D.N.R. COLLEGE (AUTONOMOUS): BHIMAVARAM

M.Sc(CS) DEPARTMENT

SOFTWARE TESTING AND QUALITY ASSURANCE

M.Sc(CS) DEPARTMENT

Presented by

K. SUPARNA

QUALITY REVOLUTION

Developing quality products on tighter schedules is critical for a company to be successful in the new global economy. Traditionally, efforts to improve quality have centered around the end of the product development cycle by emphasizing the detection and correction of defects. On the contrary, the new approach to enhancing quality encompasses all phases of a product development process—from a requirements analysis to the final delivery of the product to the customer. Every step in the development process must be performed to the highest Possible standard.

An effective quality process must focus on [1]:

- Paying much attention to customer's requirements
- Making efforts to continuously improve quality
- Integrating measurement processes with product design and development
- Pushing the quality concept down to the lowest level of the organization
- Developing a system-level perspective with an emphasis on methodology and process
- Eliminating waste through continuous improvement

In July 1950, Deming gave an eight-day seminar based on the Shewhart methods of statistical quality control [4, 5] for Japanese engineers and executives. He introduced the *plan–do–check–act* (PDCA) cycle in the seminar, which he called the Shewhart cycle (Figure 1.1). The Shewhart cycle illustrates the following activity sequence: setting goals, assigning them to measurable milestones, and assessing the progress against thosemilestones.

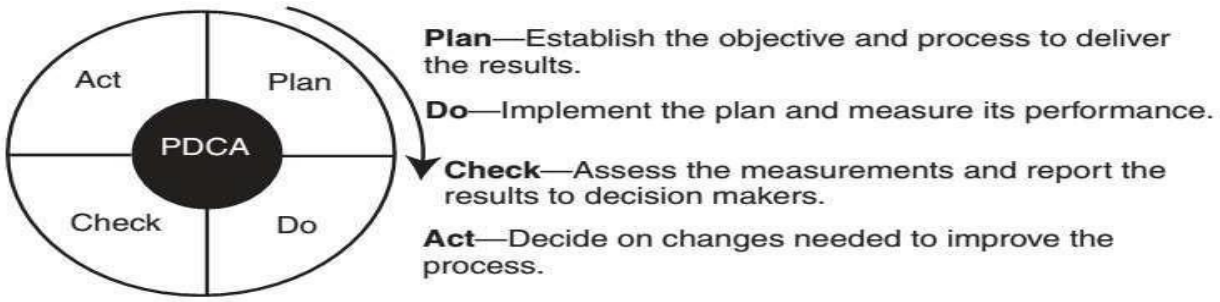


Figure 1.1 Shewhart cycle.

SOFTWARE QUALITY

Garvin [18] has analyzed how software quality is perceived in

different ways in different domains, such as philosophy, economics, marketing, and management. Kitchenham and Pfleeger's article [60] on software quality gives a succinct exposition of software quality. They discuss five views of quality in a comprehensive manner as follows:

1. *Transcendental View*: It envisages quality as something that can be recognized but is difficult to define. The transcendental view is notspecific to software quality alone but has been applied in other complex areas
of everyday life.

2. *User View*: It perceives quality as fitness for purpose. According tothis view, while evaluating the quality of a product, one must ask the key question: "Does the product satisfy user needs and expectations?"'
3. *Manufacturing View*: Here quality is understood as conformance to the specification. The quality level of a product is determined by the extentto which the product meets its specifications.
4. *Product View*: In this case, quality is viewed as tied to the inherent characteristics of the product. Aproduct's inherent characteristics, that is, internal qualities, determine its external qualities.
5. *Value-Based View*: Quality, in this perspective, depends on the amount a customer is willing to pay for it.

In mid-1970s. McCall, Richards, and Walters

[19] were the first to study the concept of software quality in terms of *quality factors* and *quality criteria*. A quality factor represents a behavioral characteristic of a system. Some examples of high-level quality factors are *correctness*, *reliability*, *efficiency*, *testability*, *maintainability*, and *reusability*.

ROLE OF TESTING

- Testing plays an important role in achieving and assessing the quality of a software product [25]. On the one hand, we improve the quality of the products as we repeat a *test–find defects–fix* cycle during development. On the other hand, we assess how good our system is when we perform system-level tests before releasing a product.
  Generally speaking, the activities for software quality assessment can be divided into two broad categories, namely, *static analysis* and *dynamic analysis*.

- **Static Analysis:** As the term "static" suggests, it is based on theexamination of a number of documents, namely requirements documents, software models, design documents, and source code. Traditional static analysis includes code review, inspection, walk-through, algorithm analysis, and proof of correctness. It does not involve actual execution of the code under development. Instead, it examines code and reasons over all possible behaviors that might arise during run time. Compiler optimizations are standard
static analysis.

- **Dynamic Analysis:** Dynamic analysis of a software system involves actual program execution in order to expose possible program failures. The behavioral and performance properties of the program are also observed. Programs are executed with both typical and carefully chosen input values.

VERIFICATION AND VALIDATION

Verification: This kind of activity helps us in evaluating a software system by determining whether the product of a given development phase satisfies the requirements established before the start of that phase. One may note that a product can be an intermediate product, such as requirement specification, design specification, code, user manual, or even the final product. Activities that check the correctness of a development phase are called *verification activities*.

- **Validation:** Activities of this kind help us in confirming that a product meets its intended *use*. Validation activities aim at confirming that aproduct meets its customer's expectations. In other words, validation activities focus on the final product, which is extensively tested from the customer point of view. Validation establishes whether the product meets overall expectations of the users.

FAILURE, ERROR, FAULT, AND DEFECT

• **Failure:** A failure is said to occur whenever the external behavior of a system does not conform to that prescribed in the system specification.

- **Error:** An error is a *state* of the system. In the absence of any corrective action by the system, an error state could lead to a failure which would not be attributed to any event subsequent to the error.

- **Fault:** A fault is the adjudged cause of an error.

In a software context, a software system may be defective due to design issues; certain system states will expose a defect, resulting in the development of faults defined as incorrect signal values or decisions within the system.

OBJECTIVES OF TESTING

The stakeholders in a test process are the programmers, the testengineers, the project managers, and the customers. A stakeholder is a person or an organization who influences a system's behaviors or who is impacted by that system

- **It does work:** While implementing a program unit, the programmer may want to test whether or not the unit works in normal circumstances. The programmer gets much confidence if the unit works to his or her satisfaction.
- The same idea applies to an entire system as well—once a system has been integrated, the developers may want to test whether or not the system performs the basic functions.
- **It does not work:** Once the programmer (or the development team) is satisfied that a unit (or the system) works to a certain degree, more tests are conducted with the objective of finding faults in the unit (or the system). Here, the idea is to try to make the unit (or the system) fail.
- **Reduce the risk of failure:** Most of the complex software systems contain faults, which cause the system to fail from time to time. This concept of "failing from time to time" gives rise to the notion of *failure rate*. As faults are discovered and fixed while performing more and more tests, the failure rate of a system generally decreases.
- **Reduce the cost of testing:** The different kinds of costs associated with a test process include the cost of designing, maintaining, and executing test cases, the cost of analyzing, the result of executing each test case, the cost of documenting the testcases, and the cost of actually executing the systemand documenting it.

WHAT IS A TEST CASE?

- In its most basic form, a *test case* is a simple pair of < input, expected outcome >.
  If a program under test is expected to compute the square root of nonnegative numbers, then four examples of test cases are as shown in Figure 1.3.

  In stateless systems, where the outcome depends solely on the current input, test cases are very simple in structure, as shown in Figure 1.3.

TB_1: $< 0, 0 >$,
TB_2: $< 25, 5 >$,
TB_3: $< 40, 6.3245553 >$,
TB_4: $< 100.5, 10.024968 >$.

Figure 1.3 Examples of basic test cases

A program to compute the square root of nonnegative numbers is an example of a stateless system. A compiler for the Cprogramming language is another example of a stateless system. A compiler is a stateless system because to compile a program it does not need to know about the programs it compiled previously.

In state-oriented systems, where the program outcome depends both on the current state of the system and the current input, a test case may consist of
a sequence of < input, expected outcome > pairs.
A telephone switching system and an automated teller machine (ATM) are examples of state- oriented systems

For an ATM machine, a test case for testing the *withdraw*function .c

Here, we assume that the user has already entered validated inputs, such as the cash card and the personal identification number (PIN). In the test case TS1, “check balance” and “withdraw” in the first, second, and fourth tuples represent the pressing of the appropriate keys on the ATM keypad. It is assumed that the user account has $500.00 on it, and the user wants to withdraw an amount of $200.00. The expected outcome “$200.00” in the third tuple represents

the cash dispensed by the ATM. After the withdrawal operation, the user makes sure that the remaining balance is

$300.00

EXPECTED OUTCOME

- An *outcome* of program execution is a complex entity that may include the following:
  - **Values produced by the program:** Outputs for local observation (integer, text, audio, image) Outputs (messages) for remote storage, manipulation, or observation
- **State change:** State change of the program

State change of the database (due to add, delete, and update operations)

- A sequence or set of values which must be interpreted together for the outcome to be valid.
- An important concept in test design is the concept of an *oracle*. An oracle is any entity—program, process, human expert, or body of data—that tells us the expected outcome of a particular test or set of tests [40]. A test case is meaningful

only if it is possible to decide on the acceptability of the result produced by the program under test.

identify expected outcomes by examining the actual test outcomes, as explained in the following:

1. Execute the program with the selected input.
2. Observe the actual outcome of program execution.
3. Verify that the actual outcome is the expected outcome.
4. Use the verified actual outcome as the expected outcome in subsequent runs of the test case.

TESTING ACTIVITIES

- In order to test a program, a test engineer must perform a sequence of testing activities. Most of these activities have been shown in Figure 1.6 and are explained in the following. These explanations focus on a single test case.

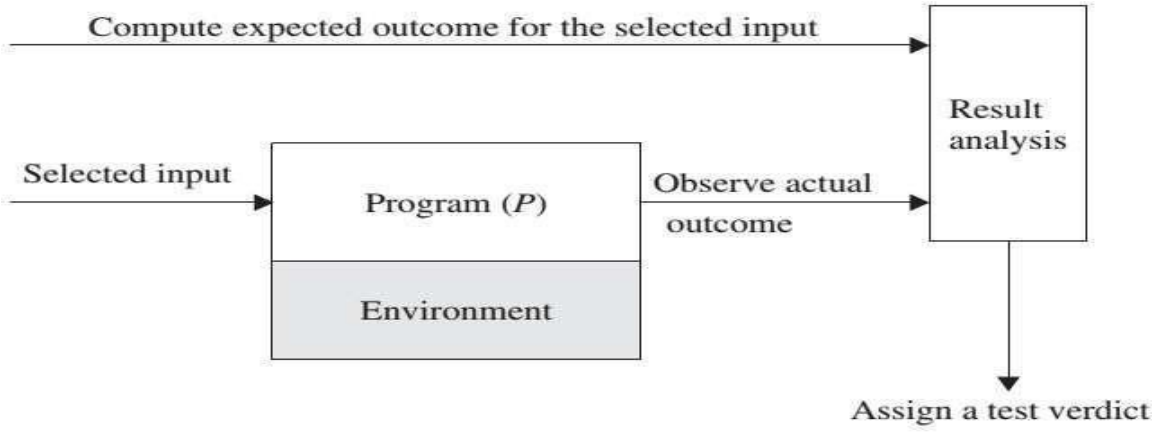


Figure 1.6 Different activities in program testing.

- **Select inputs:** The second activity is to select test inputs.

  Selection of test inputs can be based on the requirements specification, the source code, or our expectations. Test inputs are selected by keeping the test objective in mind.
- **Compute the expected outcome:** The third activity is to compute the expected outcome of the program with the selected inputs. In most cases, this can be done from an overall, high-level understanding of the test objective and the specification of the program under test.
- **Set up the execution environment of the program:** The
  Fourth step is to Prepare the right execution environment of the program. In this step all the assumptions external to the program must be satisfied.
- **Execute the program:** In the fifth step, the test engineer executes the program with the selected inputs and observes the actual outcome of the program.To execute a test case, inputs may be provided to the program at different physical locations at different times. The concept of *test coordination* is used in synchronizing different components of a test case.
- **Analyze the test result:** The final test activity is to analyze the result of test execution. Here, the main task is to compare the actual outcome of program execution with the expected outcome.

TEST LEVELS

Testing is performed at different levels involving the complete system or parts of it throughout the life cycle of a software product. A software system goes through four stages oftesting before it is actually deployed.

These four stages are known as *unit*, *integration*, *system*, and *acceptance* level testing.
The first three levels of testing are performed by a number of different stakeholders in the development organization, where as acceptance testing is performed by the customers. The four stages of testing have been illustrated in the form of what is called the classical V model in Figure 1.7.

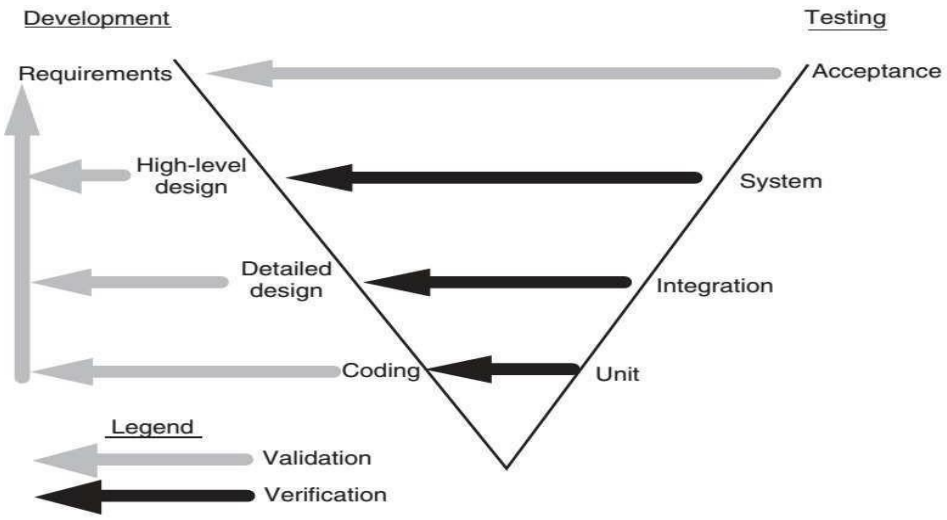


Figure 1.7 Development and testing phases in the V model.

1. **unit testing,** programmers test individual program units, such as a procedures, functions, methods, or classes, in isolation. After ensuring that individual units work to a satisfactory extent, modules are assembled to construct larger subsystems by following integration testing techniques.
2. **Integration testing** is jointly performed by software developers and integration test engineers. The objective of integration testing is to construct a reasonably stable system that can withstandthe rigor of system-level testing.
3. **System-level testing** includes a wide spectrum of testing, such as functionality testing, security testing, robustness testing, load testing, stability testing, stress testing, performance testing, and reliability testing.

There are three major kinds of test verdicts, namely, *pass*, *fail*,and *inconclusive*, as explained below. If the program produces the expected outcome and the purpose of the test case is satisfied, then a pass verdict is assigned.
If the program does not produce the expected outcome, then a fail verdict is assigned.However, in some cases it may not be possible to assign a clear pass or fail verdict.A *test report* must be written after analyzing the test result. The motivation for writing a test report is to get the fault fixed if the test revealed a fault.

System testing is a critical phase in a software development process because of the need to meet a tight schedule closeto delivery date, to discover most of the faults, and to verify that fixes are working and have not resulted in new faults. System testing comprises a number of distinct activities: creating a test plan, designing a test suite, preparing test environments, executing the tests by following a clear strategy, and monitoring the process of test execution.

4. ***Regression testing*** is another level of testing that is performedthroughout the life cycle of a system. Regression testing is performed whenever a component of the system is modified. The key idea in regression testingis to ascertain that the modification has not introduced any new faults in the portion that was not subject to modification. To be precise, regression testing is not a distinct level of testing.

Rather, it is considered as a subphase of unit, integration, and system-level testing, as illustrated in Figure 1.8 [41]. In regression testing In regression testing, new tests are not designed. Instead, tests are selected, prioritized, and executed from the existing pool of test cases to ensure that nothing is

broken in the new version of the software.

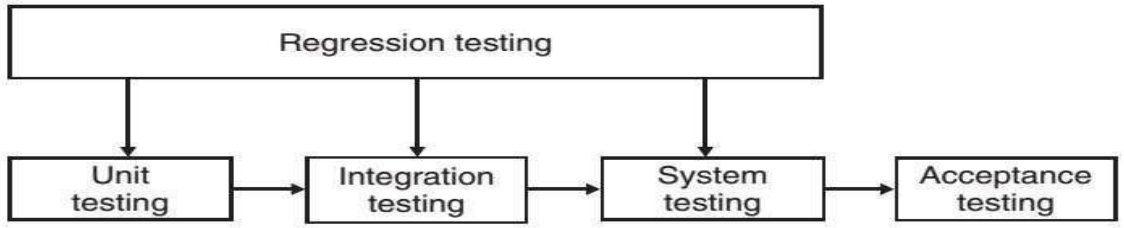


Figure 1.8 Regression testing at different software testing levels. (From
John Wiley & Sons.)

Regression testing is an expensive process and accounts for a predominant portion of testing effort in the industry. It is desirable to select a subset of the test cases from the existing pool to reduce the cost. A key question is how many andwhich test cases should be selected so that the selected test cases are more likely to uncover new faults .

After the completion of system-level testing, the product is delivered to the customer. The customer performs their own series of tests, commonly known as *acceptance testing*.

The objective of acceptance testing is to measure the quality of the product, rather than searching for the defects, which is objective of system testing. A key notion in acceptance testing is the customer's *expectations* from the system. By the time of acceptance testing, the customer should have developed their acceptance criteria based on their own expectations from the system.

There are two kinds of acceptance testing as explained in the following:

- User acceptance testing (UAT)
- Business acceptance testing (BAT)

User acceptance testing is conducted by the customer to ensure that the system satisfies the contractual acceptance criteria before being signed off as meeting user needs. On the other hand, BAT is undertaken within the supplier's development organization. The idea in having a BAT is to ensure that the system will eventually pass the user acceptance test. It is a rehearsal of UAT at the supplier's premises.

SOURCES OF INFORMATION FOR TEST CASE SELECTION

Designing test cases has continued to stay in the foci of the research community and the practitioners. A software development process generates a large body of information, such as requirements specification, design document, and source code.

In order to generate effective tests at a lower cost, test designers analyze the following sources of information:

- Requirements and functional specifications
- Source code
- Input and output domains
- Operational profile
- Fault model

- ***Requirements and Functional Specifications:*** The process of software development begins by capturing user needs. The nature and amount ofuser needs identified at the beginning of system development will vary depending on the specific life-cycle model to be followed.
- ***Source Code:*** Whereas a requirements specification describes the *intended behavior* of a system, the source code describes the *actual behavior* of the system.
  High-level assumptions and constraints take concrete form in an implementation.
- **I*nput and Output Domains:*** Some values in the input domain of a program have special meanings, and hence must be treated separately [5]. To illustrate this point, let us consider the *factorial* function. The factorial of a nonnegative integer *n* is computed as follows:
  factorial(0) = 1;
  factorial(1) = 1;
  factorial(n) = n * factorial(n-1);
  A programmer may wrongly implement the factorial function as factorial(n) = 1 * 2 * ... * n; without considering the special case of *n* = 0.

- ***Operational Profile :***As the term suggests, an *operational profile* is a quantitative characterization of how a system will be used. It was created to guide test engineers in selecting test cases (inputs) using samples of system usage. The idea is to infer, from the observed test results, the future reliability of the software when it is in actual use.

- ***Fault Model :*** Previously encountered faults are an excellent source of information in designing new test cases. The known faults are classified into different classes, such as initialization faults, logic faults, and interface faults, and stored in a repository.

  There are three types of fault-based testing: error guessing, fault seeding, and mutation analysis

Unit testing basics

- Unit testing refers to testing program units in isolation. a program unit is a piece of code, such as a function or method of class, that is invoked from outsidethe unit and that can invoke other program units.
- A program unit is tested in isolation, that is, in a stand-alone manner. There are two reasons for testing a unit in a stand-alone manner. First, errors found during testing can be attributed to a specific unit so that it can be easily fixed. Moreover, unit testing removes dependencies on other program units. Second, during unit testing it is desirable to verify that each distinct execution of a programunit produces the expected result. In terms of code details, a distinct execution refers to a distinct path in the unit.

- Unit testing is conducted in two complementaryphases:
  - Static unit testing
  - Dynamic unit testing
- In static unit testing, a programmer does not execute the unit; instead, the code is examined over all possible behaviors that might arise during run time.
- Static unit testing is also known as non-execution-based unit testing.

In static unit testing, the code of each unit is validated against requirements of the unit by reviewing the code. During the review process, potential issues are identified and resolved

Static Unit testing

- Static unit testing is conducted as a part of a larger philosophical beliefthat a software product should undergo a phase of inspection and correction at each milestone in its life cycle. At a certain milestone, the product neednot be in its final form.

  In static unit testing, code is reviewed by applying techniques commonly known as *inspection* and *walkthrough*. The original definition of inspection was coined by Michael Fagan [1] and that of walkthrough by Edward Yourdon

  [2]:

  - **Inspection:** It is a step-by-step peer group review of a work product, with each step checked against predetermined criteria.
  - **Walkthrough:** It is a review where the author leads the team through a manual or simulated execution of the product using predefined scenarios.

- Regardless of whether a review is called an inspection or a walkthrough, it is a systematic approach to examining source code in detail.

  The objective of code review is to review the code, not to evaluate the author of the code. A clash may occur between the author of the codeand the reviewers, and this may make the meetings unproductive. Therefore, code review must be
  planned and managed in a professional manner. There is a need for mutual respect, openness, trust, and sharing of expertise in the group.
  The general guidelines for performing code review consists of six steps as outlined in Figure 3.1: readiness, preparation, examination, rework, validation, and exit

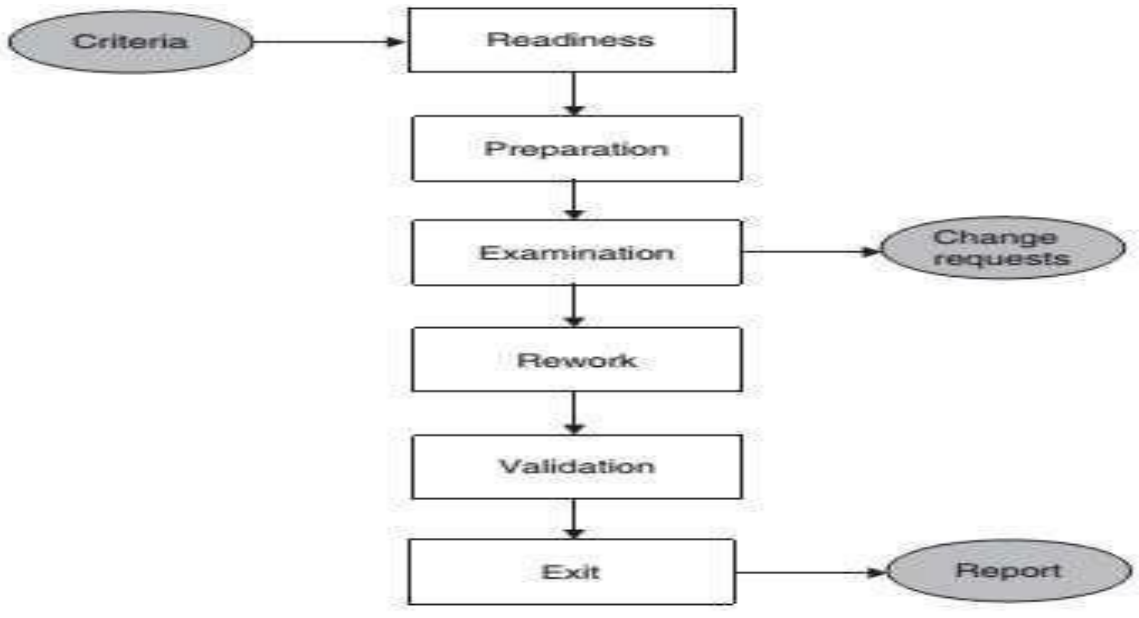


Figure 3.1 Steps in the code review process.

- These steps and documents are explained in the following.
  Step 1: Readiness The author of the unit ensures that the unit under test is ready for review. A unit is said to be ready if it satisfies the following criteria
    - **Completeness:** All the code relating to the unit to be reviewed must be available. This is because the reviewers are going to read the code and try to understand it. It is unproductive to review partially writtencode

  or code that is going to be significantly modified by the programmer.
    - **Minimal Functionality:** The code must compile and link. Moreover, the code must have been tested to some extent to make sure that it performs its basic functionalities.
    - **Readability:** Since code review involves actual reading of code by other programmers, it is essential that the code is highly readable.

  • **Complexity:** There is no need to schedule a group meeting to review straightforward code which can be easily reviewed by the programmer. The code to be reviewed must be of sufficient complexity to warrant group review.

- • **Requirements and Design Documents:** The latest approved version of the low-level design

TABLE 3.1 Hierarchy of System Documents

Requirement: High-level marketing or product proposal.
Functional specification: Software engineering response to the marketing proposal.
High-level design: Overall system architecture.
Low-level design: Detailed specification of the modules within the architecture.
Programming: Coding of the modules.

  specification or other appropriate descriptions of program requirements (see Table 3.1) should be available. These documents help the reviewers in verifying whether or not the code under review implements the expected functionalities. If the low-level design document is available, it helps the reviewers in assessing whether or not the code appropriately implements the design.

- All the people involved in the review process are informed of the group review meeting schedule two or three days before the meeting. They are also given a copy of the work package for their perusal.Reviews are conducted in bursts of 1–2 hours. Longer meetings are less and less productive because of the limited attention span of human beings.

- The rate of code review is restricted to about 125 lines of code (in a high-level language) per hour. Reviewing complex code at a higher rate will result in just glossing over the code, thereby defeating the fundamental purpose of code review. The composition of the review group involves a number of people with different roles. These roles are explained as follows:
- **Moderator:** A review meeting is chaired by the moderator. Themoderator
- is a trained individual who guides the pace of the review process.
- The moderator selects the reviewers and schedules the review meetings.

- **Author:** This is the person who has written the code to be reviewed.
- **Presenter:** A presenter is someone other than the author of the code.

- The presenter reads the code beforehand to understand it
- **Recordkeeper:** The recordkeeper documents the problems found during the review process and the follow-up actions suggested. The person should be different than the author and the moderator.
- **Reviewers:** These are experts in the subject area of the code under review. The group size depends on the content of the material under review. As a rule of thumb, the group size is between 3 and 7.

- **Observers:** These are people who want to learn about the code under review. These people do not participate in the review process but are simply passive observers.

- **Step 2: Preparation** Before the meeting, each reviewer carefully reviews the work package. It is expected that the reviewers read the code and understand its organization and operation before the review meeting.Each reviewer develops the following:
- **List of Questions:** A reviewer prepares a list of questions to be asked, if needed, of the author to clarify issues arising from his or her reading.• **Potential CR:** A reviewer may make a formal request to make a change. These are called change requests rather than defect reports.

  At this stage, since the programmer has not yet made the code public, it is more appropriate to make suggestions to the author to make
  changes, rather than report a defect.
  Though CRs focus on defects in the code, these reports are not included in defect statistics related to the product.

- **Suggested Improvement Opportunities:** The reviewers may suggest how to fix the problems, if there are any, in the code under review. Since reviewers are experts in the subject area of the code, it is not unusual for them to make suggestions for improvements.

Step 3: Examination The examination process consists of the following activities:

- The author makes a presentation of the procedural logic used in the code, the paths denoting major computations, and the dependency of the unit under review on other units.
- The presenter reads the code line by line. The reviewers may raise questions if the code is seen to have defects. However, problems arenot resolved in the meeting. The reviewers may make general suggestions on how to fix the defects, but it is up to the author of the code to take corrective measures after the meeting ends.
- The record keeper documents the change requests and thesuggestions for fixing the problems, if there are any. A CR includes the following details:
-

1. Give a brief description of the issue or action item.
2. Assign a priority level (major or minor) to a CR.
3. Assign a person to follow up the issue. Since a CR documents a potential problem, there is a need for interaction between theauthor of the code and one of the reviewers, possibly the reviewer who made the CR.
4. Set a deadline for addressing a CR

- The moderator ensures that the meeting remains focused on thereview process. The moderator makes sure that the meeting makes progress at a certain rate so that the objective of the meeting is achieved.
- At the end of the meeting, a decision is taken regarding whether or not to call another meeting to further review the code.

- **Step 4: Rework** At the end of the meeting, the recordkeeper produces a summary of the meeting that includes the following information:

- A list of all the CRs, the dates by which those will be fixed, and the names of the persons responsible for validating the CRs
- A list of improvement opportunities
- The minutes of the meeting (optional)

A copy of the report is distributed to all the members of the review group. After the meeting, the author works on the CRs to fix the problems. The author documents the improvements made to the code in the CRs.

- **Step 5: Validation** The CRs are independently validated by themoderator or another person designated for this purpose. The validation process involves checking the modified code as documented in the CRs and ensuring that the suggested improvements have been implemented correctly. The revised and final version of the outcome of the review meeting is distributed to all the group members.

- **Step 6: Exit** Summarizing the review process, it is said to be complete if all of the following actions have been taken:
  - Every line of code in the unit has been inspected.
  - If too many defects are found in a module, the module is once again reviewed after corrections are applied by the author. As a rule ofthumb, if more than 5% of the total lines of code are thought to be contentious, then a second review is scheduled.
  - The author and the reviewers reach a consensus that when corrections have been applied the code will be potentially free of defects.
  - All the CRs are documented and validated by the moderator or someone else. The author's follow-up actions are documented.
  - A summary report of the meeting including the CRs is distributed to all the members of the review group.

DEFECT PREVENTION

The Concept of defect prevention is adopt the concept of defect prevention during code development. In practice, defects are inadvertently introduced by programmers. Those accidents can be reduced by taking preventive measures. It is useful to develop a set of guidelines to construct code for defect minimization as explained in the following. These guidelines focus on incorporating suitable mechanisms into the code:

- Build internal diagnostic tools, also known as *instrumentation code,* into the units. Instrumentation codes are useful in providing information about the internal states of the units. These codes allow programmers to realize built-in tracking and tracing mechanisms. Instrumentation plays a passive role in dynamic unit testing.
- Use standard controls to detect possible occurrences of error conditions. Some examples of error detection in the code are divides by zero and array index out of bounds.
- Ensure that code exists for all return values, some of which may be invalid. Appropriate follow-up actions need to be taken to handle invalid return values.
- Ensure that counter data fields and buffer overflow and underflow are appropriately handled.
- Provide error messages and help texts from a common source so that changes in the text do not cause inconsistency.
- Validate input data, such as the arguments, passed to a function.
- Use assertions to detect impossible conditions, undefined uses of data,and undesirable program behavior. An assertion is a Boolean statement which should never be false or can be false only if an error has occurred.
- • Leave assertions in thecode.
- Fully document the assertions that appear to be unclear.
- After every major computation, reverse-compute the input(s) from the results in the code itself. Then compare the outcome with the actual inputs for correctness
- In systems involving message passing, buffer management is an important internal activity. Incoming messages are stored in an already allocated buffer.
- Develop a timer routine which counts down from a preset time until it either hits zero or isreset.
- Include a loop counter within each loop.
- Define a variable to indicate the branch of decision logic that will be taken.

DYNAMIC UNIT TESTING

- Execution-based unit testing is referred to as dynamic unit testing. In this testing, a program unit is actually executed in isolation, as we commonly understand it.
  However, this execution differs from ordinary execution in the following way:
  1. A unit under test is taken out of its actual executionenvironment.
  2. The actual execution environment is emulated by writing more code(explained later in this section) so that the unit and theemulated environment can be compiled together.
  3. The above compiled aggregate is executed with selected inputs. The outcome of such an execution is collected in a variety of ways, such as straightforward observation on a screen, logging on files, and software instrumentation of the code to reveal run time behavior. The result is compared with the expected outcome. Any difference between the actual and expected outcome implies a failure and the fault is in the code.

An environment for dynamic unit testing is created by emulating the context of the unit under test, as shown in Figure 3.2.

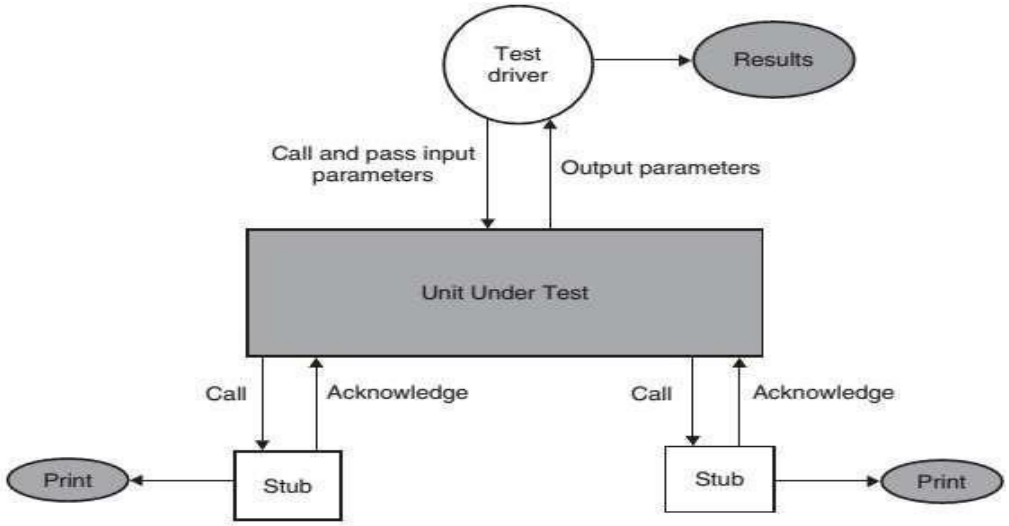


Figure 3.2 Dynamic unit test environment.

An environment for dynamic unit testing is created by emulating the context of the unit under test, as shown in Figure 3.2. The context of a unit test consists of two parts: (i) a caller of the unit and (ii) all the units called bythe unit. The environment of a unit is emulated because the unit is to be tested in isolation and the emulating environment must be a simple one so that any fault found as a result of running the unit can be solely attributed to the unit under test.The caller unit is known as a *test driver*, and all the emulations of the units called by the unit under test are called *stubs*. The test driver andthe stubs are together called *scaffolding*. The functions of a test driver and a stub are explained as follows:

- **Test Driver:** A test driver is a program that invokes the unit under test. The unit under test executes with input values received from the driver and, upon termination, returns a value to the driver. The driver compares the actual outcome, that is, the actual value returned by the unit under test, with the expected outcome from the unit and reports the ensuing test result.

- **Stubs:** A stub is a "dummy subprogram" that replaces a unit that iscalled by the unit under test. Stubs replace the units called by the unit under test. A stub performs two tasks. First, it shows an evidence that the stub was, in fact, called. Such evidence can be shown by merely printing a message. Second, the stub returns a pre computed value to the caller so that the unit under test can continue its execution. The low-level design document provides guidance for the selection ofinput test data that are likely to uncover faults. Selection of test data is broadly based on the following techniques:
- **Control Flow Testing:** The following is an outline of control flowtesting:
  (i) draw a control flow graph from a program unit; (ii) select a fewcontrol flow testing criteria; (iii) identify paths in the control flow graph tosatisfy the selection criteria; (iv) derive path predicate expressions from the selected paths; and (v) by solving the path predicate expression for apath, generate values of the inputs to the program unit that are considered as a test case to exercise the corresponding path. A thorough discussionof
  control flow testing is given in Chapter 4.

- **Data Flow Testing:** The following is an outline of data flow testing: draw a data flow graph from a program unit;
  (i) select a few data flow testing criteria;
  (ii) identify paths in the data flow graph to satisfythe selection criteria;
  (iii) derive path predicate expressions from the selected paths; and
  (iv) by solving the path predicate expression, generate values of the inputs to the program unit that are considered as a test case to exercise the corresponding path. Chapter 5 discusses data flow testing in greater detail.

- **Domain Testing:** In control flow and data flow testing, no specific types of faults are explicitly considered for detection. However, domain testing takes a new approach to fault detection. In this approach, a category of faults called *domain errors* are defined and then test data are selected to catch those faults. It is discussed in detail in Chapter 6.
- **Functional Program Testing:** In functional program testing oneperforms the following steps:
  (i) identify the input and output domains of aprogram;
  (ii) for a given input domain, select some *special* values and compute the expected outcome;
  (iii) for a given output domain, select some *special* values and compute the input values that will cause the unit to produce those output values; and
  (iv) consider various combinations of the input values chosen above. Chapter 9 discusses functional testing.

MUTATION TESTING

- Mutation testing is a technique that focuses on measuring the adequacy of test data (or test cases). The original intention behind mutation testingwas to expose and locate weaknesses in test cases. Thus, mutation testing is a way to measure the quality of test cases, and the actual testing of program units is an added benefit. Mutation testing is not a testing strategy like control flow ordata flow testing. It should be used to supplement traditional unit testing techniques.
  A mutation of a program is a modification of the program created by introducing a single, small, legal syntactic change in the code. Amodified program so obtained is called a *mutant*.
  Some mutants are *equivalent* to the given program, that is, suchmutants always produce the same output as the original program.
  1. The adequacy of an existing test suite is determined to distinguish the given program from its mutants. A given test suite may not be adequateto distinguish all the nonequivalent mutants. As explained above, those nonequivalent mutants that could not be identified by the given test suite are called stubborn mutants.
  2. New test cases are added to the existing test suite to kill the stubborn mutants. The test suite enhancement process iterates until the test suitehas reached a desired level of mutation score.

DEBUGGING

- The programmer, after a program failure, identifies the corresponding fault and fixes it. The process of determining the cause of a failure is known as *debugging*.
  Debugging occurs as a consequence of a test revealing a failure.
- **Brute Force:** The brute-force approach to debugging is preferred bymany programmers. Here, "let the computer find the error" philosophy is used. Print statements are scattered throughout the source code. These print statements provide a crude trace of the way the source code has executed. The availability of a good debugging tool makes these print statements redundant. A dynamic debugger allows the software engineer tonavigate
  by stepping through the code, observe which paths have executed, and observe how values of variables change during the controlled execution.
- **Cause Elimination:** The cause elimination approach can be best described as a process involving *induction* and *deduction* [21]. In the induction part, first, all pertinent data related to the failure are collected , such as what happened and what the symptoms are. Next, the collected data are organized in terms of behavior and symptoms, and their relationship is studied to find a pattern to isolate the causes. A cause hypothesis is devised, and the above data are used to prove or disprove the hypothesis.

In the deduction part, a list of all possible causes is developed in order of their likelihoods, and tests are conducted to eliminate or substantiate each cause in decreasing order of their likelihoods.

- **Backtracking:** In this approach, the programmer starts at a point in the code where a failure was observed and traces back the execution to the point where it occurred. This technique is frequently used by programmers, and this is useful in small programs. However, the probability of tracing back to the fault decreases as the program size increases, because the numberof potential backward paths may become too large.

one more new unit test is created, and additional code is written to passthe new test, but not more, until a new unit test is created. The process is continued until nothing is left to test. The process is illustrated in Figure3.3 and outlined below:

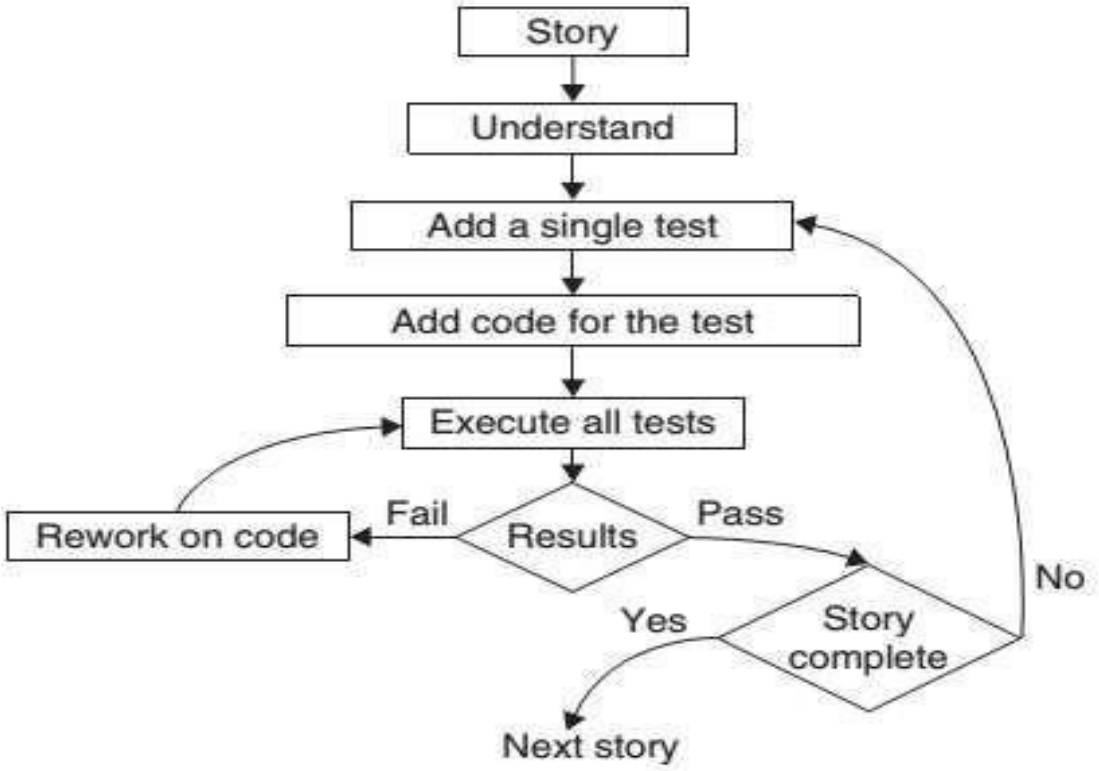


Figure 3.3 Test-first process in XP. (From ref. 24. © 2005 IEEE.)

- **Step 1:** Pick a requirement, that is, a story.
  Step 2: Write a test case that will verify a small part of the story and assign a fail verdict to it.
  Step 3: Write the code that implements a particular part of the story to pass the test.
  Step 4: Execute all tests.

  Step 5: Rework the code, and test the code until all tests pass.
  Step 6: Repeat **steps 2–5** until the story is fully implemented.

  The simple cycle in Figure 3.3 shows that, at the beginning of each cycle, the intention is for all tests to pass except the newly added test case. The new test case is introduced to *drive* the new code development. At the end of the cycle, the programmer executes all the unit tests, ensuring that each one passes and,hence, the planned task of the code still works.

TOOLS FOR UNIT TESTING

- Programmers can benefit from using tools in unit testing by reducingtesting time without sacrificing thoroughness. The well-known tools in everyday life are an editor, a compiler, an operating system, and a debugger. However, in some cases, the real execution environment of a unit may not be available to a programmer while the code is being developed. In such cases, an emulator of the environment is useful in testing and debugging the code. Other kinds of tools that facilitate effective unit testing are as follows.

1. **Code Auditor***:* This tool is used to check the quality of software toensure that it meets some minimum coding standards. It detects violations of programming, naming, and style guidelines.
2. **Bound Checker***:* This tool can check for accidental writes into the instruction areas of memory or to any other memory location outsidethe data storage area of the application.
3. **Documenters***:* These tools read source code and automaticallygenerate descriptions and caller/callee tree diagram or data model from the source code.
4. **Interactive Debuggers***:* These tools assist software developers in implementing different debugging approaches discussed in this chapter
5. **In-Circuit Emulators***:* An in-circuit emulator, commonly known as ICE,is an invaluable software development tool in embedded system design**..**
6. **Memory Leak Detectors***:* These tools test the allocation of memory to an application which requests for memory, but fails to deallocate.
7. ***Static Code (Path) Analyzer****:* These tools identify paths to test,based on the structure of the code such as McCabe's cyclomatic complexity measure
8. ***Software Inspection Support****:* Tools can help schedule groupinspections.
9. **Test Coverage Analyzer***:* These tools measure internal test coverage, often expressed in terms of the control structure of the test object, andreport the coverage metric.
10. **Test Data Generator***:* These tools assist programmers in selecting test data that cause a program to behave in a desired manner.
11. **est Harness:** This class of tools supports the execution of dynamic unit tests by making it almost painless to (i) install the unit under test in a test environment,
12. **Performance Monitors***:* The timing characteristics of software components can be monitored and evaluated by these tools.
13. **Network Analyzers***:* Network operating systems such as software that run on routers, switches, and client/server systems are tested by network analyzers.
14. .**Simulators and Emulators***:* These tools are used to replacethe real software and hardware that are currently not available. Both kinds of tools are used for training, safety, and economy reasons
15. **Traffic Generators***:* Large volumes of data needed to stress the interfaces and the integrated system are generated by trafficgenerators. These produce streams of transactions or data packets.
16. **Version Control***:* A version control system provides functionalitiesto store a sequence of revisions of the software and associated information files under development.

Control Flow Testing

The overall idea of generating test input data for performing control flow testing has been depicted in Figure 4.1. The activities performed,the intermediate results produced by those activities, and programmer preferences in the test generation process are explained below.

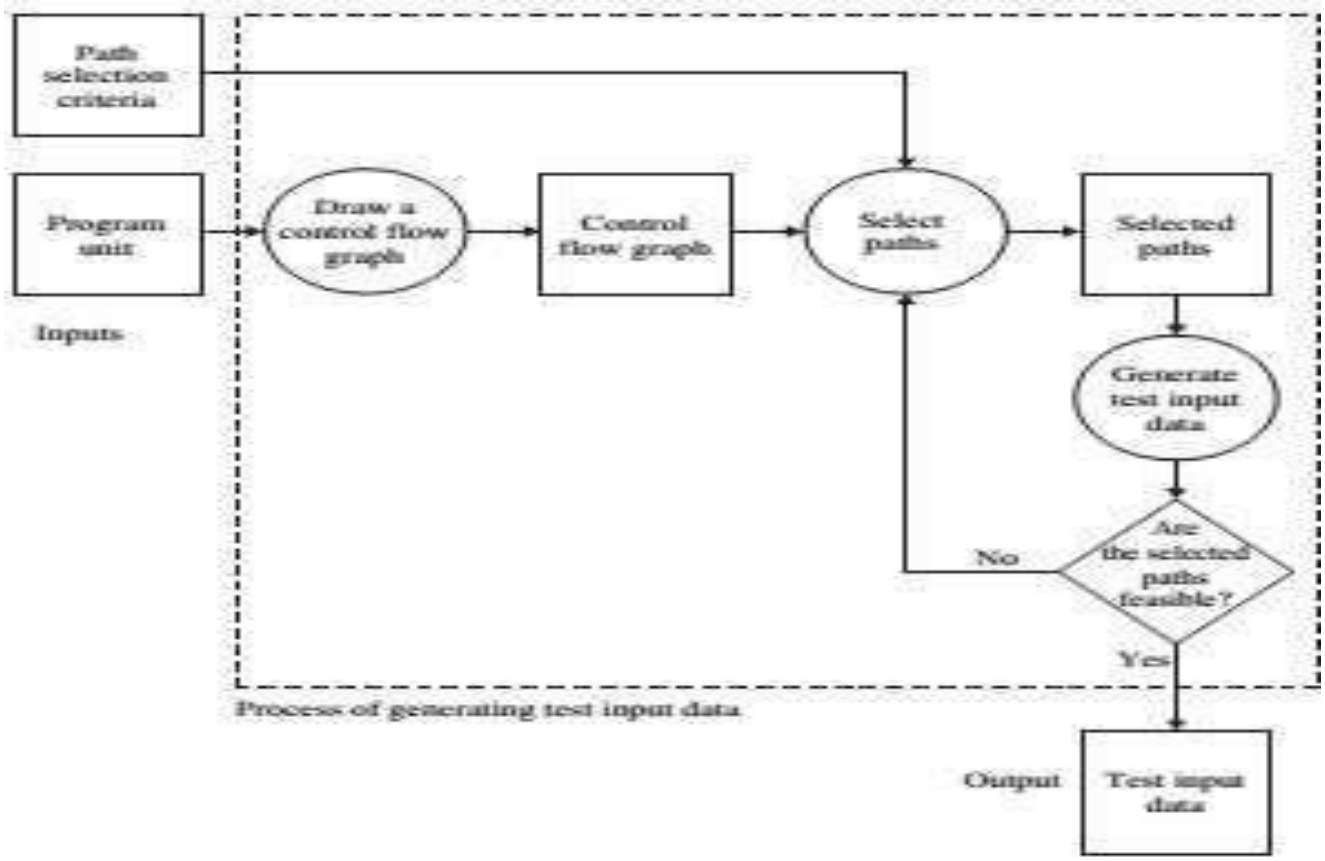


Figure 4.1 Process of generating test input data for control flow testing.

***Inputs*:** The *source code* of a program unit and a set of *path selection criteria* are the inputs to a process for generating test data. In thefollowing, two examples of path selection criteria are given.

- **Example.** Select paths such that every statement is executed at leastonce.
- **Example.** Select paths such that every conditional statement, for example, an if() statement, evaluates to *true* and *false* at least once on different occasions. A conditional statement may evaluate to true in onepath and false in a second path.

- ***Generation of a Control Flow Graph***: A control flow graph (CFG) is a detailed graphical representation of a program unit. The idea behind drawing a CFG is to be able to visualize all the paths in a program unit.The process of drawing a CFG from a program unit will be explained in the following section. If the process of test generation is automated, a compiler can be modified to produce a CFG.
- ***Selection of Paths***: Paths are selected from the CFG to satisfy the path selection criteria, and it is done by considering the structure of the CFG.
- ***Generation of Test Input Data*:** A path can be executed if and only if a certain instance of the inputs to the program unit causes all theconditional statements along the path to evaluate to true or

false as dictated by the control flow. Such a path is called a *feasible* path. Otherwise, the path is said to be *infeasible*. It is essential to identify certain values of the inputs from a given path for the path to execute.

- ***Feasibility Test of a Path***: The idea behind checking the feasibility of a selected path is to meet the path selection criteria. If some chosen paths are found to be infeasible, then new paths are selected to meet the criteria.

CONTROL FLOW GRAPH

- A CFG is a graphical representation of a program unit. Threesymbols are used to construct a CFG, as shown in Figure 4.2. A rectangle represents a sequential computation. A maximal sequential computation can be represented either by a single rectangle or by many rectangles, each corresponding to one statement inthe

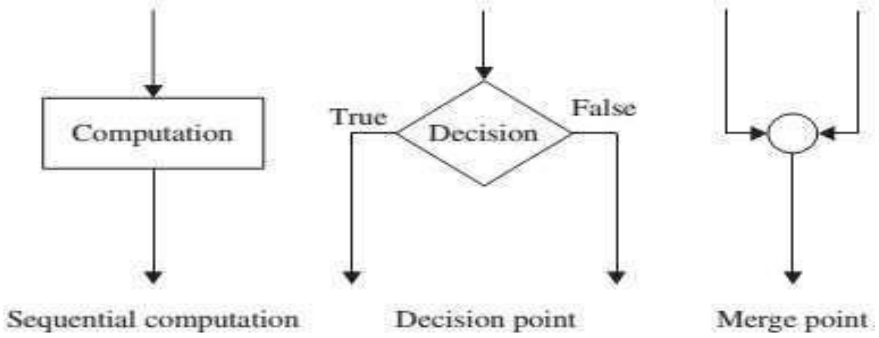


Figure 4.2 Symbols in a CFG.

We label each computation and decision box with a unique integer. The two branches of a decision box are labeled with **T** and **F** to represent the trueand false evaluations, respectively, of the condition within the box.

We will not label a merge node, because one can easily identify the paths in a CFG even without explicitly considering the merge nodes.

We assume that a CFG has exacly one entry node and exactly one exit node. The two branches of a decision node appropriately labeled with true(T) or false(F) .

We are interested in identifying entry-exit paths in a CFG. A path is represented as a sequence of computation and decision nodes from the entry node to the exit node.

PATH SELECTION CRITERIA

- A CFG, such as the one shown in Figure 4.7, can have a large number of different paths. One may be tempted to test the execution of each andevery path in a program unit. For a program unit with a small number of paths, executing all the paths maybe desirable and achievable as well.

  advantages of selecting paths based on defined criteria:

  - All program constructs are exercised at least once. The programmerneeds to observe the outcome of executing each program construct, for example, statements, Boolean conditions, and returns.
  - We do not generate test inputs which execute the same path repeatedly. Executing the same path several times is a waste of resources. However, if each execution of a program path potentially updates the *state* of the system, for example, the database state, then multiple executions of the same path may not be identical.
  - We know the program features that have been tested and those not tested. For example, we may execute an if statement only once so that it evaluates to true. If we do not execute it once again for its false evaluation, we are, at least, aware that we have not observed the outcome of the program with a false evaluation of the if statement.

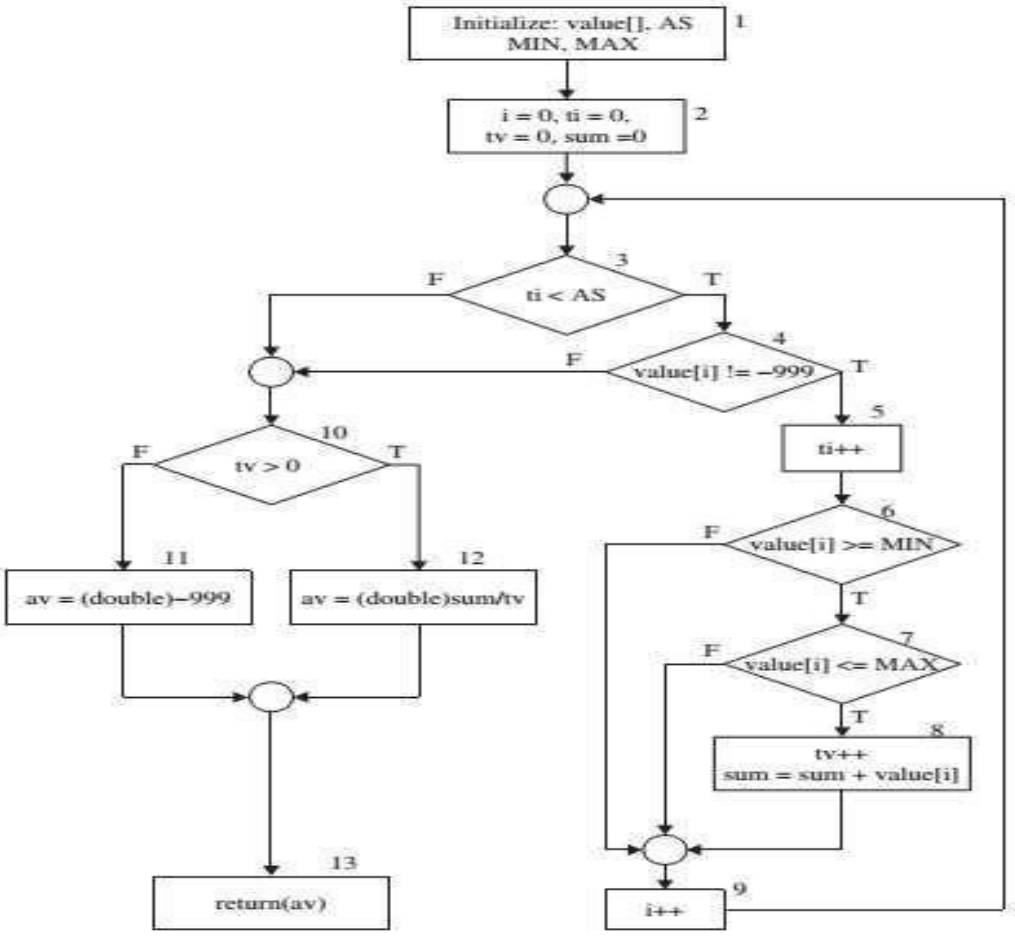


Figure 4.7 A CFG representation of ReturnAverage(). Numbers 1–13 are the nodes.

TABLE 4.1 Examples of Path in CFG of Figure 4.7

Path 1	1-2-3(F)-10(T)-12-13
Path 2	1-2-3(F)-10(F)-11-13
Path 3	1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13
Path 4	1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13

Now we explain the following well-known path selection criteria:

- Select *all* paths.
- Select paths to achieve complete *statement* coverage.
- Select paths to achieve complete *branch* coverage.
- Select paths to achieve *predicate* coverageNow we explain the following well-known path selection criteria:
- Select *all* paths.
- Select paths to achieve complete *statement* coverage.
- Select paths to achieve complete *branch* coverage.

Select paths to achieve *predicate* coverage.

All-Path Coverage Criterion

- If *all* the paths in a CFG are selected, then one can detect all faults, except those due to *missing path* errors. However, a program may contain a large number of paths, or even an infinite number of paths. The small, loop-free openfiles() function shown in Figure 4.3contains more than 25 paths.

  One does not know whether or not a path is feasible at the time of selecting paths, though only eight of all those paths are feasible. If one selects all possible paths in a program, then we say that the *all-path* selection criterion has beensatisfied.

- Let us consider the example of the openfiles() function. This function tries to open the three files file1, file2, and file3. The function returns an integer representing the number of files it has successfully opened.

  A file is said to be successfully opened with "read" access if the file exists. The existence of a file is either "yes" or "no." Thus, the input domain of the functionconsists of eight combinations of the existence of the three files, as shown in Table 4.2.

```
FILE *fptr1, *fptr2, *fptr3; /* These are global variables. */

int openfiles(){
   /*
      This function tries to open files "file1", "file2", and
      "file3" for read access, and returns the number of files
      successfully opened. The file pointers of the opened files
      are put in the global variables.
   */
     int i = 0;
     if(
         ((( fptr1 = fopen("file1", "r")) != NULL) && (i++)
                                                   && (0)) ||
         ((( fptr2 = fopen("file2", "r")) != NULL) && (i++)
                                                   && (0)) ||
         ((( fptr3 = fopen("file3", "r")) != NULL) && (i++))
     );
     return(i);
}
```

Figure 4.3 Function to open three files.

TABLE 4.2 Input Domain of openfiles()

Existence of file1	Existence of file2	Existence of file3
No	No	No
No	No	Yes
No	Yes	No
No	Yes	Yes
Yes	No	No
Yes	No	Yes
Yes	Yes	No
Yes	Yes	Yes

Statement Coverage Criterion

Statement coverage refers to executing individual program statements and observing the outcome. We say that 100% statement coverage has been achieved if *all* the statements have been executed at least once. Complete statement coverage is the *weakest* coverage criterion in program testing. Any test suite that achieves less than statement coverage for new software is considered to be unacceptable.

All program statements are represented in some form in a CFG. Referring to the ReturnAverage() method in Figure 4.6 and its CFG in Figure 4.7, the four assignment statements

i = 0;

ti = 0;

tv = 0;

sum = 0;

have been represented by node **2**. The while statement has been represented as a loop, where the loop control condition

(ti < AS && value[i] != -999)

```
public static double ReturnAverage(int value[],
                            int AS, int MIN, int MAX){
  /*
   Function: ReturnAverage  Computes the  average
   of all  those  numbers in the  input array  in
   the  positive  range  [MIN, MAX]. The  maximum
   size  of the array is AS. But, the  array size
   could be smaller than AS in which case the end
   of input is represented by -999.
  */
     int i, ti, tv, sum;
     double av;
     i = 0; ti = 0; tv = 0; sum = 0;
     while (ti < AS && value[i] != -999) {
          ti++;
          if (value[i] >= MIN && value[i] <= MAX) {
             tv++;
             sum = sum + value[i];
          }
          i++;
     }
     if (tv > 0)
        av = (double)sum/tv;
     else
        av = (double) -999;
     return (av);
}
```

Figure 4.6 Function to compute average of selected integers in an array. This program is an adaptation of "Figure 2. A sample program" in ref. 10. (With permission from the Australian Computer Society.)

Thus, covering a statement in a program means visiting one or morenodes representing the statement, more precisely, selecting a **feasible** entry–exit path that includes the corresponding nodes.

- Since a single entry–exit path includes many nodes, we need to select just a few paths to cover all the nodes of a CFG. Therefore, the basic problem is to select afew feasible paths to cover all the nodes of a CFG in order to achieve the complete statement coverage criterion.

  We follow these rules while selecting paths:

  - Select short paths.
  - Select paths of increasingly longer length. Unfold a loop several times if there is a need.
  - Select arbitrarily long, "complex" paths.

  One can select the two paths shown in Figure 4.4 to achieve complete statement coverage.

Branch Coverage Criterion

- Syntactically, a branch is an outgoing edge from a node. All the rectangle nodes have at most one outgoing branch (edge). The exit node of a CFG does not have an outgoing branch. All the diamond nodes have two outgoing branches. Covering a branch means selecting a path that includes the branch. Complete branch coverage means selecting a number of paths such that every branch is included in at least one path.

TABLE 4.5 Paths for Branch Coverage of CFG of Figure 4.7

BCPath 1	1-2-3(F)-10(F)-11-13
BCPath 2	1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13
BCPath 3	1-2-3(T)-4(F)-10(F)-11-13
BCPath 4	1-2-3(T)-4(T)-5-6(F)-9-3(F)-10(F)-11-13
BCPath 5	1-2-3(T)-4(T)-5-6(T)-7(F)-9-3(F)-10(F)-11-13

Predicate Coverage Criterion

We refer to the partial CFG of Figure 4.9*a* to explain the concept of predicate coverage. OB1, OB2, OB3, and OB are four Boolean variables. The program computes the values of the individual variables OB1, OB2, and OB3— details of their computation are irrelevant to our discussion and have been omitted.

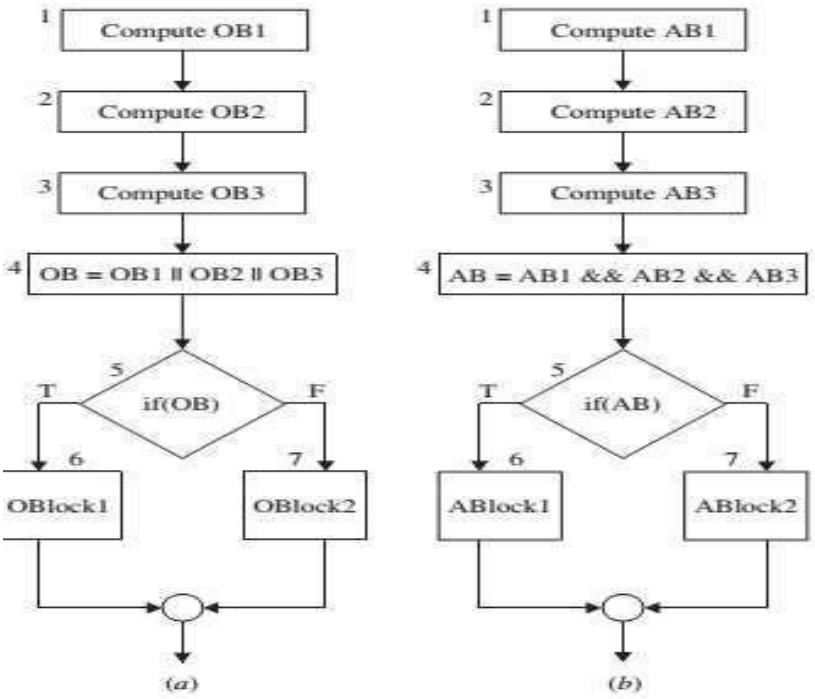


Figure 4.9 Partial CFG with (*a*) OR operation and (*b*) AND operation.

Next, OB is computed as shown in the CFG. The CFG checks the value of OB and executes either OBlock1 or OBlock2 depending on whether OB evaluates to true or false, respectively.

We need to design just two test cases to achieve both statement coverage and branch coverage. We select inputs such that the four Boolean conditions evaluate to the values shown in Table 4.6. The reader may note that we have shown just one way of forcing OB to true. If we select inputs so that these two cases hold, then we do not observe the effect of the computations taking place in nodes **2** and **3**. There may be faults in the computation parts of nodes **2** and **3** such that OB2 and OB3 always evaluate to false.

TABLE 4.6 Two Cases for Complete Statement and Branch Coverage of CFG of Figure 4.9a

Cases	OB1	OB2	OB3	OB
1	T	F	F	T
2	F	F	F	F

GENERATING TEST INPUT

1. Input Vector*:* An input vector is a collection of all data entities read by the routine whose values must be fixed prior to entering the routine. Members of an input vector of a routine can take different forms aslisted below:

- Input arguments to a routine
- Global variables and constants
- Files
- Contents of registers in assembly language programming
- Network connections
- Timers

A file is a complex input element. In one case, mere existence of a file can be considered as an input, whereas in another case, contents of the file are considered to beinputs.

2. Predicate*:* A predicate is a logical function *evaluated at a decision point.*

3. ***Path Predicate****:* A path predicate is the set of predicates associatedwith a path.

4. ***Predicate Interpretation****:* The local variables are not visible outside a function but are used to

- hold intermediate results,
- point to array elements, and
- control loop iterations.

5. Path Predicate Expression*:* An interpreted path predicate is called apath predicate expression. A path predicate expression has the following properties:

- It is void of local variables and is solely composed of elements of theinput vector and possibly a vector of constants.

- It is a set of constraints constructed from the elements of the input vector and possibly a vector of constants.
  - Path forcing input values can be generated by solving the set ofconstraints in a path predicate expression.
  - If the set of constraints cannot be solved, there exist no input which can cause the selected path to execute. In other words, the selected path issaid to be *infeasible.*

UNIT -II

Data flow testing: General idea, Data flow anomaly, Data flow graph, Data flow terms, Data flow testing criteria.
Domain testing: Domain error, Testing for domain errors, Sources of domains, Types of domain errors, ON and OFF points, Test selection criterion.
Integration testing: Concept of integration testing, Different types of interfaces and interface errors, System integration techniques: Incremental, Top down, Bottom up, Sandwich and Big Bang, Test plan for system integration, Off-the-shelf component integration.

Data Flow Testing

- **GENERAL IDEA**

- A program unit, such as a function, accepts input values, performs computations while assigning new values to local and global variables, and, finally, produces output values.
- Therefore, one can imagine a kind of "flow" of data values between variables along a path of program execution. A data value computed in a certain step of program execution is expected to be used in a later step.
- For example, a program may open a file, thereby obtaining a value for a file pointer; in a later step, the file pointer is expected to be used.

- There are two motivations for data flow testing as follows.
- **First,** a memory location corresponding to a program variable is accessed in a desirable way. For example, a memory location may not be read before writing into the location.
- **Second,** it is desirable to verify the correctness of a data value generated for a variable—this is performed by observing that all theuses of the value produce the desired results.

- The above basic idea about data flow testing tells us that a programmer can perform a number of tests on data values, which are collectively known as data flow testing. Data flow testing can be performed at two conceptual levels: *static data flow testing* and *dynamic data flow testing*.

- Static data flow testing is performed to reveal potential defects in programs. The potential program defects are commonly known as *data flow anomaly*.
- On the other hand, dynamic data flow testing involves identifying program paths from source code based on a class of *data flow testing criteria*

DATA FLOW ANOMALY

- An anomaly is a deviant or abnormal way of doing something. For example, it is an abnormal situation to successively assign two values to a variable without using the first value. we explain three types of abnormal situations concerning the generation and use of data values. The three abnormal situations are called **type 1**, **type 2**, and **type 3** anomalies [1]. These anomalies could be manifestations of potential programming errors.

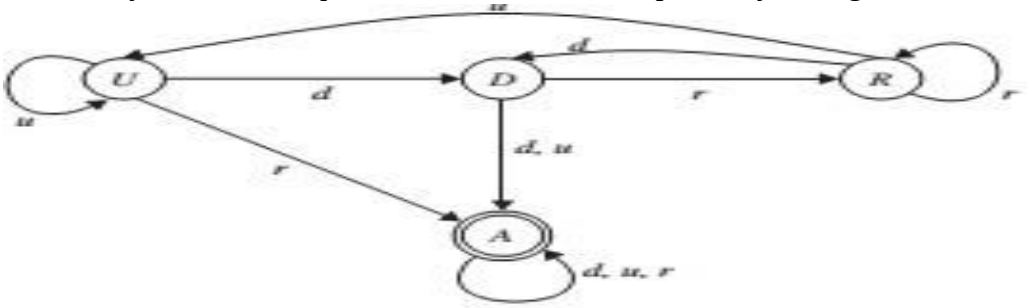
- *Defined and Then Defined Again (Type 1)*:
  Undefined but Referenced (Type 2):
  Defined but Not Referenced (Type 3):

***Defined and Then Defined Again (Type 1)*:** Consider the partial sequence of computations shown in Figure 5.1, where f1(y) and f2(z) denote functions with the inputs *y* and *z*, respectively. We can interpret the two statements
in Figure 5.1 in several ways as follows:
• The computation performed by the first statement is redundant if the second statement performs the intended computation.
• The first statement has a fault. For example, the intended first computation might be w = f1(y).
• The second statement has a fault. For example, the intended second computation might be v = f2(z).
• A fourth kind of fault can be present in the given sequence in the form of a missing statement between the two. For example, v = f3(x) may be the desired statement that should go in between the two given statements.

***Undefined but Referenced (Type 2)*:** A second form of data flow anomaly is to use an undefined variable in a computation, such as $x = x - y - w$,
where the variable *w* has not been *initialized* by the programmer. Here, too, one may argue that though *w* has not been initialized, the programmer intended to use another initialized variable, say *y*, in place of *w*. Whatever may be the real intention of the programmer, there exists an anomaly in the use of the variable *w*, and one must eliminate the anomaly either by initializing *w* or replacing *w* with the intended variable.

***Defined but Not Referenced (Type 3)*:** A third kind of data flow anomaly is to define a variable and then to undefine it without using it in any subsequent computation. For example, consider the statement $x = f(x, y)$ in which a new value is assigned to the variable *x*. If the value of *x* is not used in any subsequent computation, then we should be suspicious of the computation represented by $x = f(x, y)$. Hence, this form of anomaly is called "defined but not referenced."

- Now it is useful to make an association between the **type 1**, **type 2**, and **type 3** anomalies and the state transition diagram shown in Figure 5.2. The **type 1**, **type 2**, and **type 3** anomalies are denoted by the action sequences *dd*, *ur*, and *du*, respectively, in Figure 5.



Legend:

States	Actions
U: Undefined	*d*: Define
D: Defined but not referenced	*r*: Reference
R: Defined and referenced	*u*: Undefine
A: Abnormal	

Figure 5.2 State transition diagram of a program variable. (Fi

Data flow anomaly can be detected by using the idea of *program instrumentation*. Intuitively, program instrumentation means incorporating additional code in a program to monitor its execution status. For example, we can write additional code in a program to monitor the sequence of *states*, namely the *U* , *D*, *R*, and *A*, traversed by a variable. If the state sequence contains the *dd*, *ur*, and *du* subsequence, then a data flow anomaly is said to have occurred. The presence of a data flow anomaly in a program does not necessarily mean that execution of the program will result in a failure. A data flow anomaly simply means that the program *may* fail, and therefore the programmer must investigate the cause of the anomaly. Let us consider the *dd* anomaly shown in Figure 5.1.

If the real intention of the programmer was to perform the second computation and the first computation produces no side effect, then the first computation merely represents a waste of processing power. Thus, the said *dd* anomaly will not lead to program failure. On the other hand, if a statement is missing in between the two statements, then the program can possibly lead to a failure. The programmers must analyze the causes of data flow anomalies and eliminate them.

OVERVIEW OF DYNAMIC DATA FLOW TESTING

In the process of writing code, a programmer manipulates variables in order to achieve the desired computational effect. Variable manipulation occurs in several ways, such as initialization of the variable, assignment of a new value to the variable, computing a value of another variable using the value of the variable, and controlling the flow of program execution.

Rapps and Weyuker [3] convincingly tell us that one should not feel confident that a variable has been *assigned the correct value* if no test case causes the execution of a path from the assignment to a point where the value of the variable is *used*.

In the above motivation for data flow testing,

(i) assignment of a correct value means whether or not a value for the variable has been correctly generated and

(ii) (ii) use of a variable refers to further generation of values for the same or other variables and/or control of flow. A variable can be used in a predicate, that is, a condition, to choose an appropriate flow of control

- we give an outline of performing data flow testing in the following:
  - Draw a data flow graph from a program.
  - Select one or more data flow testing criteria.
  - Identify paths in the data flow graph satisfying the selection criteria.
  - Derive path predicate expressions from the selected paths and solve those expressions to derive test input.

DATA FLOW GRAPH

- **Definiton:** This occurs when a value is moved into the memory location of the variable. Referring to the C function VarTypes() in Figure 5.3, the assignment statement i = x; is an example of definition of the variable *i*.
  Undefinition or Kill: This occurs when the value and the location become unbound. Referring to the C function VarTypes(), the first
  ***Use*:** This occurs when the value is fetched from the memory location of the variable. There are two forms of *uses* of a variable as explained below.
  • **Computation use (c-use):** This directly affects the computation being performed. In a c-use, a potentially new value of another variable or of the same variable is produced. Referring to the C function VarTypes(), the statement
  *iptr = i + x; gives examples of c-use of variables *i* and *x*.
  • **Predicate use (p-use):** This refers to the use of a variable in a predicate controlling the flow of execution. Referring to the C function VarTypes(), the statement
  if (*iptr > y) ... gives examples of p-use of variables *y* and iptr.

- A data flow graph is a directed graph constructed as follows:
  • A sequence of*definitions* and c-uses is associated with each node of the graph.
  • A set of p-uses is associated with each edge of the graph.
  • The entry node has a definition of each parameter and each nonlocal variable which occurs in the subprogram.
  • The exit node has an *undefinition* of each local variable.
  Example: We show the data flow graph in Figure 5.4 for the ReturnAverage() The initial node, node **1**, represents initialization of the input vector < value, AS, MIN, MAX > . Node **2** represents the initialization of the four local variables *i*, ti, tv, and sum in the routine. Next we introduce a NULL node, node **3**, keeping in mind that control will come back to the beginning of the while loop. Node **3** also denotes the fact that program control exits from the while loop at the NULL node. The statement ti++ is represented by node **4**. The predicate associated with edge **(3, 4)** is the condition part of the while loop, namely,

((ti < AS) && (value[i] != -999)).

Example: We show the data flow graph in Figure 5.4 for the ReturnAverage() example discussed in Chapter 4, The initial node, node **1**, represents initialization of the input vector < value, AS, MIN, MAX > . Node **2** represents the initialization of the four local variables *i*, ti, tv, and sum in the routine. Next we introduce a NULL node, node **3**, keeping in mind that control will come back to the beginning of the while loop. Node **3** also denotes the fact that program control exits from the while loop at the NULL node. The statement ti++ is represented by node **4**.

The predicate associated with edge **(3, 4)** is the condition part of the while loop, namely,
((ti < AS) && (value[i] != -999))

Statements tv++ and sum = sum + value[i] are represented by node **5**. Therefore, the condition part of the first if statement forms the predicate associated with edge **(4, 5)**, namely, ((value[i] >= MIN) && (value[i] <= MAX))

The statement i++ is represented by node **6**. The predicate associated with edge **(4, 6)** is the negation of the condition part of the if statement, namely, ((value[i] >= MIN) && (value[i] <= MAX)).

The predicate associated with edge **(5, 6)** is true because there is an unconditional flow of control from node **5** to node **6**. Execution of the while loop terminates when its condition evaluates to false. Therefore, the predicate associated with edge **(3, 7)** is the negation of the predicate associated with edge **(3, 4)**, namely,
~((ti < AS) && (value[i] != -999))
It may be noted that there is no computation performed in a NULL node. Referring to the second if statement, av = (double) − 999 is represented by node **8**, and av = (double) sum/tv is represented by node **9**. Therefore, the predicate associated with edge **(7, 9)** is (tv > 0),
and the predicate associated with edge **(7, 8)** is ~(tv > 0).
Finally, the return(av) statement is represented by node **10**, and the predicate
True is associated with both the edges **(7, 8)** and **(7, 9)**

DATA FLOW TERMS

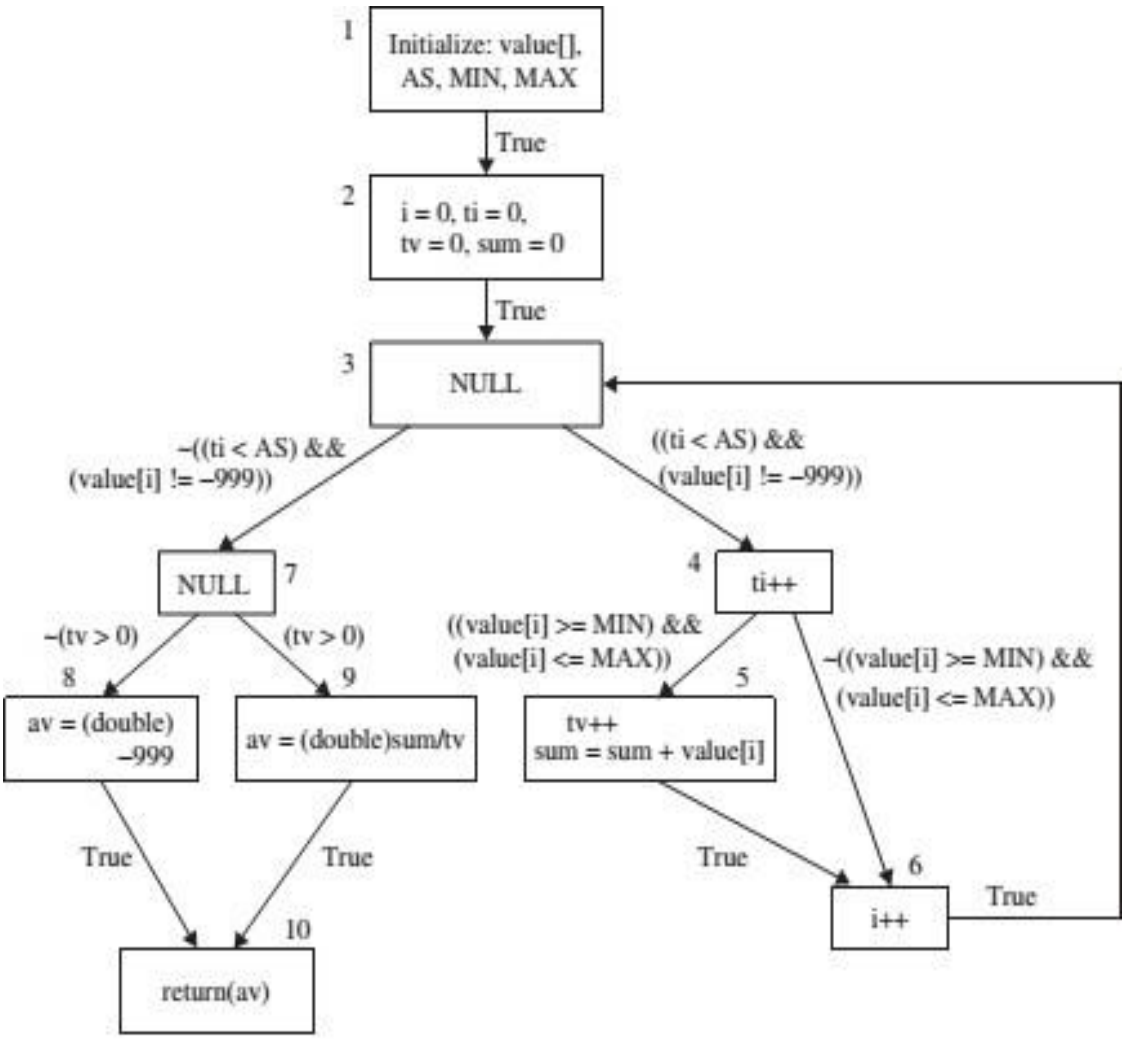


Figure 5.4 Data flow graph of ReturnAverage() example.

- ***Global c-use*:** A c-use of a variable *x* in node *i* is said to be a *global c-use* if *x* has been defined before in a node other than node *i*.
  Example: The c-use of variable tv in node **9** is a global c-use since tv has been defined in nodes **2** and **5** (Figure 5.4).
  Definition Clear Path: A path $(i - n1 - \cdots - nm - j)$, $m \geq 0$, is called a definition clear path (def-clear path) with respect to variable *x*
  • from node *i* to node *j* and
  • from node *i* to edge (*nm*, *j*)
  if *x* has been neither *defined* nor *undefined* in nodes *n*1, . . . ,*nm*. the path **2-3-4-6-3-4-6-3-4-5**, which includes a loop, is
  a def-clear path.
  Example: The paths **2-3-4-5** and **2-3-4-6** are def-clear paths with respect to variable tv from node **2** to **5** and from node **2** to **6**, respectively.
  ***Global Definition*:** A node *i* has a global definition of a variable *x* if node *i* has a definition of *x* and there is a def-clear path with respect to *x* from node *i* to some
  • node containing a global c-use or

• edge containing a p-use ofvariable *x*.

- ***Simple Path*:** A simple path is a path in which all nodes, except possibly the first and the last, are distinct.

  ***Loop-Free Path*:** A loop-free path is a path in which *all* nodes are distinct. *Complete Path*: A complete path is a path from the entry node to the exit node.

  ***Du-path*:** A path ($n1 - n2 - \cdots - nj - nk$) is a definition-use path (du-path) with respect to (w.r.t) variable *x* if node $n1$ has a global definition of *x* and *either*

  • node *nk* has a global c-use of *x* and ($n1 - n2 - \cdots - nj - nk$) is a def-clear simple path w.r.t.*x or*

  • edge (*nj* ,*nk*) has a p-use of *x* and ($n1 - n2 - \cdots - nj$) is a def-clear, loop-free path w.r.t.*x*.

  Example: Considering the global definition and global c-use of variable tv in nodes **2** and **5**, respectively, **2-3-4-5** is a du-path.

  Example: Considering the global definition and p-use of variable tv in nodes **2** and on edge **(7, 9)**, respectively, **2-3-7-9** is a du-path.

DATA FLOW TESTING CRITERIA

- ***All-defs***: For each variable *x* and for each node *i* such that *x* has a global definition in node *i*, select a complete path which includes a def-clear path from node *i* to
  • node *j* having a global c-use of *x* or
  • edge (*j* ,*k*) having a p-use of *x*.
- **Example:** Consider the variable tv, which has global definitions in nodes **2** and **5** (Figure 5.4 and Tables 5.1 and 5.2). First, we consider its global definition in node **2**. We find a global c-use of tv in node **5**, and there exists a def-clear path **2-3-4-5** from node **2** to node **5**. We choose a complete path **1-2-3-4-5-6-3-7-9-10** that includes the def-clear path **2-3-4-5** to satisfy the all-defs criterion.
- ***All-c-uses***: For each variable *x* and for each node *i*, such that *x* has a global definition in node *i*, select complete paths which include def-clear paths from node *i* to *all* nodes *j* such that there is a global c-use of *x* in *j* .
- Let us obtain paths to satisfy the all-c-uses criterion with respect to variable ti.We find two global definitions of ti in nodes **2** and **4**. Corresponding to the global definition in node **2**, there is a global c-use of ti in node **4**. However, corresponding to the global definition in node **4**, there is no global c-use of ti. From the global definition in node **2**, there is a def-clear path to the global c-use in node **4** in the form of **2-3-4**. The reader may note that there are *four* complete paths that include the def-clear path **2 -3-4** as follows:
  1-2-3-4-5-6-3-7-8-10, **1-2-3-4-5-6-3-7-9-10**,
- **1-2-3-4-6-3-7-8-10**, and **1-2-3-4-6-3-7-9-10**.
  One may choose one or more paths from among the four paths above to satisfy the all-c-uses criterion with respect to variable ti.

- ***All-p-uses*:** For each variable *x* and for each node *i* such that *x* has a global definition in node *i*, select complete paths which include def-clear paths from node *i* to *all* edges (*j* ,*k*) such that there is a p-use of *x* on edge (*j* ,*k*).
- Let us obtain paths to satisfy the all-p-uses criterion with respect to variable tv. We find two global definitions of tv in nodes **2** and **5**. Corresponding to the global definition in node **2**, there is a p-use of tv on edges **(7, 8)** and **(7, 9)**. There are def-clear paths from node **2** to edges **(7, 8)** and **(7, 9)**, namely **2-3-7-8**
  and **2-3-7-9**, respectively. Also, there are def-clear paths from node **5** to edges **(7, 8)** and **(7, 9)**, namely, **5-6-3-7-8** and **5-6-3-7-9**, respectively. In the following,
  we identify *four* complete paths that include the above four def-clear paths: **1-2-3-7-8-10**, **1-2-3-7-9-10**,
  1-2-3-4-5-6-3-7-8-10, and **1-2-3-4-5-6-3-7-9-10**.

***All-p-uses/Some-c-uses*:** This criterion is identical to the all-p-uses criterion except when a variable *x* has no p-use. If *x* has no p-use, then this criterion reduces to the some-c-uses criterion explained below.

- *Some-c-uses:* For each variable *x* and for each node *i* such that *x* has a global definition in node *i*, select complete paths which include def-clear paths from node *i* to *some* nodes *j* such that there is a global c-use of *x* in node *j* .
  Let us obtain paths to satisfy the all-p-uses/some-c-uses criterion with respect to variable *i*. We find two global definitions of *i* in nodes **2** and **6**. There is no p-use of *i* in Figure 5.4. Thus, we consider some c-uses of variable *i*. Corresponding

to the global definition of variable *i* in node **2**, there is a global c-use of *i* in node **6**, and there is a def-clear path from node **2** to node **6** in the form of **2-3-4-5-6**. Therefore, to satisfy the all-p-uses/some-c-uses criterion with respect to variable *i*, we select the complete path **1-2-3-4-5-6-3-7-9-10** that includes the def-clear path **2-3-4-5-6**.

- ***All-c-uses/Some-p-uses*:** This criterion is identical to the all-c-uses criterion except when a variable *x* has no global c-use. If *x* has no global c-use,
  then this criterion reduces to the some-p-uses criterion explained below. *Some-p-uses:* For each variable *x* and for each node *i* such that *x* has
  a global definition in node *i*, select complete paths which include def-clear paths from node *i* to *some* edges (*j* ,*k*) such that there is a p-use of *x* on edge (*j* ,*k*).
  Example: Let us obtain paths to satisfy the all-c-uses/some-p-uses criterion with respect to variable AS. We find just one global definition of AS in node **1**. There is no global c-use of AS in Figure 5.4. Thus, we consider some p-uses of AS. Corresponding to the global definition of AS in node **1**, there are p-uses of AS on edges **(3, 7)** and **(3, 4)**, and there are def-clear paths from node **1** to those two edges, namely, **1-2-3-7** and **1-2-3-4**, respectively. There are many complete paths that include those two def-clear paths. One such example path is given as **1-2-3-4-5-6-3-7-9-10**
- ***All-uses*:** This criterion is the conjunction of the all-p-uses criterion and the all-c-uses criterion discussed above.
- ***All-du-paths*:** For each variable *x* and for each node *i* such that *x* has a global definition in node *i*, select complete paths which include *all* du-paths from node *i*
  • to *all* nodes *j* such that there is a global c-use of *x* in *j* and • to *all* edges
  (*j*,*k*) such that there is a p-use of *x* on (*j* ,*k*).

Domain Testing

- **DOMAIN ERROR**

Two fundamental elements of a computer program are *input domain* and *program paths*. The input domain of a program is the set of all input data to the program. A program path is a sequence of instructions from the start of the program to some point of *interest* in the program.

In other words, a program path, or simply *path*, corresponds to some flow of control in the program.

A path is said to be *feasible* if there exists an input data which causes the program to execute the path. Otherwise, the path is said to be *infeasible*.

Howden [1] identified two broad classes of errors, namely, *computation error* and *domain error*,

- ***Computation Error*:** A computation error occurs when a specific input data causes the program to execute the correct, i.e., desired path, but the output value is wrong.
- ***Domain Error*:** A domain error occurs when a specific input data causes the program to execute a *wrong*, that is, undesired, path in the program. Ideally, for each input value, the program assigns a program path to execute; the same program path can be exclusively assigned (i.e., executed) for a subset of the input values.

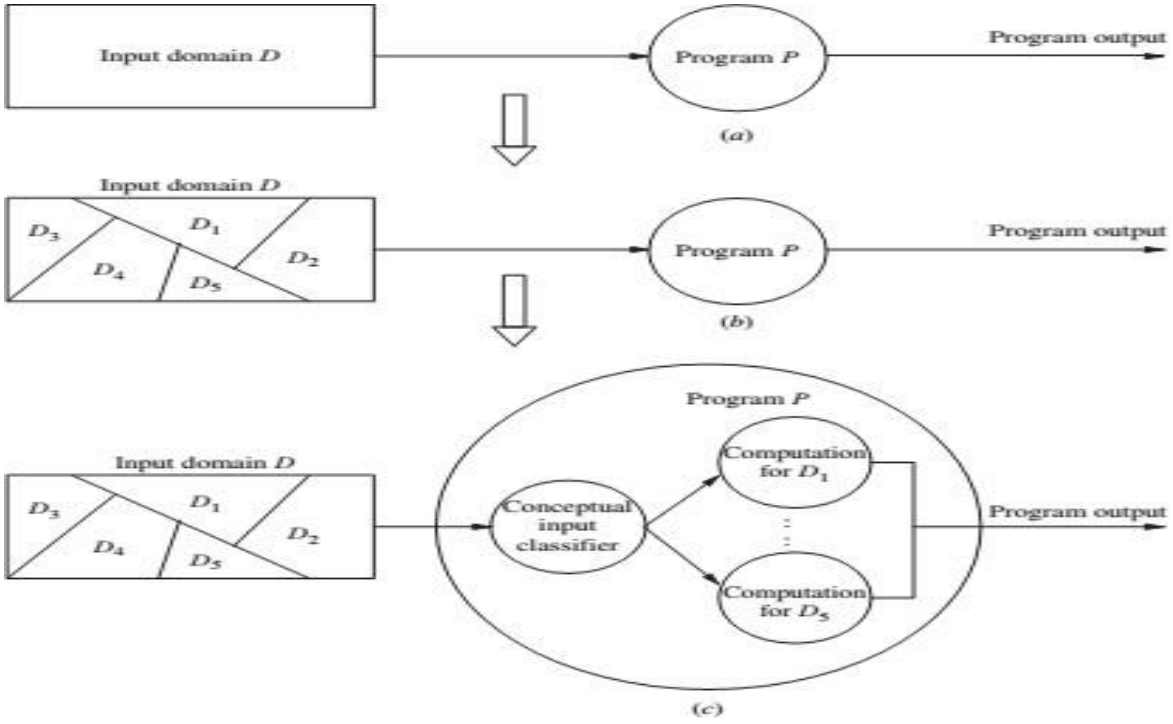


Figure 6.1 Illustration of the concept of program domains.

- A *domain* is a set of input values for which the program performs the same computation for every member of the set.

A program is said to have a *domain error* if the program incorrectly performs input classification. Assuming that adjacent domains perform different computations, a domain error will cause the program to produce

incorrect output.

- Therefore, a program will perform the wrong computation if there are faults in the input classification portion. With the above backdrop, we define the following two terms:

• A*domain* is a set of input values for which the program performs the same computation for every member of the set. We are interested in maximal domains such that the program performs different computations on *adjacent* domains.

- A program is said to have a*domain error* if the program incorrectly performs input classification. Assuming that adjacent domains perform different computations, a domain error will cause the program to produce incorrect output.

TESTING FOR DOMAIN ERRORS

- The idea of *domain testing* was first studied by White and Cohen in 1978 . There is a fundamental difference between flow graph–based testing techniques and domain testing.
  • Select paths from a control flow graph or a data flow graph to satisfy certain *coverage criteria*. To remind the reader, the*control flow* coverage criteria are *statement coverage, branch coverage*, and *predicate coverage*. • Domain testing takes an entirely new approach to fault detection. One defines a category of faults, called *domain errors*, and selects test data to detect those faults.
- We discuss the following concepts in detail:
  • **Sources of Domains:** By means of an example program, we explain how program predicates behave as an input classifier.
  • **Types of Domain Errors:** We explain how minor modifications to program predicates, which can be interpreted as programming defects, can lead to domain errors.
  • **Selecting Test Data to Reveal Domain Errors:** A test selection criterion is explained to pick input values. The test data so chosen reveal the specific kinds of domain errors.

SOURCES OF DOMAINS

- Domains can be identified from both specifications and programs.

```
int codedomain(int x, int y){
  int c, d, k
  c = x + y;
  if (c > 5) d = c - x/2;
  else       d = c + x/2;
  if (d >= c + 2) k = x + d/2;
  else            k = y + d/4;
  return(k);
}
```

Figure 6.2 A function to explain program domains.

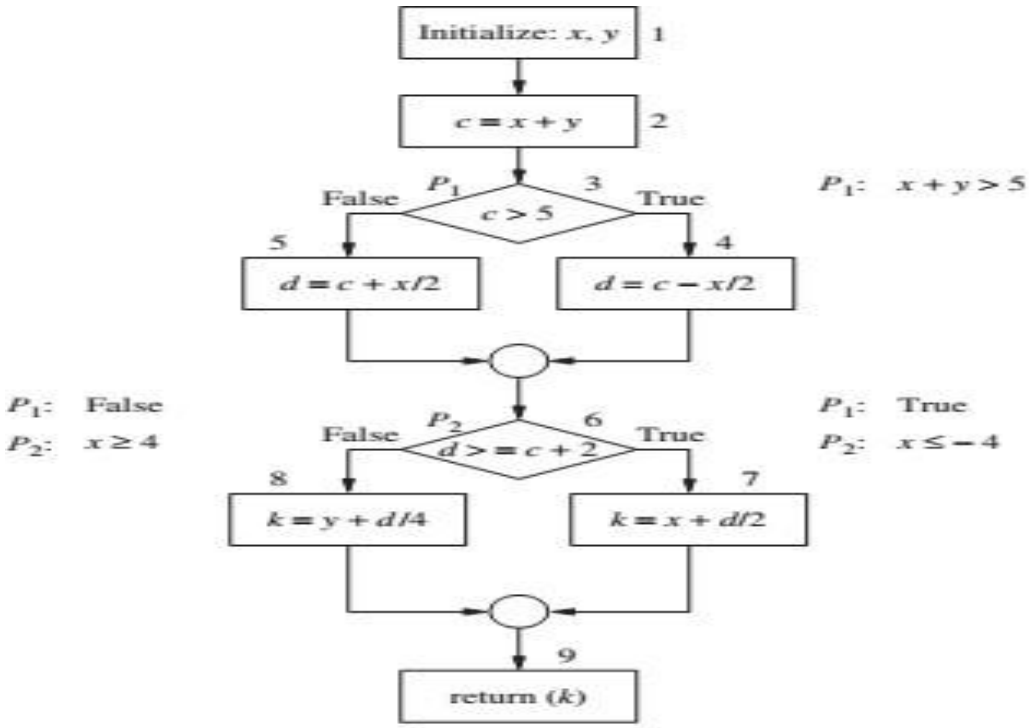


Figure 6.3 Control flow graph representation of the function in Figure 6.2.

TABLE 6.1 Two Interpretations of Second if() Statement in Figure 6.2

Evaluation of P_1	Interpretation of P_2
True	$x \leq -4$
False	$x \geq 4$

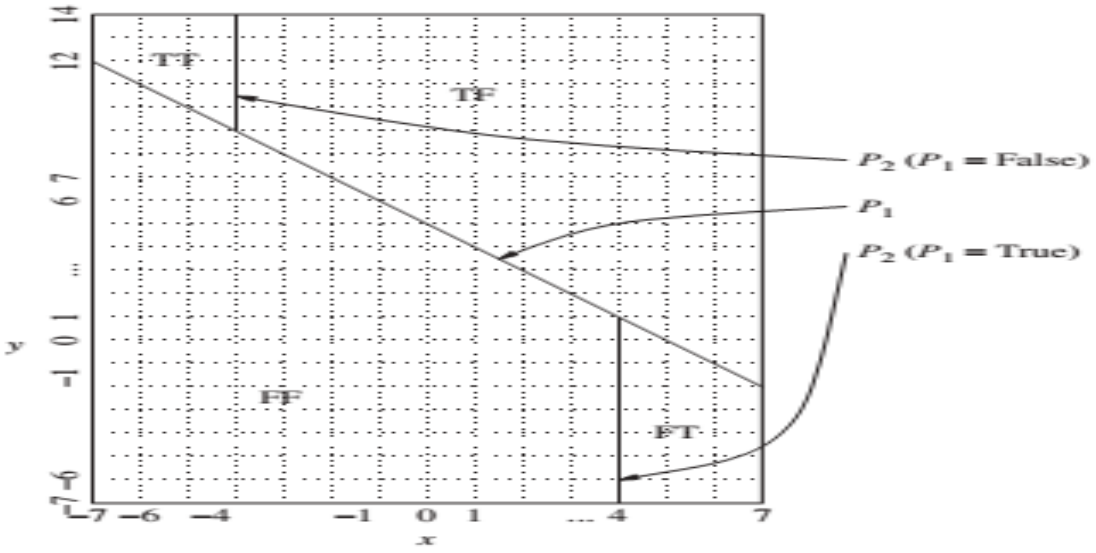


Figure 6.4 Domains obtained from interpreted predicates in Figure 6.3.

TYPES OF DOMAIN ERRORS

- The reader may recall the following properties of a domain:
  - A domain is a set of values for which the program performs identical computations.
  - A domain can be represented by a set of predicates. Individual elements of the domain satisfy the predicates of the domain.

  A domain is defined, from a geometric perspective, by a set of constraints called *boundary inequalities.*
- ***Closed Boundary*:** A boundary is said to be closed if the points on the boundary are included in the domain of interest.
  Example: Consider the domain **TT** in Figure 6.4 and its boundary defined by the inequality $P2 : x \leq -4$
- ***Open Boundary***: A boundary is said to be open if the points on the boundary do not belong to the domain of interest.
  Example: Consider the domain **TT** in Figure 6.4 and its boundary defined by the inequality $P1 : x + y > 5$
  ***Closed Domain*:** A domain is said to be closed if all of its boundaries are closed.
  Open Domain: A domain is said to be open if some of its boundaries are open. *Extreme Point*: An extreme point is a point where two or more boundaries cross.
  Adjacent Domains: Two domains are said to be adjacent if they have a boundary inequality in common.
- A program path will have a *domain error* if there is incorrect formulation of a path predicate. After an interpretation of an incorrect path predicate, the path predicate expression causes a boundary segment to
  • be shifted from its correct position or • have an incorrect relational operator.
- A domain error can be caused by
  • an incorrectly specified predicate or
  • an incorrect assignment which affects a variable used in the predicate. Now we discuss different types of domain errors:
- **Closure Error**
- **Shifted-Boundary Error**
- **Tilted-Boundary Error**
- ***Closure Error*:** A closure error occurs if a boundary is open when the intention is to have a closed boundary, or vice versa. Some examples of closure error are:
  • The relational operator $\leq$ is implemented as $<$. • The relational operator $<$ is implemented as $\leq$.
  Shifted-Boundary Error: A shifted-boundary error occurs when the implemented boundary is parallel to the intended boundary. E**xample:** Consider the boundary defined by the following predicate (Figure 6.4): $P1 : x + y > 5$ If the programmer's intention was to define a boundary represented by the predicate $P\ 1 : x + y > 4$ then the boundary defined by $P\ 1$ isE parallel, but not identical, to the boundary defined by $P1$.

- ***Tilted-Boundary Error***: If the constant coefficients of the variables in a predicate defining a boundary take up wrong values, then the tilted-boundary error occurs.
  Example: Consider the boundary defined by the following predicate (Figure 6.4):
  $P1 : x + y > 5$
  If the programmer's intention was to define a boundary represented by the predicate
  $P\ 1 : x + 0.5y > 5$
  then the boundary defined by $P\ 1$ is tilted with respect to the boundary defined by $P1$.

ON AND OFF POINTS

- *Data points on or near a boundary are most sensitive to domain errors*. In this observation, by *sensitive* we mean data points falling in the wrong domains. Therefore, the objective is to identify the data points that are most sensitive to domain errors so that errors can be detected by executing the program with those input values.
  ***ON Point*:** Given a boundary, an **ON** point is a point *on* the boundary or "very close" to the boundary.
  This definition suggests that we can choose an **ON** point in two ways. Therefore, one must know when to choose an **ON** point in which way: • If a point can be chosen to lie exactly on the boundary, then choose such a point as an **ON** point. If the boundary inequality leads to an *exact* solution, choose such an exact solution as an **ON** point.
  • If a boundary inequality leads to an *approximate* solution, choose a point very close to the boundary.
  Example: Consider the following boundary inequality. This inequality is not related to our running example of Figure 6.4.
  $P\text{ON}1 : x + 7y \geq 6$ For $x = -1$, the predicate P ON1 leads to an exact solution of $y = 1$. Therefore, the point $(-1, 1)$ lies on the boundary.

- ***OFF Point*:** An **OFF** point of a boundary lies *away* from the boundary. However, while choosing an **OFF** point, we must consider whether a boundary is *open* or *closed* with respect to a domain:
  • If the domain is *open* with respect to the boundary, then an **OFF** point of that boundary is an *interior* point inside the domain within an -distance from the boundary.
  • If the domain is *closed* with respect to the boundary, then an **OFF** point of that boundary is an *exterior* point outside the boundary within an -distance. The symbol denotes an arbitrarily small value. **Example:** Consider a domain D 1 with a *closed* boundary as follows: $P\text{OFF}1 : x + 7y \geq 6$
  Since the boundary is closed, an OFF point lies outside the domain; this means that the boundary inequality is *not* satisfied. Note that the point $(-1, 1)$ lies exactly on the boundary and it belongs to the domain.
- The above ideas of ON and OFF points lead to the following conclusions: • While testing a *closed* boundary, the ON points are in the domain under test, whereas the OFF points are in an adjacent domain.
  • While testing an *open* boundary, the ON points are in an adjacent domain, whereas the OFF points are in the domain being tested.

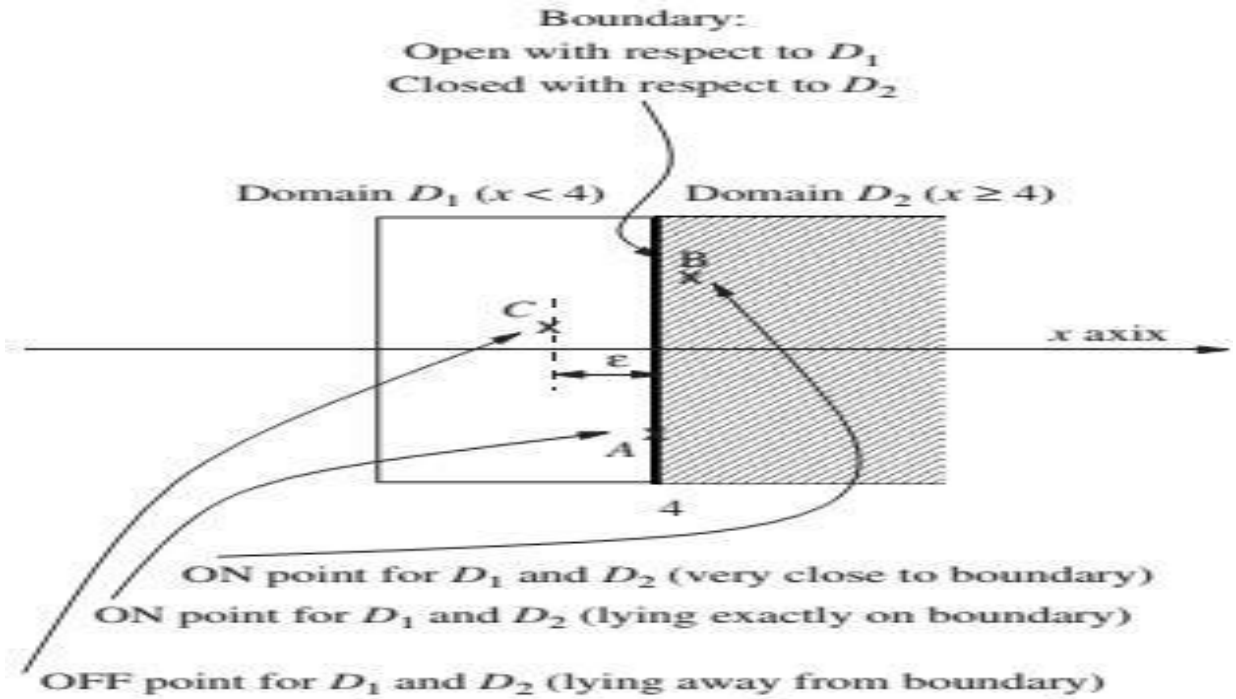


Figure 6.6 ON and OFF points.

TEST SELECTION CRITERION

- ***Test Selection Criterion*:** For each domain and for each boundary, select three points *A*, *C* , and *B* in an ON–OFF–ON sequence. This criterion generates test data that reveal domain errors. Specifically, the following kinds of errors are considered:
  1. Closed inequality boundary
  a. Boundary shift resulting in a reduced domain b.
  Boundary shift resulting in an enlarged domain c.
  Boundary tilt
  d. Closure error
  2. Open inequality boundary
  a. Boundary shift resulting in a reduced domain b.
  Boundary shift resulting in an enlarged domain c.
  Boundary tilt
  d. Closure error
  3. Equality boundary

Closed inequality boundary

- *(***CInequality) Boundary Shift Resulting in Reduced Domain***:* The boundary between *losed* the two domains *D* 1 and *D* 2 has shifted by a certain amount

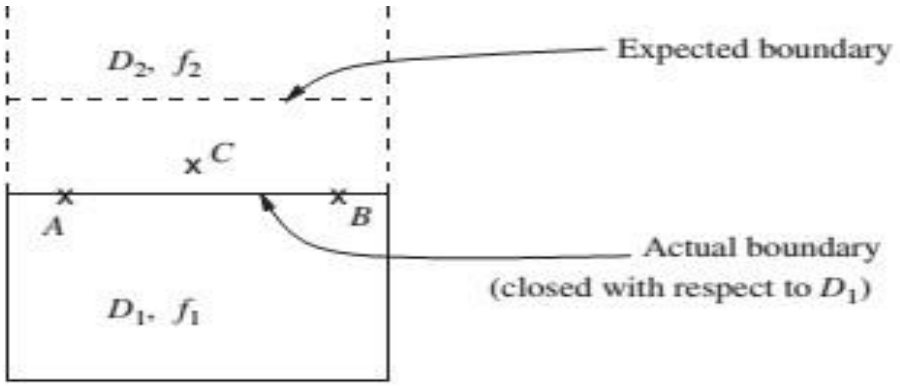


Figure 6.7 Boundary shift resulting in reduced domain (closed inequality).

TABLE 6.2 Detection of Boundary Shift Resulting in Reduced Domain (Closed Inequality)

Test Data	Actual Output	Expected Output	Fault Detected
A	$f_1(A)$	$f_1(A)$	No
B	$f_1(B)$	$f_1(B)$	No
C	$f_2(C)$	$f_1(C)$	Yes

Closed inequality boundary

- **(Closed Inequality) Boundary Shift Resulting in Enlarged Domain***:* To detect this fault, we use Figure 6.8, where the boundary between the two domains *D* 1 and *D* 2 has shifted from its expected position such that the size of the domain *D* 1 under consideration has enlarged.

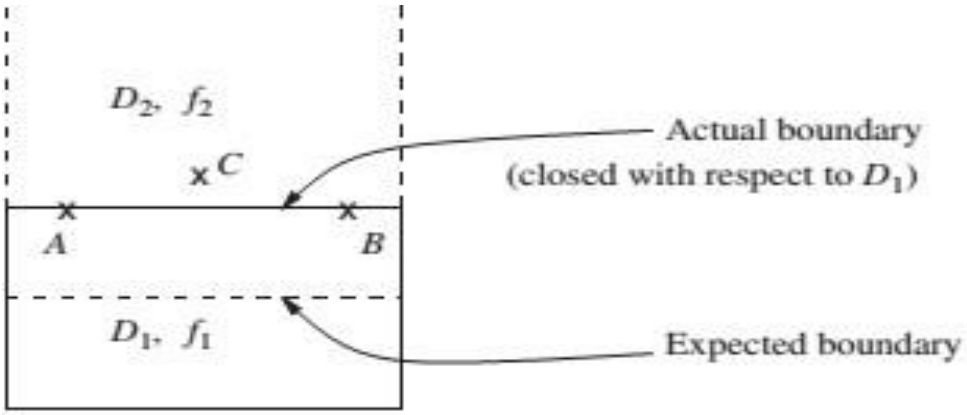


Figure 6.8 Boundary shift resulting in enlarged domain (closed inequality).

TABLE 6.3 Detection of Boundary Shift Resulting in Enlarged Domain (Closed Inequality)

Test Data	Actual Output	Expected Output	Fault Detected
A	$f_1(A)$	$f_2(A)$	Yes
B	$f_1(B)$	$f_2(B)$	Yes
C	$f_2(C)$	$f_2(C)$	No

Closed inequality boundary

- ***(Closed Inequality) Boundary Tilt:*** In Figure 6.9 the boundary between the two domains *D* 1 and *D* 2 has tilted by an appreciable amount. The boundary between the two domains is closed with respect to domain *D*1.

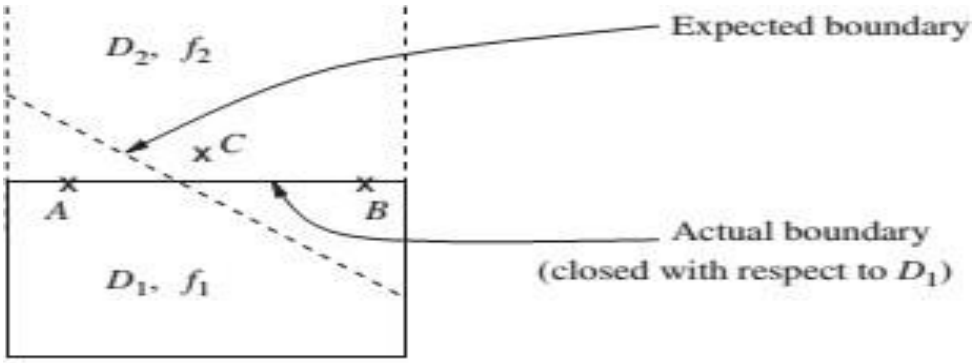


Figure 6.9 Tilted boundary (closed inequality).

TABLE 6.4 Detection of Boundary Tilt (Closed Inequality)

Test Data	Actual Output	Expected Output	Fault Detected
A	$f_1(A)$	$f_1(A)$	No
B	$f_1(B)$	$f_2(B)$	Yes
C	$f_2(C)$	$f_2(C)$	No

Closed inequality boundary

- ***(Closed Inequality) Closure Error****:* The expected boundary between the two domains in Figure 6.10 is closed with respect to domain *D* 1. However, in an actual implementation, it is open with respect to *D* 1, resulting in a closure error. The boundary between the two domains belongs to domain *D*2.

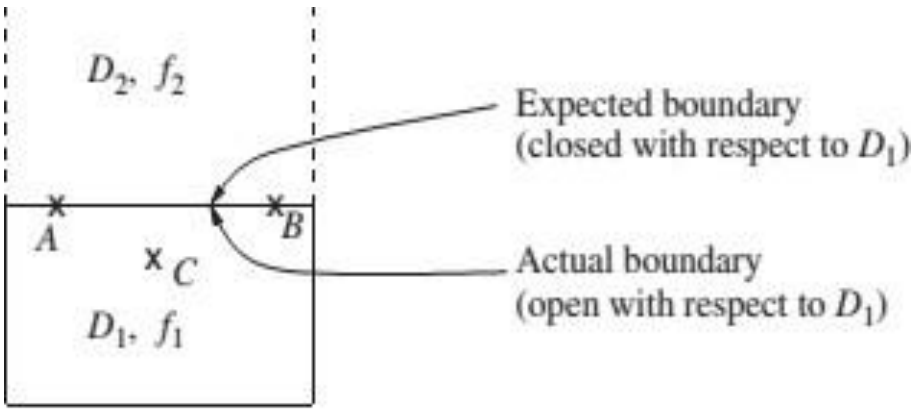


Figure 6.10 Closure error (closed inequality).

TABLE 6.5 Detection of Closure Error (Closed Inequality)

Test Data	Actual Output	Expected Output	Fault Detected
A	$f_2(A)$	$f_1(A)$	Yes
B	$f_2(B)$	$f_1(B)$	Yes
C	$f_1(C)$	$f_1(C)$	No

Open inequality boundary

- **(Open Inequality) Boundary Shift Resulting in Reduced Domain:** To explain the detection of this type of error, we use Figure 6.11, where the boundary between the two domains *D* 1 and *D* 2 has shifted by a certain amount. The boundary between the two domains is open with respect to domain *D* 1.

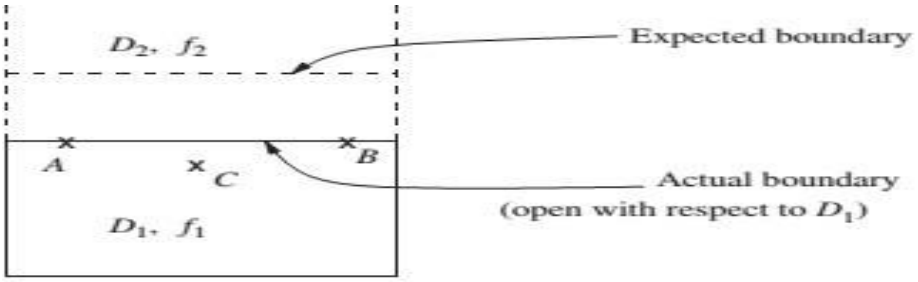


Figure 6.11 Boundary shift resulting in reduced domain (open inequality).

TABLE 6.6 Detection of Boundary Shift Resulting in Reduced Domain (Open Inequality)

Test Data	Actual Output	Expected Output	Fault Detected
A	$f_2(A)$	$f_1(A)$	Yes
B	$f_2(B)$	$f_1(B)$	Yes
C	$f_1(C)$	$f_1(C)$	No

Open inequality boundary

- ***(Open Inequality) Boundary Shift Resulting in Enlarged Domain****:* We use Figure 6.12 to explain the detection of this kind of errors. The boundary between the two domains *D* 1 and *D* 2 has shifted to enlarge the size of the domain *D* 1 under consideration. The boundary between the two domains is open with respect to domain *D* 1.

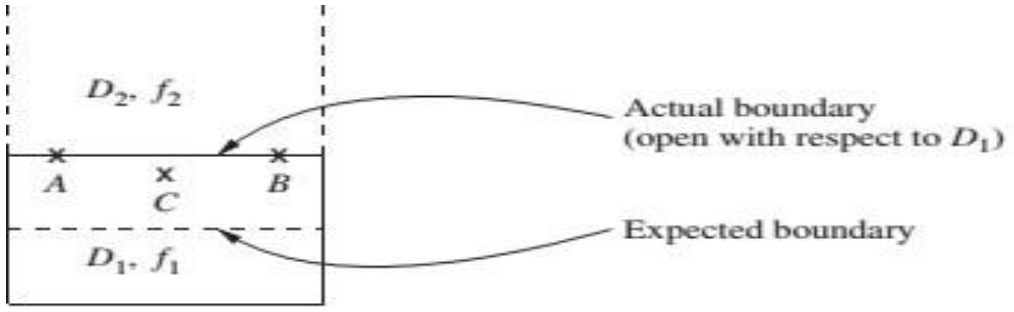


Figure 6.12 Boundary shift resulting in enlarged domain (open inequality).

TABLE 6.7 Detection of Boundary Shift Resulting in Enlarged Domain (Open Inequality)

Test Data	Actual Output	Expected Output	Fault Detected
A	$f_2(A)$	$f_2(A)$	No
B	$f_2(B)$	$f_2(B)$	No
C	$f_1(C)$	$f_2(C)$	Yes

Open inequality boundary

- ***Open Inequality) Boundary Tilt:*** We explain the boundary tilt fault by referring to Figure 6.13, where the boundary between the two domains *D* 1 and *D* 2 has tilted. Once again, we do not know the exact position of the expected boundary. The boundary between the two domains is open with respect to domain *D* 1.

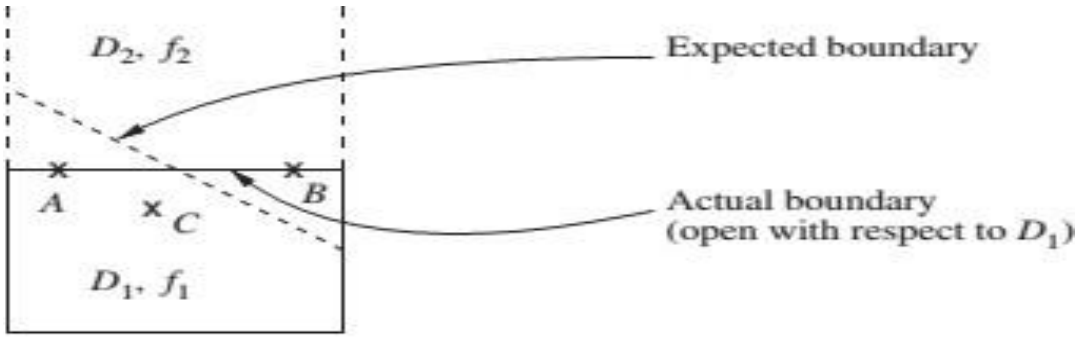


Figure 6.13 Tilted boundary (open inequality).

TABLE 6.8 Detection of Boundary Tilt (Open Inequality)

Test Data	Actual Output	Expected Output	Fault Detected
A	$f_2(A)$	$f_1(A)$	Yes
B	$f_2(B)$	$f_2(B)$	No
C	$f_1(C)$	$f_1(C)$	No

Open inequality boundary

- **(Open Inequality) Closure Error:** Detection of this kind of fault is explained by using the two domains of Figure 6.14, where the expected boundary between the two domains is open with respect to domain *D* 1. However, in an actual implementation it is closed with respect to *D* 1, resulting in a closure error.

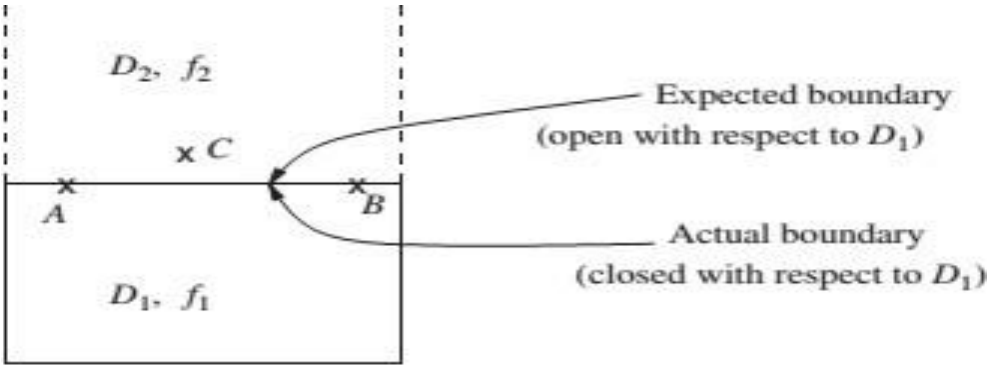


Figure 6.14 Closure error (open inequality).

TABLE 6.9 Detection of Closure Error (Open Inequality)

Test Data	Actual Output	Expected Output	Fault Detected
A	$f_1(A)$	$f_2(A)$	Yes
B	$f_1(B)$	$f_2(B)$	Yes
C	$f_2(C)$	$f_2(C)$	No

. Equality boundary

- a test selection criterion was defined to choose test points to reveal domain errors. Specifically, the selection riterion is as follows: For each domain and for each boundary, select three points *A*, *C* , and *B* in an **ON–OFF–ON** sequence.

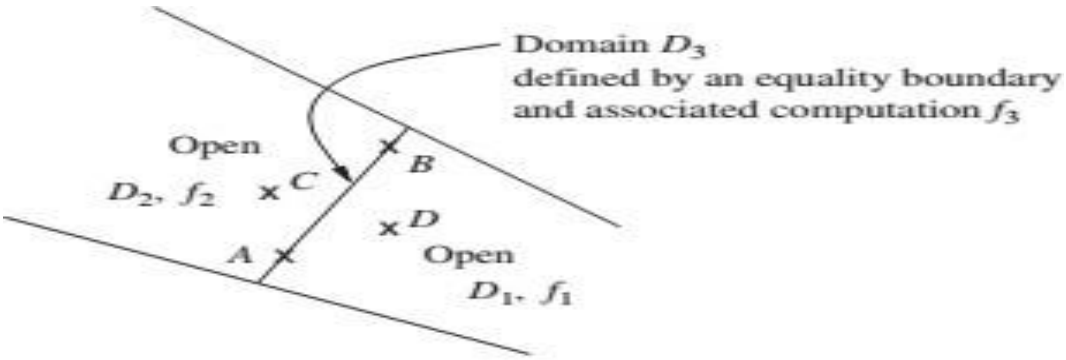


Figure 6.15 Equality border.

CONCEPT OF INTEGRATION TESTING

- A software module, or component, is a self-contained element of a system. Modules have well-defined interfaces with other modules. A module can be a subroutine, function, procedure, class, or collection of those basic elements put together to deliver a higher level service.
- A system is a collection of modules interconnected in a certain way to accomplish a tangible objective. A subsystem is an interim system that is not fully integrated with all the modules. It is also known as a subassembly.

INTEGRATION TESTING

- The objective of system integration is to build a "working" version of the system by
- (i) putting the modules together in an incremental manner and
- (ii) ensuring that the additional modules work as expected without disturbing the functionalities of the modules already put together

Integration testing is said to be complete when the system is fully integrated together, all the test cases have been executed, all the severe and moderate defects found have been fixed, and the system is retest.

INTERFACE ERRORS

- Modularization is an important principle in software design, and modules are interfaced with other modules to realize the system's functional requirements. An interface between two modules allows one module to access the service provided by the other. It implements a mechanism for passing control and data between modules. Three common paradigms for interfacing modules are as follows:
  • **Procedure Call Interface:** A procedure in one module calls a procedure in another module. The caller passes on control to the called module. The caller can pass data to the called procedure, and the called procedure can pass data to the caller while returning control back to the caller.
  • **Shared Memory Interface:** A block of memory is shared between two modules. The memory block may be allocated by one of the two modules or a third module. Data are written into the memory block by one module and are read from the block by the other.
  • **Message Passing Interface:** One module prepares a message by initializing the fields of a data structu e and sending the message to another module.This form of module interaction is common in client–server-based systemsand web-based systems.

- They found that of all errors that required a fix within one module, more than half were caused by interface errors. Perry and Evangelist have categorized interface errors as follows:
  1. *Construction*: Some programming languages, such as C, generally separate the interface specification from the implementation code.
  2. *Inadequate Functionality*: These are errors caused by implicit assumptions in one part of a system that another part of the system would perform a function.
  3. *Location of Functionality*: Disagreement on or misunderstanding about the location of a functional capability within the software leads to this
  sort of error.
  4. *Changes in Functionality*: Changing one module without correctly adjusting for that change in other related modules affects the functionality of the program.
  5. *Added Functionality*: A completely new functional module, or capability, was added as a system modification. Any added functionality after the module is checked in to the version control system with ut a CR is considered to be an error.
  6. *Misuse of Interface*: One module makes an error in using the interface of a called module. This is lik ly to occur in a procedure–call interface. Interface misuse can take the form of wrong parameter type, wro g parameter order, or wrong number of parameters passed.
  7. *Misunderstanding of Interface*: A calling module may misunderstand the
  interface specification of a called module. The called module may assume that some parameters passed to it satisfy a certain condition, whereas the caller does not ensure that the condition holds
  8. *Data Structure Alteration*: These are similar in nature to the functionality problems discussed above, but they are likely to occur at the detailed esign level. The problem arises when the size of a data structure is inadequate or it fails to contain a sufficient number of information fields.
  9. *Inadequate Error Processing*: A called module may return an error code to the calling module. However, the calling module may fail to handle the error properly.
  10. *Additions to Error Processing*: These errors are caused by changes to other modules which dictated changes in a module error handling. In
  this case either necessary functionality is missing from the current error processing that would help trace errors or current techniques of error processing require modification.
  11. *Inadequate Postprocessing*: These errors are caused by a general failure o release resources no longer required, for example, failure to deallocate memory.
  12. *Inadequate Interface Support*: The actual functionality supplied was inadequate to support the specified capabilities of the interface. For example, amodule passes a temperature value in Celsius to a module which interprets the value in Fahrenheit.

13. ***Initialization/Value Errors***: A failure to initialize, or assign, the appropriate value to a variable data structure leads to this kind of error. Problemsof this kind are usually caused by simple oversight.

14. ***Violation of Data Constraints***: A specified relationship among data items was not supported by the implementation. This can happen due to incomplete detailed design specifications.

15. ***Timing/Performance Problems***: These errors were caused by inadequate synchronization among communicating processes. A race condition is an example of these kinds of error.

16. ***Coordination of Changes***: These errors are caused by a failure to communicate changes to one software module to those responsible for other interrelated modules.

17. ***Hardware/Software Interfaces*****:** These errors arise from inadequate software handling of hardware devices.

Interface errors cannot be detected by performing unit testing on modules since unit testing causes computation to happen within a module, whereas interactions are required to happen between modules for interface errors to be detected.

SYSTEM INTEGRATION TECHNIQUES

- The common approaches to performing system integration are as follows:
  - Incremental
  - Top down
  - Bottom up
  - Sandwich
  - Big bang

Incremental

- A *software image* is a compiled software binary.

  A *build* is an interim software image for internal testing within the organization.

  Constructing a software image involves the following activities:
  - Gathering the latest unit tested, authorized versions of modules
  - Compiling the source code of those modules
  - Checking in the compiled code to the repository
  - Linking the compiled modules into subassemblies • Verifying that the subassemblies are correct
  - Exercising version control

Top Down

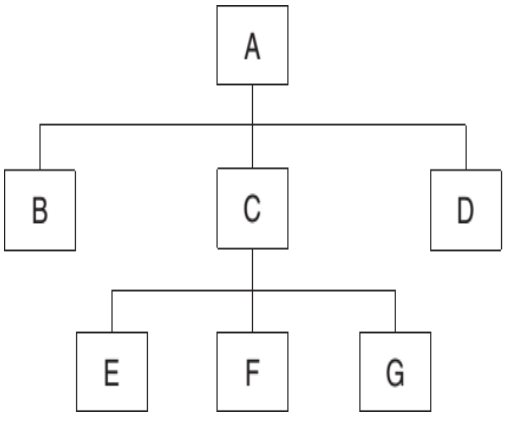


Figure 7.1 Module hierarchy with three levels and seven modules.

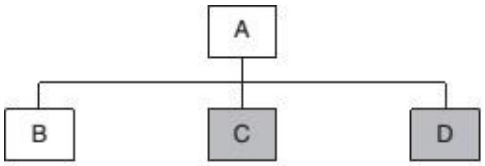


Figure 7.2 Top-down integration of modules A and B.

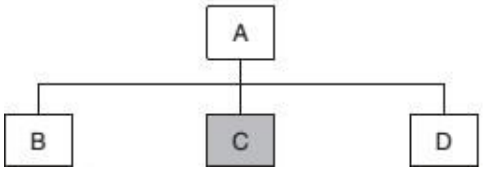


Figure 7.3 Top-down integration of modules A, B, and D.

- The integration of modules A and B by using stubs C and D (represented by grey boxes) is shown in Figu re 7.2. Interactions between modules A and B is severely constrained by the dummy nature of C and D. The interactions between Aand B are concrete, and, as a consequence, more tests are performed after additional modules are integrated. Next, as shown in Figure 7.3, stub D has been replaced with its actual instance D We perform two kinds of tests: first, test the interface between A and D; second, perform regression tests to look for interface defects between A and B in the presence of module D. Stub C has been replaced w th theactual module C, and new stubs E, F, and G have been added to the integrated system (Figure 7.4). W e perform tests as follows: First, test the interface between Aand C; second, test the combined modules A, B, and D in the presence of C
(Figure 7.4). The rest of the integration process is depicted in Figures 7.5 and 7.6 to obtain the final system of Figure 7.7.

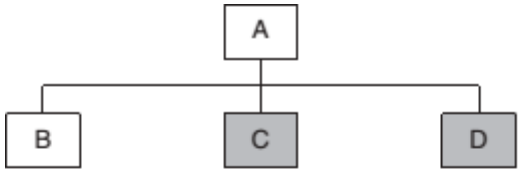


Figure 7.2 Top-down integration of modules A and B.

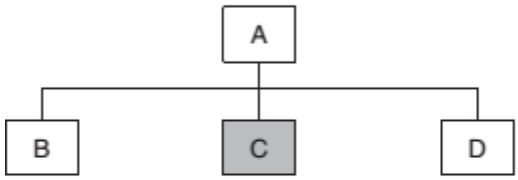


Figure 7.3 Top-down integration of modules A, B, and D.

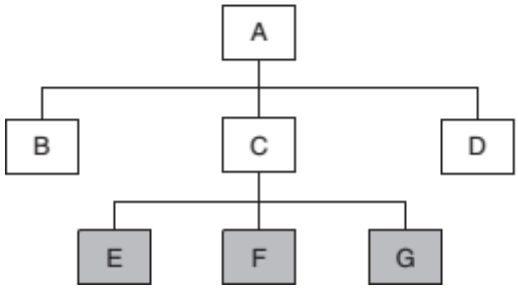


Figure 7.4 Top-down integration of modules A, B, D, and C.

Bottom Up

- Now we give an example of bottom-up integration for the module hierarchy of Figure 7.1. The lowest level modules are E, F, and G. We design a test driver to integrate these three modules, as shown in Figure 7.8. It may be noted that
  modules E, F, and G have no direct interfaces among them. However, return values generated by one module is likely to be used in another module, thus having an indirect interface. The test driver in Figure 7.8 invokes modules E, F, and G in a
  way similar to their invocations by module C. The test driver mimics module C to integrate E, F, and G in a limited way, because it is much simpler in capability than module C. The test driver is replaced with the actual module —in this case C—and a new test driver is used after the testers are satisfied with the combined behavior of E, F, and G (Figure 7.9).

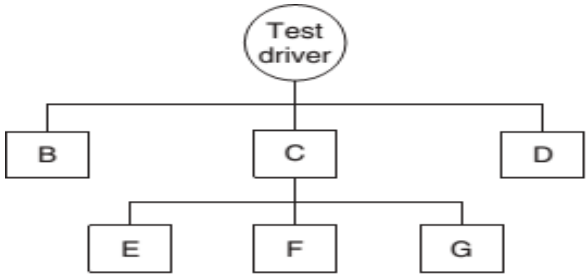


Figure 7.9 Bottom-up integration of modules B, C, and D with E, F, and G.

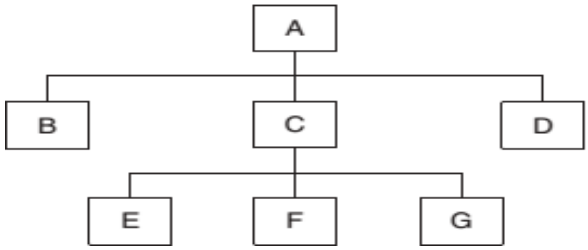


Figure 7.10 Bottom-up integration of module A with all others.

At this moment, more modules, such as B and D, are integrated with the so-far integrated system. The test driver mimics the behavior of module A. We need to include modules B and D because those are invoked by A and the test driver mimics A (Figure 7.9). The test driver is replaced with module A (Figure 7.10), and further tests are performed after the testers are satisfied with the integrated system shown in Figure 7.9.

Sandwich and Big Bang

- **In the sandwich approach**, a system is integrated by using a mix of the top-down and bottom-up approaches.
  A hierarchical system is viewed as consisting of three layers. The bottom layer contains all the modules that are often invoked. The bottom-up approach is applied to integrate the modules in the bottom layer. The top layer contains modules implementing major design decisions. These modules are integrated by using the top-down approach.
  On the other hand, if the middle layer exists, then this layer can be integrated by using the big-bang approach after the top and the bottom layers have been integrated.
- **In the big-bang approach,** first all the modules are individually tested. Next, all those modules are put together to construct the entire system which is tested as a whole.

TEST PLAN FOR SYSTEM INTEGRATION

TABLE 7.3 Framework for SIT Plan

1. Scope of testing
2. Structure of integration levels
   a. Integration test phases
   b. Modules or subsystems to be integrated in each phase
   c. Building process and schedule in each phase
   d. Environment to be set up and resources required in each phas
3. Criteria for each integration test phase *n*
   a. Entry criteria
   b. Exit criteria
   c. Integration Techniques to be used
   d. Test configuration set-up
4. Test specification for each integration test phase
   a. Test case ID number
   b. Input data
   c. Initial condition
   d. Expected results
   e. Test procedure
      How to execute this test?
      How to capture and interpret the results?
5. Actual test results for each integration test phase
6. References
7. Appendix

TABLE 7.4 Framework for Entry Criteria to Start System Integration

Softwave functional and design specifications must be writen, reviewed, and approved.
Code is reviewed and approved.
Unit test plan for each module is written, reviewed, and executed.
All of the unit tests passed.
The entire check-in request form must be completed, submitted, and approved.
Hardware design specification is written, reviewed, and approved.
Hardware design verification test is written, reviewed, and executed.
All of the design verification tests passed.
Hardware/software integration test plan is written, reviewed, and executed.
All of the hardware/software integrated tests passed.

TABLE 7.5 Framework for System Integration Exit Criteria

All code is completed and frozen and no more modules are to be integrated.
All of the system integration tests passed.
No major defect is outstanding.
All the moderate defects found in the SIT phase are fixed and retested.
Not more than 25 minor defects are outstanding.
Two weeks system uptime in system integration test environment without any anomalies, i.e. crashes.
System integration tests results are documented.

OFF-THE-SHELF COMPONENT INTEGRATION

- Instead of developing a software component from scratch, organizations occasionally purchase off-the-shelf (OTS) components form third-party vendors and integrate them with their own components
  The supporting components are *wrappers*, *glue*, and *tailoring*
  Buyer organizations perform two types of testing on an OTS component before purchasing: (i) acceptance testing of the OTS component based on the criteria discussed in Chapter 14 and (ii) integration of the component with other components
  developed in-house or purchased from a third party.
  Integration of OTS components is a challenging task because of the following characteristics identified by Basili and Boehm
  The buyer has no access to the source code.
  The vendor controls its development.
  The vendor has nontrivial installed base.
- three types of testing techniques to determine the suitability of an OTS component:
  • **Black-Box Component Testing:** This is used to determine the quality of the component.
  • **System-Level Fault Injection Testing:** This is used to determine how well a system will tolerate a failing component.
  • **Operational System Testing:** This kind of test is used to determine the tolerance of a software system when the OTS component is functioning correctly.

UNIT III

Software Quality Assurance (SQA) : The uniqueness of SQA, The environments for which SQA methods are developed, what is software, Software errors, faults and failures, classification of the causes of software errors, software quality-definition, SQA - definition and objectives.

Software quality factors: Classification of software requirements into software quality factors: Product operation, product revision, Product transition.

Software quality assurance system: The SQA system-an SQA architecture: Pre-project components, Software project life cycle components, Infrastructure components, Management SQA components**.**

The uniqueness of software quality

- No developer will declare that its software is free of defects, as major manufacturers of computer hardware are wont to do. This refusal actually reflects the essential elemental differences between software and other industrial products, such as automobiles, washing machines or radios. These differences can be categorized as follows:

  (1) Product complexity. Product complexity can be measured by the number of operational modes the product permits. An industrial product, even an advanced machine, does not allow for more than a few thousand modes of operation, created by the combinations of its different machine settings. Looking at a typical software package one can find millions of software operation possibilities. Assuring that the multitude of operational possibilities is correctly defined and developed is a major challenge to the software industry.

  (2) Product visibility. Whereas the industrial products are visible, software products are invisible. Most of the defects in an industrial product can be detected during the manufacturing process. Moreover the absence of a part in an industrial product is, as a rule, highly visible (imagine a door missing from your new car). However, defects in software products (whether stored on diskettes or CDs) are invisible, as is the fact that parts of a software package may be absent from the beginning.

  3) Product development and production process. Let us now review the phases at which the possibility of detecting defects in an industrial product may arise:

  (a) **Product development**. In this phase the designers and quality assurance (QA) staff check and test the product prototype, in order to detect its defects.

  (b) **Product production planning**. During this phase the production process and tools are designed and prepared. In some products there is a need for a special production line to be designed and built. This phase thus provides additional opportunities to inspect the product, which may reveal defects that "escaped" the reviews and tests conducted during the development phase.

  (c) **Manufacturing**. At this phase QA procedures are applied to detect failures of products themselves. Defects in the product detected in the first period of manufacturing can usually be corrected by a change in the product's design or materials or in the production tools, in a way that eliminates such defects in products manufactured in the future.

  In comparison to industrial products, software products do not benefit from the opportunities for detection of defects at all three phases of the production process. The only phase when defects can be detected is the development phase. Let us review what each phase contributes to the detection of defects:

(a) **Product development**. During this phase, efforts of the development teams and software quality assurance professionals are directed toward detecting inherent product defects. At the end of this phase an approved prototype, ready for reproduction, becomes available.
(b) **Product production planning**. This phase is not required for the software production process, as the manufacturing of software copies and printing of software manuals are conducted automatically. This applies to any software product, whether the number of copies is small, as in custom-made software, or large, as in software packages sold to the general public.
(c) **Manufacturing**. As mentioned previously, the manufacturing of software is limited to copying the product and printing copies of the software manuals. Consequently, expectations for detecting defects are quite limited during this phase.

Table 1.1: Factors affecting defect detection in software products vs. other industrial products

Characteristic	Software products	Other industrial products
Complexity	Usually, very complex product allowing for very large number of operational options	Degree of complexity much lower, allowing at most a few thousand operational options
Visibility of product	Invisible product, impossible to detect defects or omissions by sight (e.g. of a diskette or CD storing the software)	Visible product, allowing effective detection of defects by sight
Nature of development and production process	Opportunities to detect defects arise in only one phase, namely product development	Opportunities to detect defects arise in all phases of development and production: - Product development - Product production planning - Manufacturing

The environments for which SQA methods are

- The software developed by many individuals and in different situations fulfills a variety of needs:
  - ■ Pupils and students develop software as part of their education.
  - ■ Software amateurs develop software as a hobby.
  - ■ Professionals in engineering, economics, management and other fields develop software to assist them in their work, to perform calculations,summarize research and survey activities, and so forth.
  - ■ Software development professionals (systems analysts and programmers) develop software products or firmware as a professional career objective while in the employment of software houses or by software development and maintenance units (teams, departments, etc.) of large and small

  The main characteristics of this environment are as follows:

  (1) **Contractual conditions**. As a result of the commitments and conditions defined in the contract between the software developer and the customer, the activities of software development and maintenance need to cope with:

  ■ A defined list of functional requirements that the developed software and it maintenance need to fulfill.

  ■ The project budget.

  ■ The project timetable.

  The managers of software development and maintenance projects need to invest a considerable amount of effort in the oversight of activities in order to meet the contract's requirements.

  (2) **Subjection to customer–supplier relationship**. Throughout the process of software development and maintenance, activities are under the oversight of the customer. The project team has to cooperate continuously with the customer: to consider his request for changes, to discuss his criticisms about the various aspects of the project, and to get his approval for changes initiated by the development team. Such relationships do not usually exist when software is developed by non-software professionals.

  3) **Required teamwork**. Three factors usually motivate the establishment of a project team rather than assigning the project to one professional:

  ■ Timetable requirements. In other words, the workload undertaken during the project period requires the participation of more than one person if the project is to be completed on time.

  ■ The need for a variety of specializations in order to carry out the project.

  ■ The wish to benefit from professional mutual support and review for the enhancement of project quality.

  (4) **Cooperation and coordination with other software teams**. The carryingout of projects, especially large-scale projects, by more than one team is a very common event in the software industry. In these cases, cooperation may be required with:

■ Other software development teams in the same organization.
■ Hardware development teams in the same organization.
■ Software and hardware development teams of other suppliers.
■ Customer software and hardware development teams that take part in the project's development.

An outline of cooperation needs, as seen from the perspective of the development team, is shown in Figure 1.1.

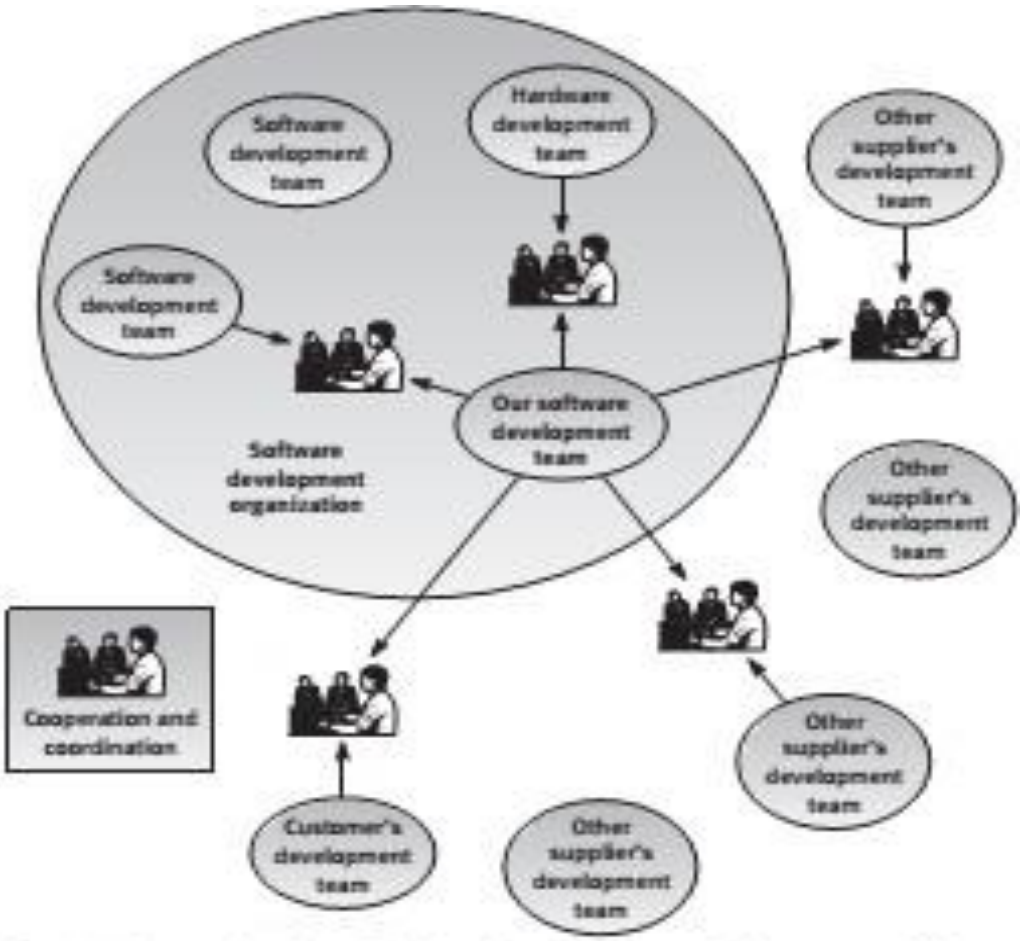


Figure 1.1: A cooperation and coordination scheme for a software development team of a large-scale project

(5) **Interfaces with other software systems**. Nowadays, most software systems include interfaces with other software packages. These interfaces allow data in electronic form to flow between the software systems, free from keying in of data processed by the other software systems. One can identify the following main types of interfaces:

■ Input interfaces, where other software systems transmit data to your software system.

■ Output interfaces, where your software system transmits processed data to other software systems.

■ Input and output interfaces to the machine's control board, as in medical and laboratory control systems, metal processing equipment, etc. Salary processing software packages provide good examples of typical input and output interfaces to other software packages.

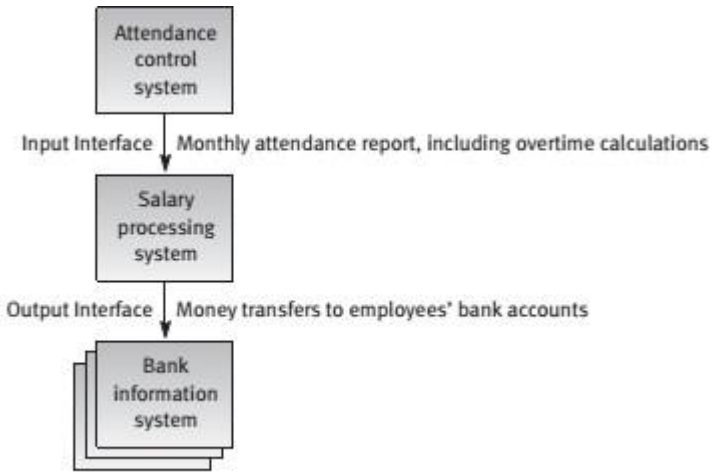


Figure 1.2: The salary software system – an example of software interfaces

(6) **The need to continue carrying out a project despite team member changes**. It is quite common for team members to leave the team during the project development period, whether owing to promotions to higher level jobs, a switch in employers, transfers to another city, and so forth. The team leader then has to replace the departing team member either by another employee or by a newly recruited employee. No matter how much effort is invested in training the new team member, "the show must go on", which means that the original project contract timetable will not change.

(7) **The need to continue carrying out software maintenance for an extended period**. Customers who develop or purchase a software system expect to continue utilizing it for a long period, usually for 5–10 years. During the service period, the need for maintenance will eventually arise. In most cases, the developer is required to supply these services directly. Internal "customers", in cases where the software has been developed in-house, share the same expectation regarding the software maintenance during the service period of the software system.

What is software? •

Software is:

Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system.

The IEEE definition of software, which is almost identical to the ISO definition ISO, 1997, Sec. 3.11 and ISO/IEC 9000-3 Sec. 3.14), lists the following four components of software:

■ Computer programs (the "code")

■ Procedures

■ Documentation

■ Data necessary for operating the software system.

All four components are needed in order to assure the quality of the software development process and the coming years of maintenance services for the following reasons:

■ Computer programs (the "code") are needed because, obviously, they activate the computer to perform the required applications.

■ Procedures are required, to define the order and schedule in which the programs are performed, the method employed, and the person responsible for performing the activities that are necessary for applying the software.

■ Various types of documentation are needed for developers, users and maintenance personnel. The development documentation (the requirements report, design reports, program descriptions, etc.) allows efficient cooperation and coordination among development team members and efficient reviews and inspections of the design and programming products. The user's documentation (the "user's manual", etc.) provides a their use.

The maintenance documentation (the "programmer's software manual", etc.) provides the maintenance team with all the required information about the code, and the structure and tasks of each software module. This information is used when trying to locate causes of software failures ("bugs") or to change or add to existing software.

■ Data including parameters, codes and name lists that adapt the software to the needs of the specific user are necessary for operating the software. Another type of essential data is the standard test data, used to ascertain that no undesirable changes in the code or software data have occurred, and what kind of software malfunctioning can be expected.

To sum up, software quality assurance always includes, in addition to code quality, the quality of the procedures, the documentation and the

Software errors, faults and failures

- The origin of software failures lies in a **software error** made by a programmer.

An error can be a grammatical error in one or more of the code lines,or a logical error in carrying out one or more of the client's requirements. However, not all software errors become **software faults**.

In other words, in some cases, the software error can cause improper functioning of the software in general or in a specific application. In many other cases, erroneous code lines will not affect the functionality of the software as a whole; in *a part* of these cases, the fault will be corrected or "neutralized" by subsequent code lines.162

What is software quality? 17

2.2 Software errors, faults and failures

We are interested mainly in the software failures that disrupt our use of the software. This requires us to examine the relationship between software faults and **software failures**.

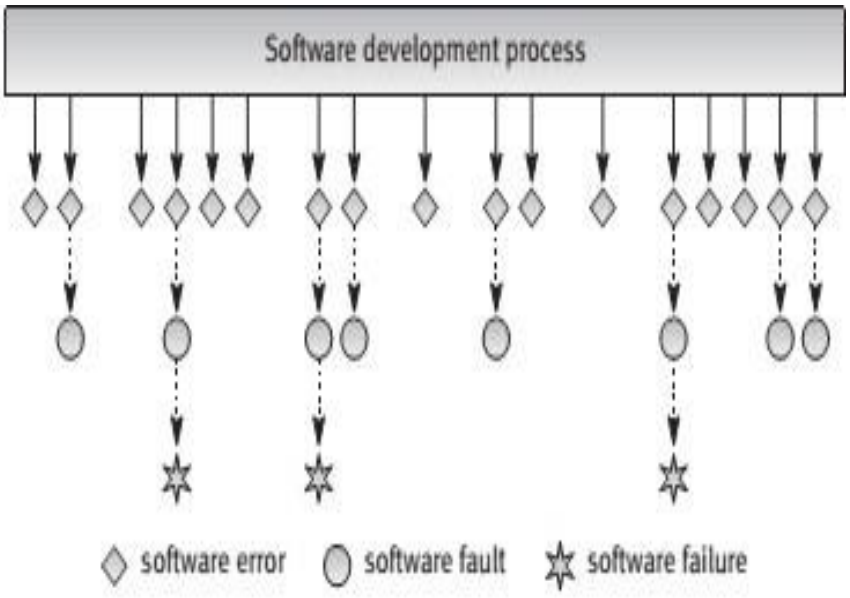


Figure 2.1: Software errors, sotware faults and software failures

Classification of the causes of software

As software errors are the cause of poor software quality, it is important to investigate the causes of these errors in order to prevent them. A software error can be "code error", a "procedure error", a "documentation error", or a "software data error".

The causes of software errors can be further classified as follows according to the stages of the software development process in which they occur.

(1) Faulty definition of requirements

The faulty definition of requirements, usually prepared by the client, is one of the main causes of software errors. The most common errors of this type are:

■ Erroneous definition of requirements.

■ Absence of vital requirements.

■ Incomplete definition of requirements. For instance, one of the requirements of a municipality's local tax software system refers to discounts granted to various segments of the population: senior citizens, parents of large families, and so forth. Unfortunately, a discount granted to students was not included in the requirements document.

■ Inclusion of unnecessary requirements, functions that are not expected to be needed in the near future.

(2) Client–developer communication failures

Misunderstandings resulting from defective client–developer communication are additional causes for the errors that prevail in the early stages of the development process:

■ Misunderstanding of the client's instructions as stated in the requirement document.

■ Misunderstanding of the client's requirements changes presented to the developer in written form during the development period.

■ Misunderstanding of the client's requirements changes presented orally to the developer during the development period.

■ Misunderstanding of the client's responses to the design problems presented by the developer. Lack of attention to client messages referring to requirements changes and to client responses to questions raised by the developer on the part of the developer.

(3) Deliberate deviations from software requirements

In several circumstances, developers may deliberately deviate from the documented requirements, actions that often cause software errors. The errors in these cases are byproducts of the changes. The most common situations of deliberate deviation are:

■ The developer reuses software modules taken from an earlier project without sufficient analysis of the changes and adaptations needed to correctly fulfill all the new requirements.

■ Due to time or budget pressures, the developer decides to omit part of the required functions in an attempt to cope with these pressures.

■ Developer-initiated, unapproved improvements to the software, introduced without the client's approval, frequently disregard requirements that seem minor to the developer. Such "minor" changes may, eventually, cause software errors.

(4) Logical design errors

Software errors can enter the system when the professionals who design the system – systems architects, software engineers, analysts, etc. – formulate the software requirements. Typical errors include:

■ Definitions that represent software requirements by means of erroneous algorithms.

■ Process definitions that contain sequencing errors.

■ Erroneous definition of boundary conditions.

■ Omission of required software system states.

■ Omission of definitions concerning reactions to illegal operation of the software system.

(5) Coding errors

A broad range of reasons cause programmers to make coding errors. These include misunderstanding the design documentation, linguistic errors in the programming languages, errors in the application of CASE and other development tools, errors in data selection, and so forth.

(6) Non-compliance with documentation and coding instructions

Almost every development unit has its own documentation and coding standards that define the content, order and format of the documents, and the code created by team members. To support this requirement, the unit develops and publicizes its templates and coding instructions. Members of the development team or unit are required to comply with these requirements.

(7) Shortcomings of the testing process

Shortcomings of the testing process affect the error rate by leaving a greater number of errors undetected or uncorrected. These shortcomings result from the following causes:

■ Incomplete test plans leave untreated portions of the software or the application functions and states of the system.

■ Failures to document and report detected errors and faults.

■ Failure to promptly correct detected software faults as a result of inappropriate indications of the reasons for the fault.

■ Incomplete correction of detected errors due to negligence or time pressures.

(8) Procedure errors

Procedures direct the user with respect to the activities required at each step of the process. They are of special importance in complex software systems where the processing is conducted in several steps, each of which may feed a variety of types of data and allow for examination of the intermediate results.

9) Documentation errors

The documentation errors that trouble the development and maintenance teams are errors in the design documents and in the documentationintegrated into the body of the software. These errors can cause additional errors in further stages of development and during maintenance.

Another type of documentation error, one that affects mainly the users, is an error in the user manuals and in the "help" displays incorporated in the software. Typical errors of this type are:

■ Omission of software functions.

■ Errors in the explanations and instructions given to users, resulting in "dead ends" or incorrect applications.

■ Listing of non-existing software functions, that is, functions planned in the early stages of development but later dropped, and functions that were active in previous versions of the software but cancelled in the current version.

Software quality – definition

- Software quality is:

  1. The degree to which a system, component, or process meets specified requirements.
  2. The degree to which a system, component, or process meets customer or user needs or expectations.

  Software quality – Pressman's definition Software quality is defined as:
  Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.

Software quality assurance – definition and objectives

- Software quality assurance – The IEEE definition Software quality assurance is:
  1. A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements.
  2. A set of activities designed to evaluate the process by which the products are developed or manufactured. Contrast with quality control.

  This definition may be characterized in the following:
  ■ Plan and implement systematically. SQA is based on planning and the application of a variety of actions that are integrated into all the stages of the software development process. This is done in order to substantiate the client's confidence that the software product will meet all the technical requirements.
  ■ Refer to the software development process.
  ■ Refer to the specifications of the technical requirements. The main deviations from the IEEE definition are:
  ■ SQA should not be limited to the development process.
  ■ SQA actions should not be limited to the technical aspects of the functional requirements, but should include also activities that deal with scheduling and the budget.

SQA – expanded definition

- SQA – expanded definition Software quality assurance is:
  A systematic, planned set of actions necessary to provide adequate confidence that the software development process or the maintenance process of a software system product conforms to established functional technical requirements as well as with the managerial requirements of keeping the schedule and operating within the budgetary confines.

The objectives of SQA activities

- The objectives of SQA activities

  Software development (process-oriented):

  1. Assuring an acceptable level of confidence that the software will conform to functional technical requirements.
  2. Assuring an acceptable level of confidence that the software will conform to managerial scheduling and budgetary requirements.
  3. Initiating and managing of activities for the improvement and greater efficiency of software development and SQA activities. This means improving the prospects that the functional and managerial requirements will be achieved while reducing the costs of carrying out the software development and SQA activities.

  Software maintenance (product-oriented):

  1. Assuring with an acceptable level of confidence that the software maintenance activities will conform to the functional technical requirements. 2. Assuring with an acceptable level of confidence that the software maintenance activities will conform to managerial scheduling and budgetary requirements.
  3. Initiating and managing activities to improve and increase the efficiency of software maintenance and SQA activities. This involves improving the prospects of achieving functional and managerial requirements while reducing costs.

Classifications of software requirements

- Several models of software quality factors and their categorization in factor categories have been suggested over the years. The classic model of software quality factors, suggested by McCall, consists of 11 factors *McCall's factor model* McCall's factor model classifies all software requirements into 11 software quality factors. The 11 factors are grouped into three categories – product operation, product revision and product transition – as follows:
  ■ **Product operation factors:** Correctness, Reliability, Efficiency, Integrity, Usability.
  ■ **Product revision factors:** Maintainability, Flexibility, Testability.
  ■ **Product transition factors:** Portability, Reusability, Interoperability. McCall's model and its categories are illustrated by the McCall model of SQF tree.

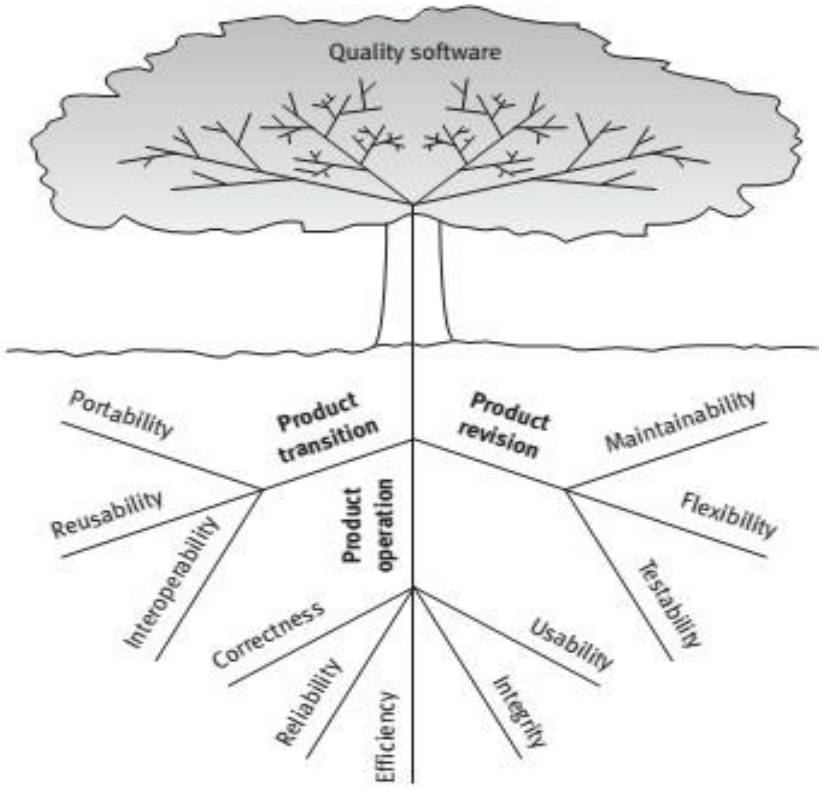


Figure 3.1: McCall's factor model tree

Source: Based on McCall *et al*., 1977

Product operation software quality factors

According to McCall's model, five software quality factors are included in the product operation category, all of which deal with requirements that directly affect the daily operation of the software. These factors are as follows.

Correctness

Correctness requirements are defined in a list of the software system's required outputs, such as a query display of a customer's balance in the sales accounting information system, or the air supply as a function of temperature specified by the firmware of an industrial control unit. Output specifications are usually multidimensional; some common dimensions include:

■ The output mission (e.g., sales invoice printout, and red alarms when temperature rises above 250°F).

■ The required accuracy of those outputs that can be adversely affected by inaccurate data or inaccurate calculations.

■ The completeness of the output information, which can be adversely affected by incomplete data.

■ The up-to-dateness of the information (defined as the time between the event and its consideration by the software system).

■ The availability of the information (the reaction time, defined as the time needed to obtain the requested information or as the requested reaction time of the firmware installed in a computerized apparatus).

■ The standards for coding and documenting the software system.

Reliability

Reliability requirements deal with failures to provide service. They determine the maximum allowed software system failure rate, and can refer to the entire system or to one or more of its separate functions. *Efficiency* Efficiency requirements deal with the hardware resources needed to perform all the functions of the software system in conformance to all other requirements. The main hardware resources to be considered are the computer's processing capabilities (measured in MIPS – million instructions per second, MHz or megahertz – million cycles per second, etc.), its data storage capability in terms of memory and disk capacity (measured in MBs – megabytes, GBs – gigabytes, TBs – terabytes, etc.) and the data communication capability of the communication lines (usually measured in KBPS – kilobits per second, MBPS – megabits per second, and GBPS – gigabits per.

Integrity

Integrity requirements deal with the software system security, that is, requirements to prevent access to unauthorized persons, to distinguish between the majority of personnel

allowed to see the information ("read permit") and alimited group who will be allowed to add and change data ("write permit"),
and so forth.

Usability

Usability requirements deal with the scope of staff resources needed to train a new employee and to operate the software system. For more about usability see Juristo *et al*. (2001), Donahue (2001) and Ferre *et al*. (2001).

Product revision

- **Product revision software quality factors**

According to the McCall model of software quality factors, three quality factors comprise the product revision category. These factors deal with those requirements that affect the complete range of software maintenance activities: corrective maintenance (correction of software faults and failures), adaptive maintenance (adapting the current software to additional circumstances and customers without changing the software) and perfective maintenance (enhancement and improvement of existing software with respect to locally limited issues). These are as follows.

Maintainability

Maintainability requirements determine the efforts that will be needed by users and maintenance personnel to identify the reasons for software failures, to correct the failures, and to verify the success of the corrections. This factor's requirements refer to the modular structure of software.

Flexibility

The capabilities and efforts required to support adaptive maintenance activities are covered by the flexibility requirements. These include the resources (i.e. in man-days) required to adapt a software package to a variety of customers of the same trade, of various extents of activities, of different ranges of products and so on. This factor's requirements also support perfective maintenance activities, such as changes and additions to the software in order to improve its service and to adapt it to changes in the firm's technical or commercial environment.

Testability

Testability requirements deal with the testing of an information system as well as with its operation. Testability requirements for the ease of testing are related to special features in the programs that help the tester, for instance by providing predefined intermediate results and log files.

Testability requirements related to software operation include automatic diagnostics performed by the software system prior to starting the system, to find out whether all components of the software system are in working order and to obtain a report about the detected faults.

Product transition

- **Product transition software quality factors**

  According to McCall, three quality factors are included in the product transition category, a category that pertains to the adaptation of software to other environments and its interaction with other software systems.

Portability

Portability requirements tend to the adaptation of a software system to other environments consisting of different hardware, different operating systems, and so forth. These requirements make it possible to continue using the same basic software in diverse situations or to use it simultaneously in diverse hardware and operating systems situations.

Reusability

Reusability requirements deal with the use of software modules originally designed for one project in a new software project currently being developed.

Interoperability

Interoperability requirements focus on creating interfaces with other software systems or with other equipment firmware (for example, the firmware of the production machinery and testing equipment interfaces with the production control software). Interoperability requirements can specify the name(s) of the software or firmware for which interface is required. They can also specify the output structure accepted as standard in a specific industry or applications area.

The SQA system – an SQA architecture

- The SQA system – an SQA architecture

An SQA system always combines a wide range of SQA components, all of which are employed to challenge the multitude of sources of software errors and to achieve an acceptable level of software quality

■ **Pre-project components**. To assure that (a) the project commitments have been adequately defined considering the resources required, the schedule and budget; and (b) the development and quality plans have been correctly determined.

■ **Components of project life cycle activities assessment** . The project life cycle is composed of two stages: the development life cycle stage and the operation–maintenance stage.

The development life cycle stage components detect design and programming errors. Its components are divided into the following four sub classes:

-Reviews

- Expert opinions

- Software testing.

-Project life cycle components

The SQA components used during the operation–maintenance phase include specialized maintenance components as well as development life cycle components, which are applied mainly for functionality improving maintenance tasks.

An additional sub-class of SQA project life cycle components deals.

■ **Components of infrastructure error prevention and improvement**.

The main objectives of these components, which are applied throughout theentire organization, are to eliminate or at least reduce the rate of errors, based on the organization's accumulated SQA experience.

■ **Components of software quality management**. This class of components is geared toward several goals, the major ones being the control of development and maintenance activities and the introduction of early

managerial support actions that mainly prevent or minimize schedule and budget failures and their outcomes.

■ **Components of standardization, certification, and SQA system assessment**.

These components implement international professional and

managerial standards within the organization. The main objectives of this

class are

(a) utilization of international professional knowledge,

(b) improvement of coordination of the organizational quality systems with other organizations, and

(c) assessment of the achievements of quality systems according to a common scale. The various standards may be classified into two main groups:

(a) quality management standards, and

(b) project process standards.

■ **Organizing for SQA – the human components**. The SQA organizational base includes managers, testing personnel, the SQA unit and practitioners interested in software quality (SQA trustees, SQA committee members and SQA forum members). All these *actors* contribute to software

Pre-project components

- The SQA components belonging here are meant to improve the preparatory steps taken prior to initiating work on the project itself:
  - Contract review
  - Development and quality plans.

Contract review

Software may be developed within the framework of a contract negotiated with a customer or in response to an internal order originating in another department. An internal order may entail a request for developing a firmware software system to be embedded within a hardware product, an order for a software product to be sold as a package, or an order for the development of administrative software to be applied within the company.

In all these instances, the development unit is committed to an agreed-upon functional specification, budget and schedule.

- Clarification of the customer's requirements
- Review of the project's schedule and resource requirement estimates
- Evaluation of the professional staff's capacity to carry out the proposed project
- Evaluation of the customer's capacity to fulfill his obligations
- Evaluation of development risks.

A similar approach is applied in the review of maintenance contracts. Such reviews take into account that besides error corrections, maintenance services include software adaptation and limited software development activities for the sake of performance improvement (termed "functionality improvement maintenance").

Development and quality plans

- Once a software development contract has been signed or a commitment made to undertake an internal project for the benefit of another department of the organization, a plan is prepared of the project ("development plan") and its integrated quality assurance activities ("quality plan").

These plans include additional details and needed revisions based on prior plans that provided the basis for the current proposal and contract. It is quite common for several months to pass between the tender submission and the signing of thecontract. During this period, changes may occur in staff availability, in professional capabilities, and so forth. The plans are then revised to reflect the changes that occurred in the interim.

- The main issues treated in the project development plan are:
  - Schedules
  - Required manpower and hardware resources
  - Risk evaluations
  - Organizational issues: team members, subcontractors and partnerships
  - Project methodology, development tools, etc
  - Software reuse plans.

  The main issues treated in the project's quality plan are:
  - Quality goals, expressed in the appropriate measurable terms
  - Criteria for starting and ending each project stage
  - Lists of reviews, tests, and other scheduled verification and validation activities.

Software project life cycle components

The project life cycle is composed of two stages: the development life cycle stage and the operation–maintenance stage. Several SQA components enter the software development project life cycle at different points. Their use should be planned prior to the project's initiation. The main components are:

- Reviews
- Expert opinions
- Software testing
- Software maintenance
- Assurance of the quality of the subcontractors' work and the customer supplied parts.

Reviews

The design phase of the development process produces a variety of documents. The printed products include design reports, software test documents, software installation plans and software manuals, among others. Reviews can be categorized as formal design reviews (DRs) and peer reviews.

i) *Formal design reviews (DRs)*

A significant portion of these documents requires formal professional approval of their quality as stipulated in the development contract and demanded by the procedures applied by the software developer.

It should be emphasized that the developer can continue to the next phase of the development process only on receipt of formal approval of these documents. Ad hoc committees whose members examine the documents presented by the development teams usually carry out formal design reviews (widely known as "DRs"). The committees are composed of senior professionals, including the project leader and, usually, the department manager, the chief software engineer, and heads of other related departments.

The majority of participants hold professional and administrative ranks higher than the project leader. On many occasions, the customer's representative will participate in a DR (this participation is generally indicated among the contractual arrangements). The DR report itself includes a list of required corrections (termed "action items").

When a design review committee sits in order to decide upon the continuation of the work completed so far, one of the following options is usually open for consideration:

- Immediate approval of the DR document and continuation to the next

development phase.

■ Approval to proceed to the next development phase after all the action items have been completed and inspected by the committee's representative.

■ An additional DR is required and scheduled to take place after all the action items have been completed and inspected by the committee's.

ii) Peer reviews

Peer reviews (inspections and walkthroughs) are directed at reviewing short documents, chapters or parts of a report, a coded printout of a software module, and the like. Inspections and walkthroughs can take several forms and use many methods; usually, the reviewers are all peers, not superiors, who provide professional assistance to colleagues.

The main objective of inspections and walkthroughs is to detect as many design and programming faults as possible. The output is a list of detected faults and, for inspections, also a defect summary and statistics to be used as a database for reviewing and improving development methods.

Expert opinions

Expert opinions support quality assessment efforts by introducing additional external capabilities into the organization's in-house development process. Turning to outside experts may be particularly useful in the following situations:

■ Insufficient in-house professional capabilities in a given area.

■ In small organizations in many cases it is difficult to find enough suitable candidates to participate in the design review teams. In such situations, outside experts may join a DR committee or, alternatively, their expert opinions may replace a DR.

■ In small organizations or in situations characterized by extreme work pressures, an outside expert's opinion can replace an inspection.

■ Temporary inaccessibility of in-house professionals .

- In case of major disagreement among the organization's senior professionals, an outside expert may support a decision.

Software testing

Software tests are formal SQA components that are targeted toward review of the actual running of the software. The tests are based on a prepared list of test cases that represent a variety of expected scenarios. Software tests examine software modules, software integration, or entire software packages (systems). Recurrent tests (usually termed "regression tests"), carried out after correction of previous test findings, are continued till satisfactory results are obtained.

The direct objective of the software tests, other than detection of software faults and other failures to fill the requirements, is the formal approval of a module or

integration setup so that either the next programming phase can be begun or the completed software system can be delivered and installed.

Software maintenance components

Software maintenance services vary in range and are provided for extensive periods, often several years. These services fall into the following categories:

■ **Corrective maintenance** – User's support services and correction of software code and documentation failures.

■ **Adaptive maintenance** – Adaptation of current software to new circumstances and customers without changing the basic software product.
These adaptations are usually required when the hardware system or its components undergo modification (additions or changes).

■ **Functionality improvement maintenance** – The functional and performance-related improvement of existing software, carried out with respect to limited issues.

Assurance of the quality of the external participant'swork

Subcontractors and customers frequently join the directly contracted developers (the "supplier") in carrying out software development projects. The larger and more complex the project, the greater the likelihood that external participants will be required, and the larger the proportion of work transmitted to them (subcontractors, suppliers of COTS software and thecustomer).

The motivation for turning to external participants lies in any number of factors, ranging from the economic to the technical to personnelrelated interests, and reflects a growing trend in the allocation of the work involved with completing complex projects.

Infrastructure components for error Prevention and improvement

- The goals of SQA infrastructure are the prevention of software faults or, at least, the lowering of software fault rates, together with the improvement of productivity. SQA infrastructure components are developed specifically to this end. These components are devised to serve a wide range of projects and software maintenance services. During recent years, we have witnessed the growing use of computerized automatic tools for the application of these components. This class of SQA components includes:
  - ■ Procedures and work instructions
  - ■ Templates and checklists
  - ■ Staff training, retraining, and certification
  - ■ Preventive and corrective actions
  - ■ Configuration management
  - ■ Documentation control.

1. Procedures and work instructions

Quality assurance procedures usually provide detailed definitions for the performance of specific types of development activities in a way that assures effective achievement of quality results. Procedures are planned to be generally applicable and to serve the entire organization. Work instructions, in contrast, provide detailed directions for the use of methods that are applied in unique instances and employed by specialized teams.

2. Supporting quality devices

One way to combine higher quality with higher efficiency is to use supporting quality devices, such as templates and checklists. These devices, based as they are on the accumulated knowledge and experience of the organization's development and maintenance professionals, contribute to meeting SQA goals by:

■ Saving the time required to define the structure of the various documents or prepare lists of subjects to be reviewed.

■ Contributing to the completeness of the documents and reviews.

■ Improving communication between development team and review committee members by standardizing documents and agendas.

3. Staff training, instruction and certification

The banality of the statement that a trained and well-instructed professional staff is the key to efficient, quality performance, does not make this observation any less true. Within the framework of SQA, keeping an

organization's human resources knowledgeable and updated at the level required is achieved mainly by:

■ Training new employees and retraining those employees who have changed assignments.

■ Continuously updating staff with respect to professional developments and the in-house, hands-on experience acquired.

■ Certifying employees after their knowledge and ability have been demonstrated.

4. Preventive and corrective actions

Systematic study of the data collected regarding instances of failure and success contributes to the quality assurance process in many ways. Among them we can list:

■ Implementation of changes that prevent similar failures in the future.

■ Correction of similar faults found in other projects and among the activities performed by other teams.

■ Implementing proven successful methodologies to enhance the probability of repeat successes.

5. Configuration management

The regular software development and maintenance operations involve intensive activities that modify software to create new versions and releases. These activities are conducted throughout the entire software service period (usually lasting several years) in order to cope with the needed corrections, adaptations to specific customer requirements, application improvements, and so forth.

Different team members carry out these activities simultaneously, although they may take place at different sites. As a result, serious dangers arise, whether of misidentification of the versions or releases, loss of the records delineating the changes implemented, or loss of documentation.

6. Documentation control

SQA requires the application of measures to ensure the efficient long-term availability of major documents related to software development ("controlled documents"). The purpose of one type of controlled document – the quality record – is mainly to provide evidence of the SQA system's performance. Documentation control therefore represents one of the building blocks of any SQA system.Documentation control functions refer mainly to customer requirement documents, contract documents, design reports, project plans, development standards, etc. Documentation control activities entail:

■ Definition of the types of controlled documents needed

- Specification of the formats, document identification methods, etc.
- Definition of review and approval processes for each controlled document
- Definition of the archive storage methods.

Management SQA components

- Managerial SQA components support the managerial control of software development projects and maintenance services. Control components include:
  - ■ Project progress control (including maintenance contract control)
  - ■ Software quality metrics
  - ■ Software qualitycosts.

Project progress control

The main objective of project progress control components is to detect the appearance of any situation that may induce deviations from the project's plans and maintenance service performance. Clearly, the effectiveness and efficiency of the corrective measures implemented is dependent on the timely discovery o f undesirable situations.

Project control activities focus on:

- ■ Resource usage
- ■ Schedules
- ■ Risk management activities
- ■ The budget

Software quality metrics

Measurement of the various aspects of software quality is considered to be an effective tool for the support of control activities and the initiation of process improvements during the development and the maintenancephases. These measurements apply to the functional quality, productivity, and organizational aspects of the project.

Among the software quality metrics available or still in the process of development, we can list metrics for:

- ■ Quality of software development and maintenance activities
- ■ Development teams' productivity
- ■ Help desk and maintenance teams' productivity
- ■ Software faults density
- ■ Schedule deviations.

Software quality costs

The quality costs incurred by software development and application are, according to the extended quality costs model, the costs of control (prevention costs, appraisal costs, and managerial preparation and control costs) combined with the costs of failure (internal failure costs, external failure costs, and managerial failure costs). Management is especially interested in the total sum of the quality costs. It is believed that up to a certain level,

expanding the resources allocated to control activities yields much larger savings in failure costs while reducing total quality costs. Accordingly, management tends to exhibit greater readiness to allocate funds to profitable proposals to improve application of existing SQA system components and further development of new components.

UNIT-IV

CASE Tools

An increasing variety of specialized computerized tools (actually software packages) have been offered to software engineering departments since the early 1990s. The purpose of these tools is to make the work of development and maintenance teams more efficient and more effective. CASE (Computer Aided Software Engineering) is a term covering a whole range of tools and methods that support software system development. These tools and methods reduce the load on developers allowing them to focus their skills on other goals. It can be used at all stages of the Software Development Life Cycle.

Fig 1: CASE Environment

CASE tools are computerized software development tools that support the developer when performing one or more phases of the software life cycle and/or support software maintenance. Compilers, interactive debugging systems, configuration management systems and automated testing systems can be considered as CASE tools. The various types of CASE tools are:

- **Diagramming tools**: enable system process, data and control structures to be represented graphically.
- **Computer display and report generators:** help prototype how systems look and feel. It makes it easier for the systems analyst to identify data requirements and relationship.
- **Analysis tools**: automatically check for importance, inconsistent, or incorrect specifications in diagrams, forms, and reports.
- **Central repository**: enables the integrated storage of specifications, diagrams, reports and project management information.
- Documentation Generators: produce technical and user documentation in standard formats.
- **Code generators:** enable the automatic generation of program and data base definition code directly from the design documents, diagrams, forms, and reports.

The comparison of CASE tools with classic real states that the computerized software development support tools such as interactive debuggers, compilers and project progress control systems can readily be considered classic CASE tools. The new tools that support the developer for a succession of several development phases of a development project are referred to as real CASE tools. When referring to real CASE tools, it is customary to

distinguish between upper CASE tools that support the analysis and design phases, and lower CASE tools that support the coding phase.

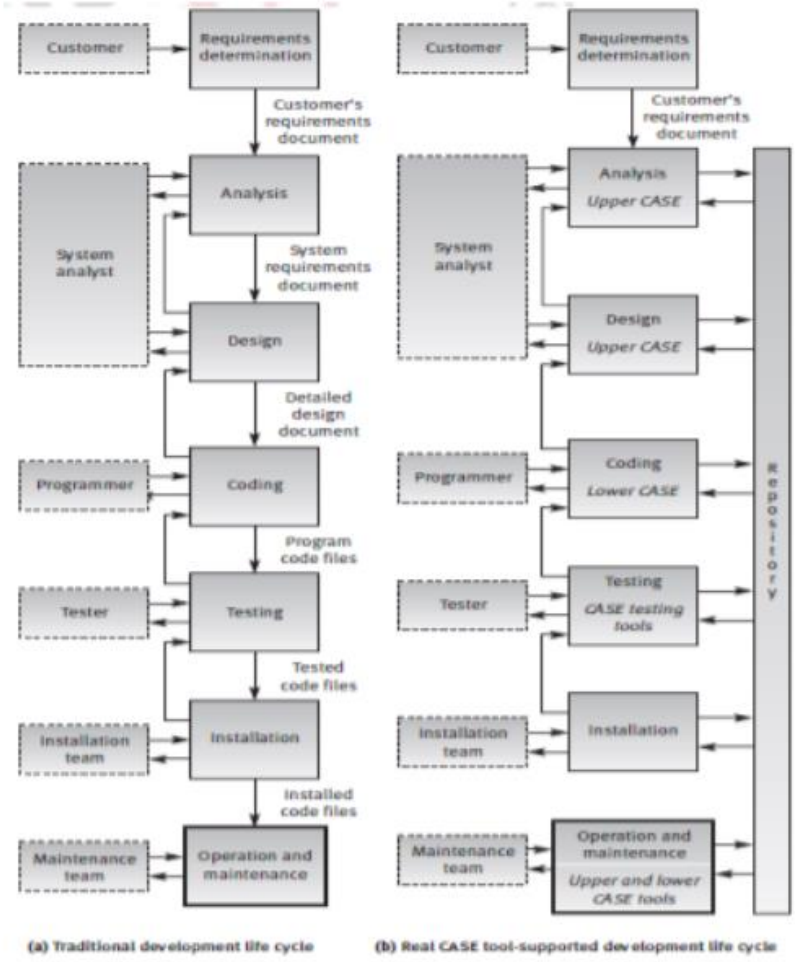


Fig 2:Traditional development SDLC vs Case Tool assisted SDLC

Contributions of CASE tools to Quality can be categorized in to:

- CASE Tool Support to Developers
- CASE Tools to improve Software Product Quality
- CASE tools to improve Software Maintenance Quality.
- CASE tools to enhance Project Management.

Type of CASE tool	Support Provided
Editing and Diagramming	Editing text and diagrams, generating design diagrams to repository records
Repository Query	Display of parts of the design texts, charts, etc.; cross referencing queries and requirements tracing
Automated Documentation	Automatic generation of requested documentation according to updated repository records
Design Support	Editing design recorded by the systems analyst and management of the data dictionary
Code Editing	Compiling, interpreting or applying interactive debugging code specific coding language or development tool
Code Generation	Transformation of design records into prototypes or application software compatible with a given software development language (or development tools)
Configuration Management	Management of design documents and software code versions, control of changes and software code.
Reverse Engineering (re-engineering)	Construction of a software repository and design documents, based on code: the "legacy" software systems. Once the repository of the legacy software is available, it can be updated and used to automatically generate new

Cause of software errors	Classic CASE tools	Real CASE tools
1. Faulty requirements definition	None	Almost none
2. Client-developer communication failures	None	Almost none
3. Deliberate deviations from software requirements	None	High
4. Logical design errors	None	High
5. Coding errors	Very high	Very high
6. Non-compliance with coding and documentation instructions	Limited	Very high
7. Shortcomings in the testing process	High	High
8. User interface and procedural errors	High	Limited
9. Documentation errors	Limited	High

Contribution of CASE tools to software product quality can be summarized as follows:

Contribution of CASE tools to Software Maintenance Quality consists of the following steps:

- **Corrective maintenance**
- **Adaptive maintenance**
- **Functional improvement maintenance**

In Corrective Maintenance CASE- generated full and updated documentation of the software enables easier and reliable identification of the cause for software failure. Cross-referenced queries enable identification of anticipated effects of any proposed correction. Correction by means of lower CASE tools provides automated

coding, with no expected coding errors. In Adaptive Maintenance Full and updated documentation of the software by CASE tools enables thorough examination of possible software package adaptations for new users and applications. In Functional Improvement Maintenance Use of the repository enables designers to assure consistency of new applications and improvements with existing software systems. Cross-referenced repository queries enable better planning of changes and additions. Changes and additions carried out by means of lower CASE or integrated CASE tools enable automated coding. The various categories of CASE tools can be of as follows:

- **Real CASE Tools**
  - Upper-CASE tools (front-end tools)
  - Lower-CASE tools (back-end tools)
  - Integrated CASE tools (I-CASE)
- **Workbenches**
- **Environments**

In Upper-CASE tools (front-end tools), it assist developer during requirements, analysis, and design workflows or activities. In Lower-CASE tools (back-end tools), it assist with implementation, testing, and maintenance workflows or activities. In Integrated CASE tools (I-CASE), it provides support for the full life cycle.

Fig 3: Real CASE Tools

In Workbenches, Collection of tools that together support the process workflows (requirements, design, etc.) and one or two activities where an activity is a related collection of tasks

Commercial examples:

- PowerBuilder
- Software through Pictures
- Software Architect

Environments support the complete software process or, at least, a large portion of the software process, which normally include several different workbenches which are integrated in some way.

Characteristics of a Successful CASE Tool

- **Standard Methodology**: CASE tools must support a standard software development methodology and standard modeling techniques.
- **Flexibility:** Flexibility in use of editors and other tools.
- **Strong Integration:** CASE tools should be integrated to support all the stages**.**
- **Integration with Testing Software:** CASE tools must provide interfaces for automatic testing tools that take care of regression and other kinds of testing software.
- **Support for reverse Engineering:** CASE tools must be able to generate complex models from already generated code.
- **On-line Help:** CASE tools must provide an online tutorial.

Objectives of quality measurement:

objectives of software quality metrics

- Main objectives of software quality metrics

  (1) To facilitate management control as well as planning and execution of the appropriate managerial interventions. Achievement of this objective isbased on calculation of metrics regarding:

  ■ Deviations of actual functional (quality) performance from planned performance

  ■ Deviations of actual timetable and budget performance from planned performance.

  (2) To identify situations that require or enable development or maintenance process improvement in the form of preventive or corrective actions

  introduced throughout the organization. Achievement of this objective is based on:

  ■ Accumulation of metrics information regarding the performance of teams, units, etc.

Comparison provides the practical basis for management's application of metrics and for SQA improvement in general. The metrics are used for comparison of performance data with *indicators*, quantitative values such as:

■ Defined software quality standards

■ Quality targets set for organizations or individuals

■ Previous year's quality achievements

■ Previous project's quality achievements

■ Average quality levels achieved by other teams applying the same development tools in similar development environments

■ Average quality achievements of the organization

■ Industry practices for meeting quality requirements.

Classification of software quality metrics

Software quality metrics can fall into a number of categories. Here we use a two-level system.

The first classification category distinguishes between life cycle and other phases of the software system:

■ Process metrics, related to the software development process (see Section 21.3)

■ Product metrics, related to software maintenance (see Section 21.4).

The second classification category refers to the subjects of the measurements:

■ Quality

■ Timetable

■ Effectiveness (of error removal and maintenance services)

■ Productivity.

Process metrics:

- Software development process metrics can fall into one of the following categories:
  - ■ Software process quality metrics
  - ■ Software process timetable metrics
  - ■ Error removal effectiveness metrics
  - ■ Software process productivity metrics.

  21.3.1 Software process quality metrics

  Software process quality metrics may be classified into two classes:
  - ■ Error density metrics
  - ■ Error severity metrics.

Table 21.1: Error density metrics

Code	Name	Calculation formula
CED	Code Error Density	$CED = \frac{NCE}{KLOC}$
DED	Development Error Density	$DED = \frac{NDE}{KLOC}$
WCED	Weighted Code Error Density	$WCED = \frac{WCE}{KLOC}$
WDED	Weighted Development Error Density	$WDED = \frac{WDE}{KLOC}$
WCEF	Weighted Code Errors per Function point	$WCEF = \frac{WCE}{NFP}$
WDEF	Weighted Development Errors per Function point	$WDEF = \frac{WDE}{NFP}$

- ***Error seveity metrics***

  The metrics belonging to this group are used to detect adverse situations of increasing numbers of severe errors in situations where errors and weight ederrors, as measured by error density metrics, are generally decreasing. Two error severity metrics are presented in Table 21.2.

Table 21.2: Error severity metrics

Code	Name	Calculation formula
ASCE	Average Severity of Code Errors	$ASCE = \frac{WCE}{NCE}$
ASDE	Average Severity of Development Errors	$ASDE = \frac{WDE}{NDE}$

Software process timetable metrics

Software process timetable metrics may be based on accounts of success (completion of milestones per schedule) in addition to failure events (noncompletion per schedule). An alternative approach calculates the average delay in completion of milestones. The metrics presented here are based on the two approaches illustrated in Table 21.3.

Table 21.3: Software process timetable metrics

Code	Name	Calculation formula
TTO	Time Table Observance	$TTO = \frac{MSOT}{MS}$
ADMC	Average Delay of Milestone Completion	$ADMC = \frac{TCDAM}{MS}$

- Key:
  - ■ MSOT = milestones completed on time.
  - ■ MS = total number of milestones.
  - ■ TCDAM = total Completion Delays (days, weeks, etc.) for All Milestones. To calculate this measure, delays reported for all relevant milestones are summed up. Milestones completed on time or before schedule are considered “O” delays. Some professionals refer to completion of milestones before schedule as “minus” delays. These are considered to balance the effect of accounted-for delays (we might call the latter “plus” delays). In these cases, the value of the ADMC may be lower than the value obtained according to the metric originally suggested.

Error removal effectiveness metrics
Software developers can measure the effectiveness of error removal by the software quality assurance system after a period of regular operation (usually 6 or 12 months) of the system. The metrics combine the error records of the development stage with the failures records compiled during the first year (or any defined period) of regular operation. Two error removal effectiveness metrics are presented in Table 21.4.

Table 21.4: Error removal effectiveness metrics

Code	Name	Calculation formula
DERE	Development Errors Removal Effectiveness	$DERE = \frac{NDE}{NDE + NYF}$
DWERE	Development Weighted Errors Removal Effectiveness	$DWERE = \frac{WDE}{WDE + WYF}$

Key:
- NYF = number of software failures detected during a year of maintenance service.
- WYF = weighted number of software failures detected during a year of maintenance service.

Software process productivity metrics This group of metrics includes "direct" metrics that deal with a project's human resources productivity as well as "indirect" metrics that focus on the extent of software use. Software reuse substantially affects productivity and effectiveness.

Table 21.5: Process productivity metrics

Code	Name	Calculation formula
DevP	Development Productivity	$DevP = \frac{DevH}{KLOC}$
FDevP	Function point Development Productivity	$FDevP = \frac{DevH}{NFP}$
CRe	Code Reuse	$CRe = \frac{ReKLOC}{KLOC}$
DocRe	Documentation Reuse	$DocRe = \frac{ReDoc}{NDoc}$

Key:
- DevH = total working hours invested in the development of the software system.
- ReKLOC = number of thousands of reused lines of code.
- ReDoc = number of reused pages of documentation.
- NDoc = number of pages of documentation.

Product metrics

- Product metrics refer to the software system's operational phase – years of regular use of the software system by customers, whether "internal" or "external" customers, who either purchased the software system or con-tracted for its evelopment. In most cases, the software developer is required to provide customer service during the software's

operational phase.

Customer services are of two main types:

■ Help desk services (HD) – software support by instructing customers regarding the method of application of the software and solution of customer implementation problems. Demand for these services depends to a great extent on the quality of the user interface (its “user friendliness”) as well as the quality of the user manual and integrated help menus.

■ Corrective maintenance services – correction of software failures identified by customers/users or detected by the customer service team prior to their discovery by customers. The number of software failures and their density are directly related to software development quality. For completeness of information and better control of failure correction, it is recommended that all software failures detected by the customer service team be recorded as corrective maintenance calls.

The array of software product metrics presented here is classified as follows:

■ HD quality metrics

■ HD productivity and effectiveness metrics

■ Corrective maintenance quality metrics

■ Corrective maintenance productivity and effectiveness metrics

HD quality metrics

The types of HD quality metrics discussed here deal with:

■ HD calls density metrics – the extent of customer requests for HD services as measured by the number of calls.

■ Metrics of the severity of the HD issues raised.

■ HD success metrics – the level of success in responding to these calls. A success is achieved by completing the required service within the time determined in the service contract.

HD calls density metrics

This section describes six different types of metrics. Some relate to the number of the errors and others to a weighted number of errors. As for size/volume measures of the software, some use number of lines of code while others apply function points. The sources of data for these and the other metrics in this group are HD reports. Three HD calls density metrics

for HD performance are presented in Table 21.6.

Product metrics(cont)

Table 21.6: HD calls density metrics

Code	Name	Calculation formula
HDD	HD calls Density	$HDD = \frac{NHYC}{KLMC}$
WHDD	Weighted HD calls Density	$WHDD = \frac{WHYC}{KLMC}$
WHDF	Weighted HD calls per Function point	$WHDF = \frac{WHYC}{NMFP}$

Key:
- NHYC = number of HD calls during a year of service.
- KLMC = thousands of lines of maintained software code.
- WHYC = weighted HD calls received during one year of service.
- NMFP = number of function points to be maintained.

- ***Severity of HD calls metrics****:* The metrics belonging to this group of measures aim at detecting one type of adverse situation: increasingly severe HD calls. The computed results may contribute to improvements in all or parts of the user interface (its "user friendliness") as well as the user manual and integrated help menus. We have selected one metric from this group for demonstration of how the entire category is employed. This metric, the Average Severity of HD Calls (ASHC), refers to failures detected during a period of one year (or any portion thereof, as appropriate):

WHYC

ASHC = ————

NHYC where WHYC and NHYC are defind as in Table 21.6.

Success of the HD services

The most common metric for the success of HD services is the capacity to solve problems raised by customer calls within the time determined in the service contract (*availability*). Thus, the metric for success of HD services compares the actual with the designated time for provision of these services.

HDS = ——————

NHYCwhere NHYOT = number of HD calls per year completed on time during one year of service.

- Corrective maintenance quality metrics Software corrective maintenance metrics deal with several aspects of the quality of maintenance services. A distinction is needed between software system failures treated by the maintenance teams and failures of the maintenance service that refer to cases where the maintenance failed to provide a repair that meets the designated standards or contract requirements. Thus, software maintenance metrics are classified as follows:

■ Software system failures density metrics – deal with the extent of demand for corrective maintenance, based on the records of failures identified during regular operation of the software system.

■ Software system failures severity metrics – deal with the severity of software system failures attended to by the corrective maintenance team.

Software quality metrics

■ Failures of maintenance services metrics – deal with cases where maintenance services were unable to complete the failure correction on time or that the correction performed failed.

■ Software system availability metrics – deal with the extent of disturbances caused to the customer as realized by periods of time where the services of the software system are unavailable or only partly available.

Software corrective maintenance productivity and effectiveness metrics While corrective maintenance productivity relates to the total of human resources invested in maintaining a given software system, corrective maintenance effectiveness relates to the resources invested in correction of a single failure. In other words, a software maintenance system displaying higher productivity will require fewer resources for its maintenance task, while a more effective software maintenance system will require fewer resources, on average, for correcting one failure. Three software corrective maintenance productivity and effectiveness metrics are presented in
Table 21.10.

Product metrics(cont)

Table 21.10: Software corrective maintenance productivity and effectiveness metrics

Code	Name	Calculation formula
CMaiP	Corrective Maintenance Productivity	$CMaiP = \frac{CMaiYH}{KLMC}$
FCMP	Function point Corrective Maintenance Productivity	$FCMP = \frac{CMaiYH}{NMFP}$
CMaiE	Corrective Maintenance Effectiveness	$CMaiE = \frac{CMaiYH}{NYF}$

Key:
- CMaiYH = total yearly working hours invested in the corrective maintenance of the software system.
- KLMC = thousands of lines of maintained software code.
- NMFP = number of function points designated for the maintained software.
- NYF = number of software failures detected during a year of maintenance service.

Implementation of software quality metrics

- The application of software quality metrics in an organization requires:

■ Definition of software quality metrics – relevant and adequate for teams, departments, etc.

■ Regular application by unit, etc.

■ Statistical analysis of collected metrics data.

■ Subsequent actions:

– Changes in the organization and methods of software development and maintenance units and/or any other body that collected the metrics data

– Change in metrics and metrics data collection

– Application of data and data analysis to planning corrective actions for all the relevant units.

- Definition of new software quality metrics

The definition of metrics involves a four–stage process:

(1) Definition of attributes to be measured: software quality, development team productivity, etc.

(2) Definition of the metrics that measure the required attributes and confirmation of its adequacy in complying with the requirements listed in Frame 21.2.

(3) Determination of comparative target values based on standards, previous year's performance, etc. These values serve as *indicators* of whether the unit measured (a team or an individual or a portion of the software) complies with the characteristics demanded of a given attribute.

(4) Determination of metrics application processes:

– Reporting method, including reporting process and frequency of reporting – Metrics data collection method.

The new metrics (updates, changes and revised applications) will be constructed following analysis of the metrics data as well as developments in the organization and its environment. The software quality metrics definition process is described in Figure 21.1.

Statistical analysis of metrics data Analysis of metrics data provides opportunities for comparing a series of project metrics. For the metrics data to be a valuable part of the SQA process, statistical analysis is required of the metrics' results. Statistical tools provide us with two levels of support, based on the type of statistics used:

■ Descriptive statistics

■ Analytical statistics.

Limitations of software metrics

Application of quality metrics is strewn with obstacles. These can be grouped as follows:

■ Budget constraints in allocating the necessary resources (manpower, funds,etc.) for development of a

quality metrics system and its regular application.

■ Human factors, especially opposition of employees to evaluation of their activities.

■ Uncertainty regarding the data's validity, rooted in partial and biased reporting.

Quality Management Standards

- Certification standards vary from assessment standards by content as well as by emphasis.

The scope of certification standards is determined by the aims of certification, which are to:

■ Enable a software development organization to demonstrate consistent ability to assure that its software products or maintenance services comply with acceptable quality requirements. This is achieved by certification granted by an external body.

■ Serve as an agreed basis for customer and supplier evaluation of the supplier's quality management system. This may be accomplished by customer performance of a quality audit of the supplier's quality management system. The audit will be based on the certification standard's requirements.

■ Support the software development organization's efforts to improve quality management system performance and enhance customer satisfaction through compliance with the standard's requirements.

- The scope of assessment standards is also determined by the aims fo assessment, which are to:

■ Serve software development and maintenance organizations as a tool forself-assessment of their ability to carry out software development projects.

■ Serve as a tool for improvement of development and maintenance processes. The standard indicates directions for process improvements.

■ Help purchasing organizations determine the capabilities of potential suppliers.

■ Guide training of assessors by delineating qualifications and training program curricula.

To sum up, while the certification standards emphasis is external – to support the supplier–customer relationships – the emphasis of the assessment

standards is internal because it focuses on software process improvement.

The scope of assessment standards is also determined by the aims fo assessment, which are to:

■ Serve software development and maintenance organizations as a tool for self-assessment of their ability to carry out software development projects.

■ Serve as a tool for improvement of development and maintenance processes. The standard indicates directions for process improvements.

■ Help purchasing organizations determine the capabilities of potential suppliers.

■ Guide training of assessors by delineating qualifications and training program curricula.

To sum up, while the certification standards emphasis is external – to support the supplier–customer relationships – the emphasis of the assessment

standards is internal because it focuses on software process improvement.

ISO 9001 and ISO 9000-3

ISO 9000-3, the Guidelines offered by the International Organization for Standardization (ISO), represent implementation of the general methodology of quality management ISO 9000 Standards to the special case of software development and maintenance. Both ISO 9001 and ISO 9000-3 are reviewed and updated once every 5–8 years, with each treated separately. As ISO 9000-3 adaptations are based on those introduced to ISO 9001, publication of the revised Guidelines follows publication of the revised Standard by a few years.

ISO 9000-3 quality management system: guiding principles

Eight principles guide the new ISO 9000-3 standard; these were originally set down in the ISO 9000:2000 standard (ISO, 2000b), as follows:

(1) **Customer focus**. Organizations depend on their customers and therefore should understand current and future customer needs.

(2) **Leadership.** Leaders establish the organization's vision. They should create and maintain an internal environment in which people can become fully involved in achieving the organization's objectives via the designated route.

(3) **Involvement of people.** People are the essence of an organization; their full involvement, at all levels of the organization, enables their abilities to be applied for the organization's benefit.

(4) **Process approach**. A desired result is achieved more efficiently when activities and resources are managed as a process.

(5) **System approach to management**. Identifying, understanding and managing processes, if viewed as a system, contributes to the organization's effectiveness and efficiency.

(6) **Continual improvement**. Ongoing improvement of overall performance should be high on the organization's agenda.

(7) **Factual approach to decision making.** Effective decisions are based on the analysis of information.

(8) **Mutually supportive supplier relationships.** An organization and its suppliers are interdependent; a mutually supportive relationship enhances the ability of both to create added value.

ISO 9000-3: requirements

The current standard edition of ISO, 9000-3 (ISO 1997) includes 20 requirements that relate to the various aspects of software quality management systems. The new ISO 9000-3 (ISO/IEC, 2001) offers a new structure, with its 22 requirements classified into the following five groups:

■ Quality management system

■ Management responsibilities

■ Resource management

■ Product realization

■ Management, analysis and improvement.

Table 23.1: ISO 9000-3 new edition – Requirements and their classification

Requirement class	Requirement subjects
4. Quality management system	4.1 General requirements 4.2 Documentation requirements
5. Management responsibilities	5.1 Management commitments 5.2 Customer focus 5.3 Quality policy 5,4 Planning 5.5 Responsibility, authority and communication 5.6 Management review
6. Resource management	6.1 Provision of resources 6.2 Human resources 6.3 Infrastructure 6.4 Work environment
7. Product realization	7.1 Planning of product realization 7.2 Customer-related processes 7.3 Design and development 7.4 Purchasing 7.5 Production and service provision 7.6 Control of monitoring and measuring devices
8. Measurement, analysis and improvement	8.1 General 8.2 Monitoring and measurement 8.3 Control of non-conforming product 8.4 Analysis of data 8.5 Improvement

Source: ISO (2000a)

ISO 9001 — application to software: the TickIT initiative

TickIT was launched in the late 1980s by the UK software industry in cooperation with the UK Department for Trade and Industry to promote development of a methodology for adapting ISO 9001 to the characteristics of the software industry known as the *TickIT initiative*. At the time of its launch, ISO 9001 had already been successfully applied in manufacturing industry; however, no significant methodology for its application to the special characteristics of the software industry was yet available. In the years to follow, the TickIT initiative, together with the efforts invested in development of ISO 9000-3, achieved this goal.

TickIT is, additionally, a leading provider of ISO 9001 certification, specializing in information technology (IT); it covers the entire range of commercial software development and maintenance services. TickIT, now managed and maintained by the DISC Department of BSI (the British Standards Institute), is accredited for certification of IT organizations in the UK and Sweden. In June 2002, TickIT reported a clientele of 1252 organizations in 42 countries, the majority in the UK (882), Sweden (54) and the United States (109). TickIT is currently authorized to accredit other organizations as certification bodies for the software industry in the UK.

TickIT activities include:

■ Publication of the *TickIT Guide*, that supports the software industry's efforts to spread ISO 9001 certification. The current guide (edition 5.0, TickIT, 2001), which includes references to ISO/IEC 12207 and ISO/IEC 15504, is distributed to all TickIT customers.

■ Performance of audit-based assessments of software quality systems and consultation to organizations on improvement of software development and maintenance processes in addition to their management.

■ Conduct of ISO 9000 certification audits.

TickIT auditors who conduct audit-based assessments and certification audits are registered by the International Register of Certificated Auditors (IRCA).

Capability Maturity Models – CMM and CMMI assessment methodology

Carnegie Mellon University's Software Engineering Institute (SEI) took the initial steps toward development of what is termed a *capability maturity model* (CMM) in 1986, when it released the first brief description of the maturity process framework. The initial version of the CMM was released in 1992, mainly for receipt of feedback from the software community. The first version for public use was released in 1993

- The principles of CMM

CMM assessment is based on the following concepts and principles:

■ Application of more elaborate management methods based on quantitative approaches increases the organization's capability to control the quality and improve the productivity of the software development process.

■ The vehicle for enhancement of software development is composed of the five-level capability maturity model. The model enables an organization to evaluate its achievements and determine the efforts needed to reach the next capability level by locating the process areas requiring improvement.

■ Process areas are generic; they define the "what", not the "how". This approach enables the model to be applied to a wide range of implementation organizations because:

– It allows use of any life cycle model

– It allows use of any design methodology, software development tool and programming language

– It does not specify any particular documentation standard.

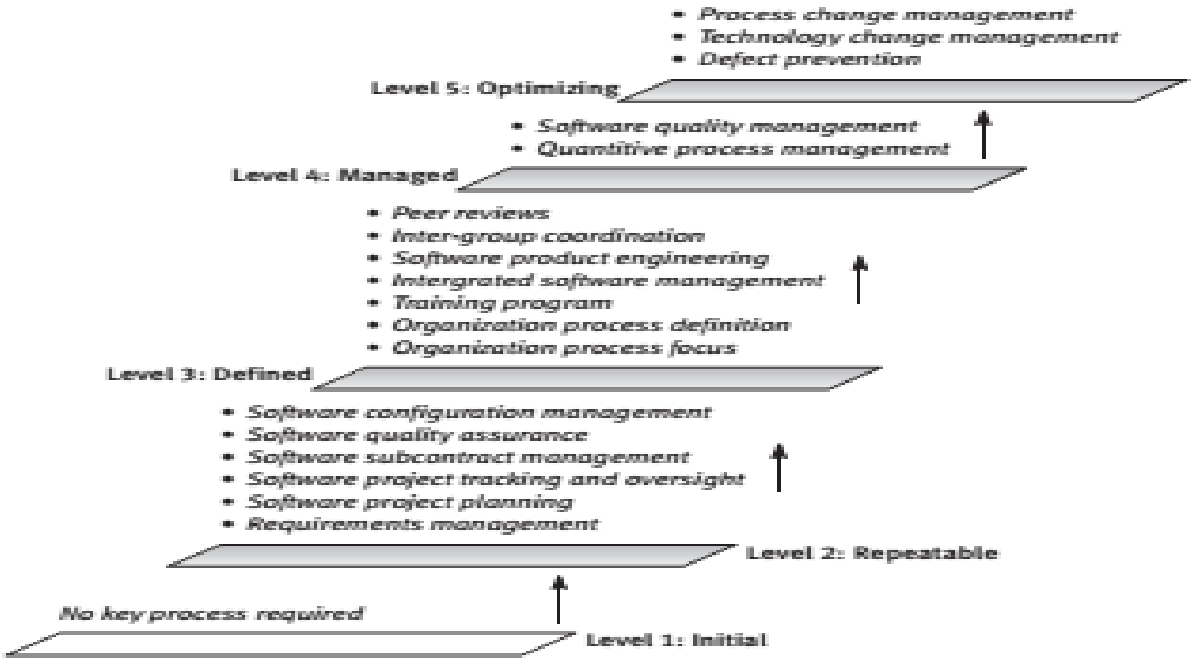


Figure 23.2: The CMM model levels and key process areas (KPAs)
Source: After Paulk *et al.* (1995)

- The evolution of CMM

After 1993, the SEI expanded the original Software Development and Maintenance Capability Maturity Model (SW-CMM) through diversification. Its main structure was retailored to fit a variety of specialized capability maturity models. The following variants have been developed:

■ **System Engineering CMM (SE-CMM)** focuses on system engineering practices related to product-oriented customer requirements. It deals with product development: analysis of requirements, design of product systems, management and coordination of the product systems and their integration. In addition, it deals with the production of the developed product: planning production lines and their operation.

■ **Trusted CMM (T-CMM)** was developed to serve sensitive and classified software systems that require enhanced software quality assurance.

■ **System Security Engineering CMM (SSE-CMM)** focuses on security aspects of software engineering and deals with secured product development processes, including security of development team members.

■ **People CMM (P-CMM)** deals with human resource development in software organizations: improvement of professional capacities, motivation, organizational structure, etc.

■ **Software Acquisition CMM (SA-CMM)** focuses on special aspects of software acquisition by treating issues – contract tracking, acquisition risk management, quantitative acquisition management, contract performance management, etc. – that touch on software purchased from external organizations.

■ **Integrated Product Development CMM (IPD-CMM)** serves as a framework for integration of development efforts related to every aspect of the product throughout the product life cycle as invested by each department.

The CMMI structure and processes areas

The CMMI model, like the original CMM models, is composed of five levels. The CMMI capability levels are the same as those of the original, apart from a minor change related to capability level 4, namely:

■ Capability maturity level 1: Initial

■ Capability maturity level 2: Managed

■ Capability maturity level 3: Defined

■ Capability maturity level 4: Quantitatively managed

■ Capability maturity level 5: Optimizing.

The Bootstrap methodology

The Bootstrap Institute, a non-profit organization that operates in Europe as part of the European Strategic Program for Research in Information Technology (ESPRIT) in cooperation with the European Software Institute (ESI), offers another route for professional SQA support to organizations, based on its Bootstrap methodology. The Bootstrap Institute provides various types of support to its licensed members:

(1) Access to the Bootstrap methodology for assessment and improvement of software development processes. The Institute constantly updates and improves its methodology.

(2) Training and accreditation of assessors.

(3) Access to the Bootstrap database.

The Bootstrap methodology measures the maturity of an organization and its projects on the basis of 31 quality attributes grouped into three classes: process, organization and technology. A five-grade scale is applied to each of the quality attributes separately. The methodology facilitates detailed assessment of the software development process by evaluating its achievements with respect to each attribute and indicates the improvements required in the software development process and in projects. The assessment options include:

■ Evaluation of the current position of the software quality assurance system as a basis for improvement initiation

■ Evaluation of level of achievements according to the Capability Maturity Model (CMM)

■ Evaluation of achievements according to ISO 15504 (the SPICE project)

■ ISO 9000-3 gap assessment to support preparations for a certification audit.

Bootstrap trains three levels of registered assessors, namely trained assessor, assessor and lead assessor. A person can become a registered lead assessor, having overall responsibility for planning and performing a Bootstrap assessment, only after successfully performing as a trained and then a registered assessor. In order to become a trained assessor, a person has to successfully complete a basic assessor training program, after which she or he can participate in Bootstrap assessments. Trained assessors who have demonstrated knowledge in performance of assessments and been recommended by a registered lead assessor may qualify as a registered assessor.

Registered assessors are likewise required to demonstrate knowledge and competence in carrying out higher-level assessments in addition to participation in a lead assessors' training course. Only then can they applying for acceptance as lead assessors. The process is illustrated in Figure 23.3.

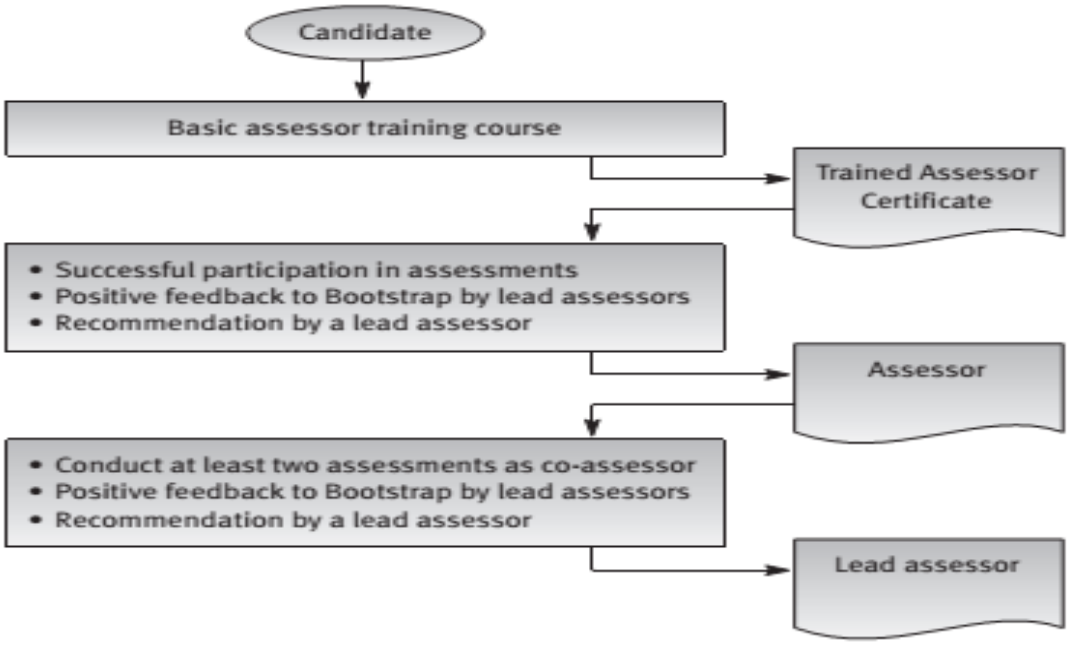


Figure 23.3: Bootstrap assessor accreditation process

The SPICE project and the ISO/IEC 15504 software process assessment standard

The parallel development of several software process assessment methodologies raised difficulties of non-standardization. A joint initiative by ISO and IEC, the SPICE (Software Process Improvement for Capability Determination) Project was established in 1993 to overcome this problem by developing a standard software process assessment methodology.

The SPICE Project released its Version 1.0 report in 1995, which became the basis for the development of the TR (technical report) version of the ISO/IEC 15504 Standard released in 1998.

The next stage in the development of the ISO/IEC 15504 Standard will be its release as an international standard. An ISO/IEC working group has been assigned the responsibility of introducing the revisions required to transform the standard from technical report status to international standard status. The working group has solicited revision proposals from the public (through a special website) as well as from national bodies. Another route taken to identify features demanding revision was the conduct of a major three-phase trial within the framework of the SPICE Project.

The next sections are dedicated to the following subjects:

- Principles behind the ISO/IEC 15504 assessment model
- Structure of the ISO/IEC 15504 assessment model
- Content of the ISO/IEC 15504 assessment model

- ISO/IEC 15504 processes
- ISO/IEC 15504 trials