documents is proper understandability by the category of users for whom the document is designed. For achieving this, Gunning's fog index is very useful. We discuss this next.

Gunning's fog index

Gunning's fog index (developed by Robert Gunning in 1952) is a metric that has been designed to measure the readability of a document. The computed metric value (fog index) of a document indicates the number of years of formal education that a person should have, in order to be able to comfortably understand that document. That is, if a certain document has a fog index of 12, any one who has completed his 12th class would not have much difficulty in understanding that document.

The Gunning's fog index of a document D can be computed as follows:

$$\text{fog}(D) = 0.4 \times \left(\frac{\text{words}}{\text{sentences}}\right) + \text{per cent of words having 3 or more syllables}$$

Observe that the fog index is computed as the sum of two different factors. The first factor computes the average number of words per sentence (total number of words in the document divided by the total number of sentences). This factor therefore accounts for the common observation that long sentences are difficult to understand. The second factor measures the percentage of complex words in the document. Note that a syllable is a group of words that can be independently pronounced. For example, the word "sentence" has three syllables ("sen", "ten", and "ce"). Words having more than three syllables are complex words and presence of many such words hamper readability of a document.

mple 10.1 Consider the following sentence: "The Gunning's fog index is based on the ise that use of short sentences and simple words makes a document easy to understand." lculate its Fog index.

The fog index of the above example sentence is

$$0.4 * (23/1) + (4/23) * 100 = 26$$

If a users' manual is to be designed for use by factory workers whose educational qualification is class 8, then the document should be written such that the Gunning's fog index of the document does not exceed 8.

10.4 TESTING

The aim of program testing is to help realiseidentify all defects in a program. However, in practice, even after satisfactory completion of the testing phase, it is not possible to guarantee that a program is error free. This is because the input data domain of most programs is very large, and it is not practical to test the program exhaustively with respect to each value that the input can assume. Consider a function taking a floating point number as argument. If a tester takes 1sec to type in a value, then even a million testers would not be able to exhaustively test it after trying for a million number of years. Even with this obvious limitation of the testing process, we should not underestimate the importance of testing. We must remember that careful testing can expose a large percentage of the defects existing in a program, and therefore provides a practical way of reducing defects in a system.

10.4.1 Basic Concepts and Terminologies

In this section, we will discuss a few basic concepts in program testing on which our subsequent discussions on program testing would be based.

How to test a program?

Testing a program involves executing the program with a set of test inputs and observing if the program behaves as expected. If the program fails to behave as expected, then the input data and the conditions under which it fails are noted for later debugging and error correction. A highly simplified view of program testing is schematically shown in Figure 10.1. The tester has been shown as a stick icon, who inputs several test data to the system and observes the outputs produced by it to check if the system fails on some specific inputs. Unless the conditions under which a software fails are noted down, it becomes difficult for the developers to reproduce a failure observed by the testers. For examples, a software might fail for a test case only when a network connection is enabled.

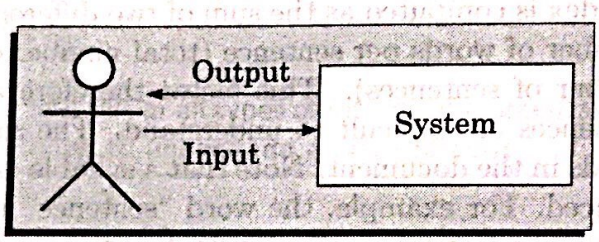

Figure 10.1: A simplified view of program testing.

Terminologies

As is true for any specialised domain, the area of software testing has come to be associated with its own set of terminologies. In the following, we discuss a few important terminologies that have been standardised by the IEEE Standard Glossary of Software Engineering Terminology [IEEE90]:

- A **mistake** is essentially any programmer action that later shows up as an incorrect result during program execution. A programmer may commit a mistake in almost any development activity. For example, during coding a programmer might commit the mistake of not initializing a certain variable, or might overlook the errors that might arise in some exceptional situations such as division by zero in an arithmetic operation. Both these mistakes can lead to an incorrect result.
- An **error** is the result of a mistake committed by a developer in any of the development activities. Among the extremely large variety of errors that can exist in a program. One example of an error is a call made to a wrong function.

> The terms *error, fault, bug,* and *defect* are considered to be synonyms in the area of program testing.

Though the terms *error, fault, bug,* and *defect* are all used interchangeably by the program testing community. Please note that in the domain of hardware testing, the term *fault* is used with a slightly different connotation [IEEE90] as compared to the terms *error* and *bug*.

Example 10.2 Can a designer's mistake give rise to a program error? Give an example of a designer's mistake and the corresponding program error.

Answer: Yes, a designer's mistake give rise to a program error. For example, a requirement might be overlooked by the designer, which can lead to it being overlooked in the code as well.

- **A failure** of a program essentially denotes an incorrect behaviour exhibited by the program during its execution. An incorrect behaviour is observed either as an incorrect result produced or as an inappropriate activity carried out by the program. Every failure is caused by some bugs present in the program. In other words, we can say that every software failure can be traced to some bug or other present in the code. The number of possible ways in which a program can fail is extremely large. Out of the large number of ways in which a program can fail, in the following we give three randomly selected examples:
  - The result computed by a program is 0, when the correct result is 10.
  - A program crashes on an input.
  - A robot fails to avoid an obstacle and collides with it.

It may be noted that mere presence of an error in a program code may not necessarily lead to a failure during its execution.

Example 10.3 Give an example of a program error that may not cause any failure.

Answer: Consider the following C program segment:

```
int markList[1:10]; /* mark list of 10 students*/
int roll;           /* student roll number*/
        ...
if(roll>0)
        markList[roll]=mark;
else
        markList[roll]=0;
```

In the above code, if the variable *roll* assumes zero or some negative value under some circumstances, then an *array index out of bound* type of error would result. However, it may be the case that for all allowed input values the variable *roll* is always assigned positive values. Then, the else clause is unreachable and no failure would occur. Thus, even if an error is present in the code, it does not show up as an error since it is unreachable for normal input values.

Explanation: An *array index out of bound* type of error is said to occur, when the array index variable assumes a value beyond the array bounds.

- **A test case** is a triplet $[I, S, R]$, where I is the data input to the program under test, S is the state of the program at which the data is to be input, and R is the result expected to be produced by the program. The state of a program is also called its *execution mode*. As an example, consider the different execution modes of a certain text editor software. The text editor can at any time during its execution assume any of the following execution modes—edit, view, create, and display. In simple words, we can say that a test case is a

set of test inputs, the mode in which the input is to be applied, and the results that are expected during and after the execution of the test case.

An example of a test case is—[*input: "abc", state: edit, result: abc is displayed*], which essentially means that the input *abc* needs to be applied in the edit mode, and the expected result is that the string *abc* would be displayed.

- A **test scenario** is an abstract test case in the sense that it only identifies the aspects of the program that are to be tested without identifying the input, state, or output. A test case can be said to be an implementation of a test scenario. In the test case, the input, output, and the state at which the input would be applied is designed such that the scenario can be executed. An important automatic test case design strategy is to first design test scenarios through an analysis of some program abstraction (model) and then implement the test scenarios as test cases.
- A **test script** is an encoding of a test case as a short program. Test scripts are developed for automated execution of the test cases.
- A test case is said to be a **positive test case** if it is designed to test whether the software correctly performs a required functionality. A test case is said to be **negative test case**, if it is designed to test whether the software carries out something, that is not required of the system. As one example each of a positive test case and a negative test case, consider a program to manage user login. A positive test case can be designed to check if a login system validates a user with the correct user name and password. A negative test case in this case can be a test case that checks whether the the login functionality validates and admits a user with wrong or bogus login user name or password.
- A **test suite** is the set of all test that have been designed by a tester to test a given program.
- **Testability** of a requirement denotes the extent to which it is possible to determine whether an implementation of the requirement conforms to it in both functionality and performance. In other words, the testability of a requirement is the degree to which an implementation of it can be adequately tested to determine its conformance to the requirement.

Example 10.4 Suppose two programs have been written to implement essentially the same functionality. How can you determine which of these is more testable?

Answer: A program is more testable, if it can be adequately tested with less number of test cases. Obviously, a less complex program is more testable. The complexity of a program can be measured using several types of metrics such as number of decision statements used in the program. Thus, a more testable program should have a lower structural complexity metric.

- A **failure mode** of a software denotes an observable way in which it can fail. In other words, all failures that have similar observable symptoms, constitute a failure mode. As an example of the failure modes of a software, consider a railway ticket booking software that has three failure modes—failing to book an available seat, incorrect seat booking (e.g., booking an already booked seat), and system crash.
- **Equivalent faults** denote two or more bugs that result in the system failing in the same failure mode. As an example of equivalent faults, consider the following two faults in C

language—division by zero and illegal memory access errors. These two are equivalent faults, since each of these leads to a program crash.

Verification *versus* validation

The objectives of both verification and validation techniques are very similar since both these techniques are designed to help remove errors in a software. In spite of the apparent similarity between their objectives, the underlying principles of these two bug detection techniques and their applicability are very different. We summarise the main differences between these two techniques in the following:

- Verification is the process of determining whether the output of one phase of software development conforms to that of its previous phase; whereas validation is the process of determining whether a fully developed software conforms to its requirements specification. Thus, the objective of verification is to check if the work products produced after a phase conform to that which was input to the phase. For example, a verification step can be to check if the design documents produced after the design step conform to the requirements specification. On the other hand, validation is applied to the fully developed and integrated software to check if it satisfies the customer's requirements.
- The primary techniques used for verification include review, simulation, formal verification, and testing. Review, simulation, and testing are usually considered as informal verification techniques. Formal verification usually involves use of theorem proving techniques or use of automated tools such as a model checker. On the other hand, validation techniques are primarily based on product testing. Note that we have categorised testing both under program verification and validation. The reason being that unit and integration testing can be considered as verification steps where it is verified whether the code is as per the module and module interface specifications. On the other hand, system testing can be considered as a validation step where it is determined whether the fully developed code is as per its requirements specification.
- Verification does not require execution of the software, whereas validation requires execution of the software.
- Verification is carried out during the development process to check if the development activities are proceeding alright, whereas validation is carried out to check if the right as required by the customer has been developed.

> We can therefore say that the primary objective of the verification steps are to determine *whether the steps in product development are being carried out alright,* whereas validation is carried out towards the end of the development process to determine *whether the right product has been developed.*

- Verification techniques can be viewed as an attempt to achieve phase containment of errors. Phase containment of errors has been acknowledged to be a cost-effective way to eliminate program bugs, and is an important software engineering principle. The principle of detecting errors as close to their points of commitment as possible is known

as *phase containment of errors*. Phase containment of errors can reduce the effort required for correcting bugs. For example, if a design problem is detected in the design phase itself, then the problem can be taken care of much more easily than if the error is identified, say, at the end of the testing phase. In the later case, it would be necessary not only to rework the design, but also to appropriately redo the relevant coding as well as the system testing activities, thereby incurring higher cost.

While verification is concerned with phase containment of errors, the aim of validation is to check whether the deliverable software is error free.

We can consider the verification and validation techniques to be different types of bug filters. To achieve high product reliability in a cost-effective manner, a development team needs to perform both verification and validation activities. The activities involved in these two types of bug detection techniques together are called the "V and V" activities.

Based on the above discussions, we can conclude that:

Error detection techniques = Verification techniques + Validation techniques

Example 10.5 Is it at all possible to develop a highly reliable software, using validation techniques alone? If so, can we say that all verification techniques are redundant?

Answer: It is possible to develop a highly reliable software using validation techniques alone. However, this would cause the development cost to increase drastically. Verification techniques help achieve phase containment of errors and provide a means to cost-effectively remove bugs.

10.4.2 Testing Activities

Testing involves performing the following main activities:

Test suite design: The set of test cases using which a program is to be tested is designed possibly using several test case design techniques. We discuss a few important test case design techniques later in this Chapter.

Running test cases and checking the results to detect failures: Each test case is run and the results are compared with the expected results. A mismatch between the actual result and expected results indicates a failure. The test cases for which the system fails are noted down for later debugging.

Locate error: In this activity, the failure symptoms are analysed to locate the errors. For each failure observed during the previous activity, the statements that are in error are identified.

Error correction: After the error is located during debugging, the code is appropriately changed to correct the error.

The testing activities have been shown schematically in Figure 10.2. As can be seen, the test cases are first designed, the test cases are run to detect failures. The bugs causing the failure are identified through debugging, and the identified error is corrected.Of all the above mentioned testing activities, debugging often turns out to be the most time-consuming activity.

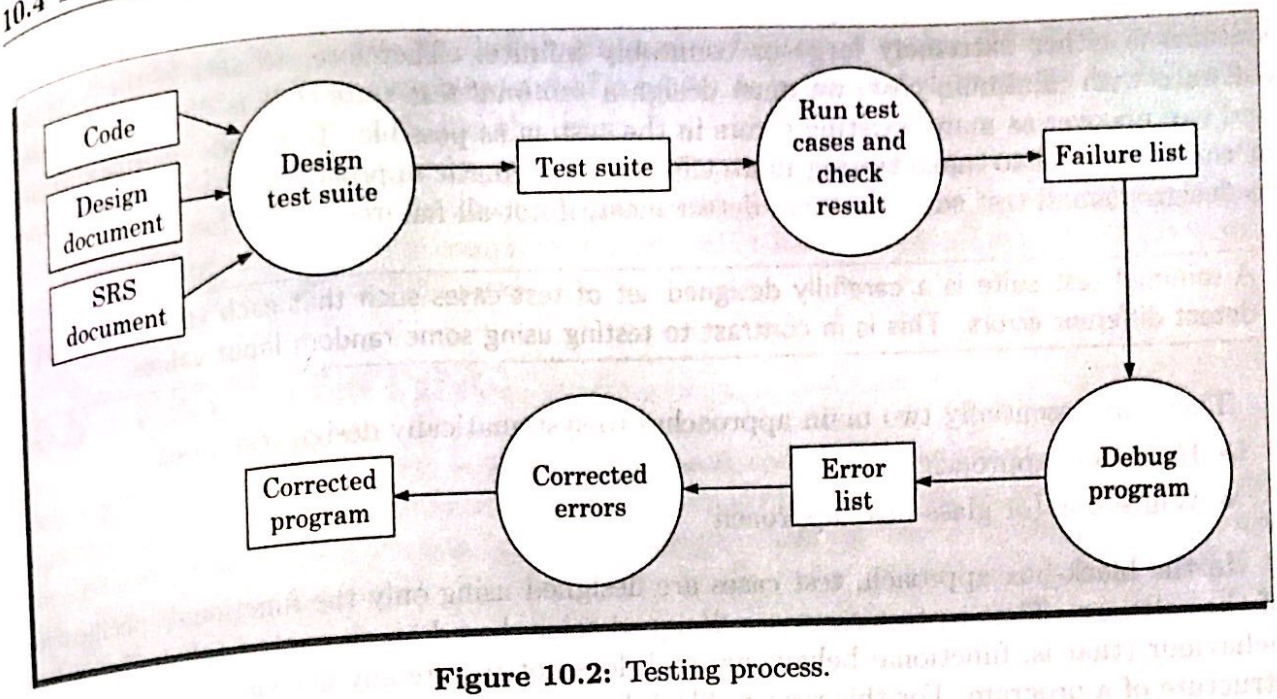

Figure 10.2: Testing process.

10.4.3 Why Design Test Cases?

Before discussing the various test case design techniques, we need to convince ourselves on the following question. Would it not be sufficient to test a software using a large number of random input values? Why design test cases? The answer to this question—this would be very costly and at the same time very ineffective way of testing due to the following reasons:

When test cases are designed based on random input data, many of the test cases do not contribute to the *significance* of the test suite, That is, they do not help detect any additional defects not already being detected by other test cases in the suite.

Testing a software using a large collection of randomly selected test cases does not guarantee that all (or even most) of the errors in the system will be uncovered. Let us try to understand why the number of random test cases in a test suite, in general, does not indicate of the effectiveness of testing. Consider the following example code segment which determines the greater of two integer values `x` and `y`. This code segment has a simple programming error:

```
if (x>y) max = x;
else max = x;
```

For the given code segment, the test suite {(x=3,y=2);(x=2,y=3)} can detect the error, whereas a larger test suite {(x=3,y=2);(x=4,y=3);(x=5,y=1)} does not detect the error. All the test cases in the larger test suite help detect the same error, while the other error in the code remains undetected. So, it would be incorrect to say that a larger test suite would always detect more errors than a smaller one, unless of course the larger test suite has also been carefully designed. This implies that for effective testing, the test suite should be carefully designed rather than picked randomly.

We have already pointed out that exhaustive testing of almost any non-trivial system is impractical due to the fact that the domain of input data values to most practical software

systems is either extremely large or countably infinite. Therefore, to satisfactorily test a software with minimum cost, we must design a *minimal test suite* that is of reasonable size and can uncover as many existing errors in the system as possible. To reduce testing cost and at the same time to make testing more effective, systematic approaches have been developed to design a small test suite that can detect most, if not all failures.

A minimal test suite is a carefully designed set of test cases such that each test case helps detect different errors. This is in contrast to testing using some random input values.

There are essentially two main approaches to systematically design test cases:

- Black-box approach
- White-box (or glass-box) approach

In the black-box approach, test cases are designed using only the functional specification of the software. That is, test cases are designed solely based on an analysis of the input/out behaviour (that is, functional behaviour) and does not require any knowledge of the internal structure of a program. For this reason, black-box testing is also known as *functional testing*. On the other hand, designing white-box test cases requires a thorough knowledge of the internal structure of a program, and therefore white-box testing is also called *structural testing*. Black-box test cases are designed solely based on the input-output behaviour of a program. In contrast, white-box test cases are based on an analysis of the code. These two approaches to test case design are complementary. That is, a program has to be tested using the test cases designed by both the approaches, and one testing using one approach does not substitute testing using the other.

10.4.4 Testing in the Large *versus* Testing in the Small

A software product is normally tested in three levels or stages:

- Unit testing
- Integration testing
- System testing

During unit testing, the individual functions (or units) of a program are tested.

Unit testing is referred to as testing in the small, whereas integration and system testing are referred to as *testing in the large*.

After testing all the units individually, the units are slowly integrated and tested after each step of integration (integration testing). Finally, the fully integrated system is tested (system testing). Integration and system testing are known as *testing in the large*.

Often beginners ask the question—"Why test each module (unit) in isolation first, then integrate these modules and test, and again test the integrated set of modules—why not just test the integrated set of modules once thoroughly?" The answer to this question is the following—There are two main reasons to it. First while testing a module, other modules with which this module needs to interface may not be ready. Moreover, it is always a good idea

to first test the module in isolation before integration because it makes debugging easier. If a failure is detected when an integrated set of modules is being tested, it would be difficult to determine which module exactly has the error.

In the following sections, we discuss the different levels of testing. It should be borne in mind in all our subsequent discussions that unit testing is carried out in the coding phase itself as soon as coding of a module is complete. On the other hand, integration and system testing are carried out during the testing phase.

10.5 UNIT TESTING

Unit testing is undertaken after a module has been coded and reviewed. This activity is typically undertaken by the coder of the module himself in the coding phase. Before carrying out unit testing, the unit test cases have to be designed and the test environment for the unit under test has to be developed. In this section, we first discuss the environment needed to perform unit testing.

Driver and stub modules

In order to test a single module, we need a complete environment to provide all relevant code that is necessary for execution of the module. That is, besides the module under test, the following are needed to test the module:

- The procedures belonging to other modules that the module under test calls.
- Non-local data structures that the module accesses.
- A procedure to call the functions of the module under test with appropriate parameters.

Modules required to provide the necessary environment (which either call or are called by the module under test) are usually not available until they too have been unit tested. In this context, *stubs* and *drivers* are designed to provide the complete environment for a module so that testing can be carried out.

Stub: The role of stub and driver modules is pictorially shown in Figure 10.3. A *stub* procedure is a dummy procedure that has the same I/O parameters as the function called by

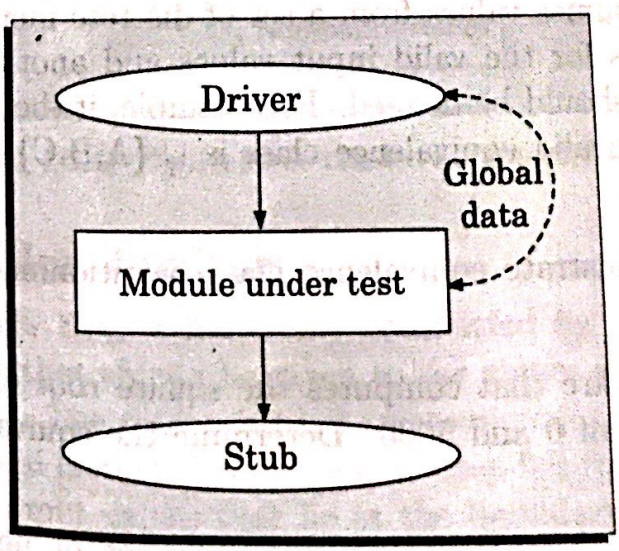

Figure 10.3: Unit testing with the help of driver and stub modules.

the unit under test but has a highly simplified behaviour. For example, a stub procedure may produce the expected behaviour using a simple table look up mechanism.

Driver: A driver module should contain the non-local data structures accessed by the module under test. Additionally, it should also have the code to call the different functions of the unit under test with appropriate parameter values for testing.

10.6 BLACK-BOX TESTING

In black-box testing, test cases are designed from an examination of the input/output values only and no knowledge of design or code is required. The following are the two main approaches available to design black box test cases:

- Equivalence class partitioning
- Boundary value analysis

In the following subsections, we will elaborate these two test case design techniques.

10.6.1 Equivalence Class Partitioning

In the equivalence class partitioning approach, the domain of input values to the program under test is partitioned into a set of equivalence classes. The partitioning is done such that for every input data belonging to the same equivalence class, the program behaves similarly.

> The main idea behind defining equivalence classes of input data is that testing the code with any one value belonging to an equivalence class is as good as testing the code with any other value belonging to the same equivalence class.

Equivalence classes for a unit under test can be designed by examining the input data and output data. The following are two general guidelines for designing the equivalence classes:

1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes need to be defined. For example, if the equivalence class is the set of integers in the range 1 to 10 (i.e., [1,10]), then the invalid equivalence classes are $[-\infty,0]$, $[11,+\infty]$.
2. If the input data assumes values from a set of discrete members of some domain, then one equivalence class for the valid input values and another equivalence class for the invalid input values should be defined. For example, if the valid equivalence classes are {A,B,C}, then the invalid equivalence class is $\cup$-{A,B,C}, where $\cup$ is the universe of possible input values.

In the following, we illustrate equivalence class partitioning-based test case generation through four examples.

Example 10.6 For a software that computes the square root of an input integer that can assume values in the range of 0 and 5000. Determine the equivalence classes and the black box test suite.

Answer: There are three equivalence classes—The set of negative integers, the set of integers in the range of 0 and 5000, and the set of integers larger than 5000. Therefore, the

test cases must include representatives for each of the three equivalence classes. A possible test suite can be: {-5,500,6000}.

Example 10.7 Design the equivalence class test cases for a program that reads two integer pairs (m_1, c_1) and (m_2, c_2) defining two straight lines of the form `y=mx+c`. The program computes the intersection point of the two straight lines and displays the point of intersection.

Answer: The equivalence classes are the following:

- Parallel lines $(m_1 = m_2, c_1 \neq c_2)$
- Intersecting lines $(m_1 \neq m_2)$
- Coincident lines $(m_1 = m_2, c_1 = c_2)$

Now, selecting one representative value from each equivalence class, we get the required equivalence class test suite {(2,2)(2,5),(5,5)(7,7), (10,10)(10,10)}.

Example 10.8 Design equivalence class partitioning test suite for a function that reads a character string of size less than five characters and displays whether it is a palindrome.

Answer: The equivalence classes are the leaf level classes shown in Figure 10.4. The equivalence classes are palindromes, non-palindromes, and invalid inputs. Now, selecting one representative value from each equivalence class, we have the required test suite: {abc,aba,abcdef}.

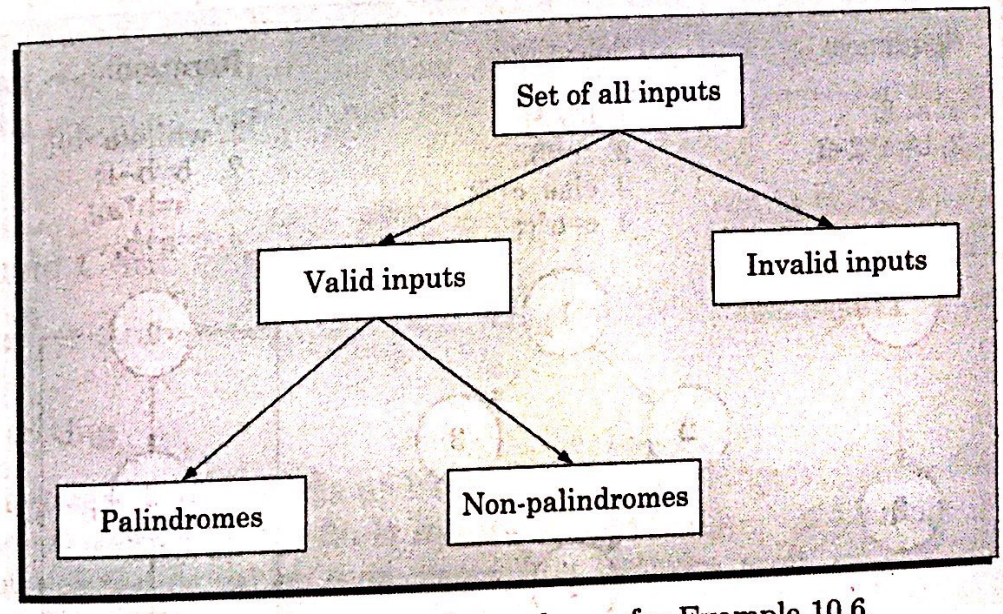

Figure 10.4: Equivalence classes for Example 10.6.

10.6.2 Boundary Value Analysis

A type of programming error that is frequently committed by programmers is missing out on the special consideration that should be given to the values at the boundaries of different equivalence classes of inputs. The reason behind programmers committing such errors might purely be due to psychological factors. Programmers often fail to properly address the special processing required by the input values that lie at the boundary of the different equivalence classes. For example, programmers may improperly use < instead of <=, or conversely <= for <, etc.

Jamie: I'm a bit concerned that we won't have enough time to conduct all the reviews. In fact, I know we won't.

Doug: Hmmm. So what do you propose?

Jamie: I say we select those elements of the analysis and design model that are most critical to *SafeHome* and review them.

Vinod: But what if we miss something in a part of the model we don't review?

Shakira: I read something about a sampling technique [Section 26.4.4] that might help us target candidates for review. (Shakira explains the approach.)

Jamie: Maybe . . . but I'm not sure we even have time to sample every element of the models.

Vinod: What do you want us to do, Doug?

Doug: Let's steal something from Extreme Programming [Chapter 4]. We'll develop the elements of each model in pairs—two people—and conduct an informal review of each as we go. We'll then target "critical" elements for a more formal team review, but keep those reviews to a minimum. That way, everything gets looked at by more than one set of eyes, but we still maintain our delivery dates.

Jamie: That means we're going to have to revise the schedule.

Doug: So be it. Quality trumps schedule on this project.

26.5 Formal Approaches to SQA

Over the past two decades, a small, but vocal, segment of the software engineering community has argued that a more formal approach to software quality assurance is required. It can be argued that a computer program is a mathematical object [SOM01]. A rigorous syntax and semantics can be defined for every programming language, and a rigorous approach to the specification of software requirements (Chapter 28) is available. If the requirements model (specification) and the programming language can be represented in a rigorous manner, it should be possible to apply mathematic proof of correctness to demonstrate that a program conforms exactly to its specifications.

Attempts to prove programs correct (Chapters 28 and 29) are not new. Dijkstra [DIJ76] and Linger, Mills, and Witt [LIN79], among others, advocated proofs of program correctness and tied these to the use of structured programming concepts (Chapter 11).

26.6 Statistical Software Quality Assurance

Statistical quality assurance reflects a growing trend throughout industry to become more quantitative about quality. For software, statistical quality assurance implies the following steps:

What steps are required to perform statistical SQA?

1. Information about software defects is collected and categorized.
2. An attempt is made to trace each defect to its underlying cause (e.g., nonconformance to specifications, design error, violation of standards, poor communication with the customer).

3. Using the Pareto principle (80 percent of the defects can be traced to 20 percent of all possible causes), isolate the 20 percent (the "vital few").
4. Once the vital few causes have been identified, move to correct the problems that have caused the defects.

This relatively simple concept represents an important step towards the creation of an adaptive software process in which changes are made to improve those elements of the process that introduce error.

> "20 percent of the code has 80 percent of the errors. Find them, fix them!"
>
> **Lowell Arthur**

26.6.1 A Generic Example

To illustrate the use of statistical methods for software engineering work, assume that a software engineering organization collects information on defects for a period of one year. Some of the defects are uncovered as software is being developed. Others are encountered after the software has been released to its end-users. Although hundreds of different defects are uncovered, all can be tracked to one (or more) of the following causes:

- Incomplete or erroneous specifications (IES).
- Misinterpretation of customer communication (MCC).
- Intentional deviation from specifications (IDS).
- Violation of programming standards (VPS).
- Error in data representation (EDR).
- Inconsistent component interface (ICI).
- Error in design logic (EDL).
- Incomplete or erroneous testing (IET).
- Inaccurate or incomplete documentation (IID).
- Error in programming language translation of design (PLT).
- Ambiguous or inconsistent human/computer interface (HCI).
- Miscellaneous (MIS).

To apply statistical SQA, the table in Figure 26.5 is built. The table indicates that IES, MCC, and EDR are the vital few causes that account for 53 percent of all errors. It should be noted, however, that IES, EDR, PLT, and EDL would be selected as the vital few causes if only serious errors are considered. Once the vital few causes are determined, the software engineering organization can begin corrective action. For example, to correct MCC, the software developer might implement facilitated requirements gathering techniques (Chapter 7) to improve the quality of customer communication and specifications. To improve EDR, the developer

FIGURE 26.5 Data collection for statistical SQA

	Total		Serious		Moderate		Minor	
Error	No.	%	No.	%	No.	%	No.	%
IES	205	22%	34	27%	68	18%	103	24%
MCC	156	17%	12	9%	68	18%	76	17%
IDS	48	5%	1	1%	24	6%	23	5%
VPS	25	3%	0	0%	15	4%	10	2%
EDR	130	14%	26	20%	68	18%	36	8%
ICI	58	6%	9	7%	18	5%	31	7%
EDL	45	5%	14	11%	12	3%	19	4%
IET	95	10%	12	9%	35	9%	48	11%
IID	36	4%	2	2%	20	5%	14	3%
PLT	60	6%	15	12%	19	5%	26	6%
HCI	28	3%	3	2%	17	4%	8	2%
MIS	56	6%	0	0%	15	4%	41	9%
Totals	942	100%	128	100%	379	100%	435	100%

might acquire tools for data modeling and perform more stringent data design reviews.

It is important to note that corrective action focuses primarily on the vital few. As the vital few causes are corrected, new candidates pop to the top of the stack.

Statistical quality assurance techniques for software have been shown to provide substantial quality improvement [ART97]. In some cases, software organizations have achieved a 50 percent reduction per year in defects after applying these techniques.

The application of the statistical SQA and the Pareto principle can be summarized in a single sentence: *Spend your time focusing on things that really matter, but first be sure that you understand what really matters!*

A comprehensive discussion of statistical SQA is beyond the scope of this book. Interested readers should see [GOH02], [SCH98], or [KAN95].

26.6.2 Six Sigma for Software Engineering

Six Sigma is the most widely used strategy for statistical quality assurance in industry today. Originally popularized by Motorola in the 1980s, the Six Sigma strategy "is a rigorous and disciplined methodology that uses data and statistical analysis to measure and improve a company's operational performance by identifying and eliminating 'defects' in manufacturing and service-related processes." [ISI03]. The term "six sigma" is derived from six standard deviations—3.4 instances (defects) per million occurrences—implying an extremely high quality standard. The Six Sigma methodology defines three core steps:

What are the core steps of the six sigma methodology?

- *Define* customer requirements, deliverables, and project goals via well-defined methods of customer communication.

- *Measure* the existing process and its output to determine current quality performance (collect defect metrics).
- *Analyze* defect metrics and determine the vital few causes.

If an existing software process is in place, but improvement is required, Six Sigma suggests two additional steps:

- *Improve* the process by eliminating the root causes of defects.
- *Control* the process to ensure that future work does not reintroduce the causes of defects.

These core and additional steps are sometimes referred to as the DMAIC (define, measure, analyze, improve, and control) method.

If an organization is developing a software process (rather than improving an existing process), the core steps are augmented as follows:

- *Design* the process to (1) avoid the root causes of defects and (2) to meet customer requirements
- *Verify* that the process model will, in fact, avoid defects and meet customer requirements.

This variation is sometimes called the DMADV (define, measure, analyze, design, and verify) method.

A comprehensive discussion of Six Sigma is best left to resources dedicated to the subject. The interested reader should see [ISI03], [SNE03], and [PAN00].

26.7 Software Reliability

Software reliability, unlike many other quality factors, can be measured directed and estimated using historical and developmental data. *Software reliability* is defined in statistical terms as "the probability of failure-free operation of a computer program in a specified environment for a specified time" [MUS87]. To illustrate, program *X* is estimated to have a reliability of 0.96 over eight elapsed processing hours. In other words, if program *X* were to be executed 100 times and require a total of eight hours of elapsed processing time (execution time), it is likely to operate correctly (without failure) 96 times.

> "The unavoidable price of reliability is simplicity."
>
> C.A.R. Hoare

Whenever soft