e.g., an excessive failure rate) and conditions for restarting or resuming an activity.

Risks and contingencies:

- Risks foreseen when designing the plan and a contingency plan for each of the identified risks.

Deliverables:

- A list all A&T artifacts and documents that must be produced.

Task and schedule:

- A complete description of analysis and test tasks, relations among them, and relations between A&T and development tasks, with resource allocation and constraints. A task schedule usually includes GANTT and PERT diagrams.

Staff and responsibilities:

- Staff required for performing analysis and test activities, the required skills, and the allocation of responsibilities among groups and individuals. Allocation of resources to tasks is described in the schedule.

Environmental needs:

- Hardware and software required to perform analysis or testing activities.

Test design specification documents describe complete test suites (i.e., sets of test cases that focus on particular aspects, elements, or phases of a software project). They may be divided into unit,

integration, system, and acceptance test suites, if we organize them by the granularity of the tests, or functional, structural, and performance test suites, if the primary organization is based on test objectives. A large project may include many test design specifications for test suites of different kinds and granularity, and for different versions or configurations of the system and its components. Test design specifications identify the features they are intended to verify and the approach used to select test cases. Features to be tested should be cross-referenced to relevant parts of a software specification or design document. A test design specification also includes description of the testing procedure and pass/fail criteria. The procedure indicates steps required to set up the testing environment and perform the tests, and includes references to scaffolding and oracles. Pass/fail criteria distinguish success from failure of a test suite as a whole. In the simplest case a test suite execution may be determined to have failed if any individual test case execution fails, but in system and acceptance testing it is common to set a tolerance level that may depend on the number and severity of failures.

Test and Analysis Reports

Reports of test and analysis results serve both developers and test designers. They identify open faults for developers and aid in scheduling fixes and revisions. They help test designers assess and refine their approach, for example, noting when some class of faults is escaping early test and analysis and showing up only in subsystem and system testing

Functional Test Design Specification of check configuration

Test Suite Identifier

WB07-15.01

Features to Be Tested

Functional test for check configuration, module specification WB02-15.32.[a]

Approach

Combinatorial functional test of feature parameters, enumerated by category- partition method over parameter table on page 3 of this document.[b]

Procedure

Designed for conditional inclusion in nightly test run. Build target T02 15 32 11 includes JUnit harness and oracles, with test reports directed to standard test log. Test environment includes table MDB 15 32 03 for loading initial test database state.

Test cases[c]

WB07-15.01.C01	malformed model number
WB07-15.01.C02	model number not in DB
…	…

WB07-15.01.C09[d]	valid model number with all legal required slots and some legal optional slots
…	…
WB07-15.01.C19	empty model DB
WB07-15.01.C23	model DB with a single element
WB07-15.01.C24	empty component DB
WB07-15.01.C29	component DB with a single element

Pass/Fail Criterion

Successful completion requires correct execution of all test cases with no violations in test log.

Test Case Specification for check configuration

Test Case Identifier

WB07-15.01.C09

Test items

Module check configuration of the Chipmunk Web presence system, business logic subsystem.

Input specification

Test Case Specification:

Model No.	valid
No. of required slots for selected model (#SMRS)	many
No. of optional slots for selected model (#SMOS)	many
Correspondence of selection with model slots	complete
No. of required components with selection ≠ empty	= No. of required slots
No. of optional components with select ≠ empty	< No. of optional slots
Required component selection	all valid
Optional component selection	all valid
No. of models in DB	many
No. of components in	DB many

Test case:

Model number	Chipmunk C20

#SMRS	5
Screen	13"
Processor	Chipmunk II plus
Hard disk	30 GB
RAM	512 MB
OS	RodentOS 3.2 Personal Edition
#SMOS	4
External storage device	DVD player

Output Specification

- return value valid

Environment Needs

Execute with ChipmunkDBM v3.4 database initialized from table MDB 15 32 03.

Special Procedural Requirements

- none

Intercase Dependencies

- none

A prioritized list of open faults is the core of an effective fault handling and repair procedure. Failure reports must be consolidated and categorized so that repair effort can be managed systematically, rather than jumping erratically from problem to problem and wasting time on duplicate reports. They must be prioritized so that effort is not squandered on faults of relatively minor importance while critical faults are neglected or even forgotten.