MODULE-01
BASIC OF SOFTWARE TESTING

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

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 f**a**ilure.

Test

Testing is obviously concerned with errors, faults, failures, and incidents. A test is the act ofexercising 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.

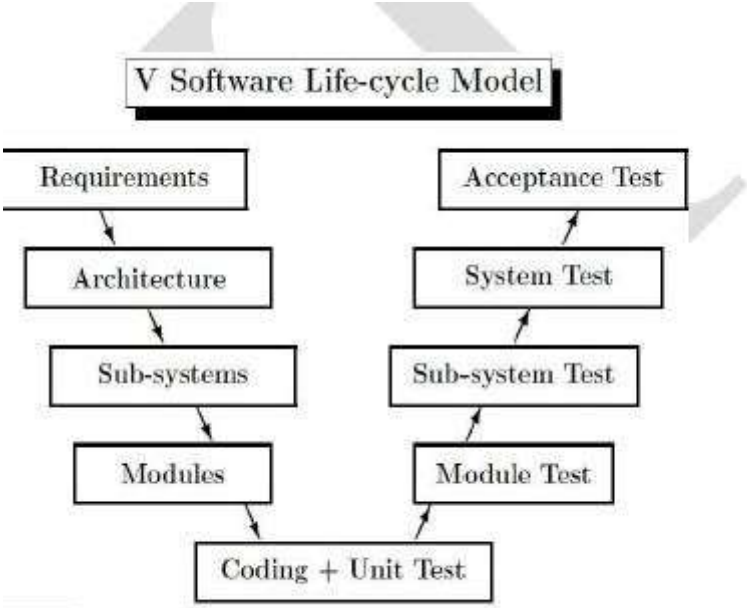

Figure 1.1 A Testing Life Cycle

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"

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.

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.

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.

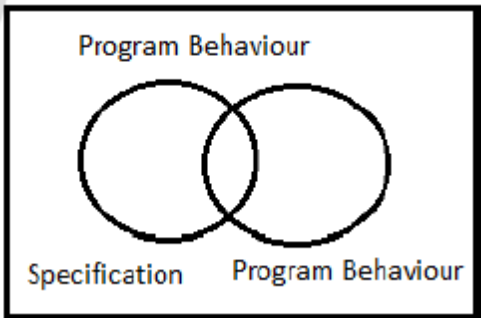

Figure 1.2 Specified and Implemented Program Behaviors

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 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.)

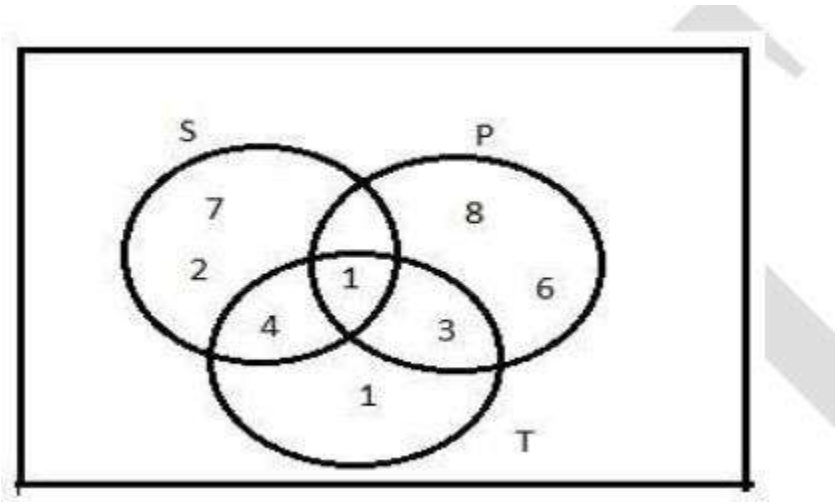

Figure 1.3 Specified, Implemented, and Tested Behaviors

The new circle in Fig. 1.3 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 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.

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.

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 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.

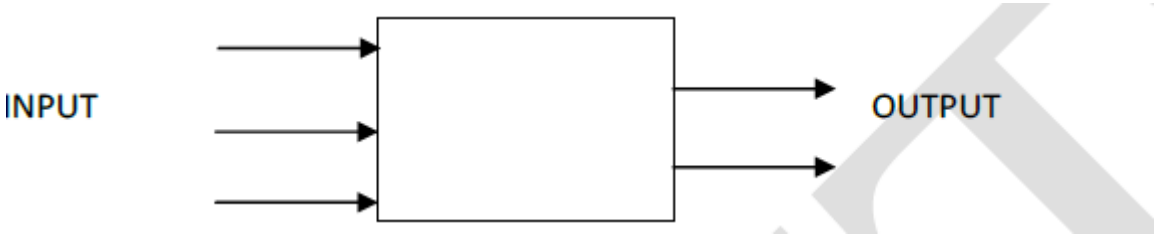

Figure 1.4 An Engineer's Black Box

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.

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 boxallows het 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.

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."

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 . My favorite is based on anomaly occurrence: one time only, intermittent, recurring, or repeatable.

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 testing is most appropriate at the unit level, while functional testing is most appropriate at the system level. While this 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

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.

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 , 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.)

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!).

```
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.
```

The Triangle Problem

The year of this writing marks the twentieth anniversary of publications using the Triangle Problem as an example.

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.

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.)

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].)

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!).

Structured Implementation

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

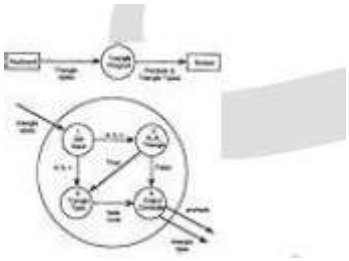

Figure 1.5 Dataflow Diagram for a Structured Triangle Program Implementation

```
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.
```

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 different kind of complexity — logical relationships among theinput variables themselves.

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 $1812 \leq year \leq 2012$.

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.

Implementation

```
PROGRAM NextDate (INPUT, OUTPUT);
TYPE monthType = 1..12;
dayType = 1..31;
yearType = 1812..2012; dateType = record
month : monthType;
day : dayType;
VAR
```

```
year : yearType
end; (*dateType record *) 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
4,6,9,11 :
ELSE Begin
tomorrow.day := 1; tomorrow.month := month + 1
End;
IF day < 30 THEN tomorrow.day := day + 1 ELSE Begin
tomorrow.day := 1; tomorrow.month := month + 1
End;
12:
2:
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;
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}
ELSE IF
tomorrow.day := 1;
tomorrow.month := 3;
End
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*)
```

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.

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.

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-timesystems,practitioners in the commercial EDP domain are finding that even traditional COBOL systems have many of the problems usually associated with real-time systems

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 1.6, 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.

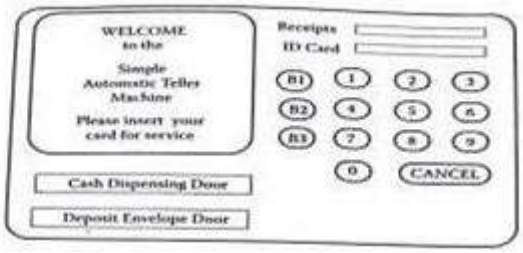

Figure 1.6 The SATM Terminal

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 transaction 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 otherinformation) 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, screen7 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.

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).

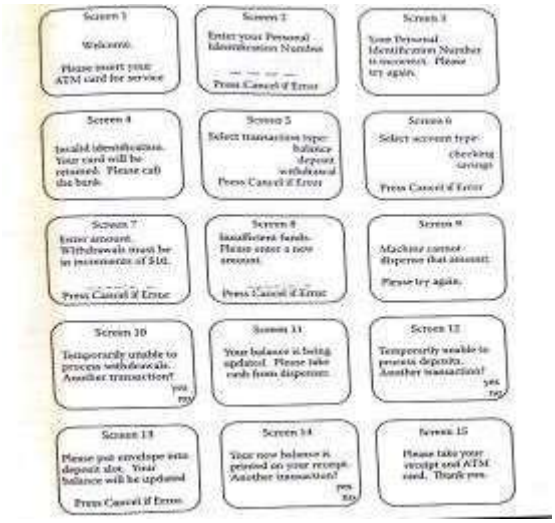

Figure1.7 SATM Screens

A plethora of questions could be resolved by a 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.

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

Figure 1.8: Lever and dial positions