CHAPTER 3: SOFTWARE QUALITY IN PERSPECTIVE

The general view of software testing is that it is an activity to "find bugs." The author believes the objectives of software testing are to qualify a software program's quality by measuring its attributes and capabilities against expectations and applicable standards. Software testing also provides valuable information to the software development effort.

Software quality is something everyone wants. Managers know that they want high quality, software developers know they want to produce a quality product, and users insist that software work consistently and be reliable.

Many software quality groups develop software quality assurance plans, which are like test plans. However, a software quality assurance plan may include a variety of activities beyond those included in a test plan. Although the quality assurance plan encompasses the entire quality scope, the test plan is one of the quality control tools of the quality assurance plan.

The objectives of this section are to:

- Describe a brief history of software testing.
- Define quality and its cost.
- Differentiate quality prevention from quality detection.
- Differentiate verification from validation.
- Outline the components of quality assurance.
- Outline common testing techniques.
- Describe how the continuous improvement process can be instrumental in achieving quality.

Section 1. A Brief History of Software Testing

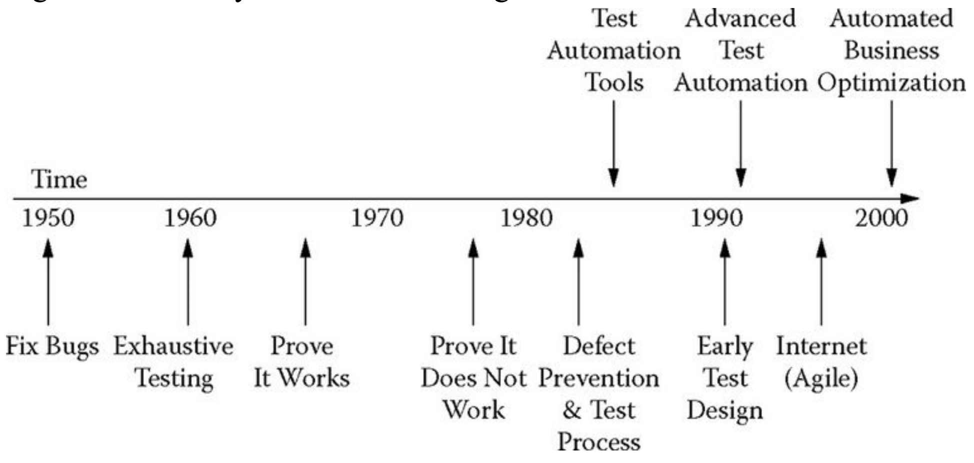
Overview

Modern testing tools are becoming more and more advanced and user-friendly. The following describes how software testing activity has evolved, and is evolving, over time. This sets the perspective on where automated testing tools are going.

Software testing is the activity of running a series of dynamic executions of software programs after the software source code has been developed. It is performed to uncover and correct as many potential errors as possible before delivery to the customer. As pointed out earlier, software testing is still an "art." It can be considered a risk management technique; the quality assurance technique, for example, represents the last defense to correct deviations from errors in the specification, design, or code.

Throughout the history of software development, there have been many definitions and advances in software testing. Figure 1.1 graphically illustrates these evolutions. In the 1950s, software testing was defined as "what programmers did to find bugs in their programs." In the early 1960s the definition of testing underwent a revision. Consideration was given to exhaustive testing of the software in terms of the possible paths through the code, or total enumeration of the possible input data variations. It was noted that it was impossible to completely test an application because (1) the domain of program inputs is too large, (2) there are too many possible input paths, and (3) design and specification issues are difficult to test. Because of the foregoing points, exhaustive testing was discounted and found to be theoretically impossible.

Figure 1.1: History of Software Testing



As software development matured through the 1960s and 1970s, the activity of software development was referred to as "computer science." Software testing was defined as "what is done to demonstrate correctness of a program" or as "the process of establishing confidence that a program or system does what it is supposed to do" in the early 1970s. A short-lived computer science technique that was proposed during the specification, design, and implementation of a software system was software verification through "correctness proof." Although this concept was theoretically promising, in practice it was too time consuming and insufficient. For simple tests, it was easy to show that the software "works" and prove that it will theoretically work. However, because most of the software was not tested using this approach, many defects remained to be discovered during actual implementation. It was soon concluded that "proof of correctness" was an inefficient method of software testing. However, even today there is still a need for correctness demonstrations, such as acceptance testing, as described in various sections of this book.

In the late 1970s it was stated that testing is a process of executing a program with the intent of finding an error, not proving that it works. The new definition emphasized that a good test case is one that has a high probability of finding an as-yet-undiscovered error. A successful test is one that uncovers an as-yet-undiscovered error. This approach was the exact opposite of that followed up to this point.

The foregoing two definitions of testing (prove that it works versus prove that it does not work) present a "testing paradox" with two underlying and contradictory objectives:

1. To give confidence that the product is working well
2. To uncover errors in the software product before its delivery to the customer (or the next state of development)

If the first objective is to prove that a program works, it was determined that "we shall subconsciously be steered toward this goal; that is, we shall tend to select test data that have a low probability of causing the program to fail."

If the second objective is to uncover errors in the software product, how can there be confidence that the product is working well, since it was just proved that it is, in fact, not working! Today it has been widely accepted by good testers that the second objective is more productive than the first objective, for if one accepts the first one, the tester will subconsciously ignore defects trying to prove that a program works.

The following good testing principles were proposed:

- A necessary part of a test case is a definition of the expected output or result.
- Programmers should avoid attempting to test their own programs.
- A programming organization should not test its own programs.
- Thoroughly inspect the results of each test.
- Test cases must be written for invalid and unexpected, as well as valid and expected input conditions.
- Examining a program to see if it does not do what it is supposed to do is only half the battle. The other half is seeing whether the program does what it is not supposed to do.
- Avoid throwaway test cases unless the program is truly a throwaway program.
- Do not plan a testing effort under the tacit assumption that no errors will be found.
- The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.

The 1980s saw the definition of testing extended to include defect prevention. Designing tests is one of the most effective bug prevention techniques known. It was suggested that a testing methodology was required, specifically, that testing must include reviews throughout the entire software development life cycle and that it should be a managed process. Promoted was the importance of testing not just a program but the requirements, design, code, tests themselves, and the program.

"Testing" traditionally (up until the early 1980s) referred to what was done to a system once working code was delivered (now often referred to as system testing); however, testing today is "greater testing," in which a tester should be involved in almost every aspect of the software development life cycle. Once code is delivered to testing, it can be tested and checked, but if anything is wrong, the previous development phases must be investigated. If the error was caused by a design ambiguity, or a programmer oversight, it is simpler to try to find the problems as soon as they occur, not wait until an actual working product is produced. Studies have shown that about 50 percent of bugs are created at the requirements (what do we want the software to do?) or design stages, and these can have a compounding effect and create more bugs during coding. The earlier a bug or issue is found in the life cycle, the cheaper it is to fix (by exponential amounts). Rather than test a program and look for bugs in it, requirements or designs can be rigorously reviewed. Unfortunately, even today, many software development organizations believe that software testing is a back-end activity.

In the mid-1980s, automated testing tools emerged to automate the manual testing effort to improve the efficiency and quality of the target application. It was anticipated that the computer could perform more tests of a program than a human could perform manually, and more reliably. These tools were initially primitive and did not have advanced scripting language facilities (see the section, "Evolution of Automated Testing Tools," later in this chapter for more details).

In the early 1990s the power of early test design was recognized. Testing was redefined to be "planning, design, building, maintaining, and executing tests and test environments." This was a quality assurance perspective of testing that assumed that good testing is a managed process, a total life-cycle concern with testability.

Also, in the early 1990s, more advanced capture/replay testing tools offered rich scripting languages and reporting facilities. Test management tools helped manage all the artifacts from requirements and test design, to test scripts and test defects. Also, commercially available performance tools arrived to test system performance. These tools tested stress and load-tested the target system to determine their breaking points. This was facilitated by capacity planning.

Although the concept of a test as a process throughout the entire software development life cycle has persisted, in the mid-1990s, with the popularity of the Internet, software was often developed without a specific testing standard model, making it much more difficult to test. Just as documents could be reviewed without specifically defining each expected result of each step of the review, so could tests be performed without explicitly defining everything that had to be tested in advance. Testing approaches to this problem are known as "agile testing." The testing techniques include exploratory testing, rapid testing, and risk-based testing.

In the early 2000s Mercury Interactive (now owned by Hewlett-Packard [HP]) introduced an even broader definition of testing when they introduced the concept of business technology optimization (BTO). BTO aligns the IT strategy and execution with business goals. It helps govern the priorities, people, and processes of IT. The basic approach is to measure and maximize value across the IT service delivery life cycle to ensure applications meet quality, performance, and availability goals. Interactive digital cockpit revealed vital business availability information in real-time to help IT and business executives prioritize IT operations and maximize business results. It provided end-to-end visibility into business availability by presenting key business process indicators in real-time, as well as their mapping to the underlying IT infrastructure.

Historical Software Testing and Development Parallels

In some ways, software testing and automated testing tools are following similar paths as traditional development. The following is a brief evolution of software development and shows how deviations from prior best practices are also being observed in the software testing process.

The first computers were developed in the 1950s, and FORTRAN was the first 1GL programming language. In the late 1960s, the concept of "structured programming" stated that any program can be written using three simple constructs: simple sequence, if-then-else, and do while statements. There were other prerequisites such as the program being a "proper program" whereby there must exist only one entry and one exit point. The focus was on the process of creating programs.

In the 1970s the development community focused on design techniques. They realized that structured programming was not enough to ensure quality—a program must be designed before it can be coded. Techniques such as Yourdon's, Myers', and Constantine's structured design and composite design techniques flourished and were accepted as best practice. The focus still had a process orientation.

The philosophy of structured design was *partitioning* and *organizing* the pieces of a system. By partitioning is meant the division of the problem into smaller sub-problems, so that each subproblem will eventually correspond to a piece of the system. Highly interrelated parts of the problem should be in the same piece of the system; that is, things that belong together should go together. Unrelated parts of the problem should reside in unrelated pieces of the system; for example, things that have nothing to do with one another do not belong together.

In the 1980s, it was determined that structured programming and software design techniques were still not enough: the requirements for the programs must first be established for the right system to be delivered to the customer. The focus was on quality that occurs when the customer receives exactly what he or she wanted in the first place.

Many requirement techniques emerged, such as data flow diagrams (DFDs). An important part of a DFD is a *store*, a representation of where the application data will be stored. The concept of a store motivated practitioners to develop a logical-view representation of the data. Previously the focus was on the physical view of data in terms of the database. The concept of a data model was then created: a simplified description of a real-world system in terms of data, for example, a logical view of data. The components of this approach included entities, relationships, cardinality, referential integrity, and normalization. These also created a controversy as to which came first: the process or data, a chicken-and-egg argument. Prior to the logical representation of data, the focus was on the processes that interfaced to databases. Proponents of the logical view of data initially insisted that the data was the first analysis focus point and then the process. With time, it was agreed that both the process and data must be considered jointly in defining the requirements of a system.

In the mid-1980s, the concept of information engineering was introduced. It was a new discipline that led the world into the information age. With this approach, there is more interest in understanding how information can be stored and represented, how information can be transmitted through networks in multimedia forms, and how information can be processed for various services and applications. Analytical problem-solving techniques, with the help of mathematics and other related theories, were applied to the engineering design problems. Information engineering stressed the importance of taking an enterprise view of application development rather than a specific application. By modeling the entire enterprise in terms of processes, data, risks, critical success factors, and other dimensions, it was proposed that management would be able to manage the enterprise in a more efficient manner.

During this same time frame, fourth-generation computers embraced microprocessor chip technology and advanced secondary storage at fantastic rates, with storage devices holding tremendous amounts of data. Software development techniques had vastly improved, and 4GLs made the development process much easier and faster. Unfortunately, the emphasis on quick turnaround of applications led to a backward trend of fundamental development techniques to “get the code out” as quickly as possible. This led to reducing the emphasis on requirement and design and persists today in many software development organizations.

Extreme Programming

Extreme programming (XP) is an example of such a trend. XP is an unorthodox approach to software development, and it has been argued that it has no design aspects. The extreme programming methodology proposes a radical departure from commonly accepted software development processes. There are really two XP rules: (1) Do a Little Design and (2) No Requirements, Just User Stories. Extreme programming disciples insist that “there really are no rules, just suggestions. XP methodology calls for small units of design, from ten minutes to half an hour, done periodically from one day between sessions to a full week between sessions. Effectively, nothing gets designed until it is time to program it.”

Although most people in the software development business understandably consider requirements documentation to be vital, XP recommends the creation of as little documentation as possible. No up-front requirement documentation is created in XP, and very little is created in the software development process.

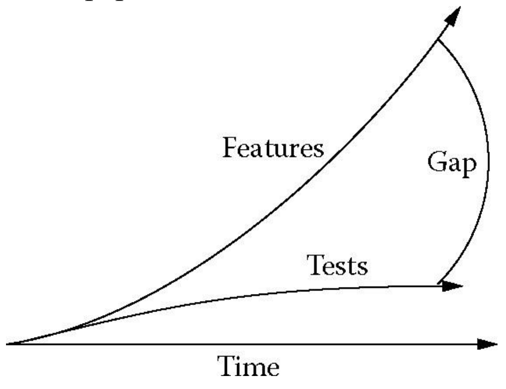
With XP, the developer comes up with test scenarios before she does anything else. The basic premise behind test-first design is that the test class is written before the real class; thus, the end purpose of the real class is not simply to fulfill a requirement, but simply to pass all the tests that are in the test class. The problem with this approach is that independent testing is needed to find out things about the product the developer did not think about or was not able to discover during her own testing.

Evolution of Automated Testing Tools

Test automation started in the mid-1980s with the emergence of automated capture/replay tools. A capture/replay tool enables testers to record interaction scenarios. Such tools record every keystroke, mouse movement, and response that was sent to the screen during the scenario. Later, the tester may replay the recorded scenarios. The capture/replay tool automatically notes any discrepancies in the expected results. Such tools improved testing efficiency and productivity by reducing manual testing efforts.

The cost justification for test automation is simple and can be expressed in a single figure (Figure 1.2). As this figure suggests, over time the number of functional features for a particular application increase owing to changes and improvements to the business operations that use the software. Unfortunately, the number of people and the amount of time invested in testing each new release either remain flat or may even decline. As a result, the test functional coverage steadily decreases, which increases the risk of failure, translating to potential business losses.

Figure 1.2: Motivation for test automation. (From "Why Automate," Linda Hayes, Worksoft, Inc. white paper, 2002, www.worksoft.com. With permission.)



For example, if the development organization adds application enhancements equal to 10 percent of the existing code, this means that the test effort is now 110 percent as great as it was before. Because no organization budgets more time and resources for testing than they do for development, it is literally impossible for testers to keep up.

This is why applications that have been in production for years often experience failures. When test resources and time cannot keep pace, decisions must be made to omit the testing of some functional features. Typically, the newest features are targeted because the oldest ones are assumed to still work. However, because changes in one area often have an unintended impact on other areas, this assumption may not be true. Ironically, the greatest risk is in the existing features, not the new ones, for the simple reason that they are already being used.

Test automation is the only way to resolve this dilemma. By continually adding new tests for new features to a library of automated tests for existing features, the test library can track the application functionality.

The cost of failure is also on the rise. Whereas in past decades software was primarily found in back-office applications, today software is a competitive weapon that differentiates many companies from their competitors and forms the backbone of critical operations. Examples abound of errors in the tens or hundreds of millions—even billions—of dollars in losses due to undetected software errors. Exacerbating the increasing risk is the decreasing cycle times. Product cycles have compressed from years into months, weeks, or even days. In these tight time frames, it is virtually impossible to achieve acceptable functional test coverage with manual testing.

Section 2. Quality Assurance Framework

What is Quality?

In Webster's dictionary, quality is defined as "the essential character of something, an inherent or distinguishing character, degree, or grade of excellence." If you look at the computer literature, you will see that there are two generally accepted meanings of quality. The first is that quality means "meeting requirements." With this definition, to have a quality product, the requirements must be measurable, and the product's requirements will either be met or not met. With this meaning, quality is a binary state; that is, a product is either a quality product or it is not. The requirements may be complete, or they may be simple, but as long as they are measurable, it can be determined whether quality requirements have or have not been met. This is the producer's view of quality as meeting the producer's requirements or specifications. Meeting the specifications becomes an end.

Another definition of quality, the customer's, is the one we use. With this definition, the customer defines quality as to whether the product or service does what the customer needs. Another way of wording it is "fit for use." There should also be a description of the purpose of the product, typically documented in a customer's "requirements specification" (see "Requirements Specification," for more details). The requirements are the most important document, and the quality system revolves around it. In addition, quality attributes are described in the customer's requirements specification. Examples include usability, the relative ease with which a user communicates with the application; portability, the capability of the system to be executed across a diverse range of hardware architectures; and reusability, the ability to transfer software components constructed in one software system into another.

Everyone is committed to quality; however, the following show some of the confusing ideas shared by many individuals that inhibit achieving a quality commitment:

- Quality requires a commitment, particularly from top management. Close cooperation between management and staff is required to make it happen.
- Many individuals believe that defect-free products and services are impossible and accept certain levels of defects as normal and acceptable.
- Quality is frequently associated with cost, meaning that high quality equals high cost. This is a confusion between quality of design and quality of conformance.
- Quality demands requirement specifications in sufficient detail that the products can be quantitatively measured against those specifications. Many organizations are not capable or willing to expend the effort to produce specifications at the level of detail required.
- Technical personnel often believe that standards stifle their creativity, and thus do not abide by standards compliance. However, to ensure quality, well-defined standards and procedures must be followed.

Prevention versus Detection

Quality cannot be achieved by simply assessing a finished product; it must be built into the process from the start to prevent defects. Quality assurance measures include using development standards, methods, and tools to ensure products are assessable. Independent testing has emerged to address undetected bugs that have caused significant losses.

Process assessments, like coding standards, backup procedures, and defect management, are crucial for quality management. Effective quality management lowers costs, as fixing defects early is cheaper. Automated testing tools also improve quality and reduce long-term maintenance costs.

The total cost of quality management includes prevention, inspection, internal failure, and external failure costs, with prevention offering the greatest return by reducing defects, improving quality, and lowering production costs.

Verification versus Validation

Verification ensures a product meets the specified requirements throughout the development process, while validation confirms the product fulfills the customer's needs at the end. Testing is closely linked to validation, traditionally viewed as a process that occurs after programming to check performance. However, integrating verification throughout the development life cycle improves results by combining it with validation.

Verification involves systematic reviews, analysis, and testing from the requirements phase through coding, ensuring software reliability and quality. Originating from the aerospace industry, verification focuses on preventing defects by ensuring the software performs all intended functions without degrading overall system performance.

Verification also ensures traceability between documentation and requirements, continuously improving software quality. Though initially costly, verification reduces long-term costs by minimizing defects and ensuring efficient error correction, leading to significant savings throughout the software's life cycle.

Software Quality Assurance
Software quality assurance (SQA) refers to systematic activities that provide evidence that a software product is fit for use and meets its requirements. It is achieved by following established quality control guidelines to ensure the software's integrity and longevity. SQA is often confused with quality control, auditing, and software testing, but they serve distinct roles.

- ✓ Quality assurance ensures processes are continuously improved to meet specifications.
- ✓ Quality control compares product quality to standards and addresses nonconformance.
- ✓ Auditing verifies compliance with policies and procedures.

SQA is a planned effort to ensure that software meets project-specific criteria, such as portability, efficiency, and reusability. It involves collaboration across the project team, not just the quality assurance group. The goal is to foster a culture of quality and ensure that all processes lead to a high-quality product.

The SQA plan outlines methods to ensure quality at each project milestone, providing management with documentation of steps taken to achieve quality. It sets realistic goals for monitoring quality, rather than striving for unrealistic standards like zero defects.

Software quality assurance is a strategy for risk management. It exists because software quality is typically costly and should be incorporated into the formal risk management of a project. Some examples of poor software quality include the following:

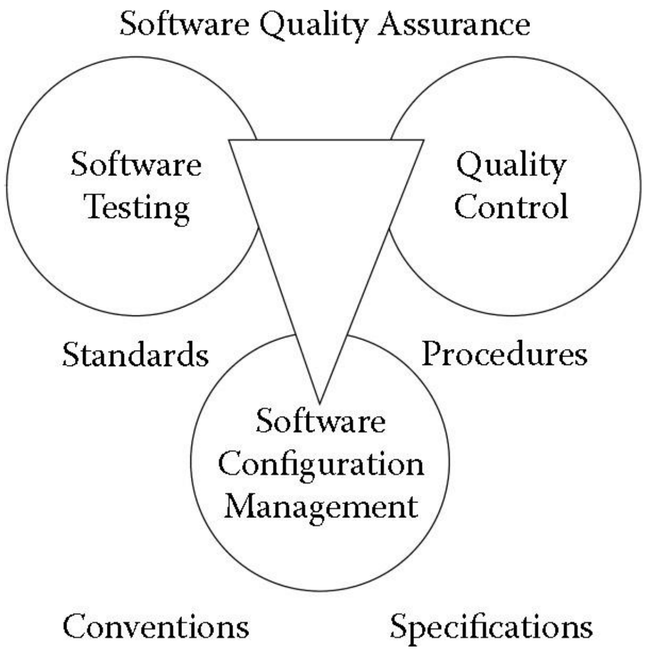
- Delivered software frequently fails.
- Consequences of system failure are unacceptable, from financial to life-threatening scenarios.
- Systems are often not available for their intended purpose.
- System enhancements are often very costly.
- Costs of detecting and removing defects are excessive.

Although most quality risks are related to defects, this only tells part of the story. A defect is a failure to comply with a requirement. If the requirements are inadequate or even incorrect, the risks of defects are more pervasive. The result is too many built-in defects and products that are not verifiable. Some risk management strategies and techniques include software testing, technical reviews, peer reviews, and compliance verification.

Components of Quality Assurance

Most software quality assurance activities can be categorized into software testing (that is, verification and validation), software configuration management, and quality control. However, the success of a software quality assurance program also depends on a coherent collection of standards, practices, conventions, and specifications, as shown in Figure 2.1.

Figure 2.1: Quality assurance components



Software Testing

Software testing is a popular risk management strategy. It is used to verify that functional requirements were met. The limitation of this approach, however, is that by the time testing occurs, it is too late to build quality into the product. Tests are only as good as the test cases, but they can be inspected to ensure that all the requirements are tested across all possible combinations of inputs and system states. However, not all defects are discovered during testing. Software testing includes the activities outlined in this text, including verification and validation activities. In many organizations, these activities, or their supervision, are included within the charter for the software quality assurance function. The extent to which personnel independent of design and coding should participate in software quality assurance activities is a matter of institutional, organizational, and project policy.

The major purpose of verification and validation activities is to ensure that software design, code, and documentation meet all the requirements imposed on them. Examples of requirements include user requirements; specifications derived from and designed to meet user requirements; code review and inspection criteria; test requirements at the modular, subsystem, and integrated software levels; and acceptance testing of the code after it has been fully integrated with hardware. During software design and implementation, verification helps determine whether the products of one phase of the software development life cycle fulfill the requirements established during the previous phase. The verification effort takes less time and is less complex when conducted throughout the development process.

Quality Control

Quality control (QC) refers to the processes and methods used to monitor and ensure that a product meets specified requirements, focusing on detecting and removing defects before product delivery. QC is typically the responsibility of the unit producing the product and can involve either the production team itself or a designated QC department within that unit.

QC involves well-defined checks at various stages of the product's life cycle, such as reviews, code inspections, and user deliverable checks, all outlined in the quality assurance (QA) plan. Inspections, conducted by peers, experts, or a configuration control board, assess whether the product adheres to project standards and milestones, identifying defects and recommending improvements. Inspection responsibilities and schedules are defined in the QA plan.

While QC focuses on detecting and fixing defects, QA is a managerial function aimed at preventing them by improving processes, advising restraint, and ensuring products meet standards from the outset.

Software Configuration Management

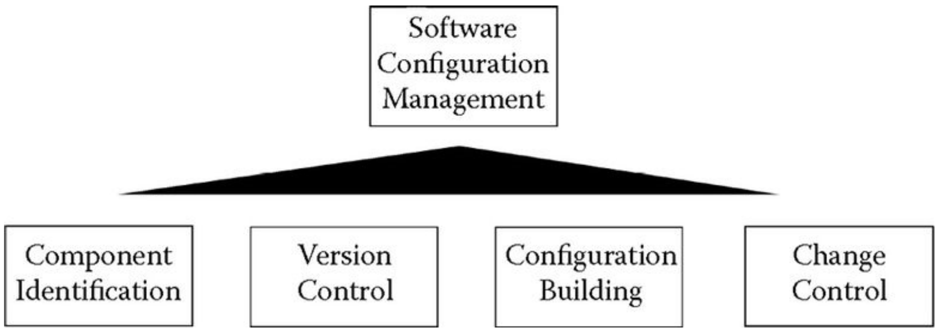
Software configuration management (SCM) focuses on labeling, tracking, and controlling changes in software elements of a system, managing the evolution of software components and their relationships. The goal of SCM is to identify and control the evolution of interrelated software components throughout various life-cycle phases, which can include development, document control, problem tracking, change control, and maintenance. By managing versions and relationships, SCM supports software reusability, leading to cost savings.

SCM involves activities that ensure design and code are defined and changes are controlled through reviews, maintaining consistency between the code and its documentation. This approach prevents unreviewed, last-minute changes. In concurrent software development, SCM organizes software, reduces the likelihood of unintended changes, and stabilizes software when there is significant change activity or risk of incorrect component selection.

Elements of Software Configuration Management

Software configuration management identifies a system configuration to systematically control changes, maintain integrity, and enforce tractability of the configuration throughout its life cycle. Components to be controlled include planning, analysis, and design documents, source code, executable code, utilities, job control language (JCL), test plans, test scripts, test cases, and development reports. The software configuration process typically consists of four elements: software component identification, software version control, configuration building, and software change control, as shown in Figure 2.2.

Figure 2.2: Software configuration management



Component Identification

A key activity in software configuration management (SCM) is identifying the software components that form a deliverable at each stage of development. SCM provides guidelines for naming software baselines, components, and configurations, helping to manage the development process through consistent identification methods and naming standards.

Software components undergo multiple revisions, and unique identification of each revision is crucial. A simple naming convention involves using discrete digits, where the first number indicates the external release version and the second indicates the internal development version. For example, moving from version 2.9 to 3.1 signals a new external release. Additional qualifiers, like dates, can be used for further differentiation.

A software configuration consists of elements that perform a major business function, such as the modules of an order system. Configurations, like individual components, can have multiple versions, each requiring distinct identification, approval status, and a description of its build process. A straightforward method to identify configurations is by storing all related components in a single library or repository and documenting the listing of these components.

Version Control
As an application evolves over time, many different versions of its software components are created, and there needs to be an organized process to manage changes in the software components and their relationships. In addition, there is usually a requirement to support parallel component development and maintenance.

Software is frequently changed as it evolves through a succession of temporary states called *versions*. A software configuration management facility for controlling versions is a software configuration management repository or library. Version control provides the tractability or history of each software change, including who did what, why, and when.

Within the software life cycle, software components evolve, and at a certain point each reaches a relatively stable state. However, as defects are corrected and enhancement features are implemented, the changes result in new versions of the components. Maintaining control of these software component versions is called *versioning*.

A component is identified and labeled to differentiate it from all other software versions of the component. When a software component is modified, both the old and new versions should be separately identifiable. Therefore, each version, except the initial one, has a predecessor. The succession of component versions is the component's history and tractability. Different versions also act as backups so that one can return to previous versions of the software.

Configuration Building
To build a software configuration, one needs to identify the correct component versions and execute the component build procedures. This is often called *configuration building*.

A software configuration consists of a set of derived software components. An example is executable object programs derived from source programs. Derived software components are correctly associated with each source component to obtain an accurate derivation. The configuration build model defines how to control the way derived software components are put together.

The inputs and outputs required for a configuration build model include the primary inputs such as the source components, the version selection procedures, and the system model, which describes how the software components are related. The outputs are the target configuration and respectively derived software components.

Software configuration management environments use different approaches to select versions. The simplest approach to version selection is to maintain a list of component versions. Other approaches entail selecting the most recently tested component versions, or those modified on a particular date.

Change Control

Change control is the process that manages modifications to software components, involving proposal, evaluation, approval or rejection, scheduling, and tracking of changes. It relies on a structured change control process, status reporting, and auditing. Through change control, some proposed changes are accepted and implemented, while others are rejected or postponed. It also includes impact analysis to assess dependencies.

Modifying a configuration involves four key elements: a change request, impact analysis, modifications or additions of components, and a method for installing these changes as a new baseline. Changes often affect multiple components, so a system beyond simple file versioning is needed, such as delta storage, which identifies a set of changes as a single modification.

Each software component follows a life cycle with specific states and transitions. When a component is changed, it must be reviewed and frozen (disallowing further changes) until a new version is approved. Approved components are stored in a software library, which serves as a repository for all software under configuration management, including various work products and models.

A derived component shares the status of its source, and a configuration's status cannot exceed that of its components. Components are stored in a configuration library and checked out for modification into a private workspace, temporarily outside configuration management control. After changes are completed, the component is checked back into the library as a new version, while previous versions are retained.

Software Quality Assurance Plan

The software quality assurance (SQA) plan is an outline of quality measures to ensure quality levels within a software development effort. The plan is used as a baseline to compare the actual levels of quality during development with the planned levels of quality. If the levels of quality are not within the planned quality levels, management will respond appropriately as documented within the plan.

The plan provides the framework and guidelines for development of understandable and maintainable code. These ingredients help ensure the quality sought in a software project. An SQA plan also provides the procedures for ensuring that quality software will be produced or maintained in-house or under contract. These procedures affect planning, designing, writing, testing, documenting, storing, and maintaining computer software. It should be organized in this way because the plan ensures the quality of the software rather than describing specific procedures for developing and maintaining it.

Steps to Develop and Implement a Software Quality Assurance Plan

Step 1: Document the Plan

The software quality assurance plan should include the following sections (see "Software Quality Assurance Plan," which contains a template for the plan):

- *Purpose Section*—This section delineates the specific purpose and scope of the particular SQA plan. It should list the names of the software items covered by the SQA plan and the intended use of the software. It states the portion of the software life cycle covered by the SQA plan for each software item specified.
- *Reference Document Section*—This section provides a complete list of documents referenced elsewhere in the text of the SQA plan.
- *Management Section*—This section describes the project's organizational structure, tasks, and responsibilities.
- *Documentation Section*—This section identifies the documentation governing the development, verification and validation, use, and maintenance of the software. It also states how the documents are to be checked for adequacy. This includes the criteria and the identification of the review or audit by which the adequacy of each document will be confirmed.
- *Standards, Practices, Conventions, and Metrics Section*—This section identifies the standards, practices, conventions, and metrics to be applied, and also states how compliance with these items is to be monitored and assured.
- *Reviews and Inspections Section*—This section defines the technical and managerial reviews, walkthroughs, and inspections to be conducted. It also states how the reviews, walkthroughs, and inspections are to be accomplished, including follow-up activities and approvals.
- *Software Configuration Management Section*—This section is addressed in detail in the project's software configuration management plan.
- *Problem Reporting and Corrective Action Section*—This section is addressed in detail in the project's software configuration management plan.
- *Tools, Techniques, and Methodologies Section*—This section identifies the special software tools, techniques, and methodologies that support SQA, states their purposes, and describes their use.
- *Code Control Section*—This section defines the methods and facilities used to maintain, store, secure, and document the controlled versions of the identified software during all phases of development. This may be implemented in conjunction with a computer program library or may be provided as a part of the software configuration management plan.

- *Media Control Section*—This section describes the methods and facilities to be used to identify the media for each computer product and the documentation required to store the media, including the copy and restore process, and protects the computer program physical media from unauthorized access or inadvertent damage or degradation during all phases of development. This may be provided by the software configuration management plan.
- *Supplier Control Section*—This section states the provisions for ensuring that software provided by suppliers meets established requirements. In addition, it should specify the methods that will be used to ensure that the software supplier receives adequate and complete requirements. For previously developed software, this section describes the methods to be used to ensure the suitability of the product for use with the software items covered by the SQA plan. For software to be developed, the supplier will be required to prepare and implement an SQA plan in accordance with this standard. This section will also state the methods to be employed to ensure that the developers comply with the requirements of this standard.
- *Records Collection, Maintenance, and Retention Section*—This section identifies the SQA documentation to be retained. It states the methods and facilities to assemble, safeguard, and maintain this documentation, and will designate the retention period. The implementation of the SQA plan involves the necessary approvals for the plan as well as development of a plan for execution. The subsequent evaluation of the SQA plan will be performed as a result of its execution.
- *Testing Methodology*—This section defines the testing approach, techniques, and automated tools that will be used.

Step 2: Obtain Management Acceptance

Management participation is necessary for the successful implementation of an SQA plan. Management is responsible for both ensuring the quality of a software project and for providing the resources needed for software development.

The level of management commitment required for implementing an SQA plan depends on the scope of the project. If a project spans organizational boundaries, approval should be obtained from all affected departments. Once approval has been obtained, the SQA plan is placed under configuration control.

In the management approval process, management relinquishes tight control over software quality to the SQA plan administrator in exchange for improved software quality. Software quality is often left to software developers. Quality is desirable, but management may express concern as to the cost of a formal SQA plan. Staff should be aware that management views the program as a means of ensuring software quality, and not as an end.

To address management concerns, software life-cycle costs should be formally estimated for projects implemented both with and without a formal SQA plan. In general, implementing a formal SQA plan makes economic and management sense.

Step 3: Obtain Development Acceptance
Because the software development and maintenance personnel are the primary users of an SQA plan, their approval and cooperation in implementing the plan are essential. The software project team members must adhere to the project SQA plan; everyone must accept it and follow it.

No SQA plan is successfully implemented without the involvement of the software team members and their managers in the development of the plan. Because project teams generally have only a few members, all team members should actively participate in writing the SQA plan. When projects become much larger (i.e., encompassing entire divisions or departments), representatives of project subgroups should provide input. Constant feedback from representatives to team members helps gain acceptance of the plan.

Step 4: Plan for Implementation of the SQA Plan
The process of planning, formulating, and drafting an SQA plan requires staff and word-processing resources. The individual responsible for implementing an SQA plan must have access to these resources. In addition, the commitment of resources requires management approval and, consequently, management support. To facilitate resource allocation, management should be made aware of any project risks that may impede the implementation process (e.g., limited availability of staff or equipment). A schedule for drafting, reviewing, and approving the SQA plan should be developed.

Step 5: Execute the SQA Plan
The actual process of executing an SQA plan by the software development and maintenance team involves determining necessary audit points for monitoring it. The auditing function must be scheduled during the implementation phase of the software product so that improper monitoring of the software project will not hurt the SQA plan. Audit points should occur either periodically during development or at specific project milestones (e.g., at major reviews or when part of the project is delivered).

Quality Standards
The following section describes the leading quality standards for IT.

Sarbanes-Oxley
The Sarbanes—Oxley Act of 2002, also known as the Public Company Accounting Reform and Investor Protection Act of 2002 and commonly called SOx or Sarbox, is a U.S. federal law enacted

on July 30, 2002, in response to a number of major corporate and accounting scandals: Enron, Tyco Internation, Adelphia, Peregrine Systems, and WorldCom.

The Sarbanes—Oxley Act is designed to ensure the following within a business:

- There are sufficient controls to prevent fraud, misuse, or loss of financial data/transactions. In many companies, most of these controls are IT-based.
- There are controls to enable speedy detection if and when such problems occur.
- Effective action is taken to limit the effects of such problems.

ISO9000

ISO9000 is a quality series and comprises a set of five documents developed in 1987 by the International Standards Organization (ISO). ISO9000 standards and certification are usually associated with non-IS manufacturing processes. However, application development organizations can benefit from these standards and position themselves for certification, if necessary. All the ISO9000 standards are guidelines and are interpretive because of their lack of stringency and rules. ISO certification is becoming more and more important throughout Europe and the United States for the manufacture of hardware. Software suppliers will increasingly be required to have certification. ISO9000 is a definitive set of quality standards, but it represents quality standards as part of a total quality management (TQM) program. It consists of ISO9001, ISO9002, or ISO9003, and it provides the guidelines for selecting and implementing a quality assurance standard.

ISO9001 is a very comprehensive standard and defines all the quality elements required to demonstrate the supplier's ability to design and deliver a quality product. ISO9002 covers quality considerations for the supplier to control design and development activities. ISO9003 demonstrates the supplier's ability to detect and control product nonconformity during inspection and testing. ISO9004 describes the quality standards associated with ISO9001, ISO9002, and ISO9003 and provides a comprehensive quality checklist.

Table 2.3 shows the ISO9000 and companion international standards.

Table 2.3: Companion ISO Standards

International	***United States***	***Europe***	***United Kingdom***
ISO9000	ANSI/ASQA	EN29000	BS5750 (Part 0.1)
ISO9001	ANSI/ASQC	EN29001	BS5750 (Part 1)
ISO9002	ANSI/ASQC	EN29002	BS5750 (Part 2)
ISO9003	ANSI/ASQC	EN29003	BS5750 (Part 3)
ISO9004	ANSI/ASQC	EN29004	BS5750 (Part 4)

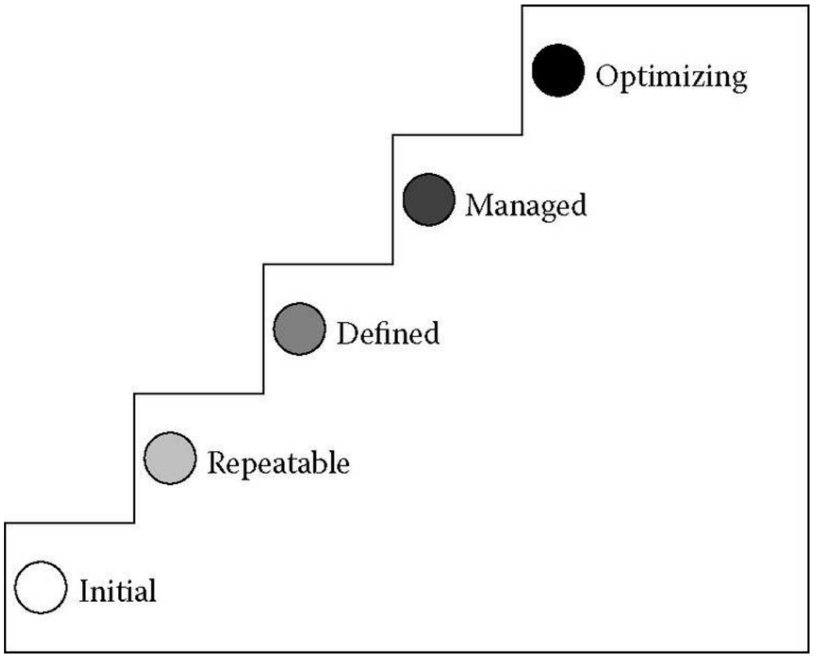
Capability Maturity Model (CMM)

The Software Engineering Institute-Capability Maturity Model (SEI-CMM) is a model for judging the maturity of the software processes of an organization and for identifying the key practices that are required to increase the maturity of these processes. As organizations enhance their software process capabilities, they progress through the various levels of maturity. The achievement of each level of maturity signifies a different component in the software process, resulting in an overall increase in the process capability of the organization. The Capability Maturity Model for Software describes the principles and practices underlying software process maturity and is intended to help software organizations improve the maturity of their software processes in terms of an evolutionary path from ad hoc chaotic processes to mature, disciplined software processes.

The CMM is organized into five maturity levels (see Figure 2.3):

1. *Initial*. The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort and heroics.
2. *Repeatable*. Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.
3. *Defined*. The software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the organization. All projects use an approved, tailored version of the organization's standard software process for developing and maintaining software.
4. *Managed*. Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.
5. *Optimizing*. Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.

Figure 2.3: Maturity levels



People CMM

The People Capability Maturity Model (People CMM) is a framework that helps organizations successfully address their critical people issues. On the basis of the best current practices in fields such as human resources, knowledge management, and organizational development, the People CMM guides organizations in improving their processes for managing and developing their workforces. The People CMM helps organizations characterize the maturity of their workforce practices, establish a program of continuous workforce development, set priorities for improvement actions, integrate workforce development with process improvement, and establish a culture of excellence. Since its release in 1995, thousands of copies of the People CMM have been distributed, and it is used worldwide by organizations small and large.

The People CMM consists of five maturity levels that establish successive foundations for continuously improving individual competencies, developing effective teams, motivating improved performance, and shaping the workforce the organization needs to accomplish its future business plans. Each maturity level is a well-defined evolutionary plateau that institutionalizes new capabilities for developing the organization's workforce. By following the maturity framework, an organization can avoid introducing workforce practices that its employees are unprepared to implement effectively.

CMMI

The CMMI Product Suite provides the latest best practices for product and service development and maintenance (Andrews and Whittaker, 2006). The CMMI models are the best process improvement models available for product and service development and maintenance. These models extend the best practices of the Capability Maturity Model for Software (SW-CMM®), the Systems Engineering Capability Model (SECM), and the Integrated Product Development Capability Maturity Model (IPD-CMM).

Organizations reported that CMMI is adequate for guiding their process improvement activities and that CMMI training courses and appraisal methods are suitable for their needs, although there are specific opportunities for improvement. The cost of CMMI is an issue that affected adoption decisions for some, but not for others. Finally, return-on-investment information is usually helpful to organizations when making the business case to adopt CMMI.

Malcolm Baldrige National Quality Award

As the National Institute of Standards and Technology (NIST) says:

In the early and mid-1980s, many industry and government leaders saw that a renewed emphasis on quality was no longer an option for American companies but a necessity for doing business in an ever-expanding, and more demanding, competitive world market. But many American businesses either did not believe quality mattered for them or did not know where to begin (Arthur, 1993).

Public Law 100-107, signed into law on August 20, 1987, created the Malcolm Baldrige National Quality Award. The Award Program led to the creation of a new public—private partnership. Principal support for the program comes from the Foundation for the Malcolm Baldrige National Quality Award, established in 1988. The Award is named for Malcolm Baldrige, who served as Secretary of Commerce from 1981 until his tragic death in a rodeo accident in 1987.

As compared to other programs such as ISO, Japan's Deming award and America's Baldrige Award:

- Focus more on results and service
- Rely on the involvement of many different professional and trade groups
- Provide special credits for innovative approaches to quality
- Include a strong customer and human resource focus
- Stress the importance of sharing information

Section 3. Overview of Testing Techniques

Software testing, as a separate process, witnessed vertical growth and received the attention of project stakeholders and business sponsors in the last decade. Various new techniques have been continuously introduced. Apart from the traditional testing techniques, various new techniques necessitated by the complicated business and development logic were realized to make software testing more meaningful and purposeful. This part discusses some of the popular testing techniques that have been adopted by the testing community.

Black-Box Testing (Functional)

In black-box, or functional testing, test conditions are developed on the basis of the program or system's functionality; that is, the tester requires information about the input data and observed output but does not know how the program or system works. Just as one does not have to know how a car works internally to drive it, it is not necessary to know the internal structure of a program to execute it. The tester focuses on testing the program's functionality against the specification. With black-box testing, the tester views the program as a black box and is completely unconcerned with the internal structure of the program or system. Some examples in this category include decision tables, equivalence partitioning, range testing, boundary value testing, database integrity testing, cause—effect graphing, orthogonal array testing, array and table testing, exception testing, limit testing, and random testing.

A major advantage of black-box testing is that the tests are geared to what the program or system is supposed to do, which is natural and understood by everyone. This should be verified with techniques such as structured walkthroughs, inspections, and joint application designs (JADs). A limitation is that exhaustive input testing is not achievable, because this requires that every possible input condition or combination be tested. In addition, because there is no knowledge of the internal structure or logic, there could be errors or deliberate mischief on the part of a programmer that may not be detectable with black-box testing. For example, suppose a payroll programmer wants to insert some job security into a payroll application he is developing. By inserting the following extra code into the application, if the employee were to be terminated, that is, his employee ID no longer existed in the system, justice would sooner or later prevail:

```
if my employee ID exists
deposit regular paycheck into my bank account
else
deposit an enormous amount of money into my bank account
erase any possible financial audit trails
erase this code
```

White-Box Testing (Structural)
In white-box, or structural testing, test conditions are designed by examining paths of logic. The tester examines the internal structure of the program or system. Test data is driven by examining the logic of the program or system, without concern for the program or system requirements. The tester knows the internal program structure and logic, just as a car mechanic knows the inner workings of an automobile. Specific examples in this category include basis path analysis, statement coverage, branch coverage, condition coverage, and branch/condition coverage.

An advantage of white-box testing is that it is thorough and focuses on the produced code. Because there is knowledge of the internal structure or logic, errors or deliberate mischief on the part of a programmer has a higher probability of being detected.

One disadvantage of white-box testing is that it does not verify that the specifications are correct; that is, it focuses only on the internal logic and does not verify the conformance of the logic to the specification. Another disadvantage is that there is no way to detect missing paths and data-sensitive errors. For example, if the statement in a program should be coded "if |a—b| < 10" but is coded "if (a—b) < 1," this would not be detectable without specification details. A final disadvantage is that white-box testing cannot execute all possible logic paths through a program because this would entail an astronomically large number of tests.

Gray-Box Testing (Functional and Structural)
Black-box testing focuses on the program's functionality against the specification. White-box testing focuses on the paths of logic. Gray-box testing is a combination of black- and white-box testing. The tester studies the requirements specifications and communicates with the developer to understand the internal structure of the system. The motivation is to clear up ambiguous specifications and "read between the lines" to design implied tests. One example of the use of gray-box testing is when it appears to the tester that a certain functionality seems to be reused throughout an application. If the tester communicates with the developer and understands the

internal design and architecture, many tests will be eliminated, because it may be possible to test the functionality only once. Another example is when the syntax of a command consists of seven possible parameters that can be entered in any order, as follows:

Command parm1, parm2, parm3, parm4, parm5, parm6, parm7
In theory, a tester would have to create 7!, or 5040 tests. The problem is compounded further if some of the parameters are optional. If the tester uses gray-box testing, by talking with the developer and understanding the parser algorithm, if each parameter is independent, only seven tests may be required.

Manual versus Automated Testing
The basis of the manual testing categorization is that it is not typically carried out by people and it is not implemented on the computer. Examples include structured walkthroughs, inspections, JADs, and desk checking.

The basis of the automated testing categorization is that it is implemented on the computer. Examples include boundary value testing, branch coverage testing, prototyping, and syntax testing. Syntax testing is performed by a language compiler, and the compiler is a program that executes on a computer.

Static versus Dynamic Testing
Static testing approaches are time independent and are classified in this way because they do not necessarily involve either manual or automated execution of the product being tested. Examples include syntax checking, structured walkthroughs, and inspections. An inspection of a program occurs against a source code listing in which each code line is read line by line and discussed. An example of static testing using the computer is a static flow analysis tool, which investigates another program for errors without executing the program. It analyzes the other program's control and data flow to discover problems such as references to a variable that has not been initialized, and unreachable code.

Dynamic testing techniques are time dependent and involve executing a specific sequence of instructions on paper or by the computer. Examples include structured walkthroughs, in which the program logic is simulated by walking through the code and verbally describing it. Boundary testing is a dynamic testing technique that requires the execution of test cases on the computer with a specific focus on the boundary values associated with the inputs or outputs of the program.

Taxonomy of Software Testing Techniques
A testing technique is a set of interrelated procedures that, together, produce a test deliverable. There are many possible classification schemes for software testing, and Table 3.1 describes one way. The table reviews formal popular testing techniques and also classifies each per the foregoing

discussion as manual, automated, static, dynamic, functional (black-box), or structural (white-box).

Table 3.1: Testing Technique Categories

Technique	*Manual*	*Automated*	*Static*	*Dynamic*	*Functional*	*Structural*
Acceptance testing	x	x		x	x	
Ad hoc testing	x				x	
Alpha testing	x			x	x	
Basis path testing		x		x		x
Beta testing	x			x	x	
Black-box testing		x		x	x	
Bottom-up testing		x		x		x
Boundary value testing		x		x	x	
Branch coverage testing		x		x		x
Branch/condition coverage		x		x		x
Cause-effect graphing		x		x	x	
Comparison testing	x	x		x	x	x
Compatibility testing	x	x				x
Condition coverage testing		x		x		x
CRUD (create, read, update, and delete) testing		x		x	x	
Database testing		x		x		x
Decision tables		x		x	x	
Desk checking	x			x		x
End-to-end testing	x	x			x	
Equivalence partitioning		x		x		
Exception testing		x		x	x	
Exploratory testing	x			x	x	
Free-form testing		x		x	x	
Gray-box testing		x		x	x	x
Histograms	x				x	
Incremental integration testing	x	x		x	x	
Inspections	x		x		x	x
Integration testing	x	x		x	x	
JADs (joint application designs)	x				x	x
Load testing	x	x		x		x
Mutation testing	x	x		x	x	
Orthogonal array testing	x		x		x	

Table 3.1: Testing Technique Categories

Technique	*Manual*	*Automated*	*Static*	*Dynamic*	*Functional*	*Structural*
Pareto analysis	x				x	
Performance testing	x	x		x	x	x
Positive and negative testing		x		x	x	
Prior defect history testing	x		x		x	
Prototyping		x		x	x	
Random testing		x		x	x	
Range testing		x		x	x	
Recovery testing	x	x		x		x
Regression testing				x	x	
Risk-based testing	x		x		x	
Run charts	x		x		x	
Sandwich testing		x		x		x
Sanity testing	x	x		x	x	
Security testing	x	x				x
State transition testing		x		x	x	
Statement coverage testing		x		x		x
Statistical profile testing	x		x		x	
Stress testing	x	x		x		
Structured walkthroughs	x			x	x	x
Syntax testing		x	x	x	x	
System testing	x	x		x	x	
Table testing		x		x		x
Thread testing		x		x		x
Top-down testing		x		x	x	x
Unit testing	x	x	x			x
Usability testing	x	x		x	x	
User acceptance testing	x	x		x	x	
White-box testing		x		x		x

Table 3.2 describes each of the software testing methods.

Table 3.2: Testing Technique Categories

Technique	*Brief Description*
Acceptance testing	Final testing based on the end-user/customer specifications, or based on use by end users/ customers over a defined period of time
Ad hoc testing	Similar to exploratory testing, but often taken to mean that the testers have significant understanding of the software before testing it

Table 3.2: Testing Technique Categories

Technique	***Brief Description***
Alpha testing	Testing of an application when development is nearing completion; minor design changes may still be made as a result of such testing. Typically done by end users or others, not by programmers or testers
Basis path testing	Identifying tests based on flow and paths of a program or system
Beta testing	Testing when development and testing are essentially completed and final bugs and problems need to be found before final release. Typically done by end users or others, not by programmers or testers
Black-box testing	Testing cases generated based on the system's functionality
Bottom-up testing	Integrating modules or programs starting from the bottom
Boundary value testing	Testing cases generated from boundary values of equivalence classes
Branch coverage testing	Verifying that each branch has true and false outcomes at least once
Branch/condition coverage testing	Verifying that each condition in a decision takes on all possible outcomes at least once
Cause-effect graphing	Mapping multiple simultaneous inputs that may affect others, to identify their conditions to test
Comparison testing	Comparing software weaknesses and strengths to competing products
Compatibility testing	Testing how well software performs in a particular hardware/software/operating system/network environment
Condition coverage testing	Verifying that each condition in a decision takes on all possible outcomes at least once
CRUD testing	Building a CRUD matrix and testing all object creations, reads, updates, and deletions
Database testing	Checking the integrity of database field values
Decision tables	Table showing the decision criteria and the respective actions
Desk checking	Developer reviews code for accuracy
End-to-end testing	Similar to system testing; the "macro" end of the test scale; involves testing of a complete application environment in a situation that mimics real-world use, such as interacting with a database, using network communications, or interacting with other hardware, applications, or systems if appropriate
Equivalence partitioning	Each input condition is partitioned into two or more groups. Test cases are generated from representative valid and invalid classes
Exception testing	Identifying error messages and exception-handling processes and conditions that trigger them

Table 3.2: Testing Technique Categories

Technique	***Brief Description***
Exploratory testing	Often taken to mean a creative, informal software test that is not based on formal test plans or test cases; testers may be learning the software as they test it
Free-form testing	Ad hoc or brainstorming using intuition to define test cases
Gray-box testing	A combination of black-box and white-box testing to take advantage of both
Histograms	A graphical representation of measured values organized according to the frequency of occurrence; used to pinpoint hot spots
Incremental integration testing	Continuous testing of an application as new functionality is added; requires that various aspects of an application's functionality be independent enough to work separately before all parts of the program are completed, or that test drivers be developed as needed; done by programmers or by testers
Inspections	Formal peer review that uses checklists, entry criteria, and exit criteria
Integration testing	Testing of combined parts of an application to determine if they function together correctly. The "parts" can be code modules, individual applications, or client/server applications on a network. This type of testing is especially relevant to client/server and distributed systems
JADs	Technique that brings users and developers together to jointly design systems in facilitated sessions
Load testing	Testing an application under heavy loads, such as testing of a Web site under a range of loads to determine at what point the system's response time degrades or fails
Mutation testing	A method for determining if a set of test data or test cases is useful, by deliberately introducing various code changes ("bugs") and retesting with the original test data/cases to determine if the bugs are detected. Proper implementation requires large computational resources
Orthogonal array testing	Mathematical technique to determine which variations of parameters need to be tested
Pareto analysis	Analyze defect patterns to identify causes and sources
Performance testing	Term often used interchangeably with stress and load testing. Ideally, performance testing (and any other type of testing) is defined in requirements documentation or QA or Test Plans
Positive and negative testing	Testing the positive and negative values for all inputs
Prior defect history testing	Test cases are created or rerun for every defect found in prior tests of the system

Table 3.2: Testing Technique Categories

Technique	*Brief Description*
Prototyping	General approach to gather data from users by building and demonstrating to them some part of a potential application
Random testing	Technique involving random selection from a specific set of input values where any value is as likely as any other
Range testing	For each input, identifies the range over which the system behavior should be the same
Recovery testing	Testing how well a system recovers from crashes, hardware failures, or other catastrophic problems
Regression testing	Testing a system in light of changes made during a development spiral, debugging, maintenance, or the development of a new release
Risk-based testing	Measures the degree of business risk in a system to improve testing
Run charts	A graphical representation of how a quality characteristic varies with time
Sandwich testing	Integrating modules or programs from the top and bottom simultaneously
Sanity testing	Typically, an initial testing effort to determine if a new software version is performing well enough to accept it for a major testing effort. For example, if the new software is crashing systems every five minutes, bogging down systems to a crawl, or destroying databases, the software may not be in a "sane" enough condition to warrant further testing in its current state
Security testing	Testing how well the system protects against unauthorized internal or external access, willful damage, etc.; may require sophisticated testing techniques
State transition testing	Technique in which the states of a system are first identified, and then test cases written to test the triggers causing a transition from one state to another
Statement coverage testing	Every statement in a program is executed at least once
Statistical profile testing	Statistical techniques are used to develop a usage profile of the system that helps define transaction paths, conditions, functions, and data tables
Stress testing	Term often used interchangeably with load and performance testing. Also used to describe such tests as system functional testing while under unusually heavy loads, heavy repetition of certain actions or inputs, input of large numerical values, or large complex queries to a database system
Structured walkthroughs	A technique for conducting a meeting at which project participants examine a work product for errors
Syntax testing	Data-driven technique to test combinations of input syntax

Table 3.2: Testing Technique Categories

Technique	*Brief Description*
System testing	Black-box type testing that is based on overall requirements specifications; covers all combined parts of a system
Table testing	Testing access, security, and data integrity of table entries
Thread testing	Combining individual units into threads of functionality that together accomplish a function or set of functions
Top-down testing	Integrating modules or programs starting from the top
Unit testing	The most "micro" scale of testing; to test particular functions or code modules. Typically done by the programmer and not by testers, as it requires detailed knowledge of the internal program design and code. Not always easily done unless the application has a well-designed architecture with tight code; may require developing test driver modules or test harnesses
Usability testing	Testing for "user-friendliness." Clearly, this is subjective, and will depend on the targeted end user or customer. User interviews, surveys, video recording of user sessions, and other techniques can be used. Programmers and testers are usually not appropriate as usability testers
User acceptance testing	Determining if software is satisfactory to an end user or customer
White-box testing	Test cases are defined by examining the logic paths of a system

Section 4. Transforming Requirements to Testable Test Cases

Introduction

Quality assurance (QA) is a holistic process involving the entire development and production process, that is, monitoring, improving, and ensuring that issues and bugs are found and fixed. Software testing is a major component of the software development life cycle. Some organizations assign responsibility for testing to their test programmers or the QA department. During the software testing process, QA project teams are typically a mix of developers, testers, and the business community who work closely together, sharing information and assigning tasks to one another.

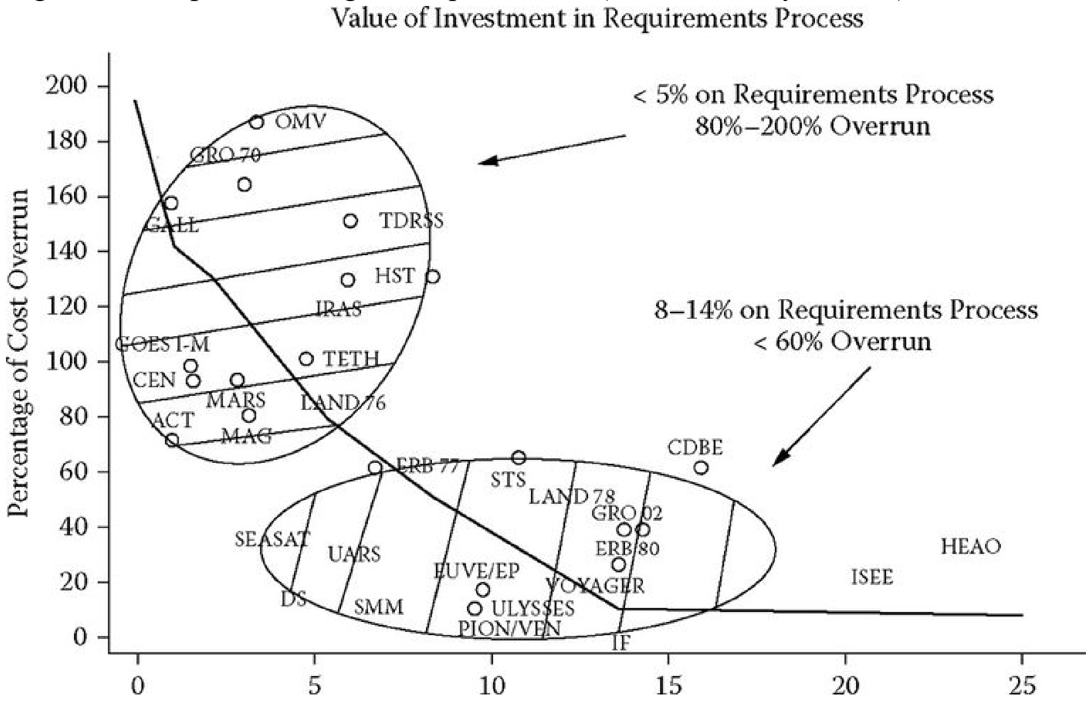
The following section provides an overview of how to create test cases when "good" requirements do exist.

Software Requirements as the Basis of Testing

Would you build a house without architecture and specific requirements? The answer is no, because of the cost of materials and manpower rework. Somehow, there is a prevalent notion that software development efforts are different, that is, put something together, declare victory, and then spend a great deal of time fixing and reengineering the software. This is called “maintenance.” According to Standish Group Statistic, American companies spend $84 billion annually on failed software projects and $138 billion on projects that significantly exceed their time and budget estimates, or have reduced functionality.

Figure 4.1 shows that the probability of project success (as measured by meeting its target cost) is greatest when 8 to 14 percent of the total project cost is invested in requirements activities.

Figure 4.1: Importance of good requirements. (Reference: Ivy Hooks.)



Requirement Quality Factors
If software testing depends on good requirements, it is important to understand some of the key elements of quality requirements.

Understandable
Requirements must be understandable. Understandable requirements are organized in a manner that facilitates reviews. Some techniques to improve understandability include the following:

- Organize requirements by their object, for example, customer, order, invoice.
- User requirements should be organized by business process or scenario. This allows the subject matter expert to see if there is a gap in the requirements.
- Separate functional from nonfunctional requirements, for example, functional versus performance.

- Organize requirements by level of detail. This determines their impact on the system, for example, "the system shall be able to take an order" versus "the system shall be able to take a retail order from the point of sale."
- Write requirements grammatically correctly and in a style that facilitates reviews. If the requirement is written in Microsoft Word, use the spell check option but beware of the context; that is, spell check may pass a word or phrase, but it may be contextually inappropriate.
- Use "shall" for requirements. Do not use "will" or "should." These are goals, not requirements. Using nonimperative words such as these makes the implementation of the requirement optional, potentially increasing cost and schedule, reducing quality, and creating contractual misunderstandings.

Necessary
The requirement must also be necessary. The following is an example of an unnecessary requirement. Suppose the following requirement is included in a requirement specification: "The system shall be acceptable if it passes 100 test cases." This is really a project process and not a requirement and should not be in a requirement specification. A requirement must relate to the target application or system being built.

Modifiable
It must be possible to change requirements and associated information. The technique used to store requirements affects modifiability. For example, requirements in a word processor are much more difficult to modify than in a requirements management tool such as CaliberRM or Doors. However, for a very small project, the cost and learning curve for the requirements management tool may make the word processor the best option.

Consistency affects modifiability. Templates and glossaries for requirements make global changes possible. Templates should be structured to make the requirements visible, thus facilitating modifiability. A good best practice is to label each requirement with a unique identifier. Requirements should also be located in a central spot and be located with ease. Any requirement dependencies should also be noted, for example, requirement "Y" may depend on requirement "X."

Nonredundant
There should not be duplicate requirements, as this causes problems. Duplicates increase maintenance; that is, every time a requirement changes, its duplicates also must be updated. Duplicate requirements also increase the potential for injecting requirement errors.

Terse
A good requirement must have no unnecessary verbiage or information. A tersely worded requirement gets right to the point; for example, “On the other hand,” “However,” “In retrospect,” and so on are pedantic.

Testable
It should be possible to verify or validate a testable requirement; that is, it should be possible to prove the intent of the requirement. Untestable requirements lend themselves to subjective interpretations by the tester. A best practice is to pretend that computers do not exist and ask yourself, could I test this requirement and know that it either works or does not?

Traceable
A requirement must also be traceable. Trace ability is key to verifying that requirements have been met. Compound requirements are difficult to trace and may cause the product to fail testing. For example, the requirement “the system shall calculate retirement and survivor benefits” is a compound requirement. The list approach avoids misunderstanding when reviewing requirements for trace ability individually.

Within Scope
All requirements must be defined in the area under consideration. The scope of a project is determined by all the requirements established for the project. The project scope is defined and refined as requirements are identified, analyzed, and baselined. A trace ability matrix will assist in keeping requirements within scope.

Numerical Method for Evaluating Requirement Quality
A best practice to ensure quality requirements is to use a numerical measure rather than subjective qualifiers such as “poor, acceptable, good, and excellent.”

The first step of this technique is to create a checklist of the requirements quality factors that will be used in your requirements review. The second step is to weight each quality factor according to its importance. The total weight of all the factors will be 100. For example:

- Quality factor 1 = 10
- Quality factor 2 = 5
- Quality factor 3 = 10
- Quality factor 4 = 5
- Quality factor 5 = 20
- Quality factor 6 = 15
- Quality factor 7 = 10
- Quality factor 8 = 25

The total score for quality starts at 100. The amount for an unmet quality factor is subtracted from the total. For example, if all quality factors are met except Quality factor 5, 20 is subtracted from 100, resulting in a final score of 80%.

Process for Creating Test Cases from Good Requirements
A technique is a process, style, and method of doing something. Examples include black box, white box, equivalence class partitioning, etc. Techniques are used within a methodology.

A methodology or process is a philosophy, guide, or blueprint that provides methods and principles for the field employing it. In the context of information systems, methodologies are strategies with a strong focus on gathering information, planning, and design elements.
The following sections outline a useful methodology for extrapolating test cases from good requirements.

Step 1: Review the Requirements
Before writing test cases, the requirements need to be reviewed to ensure that they reflect the requirements' quality factors.

An inspection is a type of formal, rigorous team manual peer review that can discover many problems than individual reviewers cannot find on their own. Informal manual peer reviews are also useful, depending on the situation. Unfortunately, reviews of requirements are not always productive (see “Waterfall Testing Review,” for more details about inspections and other types of reviews).

Two popular tools that automate the requirements process include the following:

- Smart Check™ is commercially offered by Smartware Technologies, Inc. This tool is an automated document review tool that locates anomalies and ambiguities within requirements or technical specifications based on a word, word phrases, word category, and complexity level. The tool has a glossary of words that research has shown to cause ambiguities and structural deficiencies. SmartCheck also allows the user to edit and add his or her own words, phrases, and categories to the dictionary. Reports illustrate the frequency distribution for the 18 potential anomaly types, or byword or phrase. The tool is not intended to evaluate the correctness of the specified requirements. It is an aid to writing the requirements right, not to writing the right requirements.

Step 2: Write a Test Plan
A software test plan is a document that describes the objectives, scope, approach, and focus of a software testing effort. The process of preparing a test plan is a useful way to think through the efforts needed to validate the acceptability of a software product. The completed document will

help the whole team understand the "why" and "how" of product validation. It should be thorough enough to be useful but not so thorough that no one outside the test group will read it.

The task of test planning consists of the following:

- Prioritizing quality goals for the release
- Defining the testing activities to achieve those goals
- Evaluating how well the activities support the goals
- Planning the actions needed to carry out the activities

Step 3: Identify the Test Suite

After the test plan has been completed and the requirements are "testable," an effective way of transforming the requirements to test cases is to first design the test suites. A test suite, also known as a validation suite, is a collection of test cases that are intended to be used as input to a software program to show that it has some specified set of behaviors. Test suites are used to group similar test cases together, for example, Handle Orders.

A test suite often contains detailed instructions or goals for each collection of test cases and information on the system configuration to be used during testing. A group of test cases may also contain prerequisite states or steps, and descriptions of the following tests.

A test suite (or by functionality) document is an organized table of contents for test cases. It lists the names of all test cases. The suite can be organized by listing the major product features, and then listing the test cases for each of those, as shown in Table 4.1.

Table 4.1: Function versus Test Cases

Function/Test Matrix																					
Business Function	***Test Case***																				
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**24**	**25**

Another way is to build a table in which the rows are types of business objects, and the columns are types of operations (see Table 4.2). Each cell in the grid lists test cases that test one type of operation for one type of object. For example, an Order System object is "Orders." The Orders business object would have test cases for each of the following CRUD-type operations: adding an order, list all orders, editing orders, deleting orders, searching for orders, etc. The next row might contain the "Customer" business object and have test cases for almost all the same operations.

Table 4.2: Test Suite Identification Matrix

	Types of Operations				
Business Object	***Add***	***Edit***	***Search***	***List***	***Delete***
Order	1. Create an Internet order 2. Create a POS order 3. Create a catalog order 4. Create a recurring order	1. Edit an Internet order 2. Edit a POS order 3. Edit a catalog order	1. Search an order by customer ID 2. Search an order by customer name and address 3. Search an order by zip code	1. List all orders by date 2. List all orders by customer name and address 3. List customers by state 4. List customers by products	1. Delete an Internet order 2. Delete a POS order 3. Delete a catalog order 4. Delete a recurring order 5. Delete all orders by product type
Customer	1. Create a retail customer	1. Edit a retail customer 2. Edit a wholesale customer	1. Search a customer by customer ID 2. Search a customer by	1. List all customers by data ranges date 2. List all customers by last name	1. Delete a retail customer 2. Delete a wholesale customer

Table 4.2: Test Suite Identification Matrix

	Types of Operations				
Business Object	***Add***	***Edit***	***Search***	***List***	***Delete***
			customer name and address 3. Search a customer by zip code	3. List customer by product IDs 4. List customers by gender	
Account	etc.	etc.	etc.	etc.	etc.
Coupons	etc.	etc.	etc.	etc.	etc.

The advantage of using an organized list or grid is that it gives the big picture, and it helps identify any area that needs more work. It is easy to forget to test other types of business objects and test business operations, for example, "Create Coupons." It is obvious that shoppers use coupons, but it is easy to forget to test the ability to create coupons. If it is overlooked, there will be a clearly visible blank space in the test suite document. These clear indications of missing test cases allow one to improve the test suite sooner, make more realistic estimates of testing time needed, and find more defects. These advantages allow the discovered defects to be fixed sooner and help keep management expectations in sync with reality.

Step 4: Name the Test Cases

Having an organized system test suite makes it easier to list test cases because the task is broken down into many small, specific subtasks.

There may be some list items or grid cells that really should be empty. If you cannot think of any test cases for a part of the suite that logically should have some test cases, explicitly mark it as "TBD."

The name of each test case should be a short phrase describing a general test situation. Use distinct test cases when different steps will be needed to test each situation. One test case can be used when the steps are the same and different input values are needed.

As you fill in the test suite outline, think of features or use cases that should be in the software requirements specification but are not there yet. Note any missing requirements in the requirements document as you go along.

At this point, you can already get a better feeling for the scope of the testing effort. You can already roughly prioritize the test cases. You are already starting to look at your requirements critically, and you may have identified missing or unclear requirements. Also, you can already estimate the level of specification-based test coverage that you will achieve (see "Test Case Prioritization Model" on the CD that came with the book).

Step 5: Write Test Case Descriptions and Objectives

In Step 4, you may have generated approximately one dozen test case names on your first pass. That number will go up as you continue to make your testing more systematic. The advantage of having a large number of tests is that it usually increases the coverage.

The disadvantage to creating a big test suite is simply that it is too big. It could take a long time to fully specify every test case that you have mapped out. Also, the resulting document could become too large, making it harder to maintain.

For each test case, write one or two sentences describing its purpose and objectives. The description should provide enough information so that you could come back to it after several weeks and recall the same ad hoc testing steps that you have in mind now. Later, when you actually write detailed steps in the test case, any team member can carry out the test the same way that you intended.

The act of writing the descriptions forces you to think a bit more about each test case. When describing a test case, you may realize that it should actually be split into two test cases, or merged with another test case. Again, make sure to note any requirements problems or questions that you uncover.

Step 6: Create the Test Cases

The next step is to write the test case steps and specify test data. This is where the testing techniques can help you define the test data and conditions. A rule of thumb is to create approximately ten test cases per day.

Focus on the test cases that seem most in need of additional detail. For example, select system test cases that cover the following:

- High-priority-use cases or features
- Software components that are currently available for testing
- Features that must work properly before other features can be exercised
- Features that are needed for product demos or screenshots

- Requirements that need to be clearer

Each test case should be simple enough to clearly succeed or fail. Ideally, the steps of a test case are a simple sequence: set up the test situation, exercise the system with specific test inputs, and verify the correctness of the system outputs.

Step 7: Review the Test Cases
A suite of system test cases can find many defects, but still leave many other critical defects undetected. One clear way to guard against undetected defects is to increase the coverage of your test suite.

Although a suite of unit tests might be evaluated in terms of its implementation coverage, a suite of system test cases should instead be evaluated in terms of specification coverage. Implementation coverage measures the percentage of lines of code that are executed by the unit test cases. If there is a line of code that is never executed, then there could be an undetected defect on that line. Specification coverage measures the percentage of written requirements that the system test suite covers. If there is a requirement that is not tested by any system test case, then you are not assured that the requirement has been satisfied.

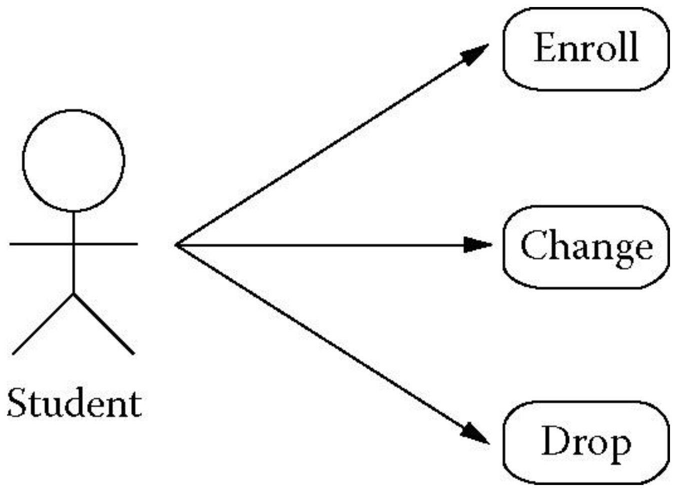
You can evaluate the coverage of your system tests at two levels: (1) the test suite itself is an organized table of contents for the test cases that can make it easy to notice parts of the system that are not being tested; and (2) within an individual test case, the set of possible input values should cover all input values. (See the "Test Case Review Checklist" located on the CD that came with the book.)

Transforming Use Cases to Test Cases
The use case, created by Ivar Jacobsen, is a scenario that describes the use of a system by an actor to accomplish work.

The following are the steps the tester can follow to create effective test cases from use cases.

Step 1: Draw a Use Case Diagram
Use cases can be represented visually with use case diagrams as shown in Figure 4.4.

Figure 4.4: Use case diagram



The ovals represent use cases, and the stick figures represent "actors," which can be either humans or other systems. The lines represent communication between an actor and a use case. Use cases provide the "big picture." Each use case represents functionality that will be implemented, and each actor represents someone or something outside our system that interacts with it.

Step 2: Write the Detailed Use Case Text

The details of each use case are then documented in text format. Table 4.3 illustrates the "Enroll" use case details consisting of the normal and alternative flows.

Table 4.3: Format for the "Enroll" Use Case Textual Description

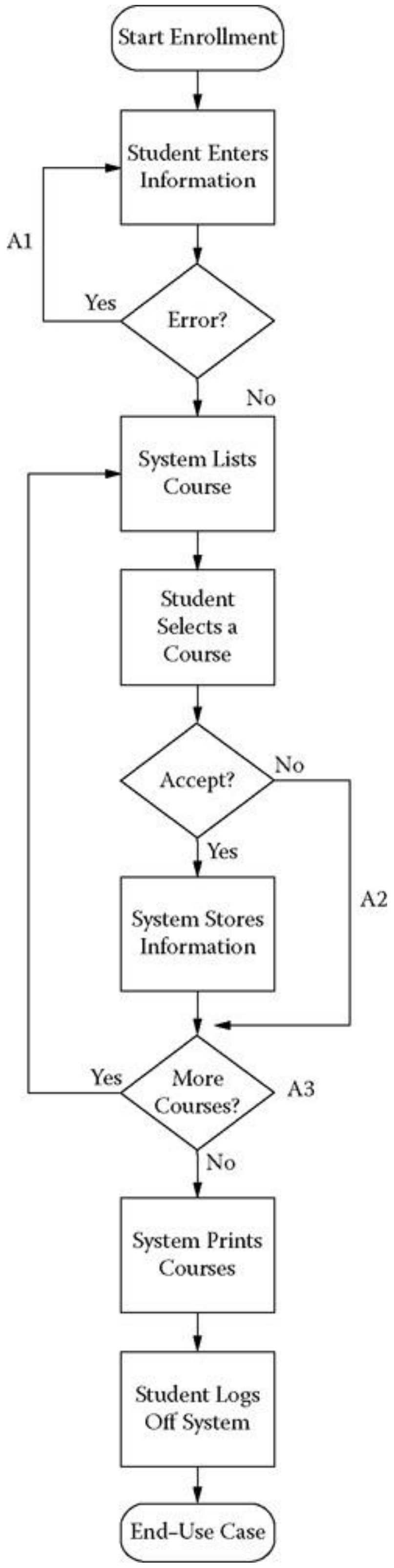
Use case ID	Enroll_001		
Use case name	Enroll a Student		
Created by	John Doe	Last updated by:	
Date created	3/15/2008	Date last updated:	
Actors	Student		
Description	Enroll a student into classes		
Trigger	Student wishes to enroll before the enrollment deadline		
Preconditions	Student has been accepted to the university Enrollment period has started		
Postconditions	Student's information has been validated and stored in the university enrollment system		

Basic flow	Enrollment: • Student enters his or her name • Student enters his or her address • Student enters his or her phone number • Student enters his or her student number • Student presses the “Submit” button • Enrollment system lists the available courses from a drop-down list • Student selects a course from a drop-down list and presses the “Accept” button • The system stores the course information and asks the student if he or she wants to select another course • The student selects “Yes,” and the enrollment process continues (Step 6) until all the courses have been selected and the student presses “No” • All selected courses and schedule are printed out • The student logs off the system
Alternative flows	A1. The “Submit” button is pressed (Step 5) and if any information is incorrect, an error message is displayed next to the error field and Step 5 is repeated A2. The student presses the “Reject” button (Step 7)
Actors	Student
Includes	N/A
Priority	High
Frequency of use	As needed
Business rules	N/A
Special requirements	N/A
Assumptions	Selected courses must not be full
Notes and Issues	N/A

Step 3: Identify Use Case Scenarios

A use case scenario is an instance of a use case, or a complete “path” through the use case. End users of a system can go down many paths as they execute the functionality specified in the use case. To illustrate this, Figure 4.5 is a flowchart of the enrollment process. The basic (or normal) path is illustrated by the dotted lines.

Figure 4.5: Enrollment flowchart



The alternate paths (or exceptions) are depicted as Al and A2. Al is the case when an error occurs when the student is entering his or her information into the system. A2 depicts the case when the student has selected a particular course but then chooses not to accept it.

Table 4.4 lists some possible combinations of scenarios for Figure 4.5. Starting with the basic flow combinations, alternative flows are added to define the scenarios. These scenarios will be used as the basis for creating test cases.

Table 4.4: Use Case Scenario

Scenario 1	Basic flow			
Scenario 2	Basic flow	Alternate flow 1		
Scenario 3	Basic flow	Alternate flow 2		
Scenario 4	Basic flow	Alternate flow 2	Alternate flow 3	

Step 4: Generating the Test Cases

A test case is a set of test inputs, execution conditions, and expected results developed for a particular objective.

Once the set of scenarios has been identified, the next step is to identify the test cases. This is accomplished by analyzing the scenarios and reviewing the use case textual descriptions. There should be at least one test case for each scenario. For each invalid test case, there should be only one invalid input.

To document the test cases, a matrix format can be used, as illustrated in Table 4.5. The first column of the first row contains the test case ID, and the second column has a brief description of the test case and the scenario being tested. All the other columns except the last one contain data elements that will be used to implement the tests. The last column contains a description of the test case's expected output. The "V" depicts a valid test input, and an "I" depicts an invalid test input.

Table 4.5: Enrollment Test Case Matrix

Test Case ID	**Scenario/Condition**	**Student Name**	**Address**	**Phone Number**	**Student Number**	**Course Rejected**	**Exit Enrollment**	**Expected Result**
Enroll 1	Scenario 1 — successful enrollment	V	V	V	V	No	Yes	Selected courses are displayed and exit system

Table 4.5: Enrollment Test Case Matrix

Test Case ID	Scenario/Condition	Student Name	Address	Phone Number	Student Number	Course Rejected	Exit Enrollment	Expected Result
Enroll 2	Scenario 2 — unidentified student	I	N/A	N/A	N/A	N/A	No	Error message; back to list of available courses
Enroll 3	Scenario 3 —rejects a course	V	V	V	V	Yes	No	Selected course is selected, rejected; back to list of available courses

Step 5: Generating Test Data

Once all of the test cases have been identified, they should be reviewed and validated to ensure accuracy and to identify redundant or missing test cases. Then, once they are approved, the final step is to substitute actual data values for the I's and V's. Table 4.6 shows a test case matrix with values substituted for the I's and V's in the previous matrix. A number of techniques can be used for identifying data values.

Table 4.6: Enrollment Test Case Details

Test Case ID	Scenario/Condition	Student Name	Address	Phone Number	Student Number	Course Selected	Expected Result
Enroll 1	Scenario 1 — successful registration	John Doe	2719 Brook Avenue, Dallas, Texas 75093	(972) 9832876	C3982	Oceanography	Courses and schedule displayed; exit system
Enroll 2	Scenario 2 — unidentified student	Invalid	2719 Brook Avenue, Dallas,	(972) 9832876	C3982	Oceanography	Error message; back to

Table 4.6: Enrollment Test Case Details

Test Case ID	Scenario/Condition	Student Name	Address	Phone Number	Student Number	Course Selected	Expected Result
			Texas 75093				login screen
Enroll 3	Scenario 2 — unidentified student	John Doe	Invalid	(972) 9832876	C3982	Oceanography	Error message; back to login screen
Enroll 4	Scenario 2 — unidentified student	John Doe	2719 Brook Avenue, Dallas, Texas 75093	Invalid	C3982	Oceanography	Error message; back to login screen
Enroll 5	Scenario 2 — unidentified student	John Doe	2719 Brook Avenue, Dallas, Texas 75093	(972) 9832876	Invalid	Oceanography	Error message; back to login screen
Enroll 6	Scenario 2 — unidentified student	John Doe	2719 Brook Avenue, Dallas, Texas 75093	(972) 9832876	C3982	Invalid	Error message; back to login screen
Enroll 7	Scenario 3 — unidentified student	John Doe	2719 Brook Avenue, Dallas, Texas 75093	(972) 9832876	C3982	Oceanography rejected	Back to login screen

Two valuable techniques are Equivalence Class Partitioning and Boundary Value Analysis (see "Software Testing Techniques," for more details).

Summary
Use cases are useful in the front end of the software development life cycle, and test cases are typically associated with the latter part of the life cycle. By leveraging use cases to generate test cases, testing teams can get started much earlier in the life cycle.

What to Do When Requirements are Nonexistent or Poor?
The following section provides an overview of how to create test cases when "good" requirements do not exist.

Depending on the project and organization, requirements may be very well written and satisfy the requirements quality factors described earlier. On the other hand, it is often the case that requirements are not clear, unambiguous, and present. In this case, other alternatives need to be considered.

Ad Hoc Testing
The Art of Ad Hoc Testing
Ad hoc testing is the least formal of test techniques. It has been criticized because it is not structured. This testing type is most often used as a complement to other types of testing. Ad hoc testing finds a place during the entire testing cycle. Early in the project, ad hoc testing provides breadth to testers' understanding of your program, thus aiding in discovery. In the middle of a project, the data obtained helps set priorities and schedules. As a project nears the ship date, ad hoc testing can be used to examine defect fixes more rigorously, as described earlier.
However, this is also a strength; that is, important things can be found quickly. Ad hoc testing is performed with improvisation in which the tester seeks to find defects with any means that seem appropriate. It is different from regression testing, which looks for a specific issue with detailed reproducible steps, with a clear expected result.

Advantages and Disadvantages of Ad Hoc Testing
One of the best uses of ad hoc testing is for discovery. Reading the requirements or specifications (if they exist) often does not provide a good sense of how a program behaves. Ad hoc testing can find holes in your test strategy, and can expose relationships between subsystems that would otherwise not be apparent. In this way, it serves as a tool for checking the completeness of your testing.

Missing cases may be found that would not otherwise be apparent with formal test cases, as these are set in concrete. Defects found while doing ad hoc testing are often examples of entire classes of forgotten test cases.

Another use for ad hoc testing is to determine the priorities for your other testing activities. Low-level housekeeping functions and basic features often do not make it into the requirements and thus have no associated test cases.

A disadvantage of ad hoc testing is that these forms of tests are not documented and, therefore, not repeatable. This limits ad hoc tests from the regression testing suite.

Exploratory Testing

The Art of Exploratory Testing

Exploratory testing is extra suitable if requirements and specifications are incomplete, or if there is lack of time. The approach can also be used to verify that previous testing has found the most important defects. It is common to perform a combination of exploratory and scripted testing, i.e., a written set of test steps to test software, where the choice is based on risk.

Exploratory testing as a technique for testing computer software does not require significant advanced planning and is tolerant of limited documentation. It relies on the skill and knowledge of the tester to guide the testing, and uses an active feedback loop to guide and calibrate the effort. According to James Bach, “The classical approach to test design, i.e., scripted testing, is like playing ‘20 Questions’ by writing out all the questions in advance.”

Exploratory testing is the tactical pursuit of software faults and defects driven by challenging assumptions. It is an approach in software testing with simultaneous learning, test design, and test execution. While the software is being tested, the tester learns things that together with experience and creativity generates new good tests to run.

Exploratory Testing Process

The basic steps of exploratory testing are as follows:

1. Identify the purpose of the product.
2. Identify functions.
3. Identify areas of potential instability.
4. Test each function and record problems.
5. Design and record a consistency verification test.

According to James Bach, “Exploratory Testing, as I practice it, usually proceeds according to a conscious plan. But not a rigorous plan… it is not scripted in detail. To the extent that the next test we do is influenced by the result of the last test we did, we are doing exploratory testing. We become more exploratory when we can't tell what tests should be run in advance of the test cycle.”

Test cases themselves are not preplanned:

- Exploratory testing can be concurrent with product development and test execution.

- Such testing is based on implicit and explicit (if they exist) specifications as well as the "as-built" product.
- Exploratory testing starts with a conjecture as to correct behavior, followed by exploration for evidence that it works/does not work.
- It is based on some kind of mental model.
- "Try it and see if it works."

Advantages and Disadvantages of Exploratory Testing

The main advantage of exploratory testing is that less preparation is needed, important bugs are found fast, and it is more intellectually stimulating than scripted testing.

Another major benefit is that testers can use deductive reasoning based on the results of previous tests to guide their future testing on the fly. They do not have to complete a current series of scripted tests before focusing in on or moving on to exploring a more target-rich environment. This also accelerates bug detection when used intelligently.

Another benefit is that, after initial testing, most bugs are discovered by some kind of exploratory testing. This can be demonstrated logically by stating, "Programs that pass certain tests tend to continue to pass the same tests and are more likely to fail other tests or scenarios that are yet to be explored."

Disadvantages are that the tests cannot be reviewed in advance (and thus cannot prevent errors in code and test cases), and that it can be difficult to show exactly which tests have been run.
When repeating exploratory tests, they will not be performed in precisely the same manner, which can be a disadvantage if it is more important to know what exact functionality.

Section 5. Quality through Continuous Improvement Process

Contribution of Edward Deming

Henry Ford and Frederick Winslow Taylor made significant contributions to factory production, but Dr. Edward Deming went further, influencing all industries, including government, schools, and hospitals. His work transformed how people think, relate to one another, and engage with customers and society.

Deming earned a Ph.D. in physics in 1928 but shifted to statistics by 1934. He contributed to the U.S. Census and introduced quality control methods to engineers during World War II. From the mid-1940s, Deming was recognized as a statistician and played a key role in post-war Japan's recovery, introducing quality management principles to both Japanese and American industries.

Role of Statistical Methods
Deming's quality method includes the use of statistical methods that he believed were essential to minimize confusion when there was variation in a process. Statistics also help us to understand the processes themselves, gain control, and improve them. This is brought home by the quote, “In God we trust. All others must use data.” Particular attention is paid to locating a problem's major causes, which, when removed, improve quality significantly. Deming points out that many statistical techniques are not difficult and require some background in mathematics. Education is a very powerful tool and is required at all levels of an organization to make it work.

The following is an outline of some statistical methods that are further described and applied to software testing.

Cause-and-Effect Diagram
Often called the “fishbone” diagram, this method can be used in brainstorming sessions to locate factors that may influence a situation. This is a tool used to identify possible causes of a problem by representing the relationship between an effect and its possible cause.

Flowchart
This is a graphical method of documenting a process. It is a diagram that shows the sequential steps of a process or of a workflow that go into creating a product or service. The justification of flowcharts is that to improve a process, one must first understand it.

Pareto Chart
This is a commonly used graphical technique in which events to be analyzed are named. The incidents are counted by name, and the events are ranked by frequency in a bar chart in ascending sequence. Pareto analysis applies the 80/20 rule. An example of this is when 20 percent of an organization's customers accounts for 80 percent of the revenue. This implies that the focus should be on the 20 percent.

Run Chart
A run chart is a graphical technique that graphs data points in chronological order to illustrate trends of a characteristic being measured, to assign a potential cause rather than random variation.

Histogram
A histogram is a graphical description of measured values organized according to the frequency or relative frequency of occurrence. It also provides the average and variation.

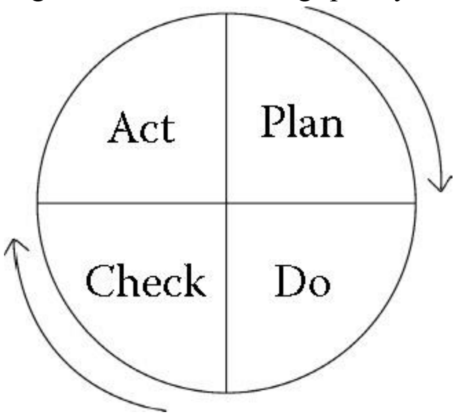
Scatter Diagram
A scatter diagram is a graph designed to show where there is a relationship between two variables or changing factors.

Control Chart
A control chart is a statistical method for distinguishing between special and common variations exhibited by processes. It is a run chart with statistically determined upper and lower limits drawn on either side of the process averages.

Continuous Improvement through the Plan, Do, Check, Act Process
The term *control* has various meanings, including supervising, governing, regulating, or restraining. The control in quality control means defining the objective of the job, developing and carrying out a plan to meet that objective, and checking to determine if the anticipated results are achieved. If the anticipated results are not achieved, modifications are made in the work procedure to fulfill the plan.

One way to describe the foregoing is with the Deming Cycle (or PDCA circle; see Figure 5.1), named after Deming in Japan because he introduced it there, although it was originated by Shewhart. It was the basis of the turnaround of the Japanese manufacturing industry, in addition to other Deming management principles. The word *management* describes many different functions, encompassing policy management, human resources management, and safety control, as well as component control and management of materials, equipment, and daily schedules. In this text, the Deming model is applied to software quality.

Figure 5.1: The Deming quality circle



In the Plan quadrant of the circle, one defines objectives and determines the conditions and methods required to achieve them. It is crucial to clearly describe the goals and policies needed to achieve the objectives at this stage. A specific objective should be documented numerically, if possible. The procedures and conditions for the means and methods to achieve the objectives are described.

In the Do quadrant of the circle, the conditions are created and the necessary training to execute the plan is imparted. It is paramount that everyone thoroughly understands the objectives and the plan. Workers need to be taught the procedures and skills required to fulfill the plan and thoroughly understand the job. The work is then performed according to these procedures.

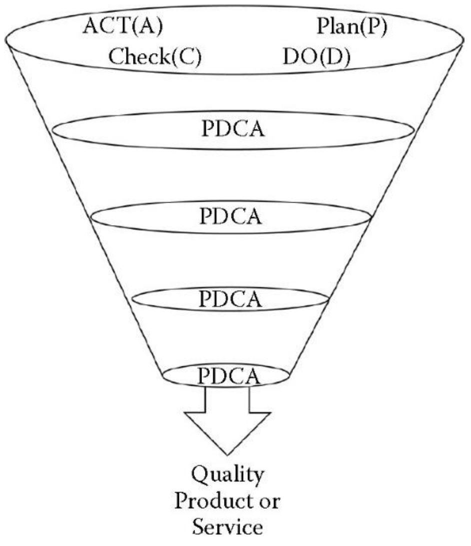
In the Check quadrant of the circle, one must check to determine whether work is progressing according to the plan and whether the expected results are obtained. The performance of the set procedures must be checked against changes in conditions, or abnormalities that may appear. As often as possible, the results of the work should be compared with the objectives. If a check detects an abnormality—that is, if the actual value differs from the target value—then a search for the cause of the abnormality must be initiated to prevent its recurrence. Sometimes, it is necessary to retrain workers and revise procedures. It is important to make sure these changes are reflected and more fully developed in the next plan.

In the Action quadrant of the circle, if the checkup reveals that the work is not being performed according to plan or results are not what was anticipated, measures must be devised for appropriate action.

Going around the PDCA Circle

The foregoing procedures not only ensure that the quality of the products meets expectations, but they also ensure that the anticipated price and delivery date are fulfilled. Sometimes our preoccupation with current concerns makes us unable to achieve optimal results. By going around the PDCA circle, we can improve our working methods and obtain the desired results. Repeated use of PDCA makes it possible to improve the quality of the work, the work methods, and the results. Sometimes this concept is depicted as an ascending spiral, as illustrated in Figure 5.2.

Figure 5.2: The ascending spiral



CHAPTER 4: WATERFALL TESTING REVIEW

The waterfall life-cycle development methodology consists of distinct phases from requirements to coding. Life-cycle testing means that testing occurs in parallel with the development life cycle and is a continuous process. Deming's continuous improvement process is applied to software testing using the quality circle, principles, and statistical techniques.

The psychology of life-cycle testing encourages testing to be performed outside the development organization. The motivation for this is that there are clearly defined requirements, and it is more efficient for a third party to verify these requirements.

The test plan is the bible of software testing. It is a document prescribing the test objectives, scope, strategy approach, and test details. There are specific guidelines for building a good test plan.

The two major quality assurance verification approaches for each life-cycle phase are technical reviews and software testing. Technical reviews are more preventive; that is, they aim to remove defects as soon as possible. Software testing verifies the actual code that has been produced.

The objectives of this section are to:

- Discuss how life-cycle testing is a parallel activity.
- Describe how Deming's process improvement is applied.
- Discuss the psychology of life-cycle development and testing.
- Discuss the components of a good test.
- List and describe how technical review and testing are verification techniques.

Section 7. Continuous Improvement “Phased” Approach

Deming's continuous improvement process, which was discussed in the previous chapter, is effectively applied to the waterfall development cycle using the Deming quality cycle, or PDCA; that is, Plan, Do, Check, and Act. It is applied from two points of view: software testing, and quality control or technical reviews.

As defined in Chapter 1, “Software Quality in Perspective,” the three major components of quality assurance are software testing, quality control, and software configuration management. The purpose of software testing is to verify and validate the activities to ensure that the software design, code, and documentation meet all the requirements imposed on them. Software testing focuses on test planning, test design, test development, and test execution. Quality control is the process and methods used to monitor work and observe whether requirements are met. It focuses on structured walkthroughs and inspections to remove defects introduced during the software development life cycle.

Psychology of Life-Cycle Testing
In the waterfall development life cycle, testing and development departments are kept separate, often with distinct reporting structures. This separation is based on the belief that testing should be handled by a dedicated quality assurance (QA) organization, which can translate requirements and design documents into test plans and cases. Key assumptions include that (1) programmers should not test their own code and (2) programming organizations should not test their own systems, as it is difficult for them to shift from development to defect detection. Independent testers, who are unbiased and objective, are believed to be more effective in ensuring software quality. QA teams were created to ensure this independence and maintain quality without conflicts of interest.

The Testing Bible: Software Test Plan
A test plan is a document outlining the approach for testing activities, acting as a service-level agreement between the quality assurance (QA) team and other stakeholders, such as development. It is developed early in the project and improves the interactions between analysis, design, and coding phases. A test plan defines objectives, scope, strategy, test cases, environment, schedules, personnel, risks, and reporting procedures.

A good test plan should be simple, comprehensive, current, and accessible, ensuring full functional coverage, logical flow, and clear roles. There are two approaches: a master test plan, providing an overview, and detailed test plans, such as for unit, integration, system, and acceptance tests. Unit test plans focus on individual code elements, while system and acceptance plans take a broader, functional view.

(See "Unit Test Plan," and "System/Acceptance Test Plan," for more details.)

The second approach is one test plan. This approach includes all the test types in one test plan, often called the acceptance/system test plan, but covers unit, integration, system, and acceptance testing, and all the planning considerations to complete the tests.

A major component of a test plan, often in the "Test Procedure" section, is a test case, as shown in Figure 6.3. (Also see "Test Case.") A test case defines the step-by-step process whereby a test is executed. It includes the objectives and conditions of the test, the steps needed to set up the test, the data inputs, the expected results, and the actual results. Other information such as the software, environment, version, test ID, screen, and test type is also provided.

Figure 6.3: Test case form

Date: _______________	Tested by: _______________
System: _____________	Environment: ___________
Objective: __________	Test ID______Reg. ID______
Function: __________	Screen: _______________
Version: ______	Test Type: ______
(Unit, Integ., System, Accept.) Condition to test: ____ ____ Data/steps to perform ____ ____ Expected results: ____ ____ ____ Actual results: Passed____Failed ____ ____ ____ ____	

Major Steps in Developing a Test Plan
A test plan is the basis for accomplishing testing and should be considered a living document; that is, as the application changes, the test plan should change.

A good test plan encourages the attitude of "quality before design and coding." It is able to demonstrate that it contains full functional coverage, and the test cases trace back to the functions being tested. It also contains workable mechanisms for monitoring and tracking discovered defects and report status. System/Acceptance Test Plan template that combines unit, integration, and system test plans into one. It is also used in this section to describe how a test plan is built during the waterfall life-cycle development methodology.

The following are the major steps that need to be completed to build a good test plan.

Step 1: Define the Test Objectives
The first step in planning any test is to establish what is to be accomplished as a result of the testing. This step ensures that all responsible individuals contribute to the definition of the test

criteria that will be used. The developer of a test plan determines what is going to be accomplished with the test, the specific tests to be performed, the test expectations, the critical success factors of the test, constraints, scope of the tests to be performed, the expected end products of the test, a final system summary report (see "System Summary Report"), and the final signatures and approvals. The test objectives are reviewed and approval for the objectives is obtained.

Step 2: Develop the Test Approach

The test plan developer outlines the overall approach or how each test will be performed. This includes the testing techniques that will be used, test entry criteria, test exit criteria, procedures to coordinate testing activities with development, the test management approach, such as defect reporting and tracking, test progress tracking, status reporting, test resources and skills, risks, and a definition of the test basis (functional requirement specifications, etc.).

Step 3: Define the Test Environment

The test plan developer examines the physical test facilities, defines the hardware, software, and networks, determines which automated test tools and support tools are required, defines the help desk support required, builds special software required for the test effort, and develops a plan to support the foregoing.

Step 4: Develop the Test Specifications

The developer of the test plan forms the test team to write the test specifications, develops test specification format standards, divides up the work tasks and work breakdown, assigns team members to tasks, and identifies features to be tested. The test team documents the test specifications for each feature and cross-references them to the functional specifications. It also identifies the interdependencies and workflow of the test specifications and reviews the test specifications.

Step 5: Schedule the Test

The test plan developer develops a test schedule based on the resource availability and development schedule, compares the schedule with deadlines, balances resources and workload demands, defines major checkpoints, and develops contingency plans.

Step 6: Review and Approve the Test Plan

The test plan developer or manager schedules a review meeting with the major players, reviews the plan in detail to ensure it is complete and workable, and obtains approval to proceed.

Components of a Test Plan

A system or acceptance test plan is based on the requirement specifications and is required in a very structured development and test environment. System testing evaluates the functionality and performance of the whole application and consists of a variety of tests, including performance, usability, stress, documentation, security, volume, recovery, and so on. Acceptance testing is a user-run test that demonstrates the application's ability to meet the original business objectives and system requirements, and usually consists of a subset of system tests.

Table 6.1 cross-references the sections of "System/Acceptance Test Plan," against the waterfall life-cycle development phases. "Start" in the intersection indicates the recommended start time, or first-cut of a test activity. "Refine" indicates a refinement of the test activity started in a previous life-cycle phase. "Complete" indicates the life-cycle phase in which the test activity is completed.

Table 6.1: System/Acceptance Test Plan versus Phase

Test Section	*Requirements Phase*	*Logical Design Phase*	*Physical Design Phase*	*Program Unit Design Phase*	*Coding Phase*
1. Introduction					
a. System description	Start	Refine	Refine	Complete	
a. Objective	Start	Refine	Refine	Complete	
a. Assumptions	Start	Refine	Refine	Complete	
a. Risks	Start	Refine	Refine	Complete	
a. Contingencies	Start	Refine	Refine	Complete	
a. Constraints	Start	Refine	Refine	Complete	
a. Approval signatures	Start	Refine	Refine	Complete	
1. Test approach and strategy					
a. Scope of testing	Start	Refine	Refine	Complete	
a. Test approach	Start	Refine	Refine	Complete	
a. Types of tests	Start	Refine	Refine	Complete	
a. Logistics	Start	Refine	Refine	Complete	
a. Regression policy	Start	Refine	Refine	Complete	
a. Test facility		Start	Refine	Complete	
a. Test procedures		Start	Refine	Complete	
a. Test organization		Start	Refine	Complete	
a. Test libraries		Start	Refine	Complete	
a. Test tools		Start	Refine	Complete	
a. Version control		Start	Refine	Complete	
a. Configuration building		Start	Refine	Complete	
a. Change control		Start	Refine	Complete	

Table 6.1: System/Acceptance Test Plan versus Phase

Test Section	*Requirements Phase*	*Logical Design Phase*	*Physical Design Phase*	*Program Unit Design Phase*	*Coding Phase*
1. Test execution setup					
a. System test process		Start	Refine	Complete	
a. Facility		Start	Refine	Complete	
a. Resources		Start	Refine	Complete	
a. Tool plan		Start	Refine	Complete	
a. Test organization		Start	Refine	Complete	
1. Test specifications					
a. Functional decomposition	Start	Refine	Refine	Complete	
a. Functions not to be tested	Start	Refine	Refine	Complete	
a. Unit test cases				Start	Complete
a. Integration test cases			Start	Complete	
a. System test cases		Start	Refine	Complete	
a. Acceptance test cases	Start	Refine	Refine	Complete	
1. Test procedures					
a. Test case, script, data development	Start	Refine	Refine	Refine	Complete
a. Test execution	Start	Refine	Refine	Refine	Complete
a. Correction	Start	Refine	Refine	Refine	Complete
a. Version control	Start	Refine	Refine	Refine	Complete
a. Maintaining test libraries	Start	Refine	Refine	Refine	Complete
a. Automated test tool usage	Start	Refine	Refine	Refine	Complete
a. Project management	Start	Refine	Refine	Refine	Complete
a. Monitoring and status reporting	Start	Refine	Refine	Refine	Complete
1. Test tools					
a. Tools to use		Start	Refine	Complete	

Table 6.1: System/Acceptance Test Plan versus Phase

Test Section	*Requirements Phase*	*Logical Design Phase*	*Physical Design Phase*	*Program Unit Design Phase*	*Coding Phase*
a. Installation and setup		Start	Refine	Complete	
a. Support and help		Start	Refine	Complete	
1. Personnel resources					
a. Required skills	Start	Refine	Refine	Complete	
a. Roles and responsibilities	Start	Refine	Refine	Complete	
a. Numbers and time required	Start	Refine	Refine	Complete	
a. Training needs	Start	Refine	Refine	Complete	
1. Test schedule					
a. Development of test plan		Start	Refine	Refine	Complete
a. Design of test cases		Start	Refine	Refine	Complete
a. Development of test cases		Start	Refine	Refine	Complete
a. Execution of test cases		Start	Refine	Refine	Complete
a. Reporting of problems		Start	Refine	Refine	Complete
a. Developing test summary report		Start	Refine	Refine	Complete
a. Documenting test summary report		Start	Refine	Refine	Complete

Technical Reviews as a Continuous Improvement Process

Quality control is a preventive aspect of quality assurance, focusing on defect removal through technical reviews during development. These reviews improve efficiency, reduce rework, and catch defects early. Developed by Michael Fagan of IBM in the 1970s, technical reviews—also called peer reviews, inspections, or structured walkthroughs—are performed at every stage of development, from requirements to coding.

Research shows that manual technical reviews are more effective than automated testing, as they catch defects earlier and reduce removal costs. Inspections can lower defect-removal costs by over two-thirds compared to dynamic testing and help identify root causes of defects. The next section outlines a framework for implementing reviews, including planning, roles, and reporting.

Motivation for Technical Reviews

A key motivation for reviews is that exhaustive software testing is impractical, and testing alone does not ensure quality or adherence to standards. Unlike testing, reviews can address these concerns. Different types of technical reviews are used depending on the project and its standards, such as stricter reviews for Department of Defense contracts compared to in-house development.

Reviews improve software quality by reducing rework, clarifying requirements, and defining test parameters. They are repeatable, effective in detecting defects, and help lower testing costs. Early error detection reduces rework, minimizes failures during testing, and ensures requirements are clear and testable.

Types of Reviews

There are formal and informal reviews. Informal reviews occur spontaneously among peers; the reviewers do not necessarily have any responsibility and do not have to produce a review report. Formal reviews are carefully planned meetings in which reviewers are held responsible for their participation, and a review report is generated that contains action items.

The spectrum of review ranges from very informal peer reviews to extremely formal and structured inspections. The complexity of a review is usually correlated to the complexity of the project. As the complexity of a project increases, the need for more formal reviews increases.

Structured Walkthroughs

A structured walkthrough is a presentation review in which a review participant, usually the developer of the software being reviewed, narrates a description of the software, and the remainder of the group provides feedback throughout the presentation. Testing deliverables such as test plans, test cases, and test scripts can also be reviewed using the walkthrough technique. These are referred to as presentation reviews because the bulk of the feedback usually occurs only for the material actually presented.

Advance preparation of the reviewers is not necessarily required. One potential disadvantage of a structured walkthrough is that, because of its informal structure, disorganized and uncontrolled reviews may result. Walkthroughs may also be stressful if the developer is conducting the walkthrough.

Inspections

The inspection technique is a formal process used to verify software products throughout their development, examining all deliverables at defined phases—from requirements to coding—to assess status and quality. A key decision during inspections is determining whether a software deliverable can advance to the next phase.

Early detection and correction of defects during inspections are crucial, as fixing issues early in development is 10 to 100 times less costly than addressing them during testing or maintenance. Inspections use exit criteria to measure product completion at the end of each phase.

Inspections offer advantages by being systematic, controlled, and less stressful, promoting egoless programming where developers are less emotionally attached to their work. The process requires a structured agenda for preparation and meetings, with rigorous entry and exit requirements for deliverables.

Unlike structured walkthroughs, inspections focus on collecting information to improve both the development and review processes, making them a more robust quality assurance technique.

Phased inspections apply the PDCA (Plan, Do, Check, and Act) quality model. Each development phase has entrance requirements; for example, how to qualify to enter an inspection and exit criteria, and how to know when to exit the inspection. In-between the entry and exit are the project deliverables that are inspected. In Table 6.2, the steps of a phased inspection and the corresponding PDCA steps are shown.

Table 6.2: PDCA Process and Inspections

Inspection Step	***Description***	***Plan***	***Do***	***Check***	***Act***
1. Planning	Identify participants, get materials together, schedule the overview	/			
2. Overview	Educate for the inspections	/			
3. Preparation	Individual preparation for the inspections		/		
4. Inspection	Actual inspection to identify defects		/	/	
5. Rework	Rework to correct any defects				/
6. Follow-up	Follow up to ensure all defects are corrected				/

The Plan step of the continuous improvement process consists of inspection planning and preparing an education overview. The strategy of an inspection is to design and implement a review process that is timely, efficient, and effective. Specific products are designated, as are acceptable criteria, and meaningful metrics are defined to measure and maximize the efficiency of the process. Inspection materials must meet inspection entry criteria. The right participants are identified and scheduled. In addition, a suitable meeting place and time are decided. The group of participants is educated on what is to be inspected and their roles.

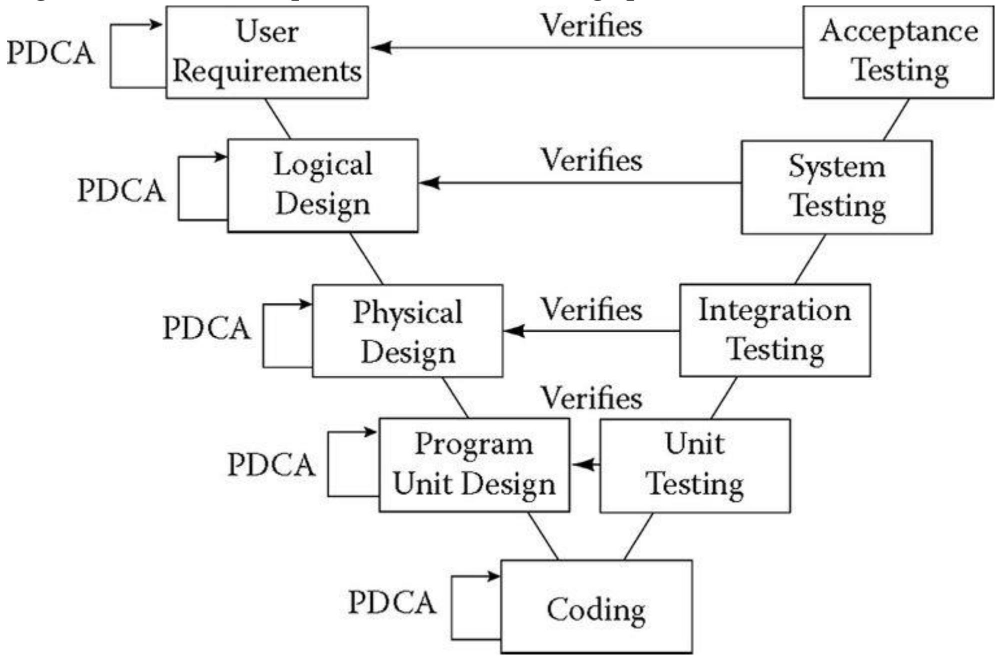
The Do step includes individual preparation for the inspections and the inspection itself. Participants learn the material and prepare for their assigned roles, and the inspection proceeds. Each review is assigned one or more specific aspects of the product to be reviewed in terms of technical accuracy, standards and conventions, quality assurance, and readability.

The Check step includes the identification and documentation of the defects uncovered. Defects are discovered during the inspection, but solution hunting and the discussion of design alternatives are discouraged. Inspections are a review process, not a solution session.

The Act step includes the rework and follow-up required to correct any defects. The author reworks all discovered defects. The team ensures that all the potential corrective actions are effective and no secondary defects are inadvertently introduced.

By going around the PDCA cycle for each development phase using inspections, we verify and improve each phase deliverable at its origin and stop it dead in its tracks when defects are discovered (see Figure 6.4). The next phase cannot Start until the discovered defects are corrected. The reason is that it is advantageous to find and correct defects as near to their point of origin as possible. Repeated application of the PDCA results in an ascending spiral, facilitating quality improvement at each phase. The end product is dramatically improved, and the bewildering task of the software testing process will be minimized; for example, a lot of the defects will have been identified and corrected by the time the testing team receives the code.

Figure 6.4: Phased inspections as an ascending spiral



Participant Roles
Roles in review methodologies, such as structured walkthroughs and inspections, are functional, allowing participants to take on multiple roles in a single review. An important aspect is the follow-up on errors identified during the review, as developers may not always correct them properly. This raises the question of who should oversee follow-ups and whether another review is necessary.

The review leader is responsible for the review process, including scheduling, conducting orderly meetings, and preparing the review report. They may also ensure that action items are addressed post-review. This role requires both technical and interpersonal skills, such as leadership, mediation, and organization. The review leader must keep the group focused and prevent the meeting from devolving into problem-solving. Preparation for the review should not take the leader more than two hours.

The recorder's role is to capture all necessary information for an accurate review report, including the essence of complex discussions and action items. This is a technical role requiring a strong understanding of the content being reviewed.

Reviewers are tasked with objectively analyzing the software, ensuring focus remains on the software itself, not its creator. The number of reviewers should be limited to six, as more can reduce productivity.

In technical reviews, the producer of the software may lead the discussion, requiring preparation and planning to present the material effectively. The producer’s attitude is crucial, and they should avoid defensiveness. The review leader should emphasize that the goal is to uncover defects and improve the product.

Steps for an Effective Review

Step 1: Plan for the Review Process
Planning can be described at both the organizational level and the specific review level. Considerations at the organizational level include the number and types of reviews that are to be performed for the project. Project resources must be allocated for accomplishing these reviews.

At the specific review level, planning considerations include selecting participants and defining their respective roles, scheduling the review, and developing a review agenda. There are many issues involved in selecting the review participants. It is a complex task normally performed by management, with technical input. When selecting review participants, care must be exercised to ensure that each aspect of the software under review can be addressed by at least some subset of the review team.

To minimize the stress and possible conflicts in the review processes, it is important to discuss the role that a reviewer plays in the organization and the objectives of the review. Focusing on the review objectives will lessen personality conflicts.

Step 2: Schedule the Review
A review should ideally take place soon after a producer has completed the software but before additional effort is expended on work dependent on the software. The review leader must state the agenda based on a well-thought-out schedule. If all the inspection items have not been completed, another inspection should be scheduled.

The problem of allocating sufficient time to a review stems from the difficulty in estimating the time needed to perform the review. The approach that must be taken is the same as that for estimating the time to be allocated for any meeting; that is, an agenda must be formulated and time estimated for each agenda item. An effective technique is to estimate the time for each inspection item on a time line.

Another scheduling problem is the duration of the review when the review is too long. This requires that review processes be focused in terms of their objectives. Review participants must understand these review objectives and their implications in terms of actual review time, as well as preparation time, before committing to the review. The deliverable to be reviewed should meet a certain set of entry requirements before the review is scheduled. Exit requirements must also be defined.

Step 3: Develop the Review Agenda
A review agenda must be developed by the review leader and the producer prior to the review. Although review agendas are specific to any particular product and the objective of its review, generic agendas should be produced for related types of products. These agendas may take the form of checklists (see "Checklists," for more details).

Step 4: Create a Review Report
The output of a review is a report. The format of the report is not important. The contents should address the management perspective, user perspective, developer perspective, and quality assurance perspective.

From a management perspective, the review report serves as a summary of the review that highlights what was reviewed, who did the reviewing, and their assessment. Management needs an estimate of when all action items will be resolved to successfully track the project.

The user may be interested in analyzing review reports for some of the same reasons as the manager. The user may also want to examine the quality of intermediate work products in an effort to monitor the development organization's progress.

From a developer's perspective, the critical information is contained in the action items. These may correspond to actual errors, possible problems, inconsistencies, or other considerations that the developer must address.

The quality assurance perspective of the review report is twofold: quality assurance must ensure that all action items in the review report are addressed, and it should also be concerned with analyzing the data on the review forms and classifying defects to improve the software development and review process. For example, a large number of specification errors might suggest a lack of rigor or time in the requirements specifications phase of the project. Another example is a large number of defects reported, suggesting that the software has not been adequately unit tested.

Section 6. Static Testing the Requirements

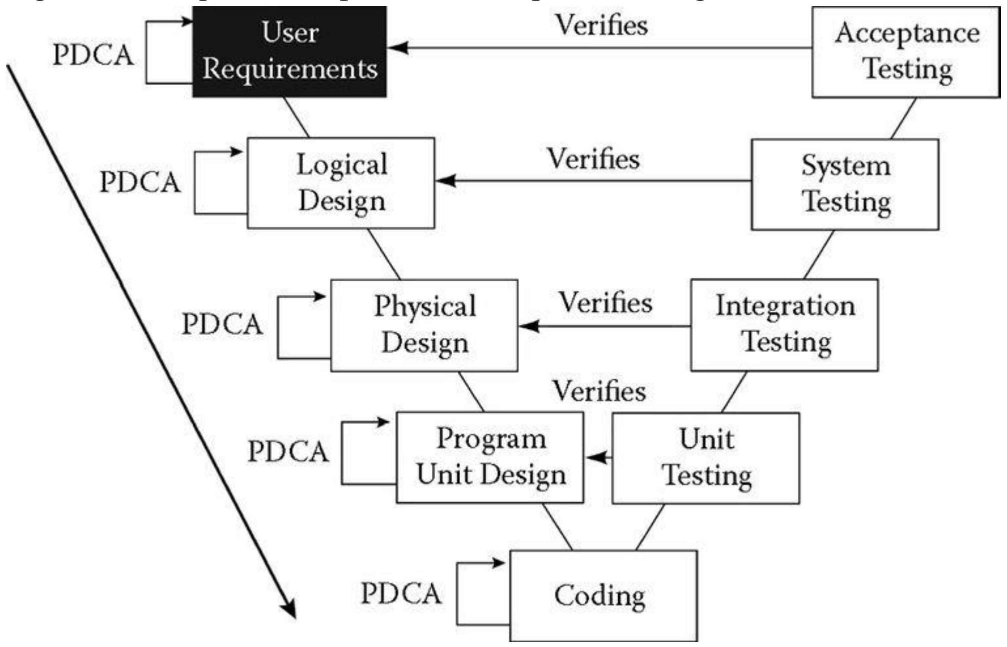
Overview

The testing process should begin early in the application development life cycle, not just at the traditional testing phase at the end of coding. Testing should be integrated with the application development phases.

During the requirements phase of the software development life cycle, the business requirements are defined on a high level and are the basis of the subsequent phases and the final implementation. Testing in its broadest sense commences during the requirements phase (see Figure 7.1), which increases the probability of developing a quality system based on the user's expectations. The result is that the requirements are verified to be correct and complete. Unfortunately, more often than not, poor requirements are produced at the expense of the application. Poor requirements ripple down the waterfall and result in a product that does not meet the user's expectations. Some characteristics of poor requirements include the following:

- Partial set of functions defined
- Performance not considered
- Ambiguous requirements
- Security not defined
- Interfaces not documented
- Erroneous and redundant requirements
- Requirements too restrictive
- Contradictory requirements

Figure 7.1: Requirements phase and acceptance testing



Functionality is the most important part of the specification and should include a hierarchic decomposition of the functions. The reason for this is that it provides a description that is described in levels to enable all the reviewers to read as much detail as needed. Specifically, this will make the task of translating the specification to test requirements much easier.

Another important element of the requirements specification is the data description (see "Requirements Specification," for more details). It should contain details such as whether the database is relational or hierarchical. If it is hierarchical, a good representation is a data model or entity relationship diagram in terms of entities, attributes, and relationships.

Another section in the requirements should be a description of the interfaces between the system and external entities that interact with the system, such as users, external software, or external hardware. A description of how users will interact with the system should be included. This would include the form of the interface and the technical capabilities of the users.

During the requirements phase, the testing organization needs to perform two functions simultaneously. It needs to build the system/acceptance test plan and also verify the requirements. The requirements verification entails ensuring the correctness and completeness of the documentation prepared by the development team.

Testing the Requirements with Ambiguity Reviews

An Ambiguity Review, developed by Richard Bender from Bender RBT, Inc., is a very powerful testing technique that eliminates defects in the requirements phase of the software life cycle, thereby avoiding those defects from propagating to the remaining phases of the software development life cycle. A QA Engineer trained in the technique performs the Ambiguity Review. The Engineer is not a domain expert (SME), and is not reading the requirements for content, but only to identify ambiguities in the logic and structure of the wording. The Ambiguity Review takes place after the requirements, or section of the requirements, reach first draft, and prior to them being reviewed for content, i.e. correctness and completeness by domain experts. The Engineer identifies all ambiguous words and phrases on a copy of the requirements. A summary of the findings is presented to the Business Analyst.

The Ambiguity Review Checklist identifies 15 common problems that occur in writing requirements.

Testing the Requirements with Technical Reviews
A software technical review is a form of peer review in which a team of qualified personnel examines the suitability of the software product for its intended use and identifies descrepancies from specifications and standards. Technical reviews may also provide recommendations of alternatives and examiniation of various alternatives. Technical reviews differ from software walkthroughs in its specific focus is on the technical quality of the product reviews. It differs from a software inspection in its ability to suggest direct alterations to the product reviewed, and its lack of a direct focus on training and process improvements (Source: Std. 1028-1997, IEEE Standard for Software Reviews, clause 3.7).

Inspections and Walkthroughs
These are formal techniques to evaluate the documentation form, interface requirements, and solution constraints as described in the previous section.

Checklists
These are oriented toward quality control and include questions to ensure the completeness of the requirements.

Methodology Checklist
This provides the methodology steps and tasks to ensure that the methodology is followed.

If the review is totally successful with no outstanding issues or defects discovered, the requirements specification is frozen, and any further refinements are monitored rigorously. If the review is not totally successful and there are minor issues during the review, the author corrects them. The corrections are reviewed by the moderator and signed off. On the other hand, if major issues and defects are discovered during the requirements review process, the defects are corrected; a new review then occurs with the same review members at a later time.

Each defect uncovered during the requirements phase review should be documented. Requirement defect trouble reports are designed to assist in the proper recording of these defects. It includes the defect category and defect type. The description of each defect is recorded under the missing, wrong, or extra columns. At the conclusion of the requirements review, the defects are summarized and totaled. Table 7.1 shows a partial requirements phase defect recording form (see "Requirements Phase Defect Checklist," for more details).

Table 7.1: Requirements Phase Defect Recording

Defect Category	*Missing*	*Wrong*	*Extra*	*Total*
1. Operating rules (or information) are inadequate or partially missing				
2. Performance criteria (or information) are inadequate or partially missing				
3. Environment information is inadequate or partially missing				
4. System mission information is inadequate or partially missing				
5. Requirements are incompatible				
6. Requirements are incomplete				
7. Requirements are missing				
8. Requirements are incorrect				
9. The accuracy specified does not conform to the actual need				
10. The data environment is inadequately described				

Requirements Traceability Matrix

A requirements traceability matrix is a document that traces user requirements from analysis through implementation. It can be used as a completeness check to verify that all requirements are present or that there are no unnecessary/extra features, and as a maintenance guide for new personnel. At each step in the development cycle, the requirements, code, and associated test cases are recorded to ensure that the user requirement is addressed in the final system. Both the user and developer have the ability to easily cross-reference the requirements to the design specifications, programming, and test cases. (See "Requirements Traceability Matrix," for more details.)

Building the System/Acceptance Test Plan

Acceptance testing ensures that a system meets the user's acceptance criteria, using black-box techniques to test the system against its specifications, typically performed by the end user in a formal test environment. The acceptance test plan, based on requirement specifications, may evolve with updates to the acceptance criteria during testing. Acceptance testing is often combined with system-level testing.

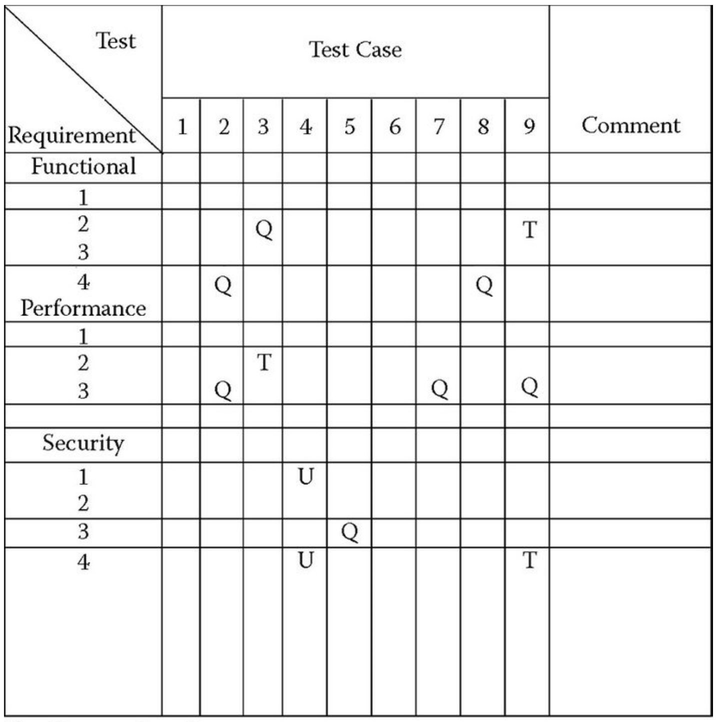
During the requirements phase, foundational test activities are outlined, including system description, acceptance test objectives, assumptions, risks, and constraints. At this stage, not all sections of the test plan will be complete due to limited information. Key elements in the Test Approach and Strategy section include:

Scope of Testing: Defines the extent of testing, such as whether the entire system or parts will be tested.

- ✓ Test Approach: Describes the design basis, including black-box, white-box, or gray-box testing.
- ✓ Types of Tests: Identifies test types to be performed, such as unit, integration, system, or acceptance tests.
- ✓ Logistics: Outlines the relationship between development and testing teams, including software delivery and defect management processes.
- ✓ Regression Policy: Ensures previously tested functions work correctly after changes.
- ✓ Testing the requirements document is challenging as testers must verify that the problem definition is accurately translated into requirements. This involves envisioning the final product and determining appropriate tests to confirm the requirements solve the problem.

A useful tool in the Test Specifications section is the requirement/test matrix, which helps analyze, review, and document the initial functional decomposition of the system. This matrix defines the testing scope, ensuring that each requirement is covered by specific tests and identifying functions that will or will not be tested.

Figure 7.2: Requirements/test matrix



Test / Requirement	Test Case									Comment
	1	2	3	4	5	6	7	8	9	
Functional										
1										
2 3			Q						T	
4 Performance		Q						Q		
1										
2 3		Q	T				Q		Q	
Security										
1 2				U						
3					Q					
4				U					T	

U – Users reviewed
Q – QA reviewed
T – Ready for testing

Some benefits of the requirements/test matrix are that it:

1. Correlates the tests and scripts with the requirements
2. Facilitates status of reviews
3. Acts as a traceability mechanism throughout the development cycle, ex. requirement, test case(s), defect(s) linkage

The requirement/test matrix in Figure 7.2 documents each requirement and correlates it with the test cases and scripts to verify it. The requirements listed on the left side of the matrix can also aid in defining the types of system tests in the Test Approach and Strategy section.

It is unusual to come up with a unique test case for each requirement and, therefore, it takes several test cases to test a requirement thoroughly. This enables reusability of some test cases to other requirements. Once the requirement/test matrix has been built, it can be reviewed, and test case design and script building can commence.

The status column is used to track the status of each test case as it relates to a requirement. For example, "Q" in the status column can indicate that the requirement has been reviewed by QA, "U" can indicate that the users had reviewed the requirement, and "T" can indicate that the test case specification has been reviewed and is ready.

In the Test Specifications section of the test plan, information about the acceptance tests is available and can be documented. These tests must be passed for the user to accept the system. A procedure is a series of related actions carried out using an operational mode, that is, one that tells how to accomplish something. The following information can be documented in the Test Procedures section: test case, script, data development, test execution, correction, version control, maintaining test libraries, automated test tool usage, project management, monitoring, and status reporting.

It is not too early to start thinking about the testing personnel resources that will be needed. This includes the required testing skills, roles and responsibilities, the numbers and time required, and the personnel training needs.

Section 8: Static Testing the Logical Design

The business requirements are defined during the requirements phase. The logical design phase refines the business requirements in preparation for a system specification that can be used during physical design and coding. The logical design phase further refines the business requirements that were defined in the requirement phase, from a functional and information model point of view.

Data Model, Process Model, and the Linkage

The logical design phase establishes a detailed framework for building an application, producing three major deliverables: a data model (entity relationship diagram), a process model, and the linkage between the two.

The data model graphically represents the information needed by the application, defining entities (such as customers, orders, and offices) and their relationships. Each entity is represented as a table with rows (specific occurrences) and columns (attributes like size, date, or address). Entities are linked by relationships:

- ✓ One-to-one: A single occurrence of one entity links to zero or one occurrence of another.
- ✓ One-to-many: One occurrence of an entity links to zero or more occurrences of another.
- ✓ Many-to-many: Multiple occurrences of entities link to each other.

The process model represents business activities, detailing inputs and outputs without describing why, how, or when processes occur. Processes, such as "accept order" or "update inventory," are decomposed into increasingly detailed levels until elementary processes (smallest meaningful activities) are defined.

A process decomposition diagram illustrates processes in a hierarchical structure, starting from the root (top-level process) and breaking down into parent (higher level) and child (lower level) processes. The process decomposition is verified using a data flow diagram, which shows all processes, data stores, and the flow of data within the system, including interactions with external entities.

An association diagram, often called a CRUD matrix or process/data matrix, links data and process models (see Figure 8.1). It helps ensure that the data and processes are discovered and assessed. It identifies and resolves matrix omissions and conflicts, and helps refine the data and process models, as necessary. It maps processes against entities, showing which processes create, read, update, or delete the instances in an entity.

Figure 8.1: CRUD matrix

Entity / Process	Entity Type									Comment
	1	2	3	4	5	6	7	8	9	
Planning	crud				cu			cu		
Selling		ud	c				c			
Scheduling	c				d	crud			d	
Compensation			cu	c	d		cu			
Shipping		crud	ud	u	c	crud				
Operations					crud		crud			
Maintenance		c			cu				cu	
Cost Planning	crud					crud				
Purchasing			ud					d		
Forecasting							c			
Receiving		c	c		c					
Ordering	d					d			cu	
Research			crud		c		crud			
• • •										

This is often called “entity life-cycle analysis.” It analyzes the birth and death of an entity and is performed by process against the entity. The analyst first verifies that there is an associated process to create instances in the entity. If there is an entity that has no associated process that creates it, a process is missing and must be defined. It is then verified that there are associated processes to update, read, or delete instances in an entity. If there is an entity that is never updated, read, or deleted, perhaps the entity may be eliminated. See “CRUD Testing,” for more details of how this can be applied to software testing.

Testing the Logical Design with Technical Reviews

The logical design phase is verified with static techniques, that is, nonexecution of the application. As utilized in the requirements phase, these techniques check the adherence to specification conventions and completeness of the models. The same static testing techniques used to verify the requirements are used in the logical design phase. The work products to be reviewed include the data model, the process model, and CRUD matrix.

Each defect discovered during the logical design review should be documented. A defect trouble report is designed to assist in the proper recording of these defects. It includes the defect category and defect type. The description of each defect is recorded under the missing, wrong, or extra columns. At the conclusion of the logical design review, the defects are summarized and totaled. Table 8.1 shows a sample logical design phase defect recording form (see "Logical Design Phase Defect Checklist," for more details).

Table 8.1: Logical Design Phase Defect Recording

Defect Category	*Missing*	*Wrong*	*Extra*	*Total*
1. The data has not been adequately defined				
2. Entity definition is incomplete				
3. Entity cardinality is incorrect				
4. Entity attribute is incomplete				
5. Normalization is violated				
6. Incorrect primary key				
7. Incorrect foreign key				
8. Incorrect compound key				
9. Incorrect entity subtype				
10. The process has not been adequately defined				

Refining the System/Acceptance Test Plan

System testing is a comprehensive evaluation of an entire application, assessing functionality, performance, security, usability, and compatibility to determine if it meets its original objectives. While the requirements phase lacked sufficient detail for defining these tests, the logical design phase provides more data and process models, allowing refinement of the testing scope and strategy. This includes detailed planning for system-level tests, testing approach, logistics, regression policy, and the setup of test facilities, procedures, and tools.

The Test Execution Setup section outlines preparations for system testing, including defining the test process, establishing the test facility, organizing required resources, and planning testing tools and team roles. Preliminary planning for software configuration management—such as version and change control—also begins, including the acquisition of tools if needed.

The Test Specifications section now includes more detailed functional information derived from data and process models, and system-level test case design is initiated. However, detailed test development—like procedures, scripts, and input/output data—is not yet completed. Acceptance test cases should be finalized during this phase.

In the Test Procedures section, initial plans from earlier phases are refined, and items related to test tools and schedules are started. This phase sets the stage for the detailed execution of system testing to ensure the application aligns with defined objectives and quality standards.

Section 9: Static Testing the Physical Design

The logical design phase translates the business requirements into system specifications that can be used by programmers during physical design and coding. The physical design phase determines how the requirements can be automated. During this phase a high-level design is created in which the basic procedural components and their interrelationships and major data representations are defined.

The physical design phase develops the architecture, or structural aspects, of the system. Logical design testing is functional; however, physical design testing is structural. This phase verifies that the design is structurally sound and accomplishes the intent of the documented requirements. It assumes that the requirements and logical design are correct and concentrates on the integrity of the design itself.

Testing the Physical Design with Technical Reviews

The logical design phase is verified using static techniques, which involve non-execution checks to ensure adherence to specifications and completeness, focusing on architectural design. Verification of the physical design uses design representation schemes such as structure charts, Warmer—Orr diagrams, Jackson diagrams, data navigation diagrams, and relational database diagrams, all mapped from the logical design phase. These schemes specify algorithms, inputs, outputs, and control flow between software modules.

Static analysis helps detect control flow errors, such as when a module requires a data item from another module but does not receive it correctly. This analysis also identifies inconsistencies between levels of design detail, such as coupling issues. Coupling measures the degree of interaction between modules, with loose coupling (minimal interaction) being preferred. Types of coupling include content, common, control, stamp, and data coupling. For example, content coupling occurs when one module alters another module’s internals, while data coupling occurs when modules communicate through shared variables.

Static analysis techniques detect both static and semantic errors, which involve issues with data decomposition, functional decomposition, and control flow. Each identified defect should be documented, categorized, recorded, reviewed by the design team for correction, and linked to the specific document where it was noted.

Table 9.1 shows a sample physical design phase defect recording form (see "Physical Design Phase Defect Checklist." for more details).

Table 9.1: Physical Design Phase Defect Recording

Defect Category	***Missing***	***Wrong***	***Extra***	***Total***
1. Logic or sequencing is erroneous				
2. Processing is inaccurate				
3. Routine does not input, or output required parameters				
4. Routine does not accept all data within the allowable range				
5. Limit and validity checks are made on input data				
6. Recovery procedures are not implemented or are not adequate				
7. Required processing is missing or inadequate				
8. Values are erroneous or ambiguous				
9. Data storage is erroneous or inadequate				
10. Variables are missing				

Creating Integration Test Cases

Integration testing is designed to test the structure and the architecture of the software and determine whether all software components interface properly. It does not verify that the system is functionally correct, only that it performs as designed.

Integration testing is the process of identifying errors introduced by combining individual program unit-tested modules. It should not begin until all units are known to perform according to the unit specifications. Integration testing can start with testing several logical units or can incorporate all units in a single integration test.

Because the primary concern in integration testing is that the units interface properly, the objective of this test is to ensure that they integrate, that parameters are passed, and the file processing is correct. Integration testing techniques include top-down, bottom-up, sandwich testing, and thread testing (see "Software Testing Techniques," for more details).

Methodology for Integration Testing

The following describes a methodology for creating integration test cases.

Step 1: Identify Unit Interfaces

The developer of each program unit identifies and documents the unit's interfaces for the following unit operations:

- External inquiry (responding to queries from terminals for information)
- External input (managing transaction data entered for processing)
- External filing (obtaining, updating, or creating transactions on computer files)

- Internal filing (passing or receiving information from other logical processing units)
- External display (sending messages to terminals)
- External output (providing the results of processing to some output device or unit)

Step 2: Reconcile Interfaces for Completeness
The information needed for the integration test template is collected for all program units in the software being tested. Whenever one unit interfaces with another, those interfaces are reconciled. For example, if program unit A transmits data to program unit B, program unit B should indicate that it has received that input from program unit A. Interfaces not reconciled are examined before integration tests are executed.

Step 3: Create Integration Test Conditions
One or more test conditions are prepared for integrating each program unit. After the condition is created, the number of the test condition is documented in the test template.

Step 4: Evaluate the Completeness of Integration Test Conditions
The following list of questions will help guide evaluation of the completeness of integration test conditions recorded on the integration testing template. This list can also help determine whether test conditions created for the integration process are complete.

- Is an integration test developed for each of the following external inquiries?
    - — Record test
    - — File test
    - — Search test
    - — Match/merge test
    - — Attributes test
    - — Stress test
    - — Control test
- Are all interfaces between modules validated so that the output of one is recorded as input to another?
- If file test transactions are developed, do the modules interface with all those indicated files?
- Is the processing of each unit validated before integration testing?
- Do all unit developers agree that integration test conditions are adequate to test each unit's interfaces?
- Are all software units included in integration testing?
- Are all files used by the software being tested included in integration testing?
- Are all business transactions associated with the software being tested included in integration testing?
- Are all terminal functions incorporated in the software being tested included in integration testing?

The documentation of integration tests is started in the Test Specifications section (see "System/Acceptance Test Plan"). Also in this section, the functional decomposition continues to be refined, but the system-level test cases should be completed during this phase.

Test items in the Introduction section are completed during this phase. Items in the Test Approach and Strategy, Test Execution Setup, Test Procedures, Test Tool, Personnel Requirements, and Test Schedule continue to be refined.

Section 10: Static Testing the Program Unit Design

The design phase develops the physical architecture, or structural aspects, of the system. The program unit design phase is refined to enable detailed design. The program unit design is the detailed design in which specific algorithmic and data structure choices are made. It is the specifying of the detailed flow of control that will make it easily translatable to program code with a programming language.

Testing the Program Unit Design with Technical Reviews

A good detailed program unit design is one that can easily be translated to many programming languages. It uses structured techniques such as while, for, repeat, if, and case constructs. These are examples of the constructs used in structured programming. The objective of structured programming is to produce programs with high quality at low cost. A structured program is one in which only three basic control constructs are used.

Sequence

Statements are executed one after another in the same order that they appear in the source listing. An example of a sequence is an assignment statement.

Selection

A condition is tested and, depending on whether the test is true or false, one or more alternative execution paths are traversed. An example of a selection is an if-then-else. With this structure, the condition is tested and, if found to be true, one set of instructions is executed. If the condition is false, another set of instructions is executed. Both sets join at a common point.

Iteration

Iteration is used to execute a set of instructions a number of times with a loop. Examples of iteration are dountil and dowhile. A dountil loop executes a set of instructions and then tests the loop termination condition. If it is true, the loop terminates and continues to the next construct. If it is false, the set of instructions is executed again until the termination logic is reached. A dowhile loop tests the termination condition. If it is true, control passes to the next construct. If it is false, a set of instructions is executed until control is unconditionally passed back to the condition logic.

Static analysis of the detailed design detects semantic errors involving information and logic control flow. Each defect uncovered during the program unit design review should be documented, categorized, recorded, presented to the design team for correction, and referenced to the specific document in which the defect was noted. Table 10.1 shows a sample program unit design phase defect recording form (see "Program Unit Design Phase Defect Checklist," for more details).

Table 10.1: Program Unit Design Phase Defect Recording

Defect Category	***Missing***	***Wrong***	***Extra***	***Total***
1. Is the if-then-else construct used incorrectly?				
1. Is the dowhile construct used incorrectly?				
1. Is the dountil construct used incorrectly?				
1. Is the case construct used incorrectly?				
1. Are there infinite loops?				
1. Is it a proper program?				
1. Are there *goto* statements?				
1. Is the program readable?				
1. Is the program efficient?				
1. Does the case construct contain all the conditions?				

Creating Unit Test Cases

Unit testing is the process of executing a functional subset of the software system to determine whether it performs its assigned function. It is oriented toward the checking of a function or a module. White-box test cases are created and documented to validate the unit logic and black-box test cases to test the unit against the specifications (see "Test Case," for a sample test case form). Unit testing, along with the version control necessary during correction and retesting, is typically performed by the developer. During unit test case development, it is important to know which portions of the code have been subjected to test cases and which have not. By knowing this coverage, the developer can discover lines of code that are never executed or program functions that do not perform according to the specifications. When coverage is inadequate, implementing the system is risky because defects may be present in the untested portions of the code (see "Software Testing Techniques," for more unit test case development techniques). Unit test case specifications are started and documented in the Test Specification section (see "System/Acceptance Test Plan"), but all other items in this section should have been completed.

All items in the Introduction, Test Approach and Strategy, Test Execution Setup, Test Tools, and Personnel Resources should have been completed prior to this phase. Items in the Test Procedures section, however, continue to be refined. The functional decomposition, integration, system, and acceptance test cases should be completed during this section. Refinement continues for all items in the Test Procedures and Test Schedule sections.

Section 11: Static Testing and Dynamic Testing the Code

The program unit design is the detailed design in which specific algorithmic and data structure choices are made. Specifying the detailed flow of control will make it easily translatable to program code with a programming language. The coding phase is the translation of the detailed design to executable code using a programming language.

Testing Coding with Technical Reviews

The coding phase produces executable source modules. The basis of good programming is programming standards that have been defined. Some good standards should include commenting, unsafe programming constructs, program layout, defensive programming, and so on. Commenting refers to how a program should be documented and to what level or degree. Unsafe programming constructions are practices that can make the program hard to maintain. An example is *goto* statements. Program layout refers to how a standard program should be laid out on a page, indentation of control constructs, and initialization. A defensive programming practice describes the mandatory components of the defensive programming strategy. An example is error condition handling and transfer of control to a common error routine. Static analysis techniques, such as structured walkthroughs and inspections, are used to ensure the proper form of the program code and documentation. This is accomplished by checking adherence to coding and documentation conventions and type checking.

Each defect uncovered during the coding phase review should be documented, categorized, recorded, presented to the design team for correction, and referenced to the specific document in which the defect was noted. Table 11.1 shows a sample coding phase defect recording form (see "Coding Phase Defect Checklist," for more details).

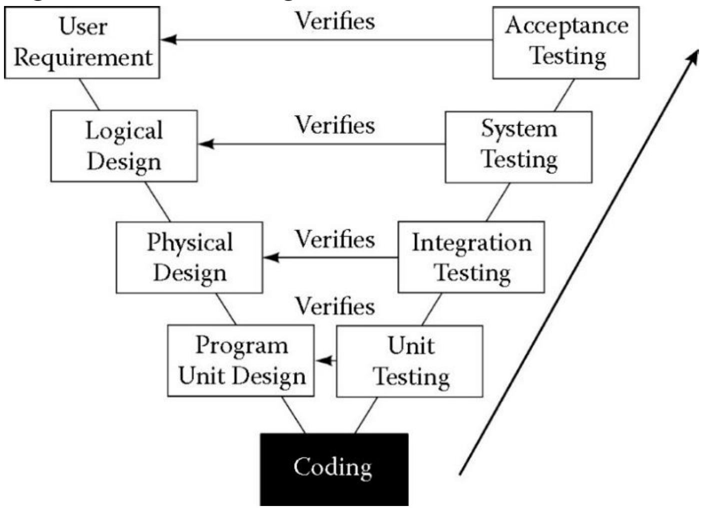
Table 11.1: Coding Phase Defect Recording

Defect Category	*Missing*	*Wrong*	*Extra*	*Total*
1. Decision logic or sequencing is erroneous or inadequate				
2. Arithmetic computations are erroneous or inadequate				
3. Branching is erroneous				
4. Branching or other testing is performed incorrectly				
5. There are undefined loop terminations				
6. Programming language rules are violated				
7. Programming standards are violated				
8. The programmer misinterprets language constructs				
9. Typographical errors exist				
10. Main storage allocation errors exist				

Executing the Test Plan
By the end of this phase, all the items in each section of the test plan should have been completed. The actual testing of software is accomplished through the test data in the test plan developed during the requirements, logical design, physical design, and program unit design phases. Because results have been specified in the test cases and test procedures, the correctness of the executions is ensured from a static test point of view; that is, the tests have been reviewed manually.

Dynamic testing, or time-dependent techniques, involves executing a specific sequence of instructions with the computer. These techniques are used to study the functional and computational correctness of the code.

Dynamic testing proceeds in the opposite order of the development life cycle. It starts with unit testing to verify each program unit independently and then proceeds to integration, system, and acceptance testing. After acceptance testing has been completed, the system is ready for operation and maintenance. Figure 11.1 briefly describes each testing type.

Figure 11.1: Executing the tests


Unit Testing
Unit testing is the basic level of testing. Unit testing focuses separately on the smaller building blocks of a program or system. It is the process of executing each module to confirm that each performs its assigned function. The advantage of unit testing is that it permits the testing and debugging of small units, thereby providing a better way to manage the integration of the units into larger units. In addition, testing a smaller unit of code makes it mathematically possible to fully test the code's logic with fewer tests. Unit testing also facilitates automated testing because the behavior of smaller units can be captured and played back with maximized reusability. A unit can be one of several types of application software. Examples include the module itself as a unit, GUI components such as windows, menus, and functions, batch programs, online programs, and stored procedures.

Integration Testing

After unit testing is completed, all modules must be integration-tested. During integration testing, the system is slowly built up by adding one or more modules at a time to the core of already-integrated modules. Groups of units are fully tested before system testing occurs. Because modules have been unit-tested prior to integration testing, they can be treated as black boxes, allowing integration testing to concentrate on module interfaces. The goals of integration testing are to verify that each module performs correctly within the control structure and that the module interfaces are correct.

Incremental testing is performed by combining modules in steps. At each step one module is added to the program structure, and testing concentrates on exercising this newly added module. When it has been demonstrated that a module performs properly with the program structure, another module is added, and testing continues. This process is repeated until all modules have been integrated and tested.

System Testing

After integration testing, the system is tested as a whole for functionality and fitness of use based on the System/Acceptance Test Plan. Systems are fully tested in the computer operating environment before acceptance testing occurs. The sources of the system tests are the quality attributes that were specified in the Software Quality Assurance Plan. System testing is a set of tests to verify these quality attributes and ensure that the acceptance test occurs in a relatively trouble-free manner. System testing verifies that the functions are carried out correctly. It also verifies that certain nonfunctional characteristics are present. Some examples include usability testing, performance testing, stress testing, compatibility testing, conversion testing, and document testing.

Black-box testing is a technique that focuses on testing a program's functionality against its specifications. White-box testing is a testing technique in which paths of logic are tested to determine how well they produce predictable results. Gray-box testing is a combination of these two approaches and is usually applied during system testing. It is a compromise between the two and is a well-balanced testing approach that is widely used during system testing.

Acceptance Testing

After systems testing, acceptance testing certifies that the software system satisfies the original requirements. This test should not be performed until the software has successfully completed systems testing. Acceptance testing is a user-run test that uses black-box techniques to test the system against its specifications. The end users are responsible for ensuring that all relevant functionality has been tested.

The acceptance test plan defines the procedures for executing the acceptance tests and should be followed as closely as possible. Acceptance testing continues even when errors are found, unless an error itself prevents continuation. Some projects do not require formal acceptance testing. This is true when the customer or user is satisfied with the other system tests, when timing requirements demand it, or when end users have been involved continuously throughout the development cycle and have been implicitly applying acceptance testing as the system is developed.
Acceptance tests are often a subset of one or more system tests. Two other ways to measure acceptance testing are as follows:

1. *Parallel Testing*—A business-transaction-level comparison with the existing system to ensure that adequate results are produced by the new system.
2. *Benchmarks*—A static set of results produced either manually or from an existing system is used as expected results for the new system.

Defect Recording

Each defect discovered during the foregoing tests is documented to assist in the proper recording of these defects. A problem report is generated when a test procedure gives rise to an event that cannot be explained by the tester. The problem report documents the details of the event and includes at least these items (see "Defect Report," for more details):

- Problem identification
- Author
- Release/build number
- Open date
- Close date
- Problem area
- Defect or enhancement
- Test environment
- Defect type
- Who detected
- How detected
- Assigned to
- Priority
- Severity
- Status

Other test reports to communicate the testing progress and results include a test case log, test log summary report, and system summary report.

A test case log documents the test cases for a test type to be executed. It also records the results of the tests, which provides the detailed evidence for the test log summary report and enables reconstructing testing, if necessary. (See "Test Case Log," for more information.)

A test log summary report documents the test cases from the tester's logs in progress or completed for the status reporting and metric collection. (See “Test Log Summary Report.”)

A system summary report should be prepared for every major testing event. Sometimes it summarizes all the tests. It typically includes the following major sections: general information (describing the test objectives, test environment, references, etc.), test results and findings (describing each test), software functions and findings, and analysis and test summary. (See “System Summary Report,” for more details.)

CHAPTER 5: SPIRAL (AGILE) SOFTWARE TESTING METHODOLOGY—PLAN, DO, CHECK, ACT

Spiral development methodologies are a reaction to the traditional waterfall systems development, in which the product evolves in sequential phases. A common problem with the life-cycle development model is that the elapsed time to deliver the product can be excessive, with user involvement only at the very beginning and very end. As a result, the system that they are given is often not what they originally requested.

By contrast, spiral development expedites product delivery. A small but functioning initial system is built and quickly delivered, and then enhanced in a series of iterations. One advantage is that the users receive at least some functionality quickly. Another advantage is that the product can be shaped by iterative feedback; for example, users do not have to define every feature correctly and in full detail at the beginning of the development cycle, but can react to each iteration.

Spiral testing is dynamic and may never be completed in the traditional sense of a delivered system's completeness. The term spiral refers to the fact that the traditional sequence of analysis—design—code—test phases is performed on a microscale within each spiral or cycle in a short period of time, and then the phases are repeated within each subsequent cycle.

The objectives of this section are to:

- Discuss the limitations of waterfall development.
- Describe the complications of client/server.
- Discuss the psychology of spiral testing.
- Describe the iterative/spiral development environment.
- Apply Deming's continuous quality improvement to a spiral development environment in terms of:
    - Information gathering
    - Test planning
    - Test case design
    - Test development
    - Test execution/evaluation
    - Traceability/coverage matrix
    - Preparing for the next spiral
    - System testing
    - Acceptance testing
    - Summarizing/reporting spiral test results

Section 12: Development Methodology Overview

Limitations of Life-Cycle Development

In Chapter 2, “Waterfall Testing Review,” the waterfall development methodology was reviewed along with the associated testing activities. The life-cycle development methodology consists of distinct phases from requirements to coding. Life-cycle testing means that testing occurs in parallel with the development life cycle and is a continuous process. Although the life-cycle or waterfall development is very effective for many large applications requiring a lot of computer horsepower, for example, DOD, financial, security-based, and so on, it has a number of shortcomings:

- The end users of the system are only involved at the very beginning and the very end of the process. As a result, the system that they were given at the end of the development cycle is often not what they originally visualized or thought they requested.
- The long development cycle and the shortening of business cycles lead to a gap between what is really needed and what is delivered.
- End users are expected to describe in detail what they want in a system, before the coding phase. This may seem logical to developers; however, there are end users who have not used a computer system before and are not certain of its capabilities.
- When the end of a development phase is reached, it is often not quite complete, but the methodology and project plans require that development press on regardless. In fact, a phase is rarely complete, and there is always more work than can be done. This results in the “rippling effect”; sooner or later, one must return to a phase to complete the work.
- Often, the waterfall development methodology is not strictly followed. In the haste to produce something quickly, critical parts of the methodology are not followed. The worst case is ad hoc development, in which the analysis and design phases are bypassed and the coding phase is the first major activity. This is an example of an unstructured development environment.
- Software testing is often treated as a separate phase starting in the coding phase as a validation technique and is not integrated into the whole development life cycle.
- The waterfall development approach can be woefully inadequate for many development projects, even if it is followed. An implemented software system is not worth very much if it is not the system the user wanted. If the requirements are incompletely documented, the system will not survive user validation procedures; that is, it is the wrong system. Another variation is when the requirements are correct, but the design is inconsistent with the requirements. Once again, the completed product will probably fail the system validation procedures.
- Because of the foregoing issues, experts began to publish methodologies based on other approaches, such as prototyping.

The Client/Server Challenge

The client/server architecture for application development divides functionality between a client and server so that each performs its task independently. The client cooperates with the server to produce the required results.

The client is an intelligent workstation used as a single user, and because it has its own operating system, it can run other applications such as spreadsheets, word processors, and file processors. The user and the server process client/server application functions cooperatively. The server can be a PC, minicomputer, local area network, or even a mainframe. The server receives requests from the clients and processes them. The hardware configuration is determined by the application's functional requirements.

Some advantages of client/server applications include reduced costs, improved accessibility of data, and flexibility. However, justifying a client/server approach and ensuring quality are difficult and present additional difficulties not necessarily found in mainframe applications. Some of these problems include the following:

- The typical graphical user interface has more possible logic paths, and thus the large number of test cases in the mainframe environment is compounded.
- Client/server technology is complicated and, often, new to the organization. Furthermore, this technology often comes from multiple vendors and is used in multiple configurations and in multiple versions.
- The fact that client/server applications are highly distributed results in a large number of failure sources and hardware/software configuration control problems.
- A short- and long-term cost—benefit analysis must be performed to justify client/ server technology in terms of the overall organizational costs and benefits.
- Successful migration to a client/server depends on matching migration plans to the organization's readiness for client/server technology.
- The effect of client/server technology on the user's business may be substantial.
- Choosing which applications will be the best candidates for a client/server implementation is not straightforward.
- An analysis needs to be performed of which development technologies and tools enable a client/server.
- Availability of client/server skills and resources, which are expensive, needs to be considered.
- Although client/server technology is more expensive than mainframe computing, cost is not the only issue. The function, business benefit, and the pressure from end users have to be balanced.

Integration testing in a client/server environment can be challenging. Client and server applications are built separately. When they are brought together, conflicts can arise no matter how clearly defined the interfaces are. When integrating applications, defect resolutions may have single or multiple solutions, and there must be open communication between quality assurance and development.

In some circles there exists a belief that the mainframe is dead and the client/ server prevails. The truth of the matter is that applications using mainframe architecture are not dead, and client/server technology is not necessarily the panacea for all applications. The two will continue to coexist and complement each other in the future. Mainframes will certainly be part of any client/server strategy.

Psychology of Client/Server Spiral Testing

The New School of Thought

The psychology of life-cycle testing encourages testing by individuals outside the development organization. The motivation for this is that with the life-cycle approach, there typically exist clearly defined requirements, and it is more efficient for a third party to verify these. Testing is often viewed as a destructive process designed to break development's work.

The psychology of spiral testing, on the other hand, encourages cooperation between quality assurance and the development organization. The basis of this argument is that, in a rapid application development environment, requirements may or may not be available, to varying degrees. Without this cooperation, the testing function would have a difficult task defining the test criteria. The only possible alternative is for testing and development to work together.

Testers can be powerful allies to development, and with a little effort, they can be transformed from adversaries into partners. This is possible because most testers want to be helpful; they just need a little consideration and support. To achieve this, however, an environment needs to be created to bring out the best of a tester's abilities. The tester and development manager must set the stage for cooperation early in the development cycle and communicate throughout the cycle.

Tester/Developer Perceptions

The relationship between testers and developers is often strained due to differing perspectives on their roles. Testing is challenging and indefinite, as testers can never be sure they have found all the issues. Many testers lack proper training, and the rigid focus on formal requirements for effective testing is often unrealistic in real-world software projects. Good testers are critical thinkers, motivated by quality, quick learners, team players, and skilled communicators.

Developers, however, see their work as tangible and often feel testing is a lesser role. This can lead to uncooperative relationships. Developers aim to build successful software, while testers focus on minimizing failure and improving quality by finding defects. The two roles can be in conflict, especially as testers are often involved late in the development process, despite their broader focus on overall software quality.

Project Goal: Integrate QA and Development
To integrate testing and development effectively, testers should not give the impression they are trying to "break the code." Instead, they should position themselves as allies in improving quality, helping developers by identifying areas for enhancement. Developers need to recognize the importance of quality and see testers as integral team members. Project managers should emphasize the value of testing throughout the development cycle.

Testers should work in parallel with development, staying informed about progress and attending planning and status meetings to prevent last-minute issues. Effective communication between testers, developers, and users is essential for improving quality and avoiding redundant tasks. Testers should review documentation and understand the objectives and status of the software, while developers need clear feedback about issues found during testing.

A collaborative relationship between testing and development is crucial, with both sides sharing knowledge and working toward the common goal of delivering a high-quality product.

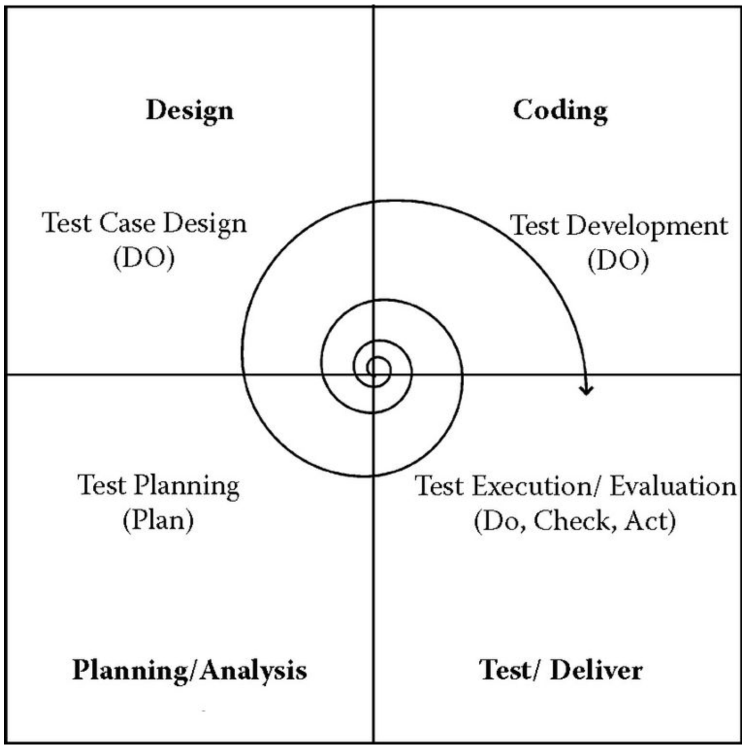
Iterative/Spiral Development Methodology
Spiral methodologies were developed in response to the limitations of the traditional waterfall model, particularly its lengthy product delivery times. In contrast, the spiral approach allows for faster product delivery by building a small, functional initial system and enhancing it through multiple iterations. This provides clients with early functionality and allows for iterative feedback, enabling product refinement over time without the need for all features to be defined upfront.

The spiral model involves repeatedly cycling through the phases of analysis, design, coding, and testing on a smaller scale in each iteration, allowing the product to evolve continuously. Unlike traditional methods, where product definitions are frozen, the specifications in a spiral development environment are constantly evolving, making it difficult to finalize requirements.

To effectively test in a spiral environment, quality assurance teams must be closely integrated with development, testing each new version as it is released. Testing iterations must be brief to avoid disrupting the frequent delivery schedule, focusing primarily on newly added or modified features. If resources permit, automated regression tests should be performed to ensure overall system integrity. Rapid turnaround on client requests is common, necessitating efficient automated regression testing before releasing new versions.

Spiral testing is a process of working from a base and building a system incrementally. Upon reaching the end of each phase, developers reexamine the entire structure and revise it. Drawing the four major phases of system development— planning/analysis, design, coding, and test/deliver—into quadrants, as shown in Figure 12.1, represents the spiral approach. The respective testing phases are test planning, test case design, test development, and test execution/evaluation.

Figure 12.1: Spiral testing process



The spiral process starts with planning and requirements analysis, followed by designing and building the system's base functionality, which is then tested. After each iteration, users provide feedback, leading to further improvements in the next iteration. This cycle continues until users and developers agree the system is ready for implementation.

The spiral approach ensures user involvement and allows the system to adapt to changing business needs. However, a potential flaw is "spiral death," where the project loops endlessly without producing a final system. Additionally, the flexibility of the spiral model may cause the development team to lose focus on user needs, leading to product failure.

Quality assurance plays a crucial role in ensuring that user requirements are met. A variation of the spiral model is the iterative methodology, which sets a fixed number of iterations and goals, forcing the system to be implemented after the final iteration, even if it isn't fully complete, making it valuable to the user.

Role of JADs
During the first spiral, the major deliverables are the objectives, an initial functional decomposition diagram, and a functional specification. The functional specification also includes an external (user) design of the system. It has been shown that errors defining the requirements and external design are the most expensive to fix later in development. It is, therefore, imperative to get the design as correct as possible the first time.

A technique that helps accomplish this is joint application design sessions (see "JADs," for more details). Studies show that JADs increase productivity over traditional design techniques. In JADs, users and IT professionals jointly design systems in facilitated group sessions. JADs go beyond the one-on-one interviews to collect information. They promote communication, cooperation, and teamwork among the participants by placing the users in the drivers' seats.

JADs are logically divided into phases: customization, session, and wrap-up. Regardless of what activity one is pursuing in development, these components will always exist. Each phase has its own objectives.

Role of Prototyping
Prototyping is an iterative approach often used to build systems that users initially are unable to describe precisely (see "Prototyping," for more details). The concept is made possible largely through the power of fourth-generation languages (4GLs) and application generators.

Prototyping is, however, as prone to defects as any other development effort, maybe more so if not performed in a systematic manner. Prototypes need to be tested as thoroughly as any other system. Testing can be difficult unless a systematic process has been established for developing prototypes. There are various types of software prototypes, ranging from simple printed descriptions of input, processes, and output to completely automated versions. An exact definition of a software prototype is impossible to find; the concept is made up of various components. Among the many characteristics identified by MIS professionals are the following:

- Comparatively inexpensive to build (i.e., less than 10 percent of the full system's development cost).
- Relatively quick development so that it can be evaluated early in the life cycle.
- Provides users with a physical representation of key parts of the system before implementation.
- Prototypes:
    - Do not eliminate or reduce the need for comprehensive analysis and specification of user requirements.
    - Do not necessarily represent the complete system.
    - Perform only a subset of the functions of the final product.

- Lack the speed, geographical placement, or other physical characteristics of the final system.

Basically, prototyping is the building of trial versions of a system. These early versions can be used as the basis for assessing ideas and making decisions about the complete and final system. Prototyping is based on the premise that, in certain problem domains (particularly in online interactive systems), users of the proposed application do not have a clear and comprehensive idea of what the application should do or how it should operate.

Often, errors or shortcomings overlooked during development appear after a system becomes operational. Application prototyping seeks to overcome these problems by providing users and developers with an effective means of communicating ideas and requirements before a significant amount of development effort has been expended. The prototyping process results in a functional set of specifications that can be fully analyzed, understood, and used by users, developers, and management to decide whether an application is feasible and how it should be developed.

Fourth-generation languages have enabled many organizations to undertake projects based on prototyping techniques. They provide many of the capabilities necessary for prototype development, including user functions for defining and managing the user—system interface, data management functions for organizing and controlling access, and system functions for defining execution control and interfaces between the application and its physical environment.
In recent years, the benefits of prototyping have become increasingly recognized. Some include the following:

- Prototyping emphasizes active physical models. The prototype looks, feels, and acts like a real system.
- Prototyping is highly visible and accountable.
- The burden of attaining performance, optimum access strategies, and complete functioning is eliminated in prototyping.
- Issues of data, functions, and user—system interfaces can be readily addressed.
- Users are usually satisfied, because they get what they see.
- Many design considerations are highlighted, and a high degree of design flexibility becomes apparent.
- Information requirements are easily validated.
- Changes and error corrections can be anticipated and, in many cases, made on the spur of the moment.
- Ambiguities and inconsistencies in requirements become visible and correctable.
- Useless functions and requirements can be quickly eliminated.

Methodology for Developing Prototypes
The following describes a methodology to reduce development time through reuse of the prototype and knowledge gained in developing and using the prototype. It does not include how to test the prototype within spiral development. This is included in the next part.

Step 1: Develop the Prototype
In the construction phase of spiral development, the external and screen designs are transformed into functional prototypes using 4GL tools like Visual Basic or PowerBuilder. These prototypes focus on the "look and feel" of the user interface, without incorporating detailed business functionality. The team builds a prototype system with data entry screens, reports, file routines, procedures, and menus, based on the logical database structure created earlier.

The iterative process of prototype development involves several steps:

1. Define database structures for test data.
2. Create report formats using query languages to automate formatting.
3. Design interactive data entry screens with appropriate prompts and validation.
4. Develop external file routines to handle batch data processing.
5. Implement algorithms and procedures needed for both the prototype and final system.
6. Build procedure selection menus for user-friendly functionality.
7. Test data validation, procedures, and overall system execution.

This process repeats through several iterations, refining the system until changes become largely cosmetic. Once the prototype is well-developed, a draft user manual is created, documenting screens, reports, database structures, and procedures to give a clear understanding of the system's operation.

Step 2: Demonstrate Prototypes to Management
The purpose of this demonstration is to give management the option of making strategic decisions about the application on the basis of the prototype's appearance and objectives. The demonstration consists primarily of a short description of each prototype component and its effects, and a walkthrough of the typical use of each component. Every person in attendance at the demonstration should receive a copy of the draft user manual, if one is available.

The team should emphasize the results of the prototype and its impact on development tasks still to be performed. At this stage, the prototype is not necessarily a functioning system, and management must be made aware of its limitations.

Step 3: Demonstrate Prototype to Users
There are pros and cons to allowing prospective users to interact with a prototype system. A key risk is that users may develop unrealistic expectations or become attached to the prototype, even when the final production system is ready. However, using the prototype can help users quickly identify procedural issues and undesirable behaviors.

Prototypes should be demonstrated to a representative group of users, with a clear explanation that it is not the final product, but a flexible version meant to identify errors from the users' perspective. Feedback from these demonstrations typically leads to requests for changes, error corrections, and suggestions for improvement. The prototyping team then iterates through the process to implement these changes.

Regular demonstrations following each iteration strengthen user involvement by showing how their feedback has been incorporated. These sessions help ensure user satisfaction and ownership while also allowing necessary changes in the system's scope or design to be made early, which is less costly than making modifications in later stages of development.

Step 4: Revise and Finalize Specifications
At this stage, the prototype includes key components such as data entry formats, report formats, file formats, a logical database structure, algorithms, procedures, selection menus, system flow, and possibly a draft user manual.

The deliverables from this phase include formal system requirements, listings of 4GL command files for each programmed object (e.g., screens, reports, database structures), sample reports, data entry screens, the logical database structure, a data dictionary, and a risk analysis. The risk analysis outlines issues and changes not incorporated into the prototype and their potential impact on full system development and operation.

The prototyping team reviews all components for inconsistencies or omissions, makes corrections, and formally documents the specifications.

Step 5: Develop the Production System
At this point, development can proceed in one of three directions:

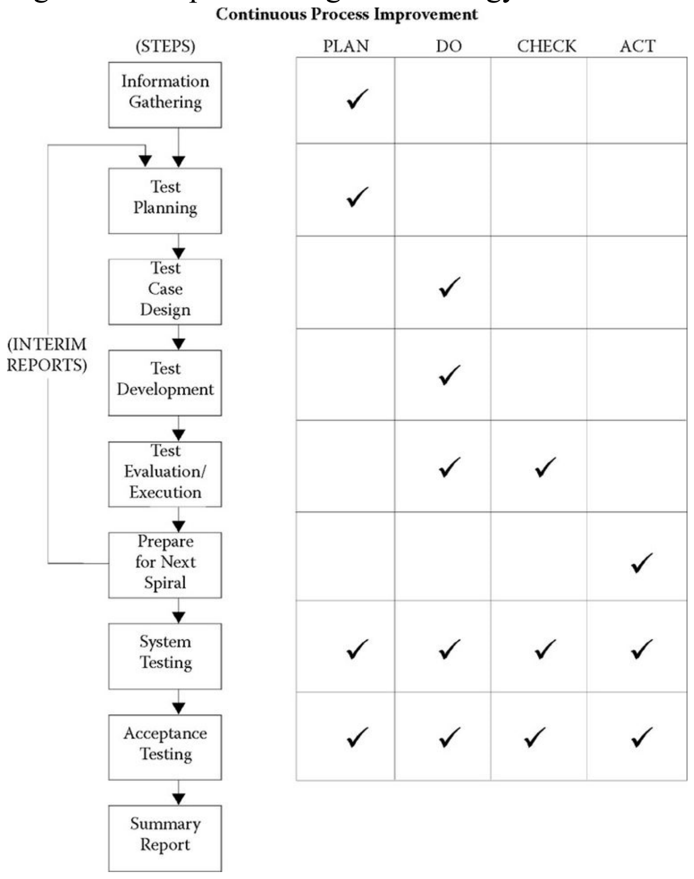
1. The project is suspended or canceled because the prototype has uncovered insurmountable problems or the environment is not ready to mesh with the proposed system.
2. The prototype is discarded because it is no longer needed or because it is too inefficient for production or maintenance.
3. Iterations of prototype development are continued, with each iteration adding more system functions and optimizing performance until the prototype evolves into the production system.

The decision on how to proceed is generally based on such factors as:

- The actual cost of the prototype
- Problems uncovered during prototype development
- The availability of maintenance resources
- The availability of software technology in the organization
- Political and organizational pressures
- The amount of satisfaction with the prototype
- The difficulty in changing the prototype into a production system
- Hardware requirements

Figure 12.3 provides an overview of the spiral testing methodology by relating each step to the PDCA quality model. “Spiral Testing Methodology,” provides a detailed representation of each part of the methodology. The methodology provides a framework for testing in this environment. The major steps include information gathering, test planning, test design, test development, test execution/ evaluation, and preparing for the next spiral. It includes a set of tasks associated with each step or a checklist from which the testing organization can choose based on its needs. The spiral approach flushes out the system functionality. When this has been completed, it also provides for classical system testing, acceptance testing and summary reports.

Figure 12.3: Spiral testing methodology



(STEPS)	PLAN	DO	CHECK	ACT
Information Gathering	✓			
Test Planning	✓			
Test Case Design		✓		
Test Development		✓		
Test Evaluation/ Execution		✓	✓	
Prepare for Next Spiral				✓
System Testing	✓	✓	✓	✓
Acceptance Testing	✓	✓	✓	✓
Summary Report				

Section 13: Information Gathering (Plan)

Overview

You will recall that, in the spiral development environment, software testing is described as a continuous improvement process that must be integrated into a rapid application development methodology. Deming's continuous improvement process using the PDCA model (see Figure 13.1) is applied to the software testing process. We are now in the Plan part of the spiral model.

Figure 13.1: Spiral testing and continuous improvement

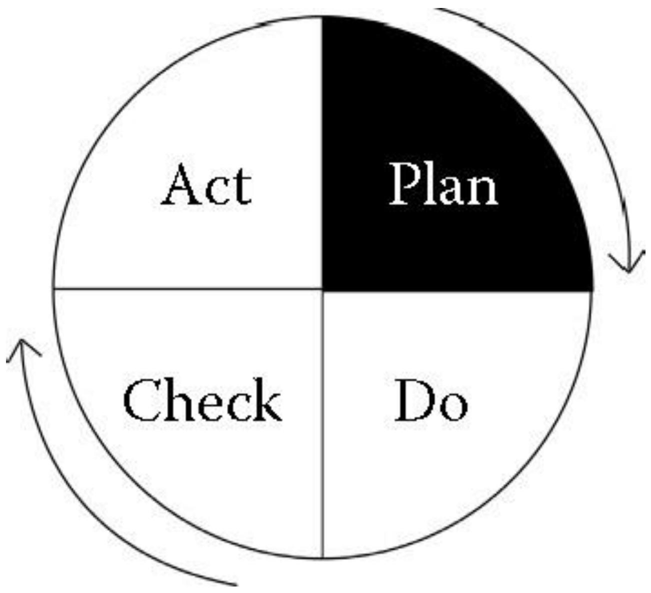
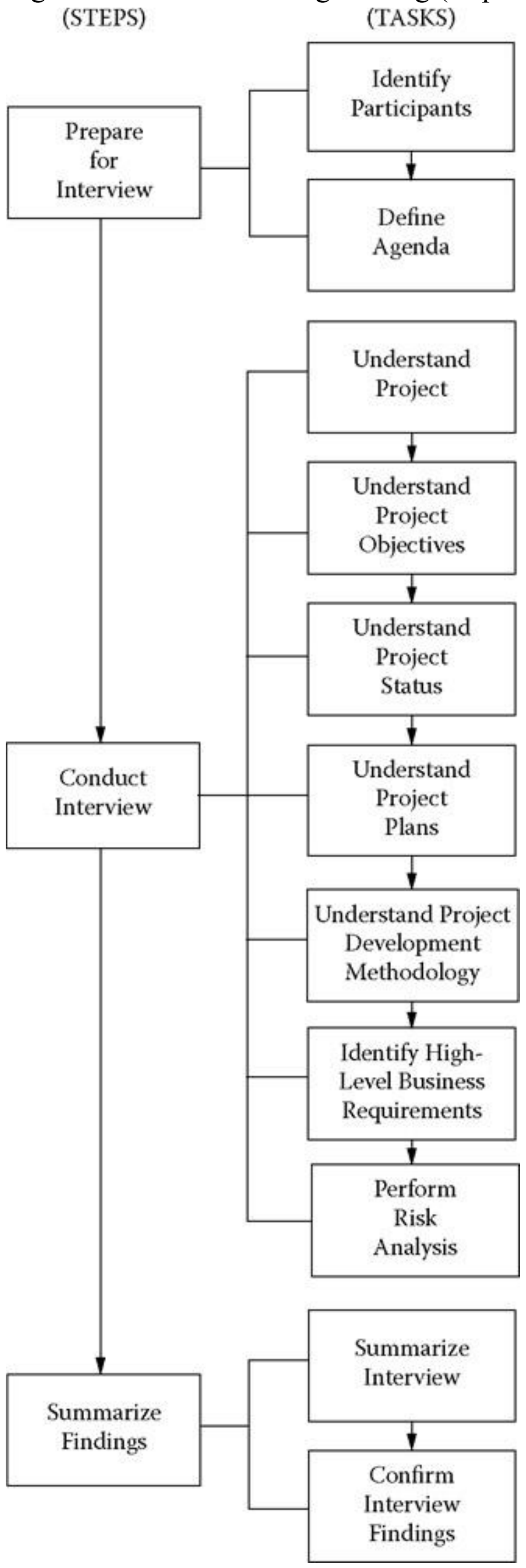


Figure 13.2 outlines the steps and tasks associated with information gathering within the Plan part of spiral testing. Each step and task is described along with valuable tips and techniques.

Figure 13.2: Information gathering (steps/tasks)



The purpose of gathering information is to obtain information relevant to the software development project and organize it, to understand the scope of the development project and start building a test plan. Other interviews may occur during the development project, as necessary.

Proper preparation is critical to the success of the interview. Before the interview, it is important to clearly identify the objectives of the interview and communicate them to all parties, identify the quality assurance representative who will lead the interview, and identify the scribe; schedule a time and place; prepare any required handouts; and communicate what is required from development.
Although many interviews are unstructured, the interviewing steps and tasks shown in Figure 13.2 will be helpful.

Step 1: Prepare for the Interview

Task 1: Identify the Participants
It is recommended that there be no more than two interviewers representing quality assurance. It is helpful for one of these to assume the role of questioner, with the other taking detailed notes. This will allow the interviewer to focus on soliciting information. Ideally, the interviewer should be the manager responsible for the project-testing activities. The scribe, or note taker, should be a test engineer or lead tester assigned to the project; the scribe supports the interviewer and records each pertinent piece of information and lists the issues, the assumptions, and questions.

The recommended development participants attending include the project sponsor, development manager, or a senior development team member. Although members of the development team can take notes, this is the responsibility of the scribe. Having more than one scribe can result in confusion, because multiple sets of notes will eventually have to be consolidated. The most efficient approach is for the scribe to take notes, and summarize at the end of the interview. (See "Project Information Gathering Checklist," which can be used to verify the information available and required at the beginning of the project.)

Task 2: Define the Agenda
The key factor for a successful interview is a well-thought-out agenda. It should be prepared by the interviewer ahead of time and agreed upon by the development leader. The agenda should include an introduction, specific points to cover, and a summary section. The main purpose of an agenda is to enable the testing manager to gather enough information to scope out the quality assurance activities and begin a test plan. Table 13.1 depicts a sample agenda (details are described in "Step 2: Conduct the Interview").

Table 13.1: Interview Agenda

I.	Introductions
II.	Project Overview
III.	Project Objectives
IV.	Project Status
V.	Project Plans
VI.	Development Methodology
VII.	High-Level Requirements
VIII.	Project Risks and Issues
IX.	Summary

Step 2: Conduct the Interview

A good interview contains certain elements. The first is defining what will be discussed, or "talking about what we are going to talk about." The second is discussing the details, or "talking about it." The third is summarizing, or "talking about what we talked about." The final element is timeliness. The interviewer should state up front the estimated duration of the interview and set the ground rule that if the allotted time expires before completing all items on the agenda, a follow-on interview will be scheduled. This is difficult, particularly when the interview is into the details, but nonetheless it should be followed.

Task 1: Understand the Project

Before getting into the project details, the interviewer should state the objectives of the interview and present the agenda. As with any type of interview, he or she should indicate that only one individual should speak, no interruptions should occur until the speaker acknowledges a request, and the focus should be on the material being presented.

The interviewer should then introduce himself or herself, introduce the scribe, and ask the members of the development team to introduce themselves. Each should indicate name, title, specific roles and job responsibilities, as well as expectations of the interview. The interviewer should point out that the purpose of this task is to obtain general project background information.

The following general questions should be asked to solicit basic information:

- What is the name of the project?
- What are the high-level project objectives?
- Who is the audience (users) of the system to be developed?
- When was the project started?
- When is it anticipated to be complete?
- What is the status of the project?
- What is the projected effort in person-months?
- Is this a new, maintenance, or package development project?

- What are the major problems, issues, and concerns?
- Are there plans to address problems and issues?
- Is the budget on schedule?
- Is the budget too tight, too loose, or about right?
- What organizational units are participating in the project?
- Is there an established organization chart?
- What resources are assigned to each unit?
- What is the decision-making structure; that is, who makes the decisions?
- What are the project roles and the responsibilities associated with each role?
- Who is the resource with whom the test team will communicate on a daily basis?
- Has a quality management plan been developed?
- Has a periodic review process been set up?
- Has a representative from the user community been appointed to represent quality?

Task 2: Understand the Project Objectives
To develop a test plan for a development project, it is important to understand the objectives of the project. The purpose of this task is to understand the scope, needs, and high-level requirements of this project.

The following questions should be asked to solicit basic information:
- Purpose:
  - — What type of system is being developed, for example, payroll, order entry, inventory, or accounts receivable/payable?
  - — Why is the system being developed?
  - — What subsystems are involved?
  - — What are the subjective requirements, for example, ease of use, efficiency, morale, flexibility?
- Scope:
  - — Who are the users of the system?
  - — What are the users' job titles and roles?
  - — What are the major functions and subfunctions of the system?
  - — What functions will not be implemented?
  - — What business procedures are within the scope of the system?
  - — Are there analysis diagrams, such as business flow diagrams, data flow diagrams, or data models, to describe the system?
  - — Have project deliverables been defined along with completeness criteria?
- Benefits:
  - — What are the anticipated benefits that will be provided to the user with this system?
    - Increased productivity

        - Improved quality
        - Cost savings
        - Increased revenue
        - More competitive advantage
- Strategic:
    - — What are the strategic or competitive advantages?
    - — What impact will the system have on the organization, customers, legal, government, and so on?
- Constraints:
    - — What are the financial, organizational, personnel, technological constraints, or limitations of the system?
    - — What business functions and procedures are out of the scope of the system?

Task 3: Understand the Project Status
The purpose of this task is to understand where the project is at this point, which will help define how to plan the testing effort. For example, if this is the first interview and the project is at the stage of coding the application, the testing effort is already behind schedule. The following questions should be asked to solicit basic information:

- Has a detailed project work plan, including activities, tasks, dependencies, resource assignments, work effort estimates, and milestones, been developed?
- Is the project on schedule?
- Is the completion time too tight?
- Is the completion time too loose?
- Is the completion time about right?
- Have there been any major slips in the schedule that will have an impact on the critical path?
- How far is the project from meeting its objectives?
- Are the user functionality and quality expectations realistic and being met?
- Are the project work effort hours trends on schedule?
- Are the project costs trends within the budget?
- What development deliverables have been delivered?

Task 4: Understand the Project Plans
Because the testing effort needs to track development, it is important to understand the project work plans. The following questions should be asked to solicit basic information:

- Work breakdown:
    - — Has a Microsoft Project (or other tool) plan been developed?
    - — How detailed is the plan; for example, how many major and bottom-level tasks have been identified?
    - — What are the major project milestones (internal and external)?

- Assignments:
    - — Have appropriate resources been assigned to each work plan?
    - — Is the work plan well balanced?
    - — What is the plan to stage resources?
- Schedule:
    - — Is the project plan on schedule?
    - — Is the project plan behind schedule?
    - — Is the plan updated periodically?

Task 5: Understand the Project Development Methodology
The testing effort must integrate with the development methodology. If considered a separate function, it may not receive the appropriate resources and commitment. When testing is integrated with development, the latter should not proceed without the former. Testing steps and tasks need to be integrated into the systems development methodology through addition or modification of tasks. Specifically, the testing function needs to know when in the development methodology test design can start. It also needs to know when the system will be available for execution and the recording and correction of defects.

The following questions should be asked to solicit basic information:

- What is the methodology?
- What development and project management methodology does the development organization use?
- How well does the development organization follow the development methodology?
- Is there room for interpretation or flexibility?
- Standards:
    - — Are standards and practices documented?
    - — Are the standards useful or do they hinder productivity?
    - — How well does the development organization enforce standards?

Task 6: Identify the High-Level Business Requirements
A software requirements specification defines the functions of a particular software product in a specific environment. Depending on the development organization, it may vary from a loosely defined document with a generalized definition of what the application will do to a very detailed specification, as shown in "Requirements Specification." In either case, the testing manager must assess the scope of the development project to start a test plan.

The following questions should be asked to solicit basic information:

- *What are the high-level functions?* The functions at a high level should be enumerated. Examples include order processing, financial processing, reporting capability, financial planning, purchasing, inventory control, sales administration, shipping, cash flow analysis, payroll, cost accounting, and recruiting. This list defines what the application is supposed to do and gives the testing manager an idea of the level of test design and implementation required. The interviewer should solicit as much detail as possible, including a detailed breakdown of each function. If this detail is not available during the interview, a request for a detailed functional decomposition should be made, and it should be pointed out that this information is essential for preparing a test plan.

- *What are the system (minimum) requirements?* A description of the operating system version (Windows, etc.) and minimum microprocessor, disk space, RAM, and communications hardware should be provided.
- *What are the Windows or external interfaces?* The specification should define how the application should behave from an external viewpoint, usually by defining the inputs and outputs. It also includes a description of any interfaces to other applications or subsystems.
- *What are the performance requirements?* This includes a description of the speed, availability, data volume throughput rate, response time, and recovery time of various functions, stress, and so on. This serves as a basis for understanding the level of performance and stress testing that may be required.
- *What other testing attributes are required?* This includes such attributes as portability, maintainability, security, and usability. This serves as a basis for understanding the level of other system-level testing that may be required.
- *Are there any design constraints?* This includes a description of any limitation on the operating environments, database integrity, resource limits, implementation language standards, and so on.

Task 7: Perform Risk Analysis

The purpose of this task is to measure the degree of business risk in an application system to improve testing. This is accomplished in two ways: high-risk applications can be identified and subjected to more extensive testing, and risk analysis can help identify the error-prone components of an individual application so that testing can be directed at those components. This task describes how to use risk assessment techniques to measure the risk of an application under testing.

Computer Risk Analysis

Risk analysis is a formal method for identifying vulnerabilities (i.e., areas of potential loss). Any weakness that could be misused, intentionally or accidentally, and result in a loss to the organization is a vulnerability. Identification of risks allows the testing process to measure the potential effect of those vulnerabilities (e.g., the maximum loss that could occur if the risk or vulnerability were exploited).

Risk has always been a testing consideration. Individuals naturally try to anticipate problems and then test to determine whether additional resources and attention need to be directed at those problems. Often, however, risk analysis methods are both informal and ineffective.

Through proper analysis, the test manager should be able to predict the probability of such unfavorable consequences as the following:

- Failure to obtain all, or even any, of the expected benefits
- Cost and schedule overruns
- An inadequate system of internal control
- Technical performance of the resulting system that is significantly below the estimate
- Incompatibility of the system with the selected hardware and software

The following reviews the various methods used for risk analysis and the dimensions of computer risk, and then describes the various approaches to assigning risk priorities. There are three methods of performing risk analysis.

Method 1: Judgment and Instinct
This method of determining how much testing to perform enables the tester to compare the project with past projects to estimate the magnitude of the risk. Although this method can be effective, the knowledge and experience it relies on are not transferable but must be learned over time.

Method 2: Dollar Estimation
Risk is the probability of incurring loss. That probability is expressed through this formula:

$$(\text{Frequency of occurrence}) \times (\text{loss per occurrence}) = (\text{annual loss expectation})$$

Business risk based on this formula can be quantified in dollars. Often, however, the concept, not the formula, is used to estimate how many dollars might be involved if problems were to occur. The disadvantages of projecting risks in dollars are that such numbers (i.e., frequency of occurrence and loss per occurrence) are difficult to estimate and the method implies a greater degree of precision than may be realistic.

Method 3: Identifying and Weighting Risk Attributes
Experience has demonstrated that the major attributes causing potential risks are the project size, experience with the technology, and project structure. The larger the project is in dollar expense, staffing levels, elapsed time, and number of departments affected, the greater the risk.

Because of the greater likelihood of unexpected technical problems, project risk increases as the project team's familiarity with the hardware, operating systems, database, and application languages decreases. A project that has a slight risk for a leading-edge, large systems development department may carry a very high risk for a smaller, less technically advanced group. The latter group, however, can reduce its risk by purchasing outside skills for an undertaking that involves a technology in general commercial use.

In highly structured projects, the nature of the task defines the output completely, from the beginning. Such output is fixed during the life of the project. These projects carry much less risk than those whose output is more subject to the manager's judgment and changes.

The relationship among these attributes can be determined through weighting, and the testing manger can use weighted scores to rank application systems according to their risk. For example, this method can show application A is a higher risk than application B.

Risk assessment is applied by first weighting the individual risk attributes. For example, if an attribute is twice as important as another, it can be multiplied by the weight of two. The resulting score is compared with other scores developed for the same development organization and is used to determine a relative risk measurement among applications, but it is not used to determine an absolute measure.

Table 13.2 compares three projects using the weighted risk attribute method. Project size has a weight factor of 2, experience with technology has a weight factor of 3, and project structure has a weight factor of 1. When the project scores are each multiplied by each of the three weight factors, it becomes clear that project A has the highest risk.

Table 13.2: Identifying and Weighting Risk Attributes

Weighting Factor	**Project A (Score × Weight)**	**Project B (Score × Weight)**	**Project C (Score × Weight)**
Project size (2)	5 × 2 = 10	3 × 2 = 6	2 × 2 = 4
Experience with technology (3)	7 × 3 = 21	1 × 3 = 3	5 × 3 = 15
Project structure (1)	4 × 1 = 4	6 × 1 = 6	3 × 1 = 3
Total score	35	15	22

Information gathered during risk analysis can be used to allocate test resources to test application systems. For example, high-risk applications should receive extensive testing; medium-risk systems, less testing; and low-risk systems, minimal testing. The area of testing can be selected on the basis of high-risk characteristics. For example, if computer technology is a high-risk characteristic, the testing manager may want to spend more time testing how effectively the development team is using that technology.

Step 3: Summarize the Findings

Task 1: Summarize the Interview

After the interview is completed, the interviewer should review the agenda and outline the main conclusions. If a follow-up session is needed, one should be scheduled at this point while the members are present.

Typically, during the interview, the notes are unstructured and hard to follow by anyone except the note taker. However, the notes should have at least followed the agenda. After the interview concludes, the notes should be formalized into a summary report. This should be performed by the scribe note taker. The goal is to make the results of the session as clear as possible for quality assurance and the development organization. However, the interview leader may have to embellish the material or expand certain areas. (See "Minutes of the Meeting," which can be used to document the results and follow-up actions for the project information-gathering session).

Task 2: Confirm the Interview Findings

The purpose of this task is to bring about agreement between the interviewer and the development organization, to ensure an understanding of the project. After the interview notes are formalized, it is important to distribute the summary report to the other members who attended the interview. A sincere invitation for their comments or suggestions should be communicated. The interviewer should then actively follow up interview agreements and disagreements. Any changes should then be implemented. Once there is full agreement, the interviewer should provide a copy of the summary report.

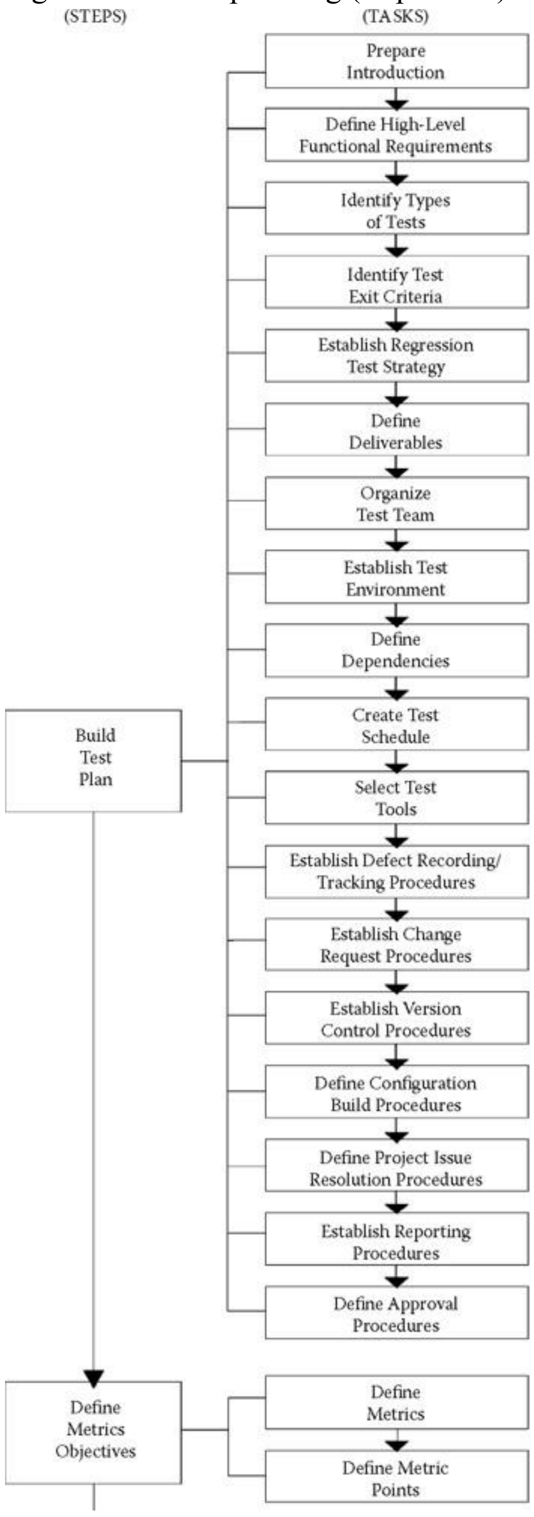
Section 14: Test Planning (Plan)

The test project plan is an ongoing document, particularly in the spiral environment, because the system is constantly changing. As the system changes, so does the test plan. A good test plan is one that:

- Has a good chance of detecting a majority of the defects
- Provides test coverage for most of the code
- Is flexible
- Is executed easily and automatically, and is repeatable
- Defines the types of tests to be performed
- Clearly documents the expected results
- Provides for defect reconciliation when a defect is discovered
- Clearly defines the test objectives
- Clarifies the test strategy
- Clearly defines the test exit criteria
- Is not redundant
- Identifies the risks
- Documents the test requirements
- Defines the test deliverables

Although there are many ways a test plan can be created, Figure 14.1 provides a framework that includes most of the essential planning considerations. It can be treated as a checklist of test items to consider. Some of the items, such as defining the test requirements and test team, are obviously required; however, others may not be. It depends on the nature of the project and the time constraints.

Figure 14.1: Test planning (steps/tasks)


(STEPS)
(TASKS)
Prepare Introduction
Define High-Level Functional Requirements
Identify Types of Tests
Identify Test Exit Criteria
Establish Regression Test Strategy
Define Deliverables
Organize Test Team
Establish Test Environment
Define Dependencies
Create Test Schedule
Select Test Tools
Establish Defect Recording/ Tracking Procedures
Establish Change Request Procedures
Establish Version Control Procedures
Define Configuration Build Procedures
Define Project Issue Resolution Procedures
Establish Reporting Procedures
Define Approval Procedures
Build Test Plan
Define Metrics Objectives
Define Metrics
Define Metric Points

The planning test methodology includes three steps: building the test project plan, defining the metrics, and reviewing/approving the test project plan. Each of these is then broken down into its respective tasks, as shown in Figure 14.1.

Step 1: Build a Test Plan

Task 1: Prepare an Introduction
The first bit of test plan detail is a description of the problems to be solved by the application of the associated opportunities. This defines the summary background, describing the events or current status leading up to the decision to develop the application. Also, the application's risks, purpose, objectives, and benefits, and the organization's critical success factors should be documented in the introduction. A critical success factor is a measurable item that will have a major influence on whether a key function meets its objectives. An objective is a measurable end state that the organization strives to achieve. Examples of objectives include the following:

- New product opportunity
- Improved efficiency (internal and external)
- Organizational image
- Growth (internal and external)
- Financial (revenue, cost profitability, etc.)
- Competitive position
- Market leadership

The introduction should also include an executive summary description. The executive sponsor (often called the project sponsor) is the individual who has ultimate authority over the project. This individual has a vested interest in the project in terms of funding, project results, and resolving project conflicts, and is responsible for the success of the project. An executive summary describes the proposed application from an executive's point of view. It should describe the problems to be solved, the application goals, and the business opportunities. The objectives should indicate whether the application is a replacement of an old system and document the impact the application will have, if any, on the organization in terms of management, technology, and so on.

Any available documentation should be listed, and its status described. Examples include requirements specifications, functional specifications, project plan, design specification, prototypes, user manual, business model/flow diagrams, data models, and project risk assessments. In addition to project risks, which are the potential adverse effects on the development project, the risks relating to the testing effort should be documented. Examples include the lack of testing skills, scope of the testing effort, lack of automated testing tools, and the like. See “Test Plan (Client/Server and Internet Spiral Testing),” for more details.

Task 2: Define the High-Level Functional Requirements (in Scope)
A functional specification consists of the hierarchical functional decomposition, the functional window structure, the window standards, and the minimum system requirements of the system to be developed. An example of window standards is the Windows 95 GUI Standards. An example of a minimum system requirement could be Windows 95, a Pentium II microprocessor, 24 MB RAM, 3 GB disk space, and a modem. At this point in development, a full functional specification may not have been defined. However, a list of at least the major business functions of the basic window structure should be available.

A basic functional list contains the main functions of the system with each function named and described with a verb—object paradigm. This list serves as the basis for structuring functional testing (see Figure 14.2).

Figure 14.2: High-level business functions
Order processing (ex. create new order, edit order, etc.)
Customer processing (create new customer, edit customer, etc.)
Financial processing (receive payment, deposit payment, etc.)
Inventory processing (acquire products, adjust product price, etc.)
Reports (create order report, create account receivable report, etc.)

A functional window structure describes how the functions will be implemented in the windows environment. At this point, a full functional window structure may not be available, but a list of the major windows should be (see Figure 14.3).

Figure 14.3: Functional window structure
The Main-Window (menu bar, customer order window, etc.)
The Customer-Order-Window (order summary list, etc.)
The Edit-Order-Window (create order, edit order, etc.)
The Menu Bar (File, Order, View, etc.)
The Tool Bar with icons (FileNew, OrderCreate)

Task 3: Identify Manual/Automated Test Types
The types of tests that need to be designed and executed depend only on the objectives of the application, that is, the measurable end state the organization is striving to achieve. For example, if the application is a financial application used by a large number of individuals, special security and usability tests need to be performed. However, three types of tests that are nearly always required are function, user interface, and regression testing. Function testing comprises the majority of the testing effort and is concerned with verifying that the functions work properly. It is a black-box-oriented activity in which the tester is completely unconcerned with the internal behavior and structure of the application. User interface testing, or GUI testing, checks the user's

interaction or functional window structure. It ensures that object state dependencies work properly and provide useful navigation through the functions. Regression testing tests the application in light of changes made during debugging, maintenance, or the development of a new release.

Other types of tests that need to be considered include system and acceptance testing. System testing is the highest level of testing and evaluates functionality as a total system, its performance, and overall fitness of use. Acceptance testing is an optional user-run test that demonstrates the ability of the application to meet the user's requirements. This test may or may not be performed, depending on the formality of the project. Sometimes the system test suffices.

Finally, the tests that can be automated with a testing tool need to be identified. Automated tests provide three benefits: repeatability, leverage, and increased functionality. Repeatability enables automated tests to be executed more than once, consistently. Leverage comes from repeatability, from tests previously captured and tests that can be programmed with the tool, which may not have been possible without automation. As applications evolve, more and more functionality is added. With automation, the functional coverage is maintained with the test library.

Task 4: Identify the Test Exit Criteria

One of the most difficult and political problems is deciding when to stop testing, because it is impossible to know when all the defects have been detected. There are at least four criteria for exiting testing:

1. *Scheduled testing time has expired*—This criterion is very weak, inasmuch as it has nothing to do with verifying the quality of the application. This does not take into account that there may be an inadequate number of test cases or the fact that there may not be any more defects that are easily detectable.
2. *Some predefined number of defects discovered*—The problems with this is knowing the number of errors to detect and also overestimating the number of defects. If the number of defects is underestimated, testing will be incomplete. Potential solutions include experience with similar applications developed by the same development team, predictive models, and industry-wide averages. If the number of defects is overestimated, the test may never be completed within a reasonable time frame. A possible solution is to estimate completion time, plotting defects detected per unit of time. If the rate of defect detection is decreasing dramatically, there may be "burnout," an indication that a majority of the defects have been discovered.
3. *All the formal tests execute without detecting any defects*—A major problem with this is that the tester is not motivated to design destructive test cases that force the tested program to its design limits; for example, the tester's job is completed when the test program fields no more errors. The tester is motivated to not find errors and may subconsciously write test cases that show the program is error free. This criterion is only valid if there is a rigorous and totally comprehensive test case suite created that approaches 100 percent coverage.

The problem with this is determining when there is a comprehensive suite of test cases. If it is felt that this is the case, a good strategy at this point is to continue with ad hoc testing. Ad hoc testing is a black-box testing technique in which the tester lets his or her mind run freely to enumerate as many test conditions as possible. Experience has shown that this technique can be a very powerful supplemental or add-on technique.

4. *Combination of the foregoing criteria*—Most testing projects utilize a combination of the foregoing exit criteria. It is recommended that all the tests be executed, but any further ad hoc testing will be constrained by time.

Task 5: Establish Regression Test Strategy

Regression testing tests the application in light of changes made during a development spiral, debugging, maintenance, or the development of a new release. This test must be performed after functional improvements or repairs have been made to a system to confirm that the changes have no unintended side effects. Correction of errors relating to logic and control flow, computational errors, and interface errors are examples of conditions that necessitate regression testing. Cosmetic errors generally do not affect other capabilities and do not require regression testing.

It would be ideal if all the tests in the test suite were rerun for each new spiral; however, owing to time constraints, this is probably not realistic. A good regression strategy during spiral development is for some regression testing to be performed during each spiral to ensure that previously demonstrated capabilities are not adversely affected by later development spirals or error corrections. During system testing, after the system is stable and the functionality has been verified, regression testing should consist of a subset of the system tests. Policies need to be created to decide which tests to include. (See "Test Strategy")

A retest matrix is an excellent tool that relates test cases to functions (or program units), as shown in Table 14.1. A check entry in the matrix indicates that the test case is to be retested when the function (or program unit) has been modified due to enhancements or corrections. An empty cell means that the test does not need to be retested. The retest matrix can be built before the first testing spiral, but needs to be maintained during subsequent spirals. As functions (or program units) are modified during a development spiral, existing or new test cases need to be created and checked in the retest matrix in preparation for the next test spiral. Over time, with subsequent spirals, some functions (or program units) may remain stable with no recent modifications. Consideration to selectively remove their check entries should be undertaken between testing spirals. (Also see "Retest Matrix.")

Table 14.1: Retest Matrix

	Test Case				
	1	**2**	**3**	**4**	**5**
Business function					
Order processing					
Create new order	/	/	/	/	
Fulfill order					
Edit order	/			/	
Delete order					
Customer processing					
Create new customer					
Edit customer					
Delete customer		/			
Financial processing					
Receive customer payment		/	/		/
Deposit payment					
Pay vendor					
Write a check	/	/	/	/	/
Display register					
Inventory processing					
Acquire vendor products					
Maintain stock					
Handle back orders	/	/	/	/	/
Audit inventory					
Adjust product price					
Reports					
Create order report					
Create account receivables report	/	/	/	/	/
Create account payables report					
Create inventory report					

Other considerations of regression testing are as follows:

- Regression tests are potential candidates for test automation when they are repeated over and over in every testing spiral.
- Regression testing needs to occur between releases after the initial release of the system.
- A test that uncovers an original defect should be rerun after the defect has been corrected.
- An in-depth effort should be made to ensure that the original defect was corrected, and not just the symptoms.

- Regression tests that repeat other tests should be removed.
- Other test cases in the functional (or program unit) area where a defect is uncovered should be included in the regression test suite.
- Client-reported defects should have high priority and should be regression-tested thoroughly.

Task 6: Define the Test Deliverables

Test deliverables result from test planning, test design, test development, and test defect documentation. Some spiral test deliverables from which you can choose include the following:

- Test plan: Defines the objectives, scope, strategy, types of tests, test environment, test procedures, exit criteria, and so on (see “Sample Template”).
- Test design: Tests for the application's functionality, performance, and appropriateness for use. The tests demonstrate that the original test objectives are satisfied.
- Change request: A documented request to modify the current software system, usually supplied by the user (see “Change Request Form,” for more details). It is typically different from a defect report, which reports an anomaly in the system.
- Metrics: The measurable indication of some quantitative aspect of a system. Examples include the number of severe defects, and the number of defects discovered as a function of the number of testers.
- Test case: A specific set of test data and associated procedures developed for a particular objective. It provides a detailed blueprint for conducting individual tests and includes specific input data values and the corresponding expected results (see “Test Case,” for more details).
- Test log summary report: Specifies the test cases from the tester's individual test logs that are in progress or completed for status reporting and metric collection (see “Test Log Summary Report”).
- Test case log: Specifies the test cases for a particular testing event to be executed during testing. It is also used to record the results of the test performed, to provide the detailed evidence for the summary of test results, and to provide a basis for reconstructing the testing event if necessary (see “Test Case Log”).
- Interim test report: A report published between testing spirals, indicating the status of the testing effort (see Part 18, Step 3, Publish Interim Report).
- System summary report: A comprehensive test report after all spiral testing has been completed (see “System Summary Report”).
- Defect report: Documents defects discovered during spiral testing (see “Defect Report”).

Task 7: Organize the Test Team

The people component includes human resource allocations and the required skill sets. The test team should comprise the highest-caliber personnel possible. They are usually extremely busy and are in great demand because of their talents, and it therefore is vital to build the best case possible for using these individuals for test purposes. A test team leader and test team need to have the right skills and experience, and be motivated to work on the project. Ideally, they should be professional quality assurance specialists, but can represent the executive sponsor, users, technical operations, database administration, computer center, independent parties, and so on. In any event, they should not represent the development team, for they may not be as unbiased as an outside party. This is not to say that developers should not test; they should unit and function test their code extensively before handing it over to the test team.

There are two areas of responsibility in testing: testing the application, which is the responsibility of the test team, and the overall testing processes, which is handled by the test manager. The test manager directs one or more testers, is the interface between quality assurance and the development organization, and manages the overall testing effort. Responsibilities include the following:

- Setting up the test objectives
- Defining test resources
- Creating test procedures
- Developing and maintaining the test plan
- Designing test cases
- Designing and executing automated testing tool scripts
- Test case development
- Providing test status
- Writing reports
- Defining the roles of the team members
- Managing the test resources
- Defining standards and procedures
- Ensuring quality of the test process
- Training the team members
- Maintaining test statistics and metrics

The test team must be a set of team players and have these responsibilities:

- Executing test cases according to the plan
- Evaluating the test results
- Reporting errors
- Designing and executing automated testing tool scripts
- Recommending application improvements
- Recording defects

The main function of a team member is to test the application and report defects to the development team by documenting them in a defect-tracking system. Once the development team corrects the defects, the test team reexecutes the tests that discovered the original defects.

The roles of the test manager and team members are not mutually exclusive. Some of the team leader's responsibilities are shared with the team members, and vice versa.

The basis for allocating dedicated testing resources is the scope of the functionality and the development time frame; for example, a medium development project will require more testing resources than a small one. If project A of medium complexity requires a testing team of five, project B with twice the scope would require ten testers (given the same resources).

Another rule of thumb is that the testing costs approach 25 percent of the total budget. Because the total project cost is known, the testing effort can be calculated and translated to tester headcount. The best estimate is a combination of the project scope, test team skill levels, and project history. A good measure of required testing resources for a particular project is the histories of multiple projects, that is, testing resource levels and performance compared to similar projects.

Task 8: Establish a Test Environment
The purpose of the test environment is to provide a physical framework necessary for the testing activity. For this task, the test environment needs are established and reviewed before implementation.

The main components of the test environment include the physical test facility, technologies, and tools. The test facility component includes the physical setup. The technologies component includes the hardware platforms, physical network and all its components, operating system software, and other software such as utility software. The tools component includes any specialized testing software such as automated test tools, testing libraries, and support software.

The testing facility and workplace need to be established. This may range from an individual workplace configuration to a formal testing laboratory. In any event, it is important that the testers be together and in close proximity to the development team. This facilitates communication and the sense of a common goal. The testing tools that were acquired need to be installed.

The hardware and software technologies need to be set up. This includes the installation of test hardware and software, and coordination with vendors, users, and information technology personnel. It may be necessary to test the hardware and coordinate with hardware vendors. Communication networks need to be installed and tested.

Task 9: Define the Dependencies
A good source of information is previously produced test plans on other projects. If available, the sequence of tasks in the project work plans can be analyzed for activity and task dependencies that apply to this project.
Examples of test dependencies include the following:

- Code availability
- Tester availability (in a timely fashion)
- Test requirements (reasonably defined)
- Test tools availability
- Test group training
- Technical support
- Defects fixed in a timely manner
- Adequate testing time
- Computers and other hardware
- Software and associated documentation
- System documentation (if available)
- Defined development methodology
- Test laboratory space availability
- Agreement with development (procedures and processes)

The support personnel need to be defined and committed to the project. This includes members of the development group, technical support staff, network support staff, and database administrator support staff.

Task 10: Create a Test Schedule

A test schedule should be produced that includes the testing steps (and perhaps tasks), target start and end dates, and responsibilities. It should also describe how it will be reviewed, tracked, and approved. A simple test schedule format, as shown in Table 14.2, follows the spiral methodology.

Table 14.2: Test Schedule

Test Step	**Begin Date**	**End Date**	**Responsible Staff Member**
First Spiral Information gathering			
Prepare for interview	6/1/04	6/2/04	Smith, test manager
Conduct interview	6/3/04	6/3/04	Smith, test manager
Summarize findings	6/4/04	6/5/04	Smith, test manager
Test planning			
Build test plan	6/8/04	6/12/04	Smith, test manager
Define the metric objectives	6/15/04	6/17/04	Smith, test manager
Review/approve plan	6/18/04	6/18/04	Smith, test manager
Test case design			
Design function tests	6/19/04	6/23/04	Smith, test manager
Design GUI tests	6/24/04	6/26/04	Smith, test manager
Define the system/acceptance			

Table 14.2: Test Schedule

Test Step	**Begin Date**	**End Date**	**Responsible Staff Member**
Tests	6/29/04	6/30/04	Smith, test manager
Review/approve design	7/3/04	7/3/04	Smith, test manager
Test development			
Develop test scripts	7/6/04	7/16/04	Jones, Baker, Brown, testers
Review/approve test development	7/17/04	7/17/04	Jones, Baker, Brown, testers
Test execution/evaluation			
Setup and testing	7/20/04	7/24/04	Smith, Jones, Baker, Brown, testers
Evaluation	7/27/04	7/29/04	Smith, Jones, Baker, Brown, testers
Prepare for the Next Spiral			
Refine the tests	8/3/04	8/5/04	Smith, test manager
Reassess team, procedures, and test environment	8/6/04	8/7/04	Smith, test manager
Publish interim report	8/10/04	8/11/04	Smith, test manager
•			
•			
•			
Last Spiral…			
Test execution/evaluation			
Setup and testing	10/5/04	10/9/04	Jones, Baker, Brown, testers
Evaluation	10/12/04	10/14/04	Smith, test manager
•			
•			
•			
Conduct system testing			
Complete system test plan	10/19/04	10/21/04	Smith, test manager
Complete system test cases	10/22/04	10/23/04	Smith, test manager
Review/approve system tests	10/26/04	10/30/04	Jones, Baker, Brown, testers
Execute the system tests	11/2/04	11/6/04	Jones, Baker, Brown, testers
Conduct acceptance testing			
Complete acceptance test plan	11/9/04	11/10/04	Smith, test manager
Complete acceptance test cases	11/11/04	11/12/04	Smith, test manager
Review/approve acceptance			
Test plan	11/13/04	11/16/04	Jones, Baker, Brown, testers

Table 14.2: Test Schedule

Test Step	**Begin Date**	**End Date**	**Responsible Staff Member**
Execute the acceptance tests	11/17/04	11/20/04	
Summarize/report spiral test results			
Perform data reduction	11/23/04	11/26/04	Smith, test manager
Prepare final test report	11/27/04	11/27/04	Smith, test manager
Review/approve the final			
Test report	11/28/04	11/29/04	Smith, test manager Baylor, sponsor

Also, a project management tool such as Microsoft Project can format a Gantt chart to emphasize the tests and group them into test steps. A Gantt chart consists of a table of task information and a bar chart that graphically displays the test schedule. It also includes task time duration and links the task dependency relationships graphically. People resources can also be assigned to tasks for workload balancing. See "Test Schedule," and template file Gantt spiral testing methodology template.

Another way to schedule testing activities is with "relative scheduling," in which testing steps or tasks are defined by their sequence or precedence. It does not state a specific start or end date but does have a duration, such as days. (Also see "Test Execution Plan," which can be used to plan the activities for the Execution phase, and "PDCA Test Schedule," which can be used to plan and track the Plan—Do—Check—Act test phases.)

It is also important to define major external and internal milestones. External milestones are events that are external to the project but may have a direct impact on the project. Examples include project sponsorship approval, corporate funding, and legal authorization. Internal milestones are derived for the schedule work plan and typically correspond to key deliverables that need to be reviewed and approved. Examples include test plan, design, and development completion approval by the project sponsor and the final spiral test summary report. Milestones can be documented in the test plan in table format as shown in Table 14.3. (Also see "Test Project Milestones," which can be used to identify and track the key test milestones.)

Table 14.3: Project Milestones

Project Milestone	***Due Date***
Sponsorship approval	7/1/04
First prototype available	7/20/04
Project test plan	6/18/04
Test development complete	7/1704

Table 14.3: Project Milestones

Project Milestone	*Due Date*
Test execution begins	7/20/04
Final spiral test summary report published	11/27/04
System ship date	12/1/04

Task 11: Select the Test Tools

Test tools range from relatively simple to sophisticated software. New tools are being developed to help provide the high-quality software needed for today's applications.

Because test tools are critical to effective testing, those responsible for testing should be proficient in using them. The tools selected should be most effective for the environment in which the tester operates and the specific types of software being tested. The test plan needs to name specific test tools and their vendors. The individual who selects the test tool should also conduct the test and be familiar enough with the tool to use it effectively. The test team should review and approve the use of each test tool, because the tool selected must be consistent with the objectives of the test plan.

Task 12: Establish Defect Recording/Tracking Procedures

During the testing process, when a defect is discovered, it needs to be recorded. A defect is related to individual tests that have been conducted, and the objective is to produce a complete record of those defects. The overall motivation for recording defects is to correct them and record metric information about the application. Development should have access to the defects reports, which they can use to evaluate whether there is a defect and how to reconcile it. The defect form can either be manual or electronic, with the latter being preferred. Metric information such as the number of defects by type or open time for defects can be very useful in understanding the status of the system.

Defect control procedures need to be established to control this process from initial identification to reconciliation. Table 14.4 shows some possible defect states, from open to closed with intermediate states. The testing department initially opens a defect report and also closes it. A "Yes" in a cell indicates a possible transition from one state to another. For example, an "Open" state can change to "Under Review," "Returned by Development," or "Deferred by Development." The transitions are initiated by either the testing department or by development.

Table 14.4: Defect States

	Open	*Under Review*	*Returned by Development*	*Ready for Testing*	*Returned by QA*	*Deferred by Development*	*Closed*
Open	—	Yes	Yes	—	—	Yes	—

Table 14.4: Defect States

	Open	*Under Review*	*Returned by Development*	*Ready for Testing*	*Returned by QA*	*Deferred by Development*	*Closed*
Under review	—	—	Yes	Yes	—	Yes	Yes
Returned by development	—	—	—	—	Yes	—	Yes
Ready for testing	—	—	—	—	Yes	—	Yes
Returned by QA	—	—	Yes	—	—	Yes	Yes
Deferred by development	—	Yes	Yes	Yes	—	—	Yes
Closed	Yes	—	—	—	—	—	—

A defect report form also needs to be designed. The major fields of a defect form include (see "Defect Report," for more details) the following:

- Identification of the problem, for example, functional area, problem type, and so on
- Nature of the problem, for example, behavior
- Circumstances that led to the problem, for example, inputs and steps
- Environment in which the problem occurred, for example, platform, and so on
- Diagnostic information, for example, error code, and so on
- Effect of the problem, for example, consequence

It is quite possible that a defect report and a change request form are the same. The advantage of this approach is that it is not always clear whether a change request is a defect or an enhancement request. The differentiation can be made with a form field that indicates whether it is a defect or enhancement request. On the other hand, a separate defect report can be very useful during the maintenance phase when the expected behavior of the software is well known and it is easier to distinguish between a defect and an enhancement.

Task 13: Establish Change Request Procedures

If it were a perfect world, a system would be built and there would be no future changes. Unfortunately, it is not a perfect world and after a system is deployed, there are change requests. Some of the reasons for change are the following:

- The requirements change.
- The design changes.
- The specification is incomplete or ambiguous.
- A defect is discovered that was not discovered during reviews.
- The software environment changes, for example, platform, hardware, and so on.

Change control is the process by which a modification to a software component is proposed, evaluated, approved or rejected, scheduled, and tracked. It is a decision process used in controlling the changes made to software. Some proposed changes are accepted and implemented during this process. Others are rejected or postponed, and are not implemented. Change control also provides for impact analysis to determine the dependencies (see "Change Request Form," for more details).

Each software component has a life cycle. A life cycle consists of states and allowable transitions between those states. Any time a software component is changed, it should always be reviewed. During the review, it is frozen from further modifications and the only way to change it is to create a new version. The reviewing authority must approve the modified software component or reject it. A software library should hold all components as soon as they are frozen and also act as a repository for approved components.

The formal title of the organization that manages changes is a configuration control board, or CCB. The CCB is responsible for the approval of changes and for judging whether a proposed change is desirable. For a small project, the CCB can consist of a single person, such as a project manager. For a more formal development environment, it can consist of several members from development, users, quality assurance, management, and the like.

All components controlled by software configuration management are stored in a software configuration library, including work products such as business data and process models, architecture groups, design units, tested application software, reusable software, and special test software. When a component is to be modified, it is checked out of the repository into a private workspace. It evolves through many states that are temporarily outside the scope of configuration management control.

When a change is completed, the component is checked into the library and becomes a new component version. The previous component version is also retained.

Change control is based on the following major functions of a development process: requirements analysis, system design, program design, testing, and implementation. At least six control procedures are associated with these functions and need to be established for a change control system (see "Software Quality Assurance Plan," for more details):

1. *Initiation procedures*—-This includes procedures for initiating a change request through a change request form, which serves as a communication vehicle. The objective is to gain consistency in documenting the change request document and routing it for approval.
2. *Technical assessment procedures*—This includes procedures for assessing the technical feasibility and technical risks, and scheduling a technical evaluation of a proposed change.

The objectives are to ensure integration of the proposed change, the testing requirements, and the ability to install the change request.

3. *Business assessment procedures*—This includes procedures for assessing the business risk, effect, and installation requirements of the proposed change. The objectives are to ensure that the timing of the proposed change is not disruptive to the business goals.
4. *Management review procedures*—This includes procedures for evaluating the technical and business assessments through management review meetings. The objectives are to ensure that changes meet technical and business requirements and that adequate resources are allocated for testing and installation.
5. *Test tracking procedures*—This includes procedures for tracking and documenting test progress and communication, including steps for scheduling tests, documenting the test results, deferring change requests based on test results, and updating test logs. The objectives are to ensure that testing standards are utilized to verify the change, including test plans and test design, and that test results are communicated to all parties.
6. *Installation tracking procedures*—This includes procedures for tracking and documenting the installation progress of changes. It ensures that proper approvals have been completed, adequate time and skills have been allocated, installation and backup instructions have been defined, and proper communication has occurred. The objectives are to ensure that all approved changes have been made, including scheduled dates, test durations, and reports.

Task 14: Establish Version Control Procedures

A method for uniquely identifying each software component needs to be established via a labeling scheme. Every software component must have a unique name. Software components evolve through successive revisions, and each needs to be distinguished. A simple way to distinguish component revisions is with a pair of integers, 1.1, 1.2, …, that define the release number and level number. When a software component is first identified, it is revision 1 and subsequent major revisions are 2, 3, and so on.

In a client/server environment, it is highly recommended that the development environment be different from the test environment. This requires the application software components to be transferred from the development environment to the test environment. Procedures need to be set up.

Software needs to be placed under configuration control so that no changes are being made to the software while testing is being conducted. This includes source and executable components. Application software can be periodically migrated into the test environment. This process must be controlled to ensure that the latest version of software is tested. Versions will also help control the repetition of tests to ensure that previously discovered defects have been resolved.

For each release or interim change between versions of a system configuration, aversion description document should be prepared to identify the software components.

Task 15: Define Configuration Build Procedures
Assembling a software system involves tools to transform the source components, or source code, into executable programs. Examples of tools are compilers and linkage editors.

Configuration build procedures need to be defined to identify the correct component versions and execute the component build procedures. The configuration build model addresses the crucial question of how to control the way components are built.

A configuration typically consists of a set of derived software components. An example of derived software components is executable object programs derived from source programs. Derived components must be correctly associated with each source component to obtain an accurate derivation. The configuration build model addresses the crucial question of how to control the way derived components are built.

The inputs and outputs required for a configuration build model include primary inputs and primary outputs. The primary inputs are the source components, which are the raw materials from which the configuration is built; the version selection procedures; and the system model, which describes the relationship between the components. The primary outputs are the target configuration and derived software components.

Different software configuration management environments use different approaches for selecting versions. The simplest approach to version selection is to maintain a list of component versions. Other automated approaches allow for the most recently tested component versions to be selected, or those updated on a specific date. Operating system facilities can be used to define and build configurations, including the directories and command files.

Task 16: Define Project Issue Resolution Procedures
Testing issues can arise at any point in the development process and must be resolved successfully. The primary responsibility of issue resolution is with the project manager, who should work with the project sponsor to resolve those issues.

Typically, the testing manager will document test issues that arise during the testing process. The project manager or project sponsor should screen every issue that arises. An issue can be rejected or deferred for further investigation, but should be considered relative to its impact on the project. In any case, a form should be created that contains the essential information. Examples of testing issues include lack of testing tools, lack of adequate time to test, inadequate knowledge of the requirements, and so on.

Issue management procedures need to be defined before the project starts. The procedures should address how to:

- Submit an issue
- Report an issue
- Screen an issue (rejected, deferred, merged, or accepted)
- Investigate an issue
- Approve an issue
- Postpone an issue
- Reject an issue
- Close an issue

Task 17: Establish Reporting Procedures

Test reporting procedures are critical to manage the testing progress and manage the expectations of the project team members. This will keep the project manager and sponsor informed of the testing project progress and minimize the chance of unexpected surprises. The testing manager needs to define who needs the test information, what information they need, and how often the information is to be provided. The objectives of test status reporting are to report the progress of the testing toward its objectives and report test issues, problems, and concerns.

Two key reports that need to be published are:

1. *Interim Test Report*—An interim test report is a report published between testing spirals indicating the status of the testing effort.
2. *System Summary Report*—A test summary report is a comprehensive test report after all spiral testing has been completed.

Task 18: Define Approval Procedures

Approval procedures are critical in a testing project. They help provide the necessary agreement between members of the project team. The testing manager should define who needs to approve a test deliverable, when it will be approved, and what the backup plan is if an approval cannot be obtained. The approval procedure can vary from a formal sign-off of a test document to an informal review with comments. Table 14.5 shows test deliverables for which approvals are required or recommended, and by whom. (Also see "Test Approvals," for a matrix that can be used to formally document management approvals for test deliverables.)

Table 14.5: Deliverable Approvals

Test Deliverable	***Approval Status***	***Suggested Approver***
Test plan	Required	Project Manager, Development Manager, Sponsor
Test design	Required	Development Manager
Change request	Required	Development Manager
Metrics	*Recommended*	*Development Manager*

Table 14.5: Deliverable Approvals

Test Deliverable	***Approval Status***	***Suggested Approver***
Test case	Required	Development Manager
Test log summary report	Recommended	Development Manager
Interim test report	Required	Project Manager, Development Manager
System summary report	Required	Project Manager, Development Manager, Sponsor
Defect report	Required	Development Manager

Step 2: Define the Metric Objectives

"You can't control what you can't measure." This is a quote from Tom DeMarco's book, *Controlling Software Projects*, in which he describes how to organize and control a software project so that it is measurable in the context of time and cost projections. Control is the extent to which a manager can ensure minimum surprises. Deviations from the plan should be signaled as early as possible for timely corrective action. Another quote from DeMarco's book, "The only unforgivable failure is the failure to learn from past failure," stresses the importance of estimating and measurement. Measurement is a recording of past effects to quantitatively predict future effects.

Task 1: Define the Metrics

Software testing as a test development project has deliverables such as test plans, test design, test development, and test execution. The objective of this task is to apply the principles of metrics to control the testing process. A metric is a measurable indication of some quantitative aspect of a system and has the following characteristics:

- *Measurability*—A metric point must be measurable for it to be a metric, by definition. If the phenomenon cannot be measured, there is no way to apply management methods to control it.
- *Independence*—Metrics need to be independent of human influence. There should be no way of changing the measurement other than by changing the phenomenon that produced the metric.
- *Accountability*—Any analytical interpretation of the raw metric data rests on the data itself and it is, therefore, necessary to save the raw data and the methodical audit trail of the analytical process.
- *Precision*—Precision is a function of accuracy. The key to precision is, therefore, that a metric be explicitly documented as part of the data collection process. If a metric varies, it can be measured as a range or tolerance.

A metric can be a "result" or a "predictor." A result metric measures a completed event or process. Examples include actual total elapsed time to process a business transaction or total test costs of a project. A predictor metric is an early-warning metric that has a strong correlation to some later result. An example is the predicted response time through statistical regression analysis when more

terminals are added to a system when the number of terminals has not yet been measured. A result or predictor metric can also be a derived metric. A derived metric is one that is derived from a calculation or graphical technique involving one or more metrics.
The motivation for collecting test metrics is to make the testing process more effective. This is achieved by carefully analyzing the metric data and taking the appropriate action to correct problems. The starting point is to define the metric objectives of interest. Some examples include the following:

- *Defect analysis*—Every defect must be analyzed to answer such questions as the root causes, how it was detected, when it was detected, who detected it, and so on.
- *Test effectiveness*—How well is testing doing, for example, return on investment?
- *Development effectiveness*—How well is development fixing defects?
- *Test automation*—How much effort is expended on test automation?
- *Test cost*—What are the resources and time spent on testing?
- *Test status*—Another important metric is status tracking, or where are we in the testing process?
- *User involvement*—How much is the user involved in testing?

Task 2: Define the Metric Points
Table 14.6 lists some metric points associated with the general metrics selected in the previous task and the corresponding actions to improve the testing process. Also shown is the source, or derivation, of the metric point.

Table 14.6: Metric Points

Metric	Metric Point	Derivation
Defect analysis	Distribution of defect causes	Histogram, Pareto
	Number of defects by cause over time	Multiline graph
	Number of defects by how found over time	Multiline graph
	Distribution of defects by module	Histogram, Pareto
	Distribution of defects by priority (critical, high, medium, low)	Histogram
	Distribution of defects by functional area	Histogram
	Distribution of defects by environment (platform)	Histogram, Pareto
	Distribution of defects by type (architecture, connectivity, consistency, database integrity, documentation, GUI, installation, memory, performance, security, standards and conventions, stress, usability, bad fixes)	Histogram, Pareto
	Distribution of defects by who detected (external customer, internal customer, development, QA, other)	Histogram, Pareto

Table 14.6: Metric Points

Metric	Metric Point	Derivation
	Distribution by how detected (technical review, walkthroughs, JAD, prototyping, inspection, test execution)	Histogram, Pareto
	Distribution of defects by severity (high, medium, low defects)	Histogram
Development effectiveness	Average time for development to repair defect	Total repair time ÷ number of repaired defects
Test automation	Percentage of manual versus automated testing	Cost of manual test effort ÷ total test cost
Test cost	Distribution of cost by cause	Histogram, Pareto
	Distribution of cost by application	Histogram, Pareto
	Percentage of costs for testing	Test testing cost ÷ total system cost
	Total costs of testing over time	Line graph
	Average cost of locating a defect	Total cost of testing ÷ number of defects detected
	Anticipated costs of testing versus actual cost	Comparison
	Average cost of locating a requirements defect with requirements reviews	Requirements review costs ÷ number of defects uncovered during requirement reviews
	Average cost of locating a design defect with design reviews	Design review costs ÷ number of defects uncovered during design reviews
	Average cost of locating a code defect with reviews	Code review costs ÷ number of defects uncovered during code reviews
	Average cost of locating a defect with test execution	Test execution costs ÷ number of defects uncovered during test execution
	Number of testing resources over time	Line plot
Test effectiveness	Percentage of defects discovered during maintenance	Number of defects discovered during maintenance ÷ total

Table 14.6: Metric Points

Metric	**Metric Point**	**Derivation**
		number of defects uncovered
	Percentage of defects uncovered due to testing	Number of detected errors through testing ÷ total system defects
	Average effectiveness of a test	Number of tests ÷ total system defects
	Value returned while reviewing requirements	Number of defects uncovered during requirements review ÷ requirements test costs
	Value returned while reviewing design	Number of defects uncovered during design review ÷ design test costs
	Value returned while reviewing programs	Number of defects uncovered during program review ÷ program test costs
	Value returned during test execution	Number of defects uncovered during testing ÷ test costs
	Effect of testing changes	Number of tested changes ÷ problems attributable to the changes
	People's assessment of effectiveness of testing	Subjective scaling (1-10)
	Average time for QA to verify fix	Total QA verification time ÷ total number of defects to verify
	Number of defects over time	Line graph
	Cumulative number of defects over time	Line graph
	Number of application defects over time	Multiline graph
Test extent	Percentage of statements executed	Number of statements executed ÷ total statements
	Percentage of logical paths executed	Number of logical paths ÷ total number of paths
	Percentage of acceptance criteria tested	Acceptance criteria tested ÷ total acceptance criteria

Table 14.6: Metric Points

Metric	**Metric Point**	**Derivation**
	Number of requirements tested over time	Line plot
	Number of statements executed over time	Line plot
	Number of data elements exercised overtime	Line plot
	Number of decision statements executed over time	Line plot
Test status	Number of tests ready to run over time	Line plot
	Number of tests run over time	Line plot
	Number of tests run without defects uncovered	Line plot
	Number of defects corrected over time	Line plot
User involvement	Percentage of user testing	User testing time ÷ total test time

Step 3: Review/Approve the Plan

Task 1: Schedule/Conduct the Review

The test plan review should be scheduled well in advance of the actual review, and the participants should have the latest copy of the test plan.

As with any interview or review, it should contain certain elements. The first is defining what will be discussed, or "talking about what we are going to talk about." The second is discussing the details, or "talking about it." The third is summarization, or "talking about what we talked about." The final element is timeliness. The reviewer should state up front the estimated duration of the review and set the ground rule that if time expires before completing all items on the agenda, a follow-on review will be scheduled.

The purpose of this task is for development and the project sponsor to agree and accept the test plan. If there are any suggested changes to the test plan during the review, they should be incorporated into the test plan.

Task 2: Obtain Approvals

Approval is critical in a testing effort, for it helps provide the necessary agreements between testing, development, and the sponsor. The best approach is with a formal sign-off procedure of a test plan. If this is the case, use the management approval sign-off forms. However, if a formal agreement procedure is not in place, send a memo to each key participant, including at least the project manager, development manager, and sponsor. In the document, attach the latest test plan and point out that all their feedback comments have been incorporated and that if you do not hear from them, it is assumed that they agree with the plan. Finally, indicate that in a spiral development environment, the test plan will evolve with each iteration but that you will include them in any modification.

Section 15: Test Case Design (Do)

You will recall that in the spiral development environment, software testing is described as a continuous improvement process that must be integrated into a rapid application development methodology. Deming's continuous improvement process using the PDCA model is applied to the software testing process. We are now in the Do part of the spiral model (see Figure 15.1).

Figure 15.1: Spiral testing and continuous improvement

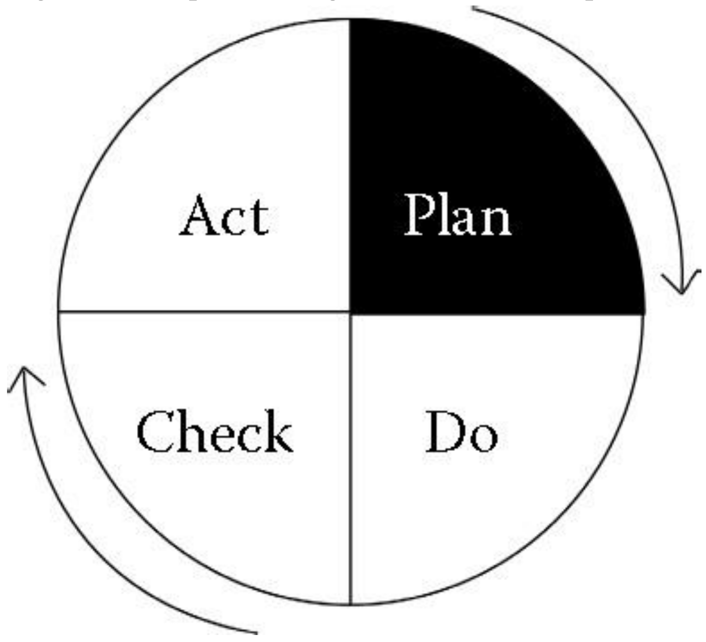
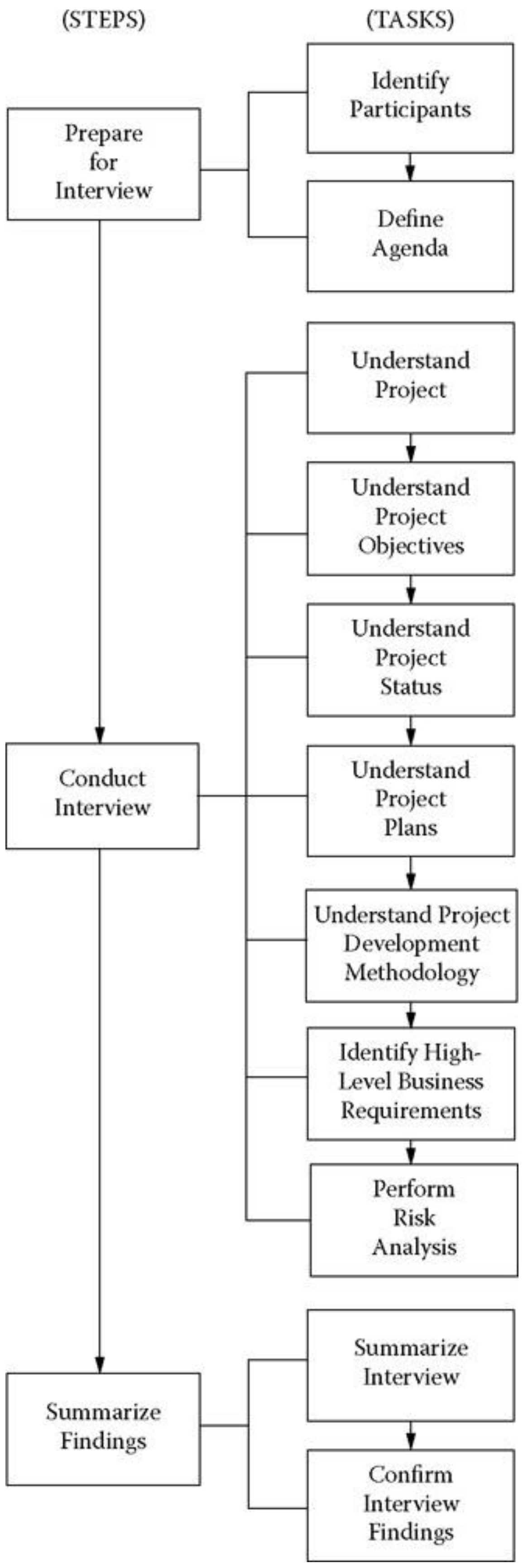


Figure 15.2 outlines the steps and tasks associated with the Do part of spiral testing. Each step and task are described, and valuable tips and techniques are provided.

Figure 15.2: Test design (steps/tasks)


(STEPS)
(TASKS)
Prepare for Interview
Identify Participants
Define Agenda
Conduct Interview
Understand Project
Understand Project Objectives
Understand Project Status
Understand Project Plans
Understand Project Development Methodology
Identify High-Level Business Requirements
Perform Risk Analysis
Summarize Findings
Summarize Interview
Confirm Interview Findings

Step 1: Design Function Tests

Task 1: Refine the Functional Test Requirements

At this stage, the functional specification should be complete. It includes:

- **Hierarchical functional decomposition**, detailing business functions.
- **Functional window structure** and **window standards**, such as Windows 2000 GUI Standards.
- **Minimum system requirements**, like operating system, processor, memory, and storage needs (e.g., Windows 2000, Pentium IV, 1 GB RAM).

The functional breakdown lists discrete business functions described using a verb-object format (e.g., "approve customer credit"). Each function's success criteria are outlined, and the hierarchy serves as the basis for testing. Functions include activities like handling orders or creating invoices, representing the full application, including interfaces. Sources for these functions might include process decomposition, data flow diagrams, or CRUD matrices gathered during interviews.

The requirements serve as the basis for creating test cases. The following quality assurance test checklists can be used to ensure that the requirements are clear and comprehensive:

- Clarification Request, which can be used to document questions that may arise while the tester analyzes the requirements.
- Ambiguity Review Checklist, which can be used to assist in the review of a functional specification of structural ambiguity (not to be confused with content reviews).
- Architecture Review Checklist, which can be used to review the architecture for completeness and clarity.
- Data Design Review Checklist, which can be used to review the logical and physical design for clarity and completeness.
- Functional Specification Review Checklist, which can be used in functional specification for content completeness and clarity (not to be confused with ambiguity reviews).
- Prototype Review Checklist, which can be used to review a prototype for content completeness and clarity.
- Requirements Review Checklist, which can be used to verify that the testing project requirements are comprehensive and complete.
- Technical Design Review Checklist, which can be used to review the technical design for clarity and completeness.

A functional breakdown is used to illustrate the processes in a hierarchical structure showing successive levels of detail. It is built iteratively as processes and non-elementary processes are decomposed (see Figure 15.3).

Figure 15.3: Functional breakdown

Functional Breakdown

Functional Test Requirements (Breakdown)

Order Processing
Create new order
Fulfill order
Edit order
Delete order
Customer Processing
Create new customer
Edit customer
Delete customer
Financial Processing
Receive customer payment
Deposit payment
Pay vendor
Write a check
Display register
Inventory Processing
Acquire vendor products
Maintain stock
Handle back orders
Audit inventory
Adjust product price

Reports

Create order report
Create account receivable report
Create account payable report
Create inventory report

A data flow diagram shows processes and the flow of data among these processes. It is used to define the overall data flow through a system and consists of external agents that interface with the system, processes, data flow, and stores depicting where the data is stored or retrieved. A data flow diagram should be reviewed, and each major and leveled function should be listed and organized into a hierarchical list.

A CRUD matrix, or association matrix, links data and process models. It identifies and resolves matrix omissions and conflicts and helps refine the data and process models, as necessary.
A functional window structure describes how the functions will be implemented in the windows environment. Figure 15.4 shows a sample functional window structure for order processing.

Figure 15.4: Functional window structure

Functional Window Structure

The Main Window

a. The top line of the main window has the standard title bar with Min/Max controls.
b. The next line contains the standard Windows menu bar.
c. The next line contains the standard Windows tool bar.
d. The rest of the Main-Application Window is filled with the Customer-Order Window.

The Customer-Order Window

a. This window shows a summary of each previously entered order.
b. Several orders will be shown at one time (sorted by order number and customer name). For each customer order, this window will show:
    1. Order Number
    2. Customer Name
    3. Customer Number
    4. Date
    5. Invoice Number
    6. Model Number
    7. Product Number
    8. Quantity Shipped
    9. Price
c. The scroll bar will be used to select which orders are to be viewed.
d. This window is read-only for viewing.
e. Double-clicking an order will display the Edit-Order Dialog where the order can be modified.

The Edit-Order Window

a. This dialog is used to create new orders or for making changes to previously created orders.
b. This dialog will be centered over the Customer-Order Window. The layout of this dialog will show the following:
    1. Order Number (automatically filled in)
    2. Edit field for: Customer Name
    3. Edit field for: Customer Number
    4. Date (initialized)
    5. Edit field for: Invoice Number
    6. Edit field for: Model Number

7. Edit field for: Product Number
8. Edit field for: Quantity Shipped
9. Price (automatically filled in)
10. Push buttons for: OK and Cancel

The Menu Bar Will Include the Following Menus:
File:

- New:
  - Used to create a new order file
- Open:
  - Used to open the order file

Task 2: Build a Function/Test Matrix
The function/test matrix cross-references the tests to the functions. This matrix provides proof of the completeness of the test strategies, illustrating in graphic format which tests exercise which functions. (See Table 15.1 and “Function/Test Matrix,” for more details.)

Table 15.1: Functional/Test Matrix

	Test Case				
Business Function	***1***	***2***	***3***	***4***	***5***
Order processing					
Create new order	CNO01	CNO02			
Fulfill order	AO01				
Edit order	EO01	EO02	EO03	EO04	
Delete order	DO01	DO02	DO03	DO04	DO05
Customer processing					
Create new customer	ANC01	ANC02	ANC03		
Edit customer	EC01	EC02	EC03	EC04	EC05
Delete customer	DC01	DC02			
Financial processing					
Receive customer payment	RCP01	RCP02	RCP03	RCP04	
Deposit payment	AP01	AP02			
Pay vendor	PV01	PV02	PV03	PV04	PV05
Write a check	WC01	WC02			
Display register	DR01	DR02			
Inventory processing					
Acquire vendor products	AP01	AP02	AP03		
Maintain stock	MS01	MS02	MS03	MS04	MS05
Handle back orders	HB01	HB02	HB03		

Table 15.1: Functional/Test Matrix

	Test Case				
Business Function	***1***	***2***	***3***	***4***	***5***
Audit inventory	AIOI	AI02	AI03	AI04	
Adjust product price	ACOI	AC02	AC03		
Reports					
Create order report	CO0L	CO02	CO03	CO04	CO05
Create account receivables report	CAOI	CA02	CA03		
Create account payables	AYOI	AY02	AY03		
Create inventory report	CIOI	CI02	CI03	CI04	

The matrix is used as a control sheet during testing and can also be used during maintenance. For example, if a function is to be changed, the maintenance team can refer to the function/test matrix to determine which tests need to be run or changed. The business functions are listed vertically, and the test cases are listed horizontally. The test case name is recorded on the matrix along with the number. (Also see "Test Condition versus Test Case," Matrix I, which can be used to associate a requirement with each condition that is mapped to one or more test cases.)

It is also important to differentiate those test cases that are manual from those that are automated. One way to accomplish this is to come up with a naming standard that will highlight an automated test case; for example, the first character of the name is "A." Table 15.1 shows an example of a function/test matrix.

Step 2: Design GUI Tests
The goal of a good graphical user interface (GUI) design should be consistency in "look and feel" for the users of the application. Good GUI design has two key components: interaction and appearance. Interaction relates to how the user interacts with the application. Appearance relates to how the interface looks to the user.

GUI testing involves confirming that the navigation is correct; for example, when an icon, menu choice, or radio button is clicked, the desired response occurs. The following are some good GUI design principles the tester should look for while testing the application.

Ten Guidelines for Good GUI Design

1. Involve users.
2. Understand the user's culture and experience.
3. Prototype continuously to validate the requirements.
4. Let the user's business workflow drive the design.
5. Do not overuse or underuse GUI features.
6. Create the GUI, help files, and training concurrently.

7. Do not expect users to remember secret commands or functions.
8. Anticipate mistakes, and do not penalize the user for making them.
9. Continually remind the user of the application status.
10. Keep it simple.

Task 1: Identify the Application GUI Components

GUI provides multiple channels of communication using words, pictures, animation, sound, and video. Five key foundation components of the user interface are windows, menus, forms, icons, and controls.

1. *Windows*—In a windowed environment, all user interaction with the application occurs through the windows. These include a primary window, along with any number of secondary windows generated from the primary one.
2. *Menus*—Menus come in a variety of styles and forms. Examples include action menus (push button, radio button), pull-down menus, pop-up menus, option menus, and cascading menus.
3. *Forms*—Forms are windows or screens into which the user can add information.
4. *Icons*—Icons, or "visual push buttons," are valuable for instant recognition, ease of learning, and ease of navigation through the application.
5. *Controls*—A control component appears on a screen that allows the user to interact with the application, and is indicated by its corresponding action. Controls include menu bars, pull-down menus, cascading menus, pop-up menus, push buttons, check boxes, radio buttons, list boxes, and drop-down list boxes.

A design approach to GUI test design is to first define and name each GUI component by name within the application, as shown in Table 15.2. In the next step, a GUI component checklist is developed that can be used to verify each component in this table. (Also see "GUI Component Test Matrix.")

Table 15.2: GUI Component Test Matrix

	GUI Type							
Name	***Window***	***Menu***	***Form***	***ICON***	***Control***	***P/F***	***Date***	***Tester***
Main window	/							
Customer-order window	/							
Edit-order window	/							
Menu bar		/						
Tool bar					/			

Task 2: Define the GUI Tests

In the previous task, the application GUI components were defined, named, and categorized in the GUI component test matrix. In the present task, a checklist is developed against which each GUI component is verified. The list should cover all possible interactions and may or may not apply to a particular component. Table 15.3 is a partial list of the items to check. (See "Screen Data Mapping," which can be used to document the properties of the screen data, and "Test Case Preparation Review Checklist," which can be used to ensure that test cases have been prepared as per specifications.)

Table 15.3: GUI Component Checklist

Access via Double-Click	*Multiple Windows Open*	*Tabbing Sequence*
Access via menu	Ctrl menu (move)	Push buttons
Access via toolbar	Ctrl + function keys	Pull-down menu and
		submenus options
Right-mouse options	Color	Dialog controls
Help links	Accelerators and hot keys	Labels
Context-sensitive help	Cancel	Chevrons
Button bars	Close	Ellipses
Open by double-click	Apply	Cray-out unavailability
Screen images and graphics	Exit	Check boxes
Open by menu	OK	Filters
Open by toolbar	Tile horizontal/vertical	Spin boxes
Icon access	Arrange icons	Sliders
Access to DOS	Toggling	Fonts
Access via single-click	Expand/contract tree	Drag/drop
Resize window panels	Function keys	Horizontal/vertical scrolling
Fields accept allowable values	Minimize the window Maximize the window	Cascade Window open
Fields handle invalid values	Tabbing sequence	

In addition to the GUI component checks, if there is a GUI design standard, it should be verified as well. GUI standards are essential to ensure that the internal rules of construction are followed to achieve the desired level of consistency. Some of the typical GUI standards that should be verified include the following:

- Forms "enterable" and display-only formats
- Wording of prompts, error messages, and help features
- Use of color, highlight, and cursors
- Screen layouts

- Function and shortcut keys, or "hot keys"
- Consistently locating screen elements on the screen
- Logical sequence of objects
- Consistent font usage
- Consistent color usage

It is also important to differentiate manual from automated GUI test cases. One way to accomplish this is to use an additional column in the GUI component matrix that indicates if the GUI test is manual or automated.

Step 3: Define the System/Acceptance Tests

Task 1: Identify Potential System Tests

System testing is the highest level of testing and evaluates the functionality as a total system, its performance, and overall fitness of use. This test is usually performed by the internal organization and is oriented to systems' technical issues rather than acceptance, which is a more user-oriented test.

Systems testing consists of one or more tests that are based on the original objectives of the system that were defined during the project interview. The purpose of this task is to select the system tests that will be performed, not how to implement the tests. Some common system test types include the following:

- *Performance testing*—Verifies and validates that the performance requirements have been met; measures response times, transaction rates, and other time-sensitive requirements.
- *Security testing*—Evaluates the presence and appropriate functioning of the security of the application to ensure the integrity and confidentiality of the data.
- *Volume testing*—Subjects the application to heavy volumes of data to determine if it can handle the volume of data.
- *Stress testing*—Investigates the behavior of the system under conditions that overload its resources. Of particular interest is the impact that this has on system processing time.
- *Compatibility testing*—Tests the compatibility of the application with other applications or systems.
- *Conversion testing*—Verifies the conversion of existing data and loads a new database.
- *Usability testing*—Determines how well the user will be able to use and understand the application.
- *Documentation testing*—Verifies that the user documentation is accurate and ensures that the manual procedures work correctly.
- *Backup testing*—Verifies the ability of the system to back up its data in the event of a software or hardware failure.
- *Recovery testing*—Verifies the system's ability to recover from a software or hardware failure.
- *Installation testing*—Verifies the ability to install the system successfully.

Task 2: Design System Fragment Tests

System fragment tests are partial system tests performed during each spiral iteration to identify potential problems early. These tests cover aspects like function, performance, security, usability, documentation, and procedures. Some fragment tests, such as functional and security tests, should be formally conducted during each spiral. Other tests, like usability and documentation, are more informal but still crucial and ongoing throughout development.

Function tests are performed as the system integrates new features, requiring test cases to be created and run for each new functionality. Security mechanisms are introduced early, so security tests are developed and updated with each iteration.

Usability testing is informal but continuous, with issues documented in the defect-tracking system. End users' reviews of the prototype serve as formal usability tests. Similarly, documentation and procedures are developed and tested in parallel with system development to prevent last-minute issues.

Performance testing occurs at a basic level (e.g., one user) during each spiral. Baseline measurements of key functions are established, and subsequent spirals compare performance to these benchmarks to track progress.

Table 15.4: Baseline Performance Measurements

Business Function	***Baseline Seconds—Rel 1.0 (1/1/2004)***	***Seconds—Rel 1.1 (2/1/2004)***	***Measure and Delta Seconds—Rel 1.2 (2/15/2004)***	***Measure and Delta Seconds —Rel 1.3 (3/1/2004)***	***Measure and Delta Seconds—Rel 1.4 (3/15/2004)***	***Measure and Delta Seconds—Rel 1.5 (4/1/2004)***
Order processing						
Create new order	1.0	1.5	1.3	1.0	.9	.75
	(+50%)	(−13%)	(−23%)	(−10%)	(−17%)	
Fulfill order	2.5	2.0	1.5	1.0	1.0	1.0
	(−20%)	(−25%)	(−33%)	(0%)	(0%)	
Edit order	1.76	2.0	2.5	1.7	1.5	1.2
	(+14%)	(+25%)	(−32%)	(−12%)	(−20%)	
Delete order	1.1	1.1	1.4	1.0	.8	.75
	(0%)	(+27%)	(−29%)	(−20%)	(−6%)	
•	•	•	•	•	•	•

Task 3: Identify Potential Acceptance Tests

Acceptance testing is an optional user-run test that demonstrates the ability of the application to meet the user's requirements. The motivation for this test is to demonstrate rather than be destructive, that is, to show that the system works. Less emphasis is placed on technical issues, and more is placed on the question of whether the system is a good business fit for the end user. The test is usually performed by users, if performed at all. Typically, 20 percent of the time, this test is rolled into the system test. If performed, acceptance tests typically are a subset of the system tests. However, the users sometimes define "special tests," such as intensive stress or volume tests, to stretch the limits of the system even beyond what was tested during the system test.

Step 4: Review/Approve Design

Task 1: Schedule/Prepare for Review

The test design review should be scheduled well in advance of the actual review, and the participants should have the latest copy of the test design.

As with any interview or review, it should contain certain elements. The first is defining what will be discussed, or "talking about what we are going to talk about." The second is discussing the details, or "talking about it." The third is summarization, or "talking about what we talked about." The final element is timeliness. The reviewer should state up front the estimated duration of the review and set the ground rule that if time expires before completing all items on the agenda, a follow-on review will be scheduled.

The purpose of this task is for development and the project sponsor to agree and accept the test design. If there are any suggested changes to the test design during the review, they should be incorporated into the design.

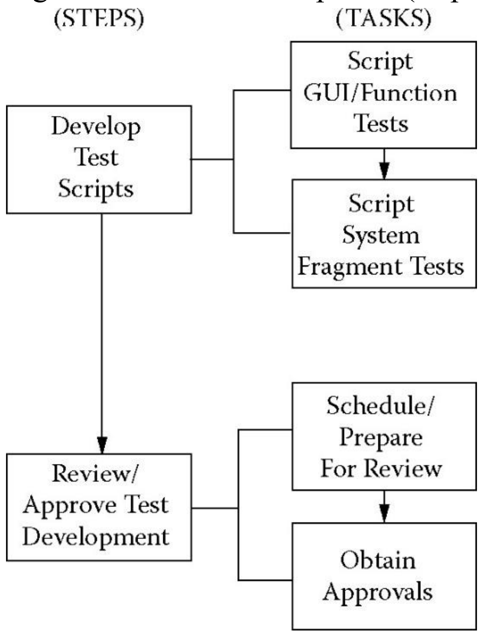
Task 2: Obtain Approvals

Approval is critical in a testing effort, because it helps provide the necessary agreements among testing, development, and the sponsor. The best approach is with a formal sign-off procedure of a test design. If this is the case, use the management approval sign-off forms. However, if a formal agreement procedure is not in place, send a memo to each key participant, including at least the project manager, development manager, and sponsor. In the document, attach the latest test design and point out that all their feedback comments have been incorporated and that if you do not hear from them, it is assumed that they agree with the design. Finally, indicate that in a spiral development environment, the test design will evolve with each iteration but that you will include them in any modification.

Chapter 16: Test Development (Do)

Figure 16.1 outlines the steps and tasks associated with the Do part of spiral testing. Each step and task is described, and valuable tips and techniques are provided.

Figure 16.1: Test development (steps/tasks)


(STEPS)
(TASKS)
Develop Test Scripts
Script GUI/Function Tests
Script System Fragment Tests
Review/ Approve Test Development
Schedule/ Prepare For Review
Obtain Approvals

Step 1: Develop Test Scripts
Task 1: Script the Manual/Automated GUI/Function Tests
In Section 15, a GUI/Function Test Matrix was built that cross-references the tests to the functions. The business functions are listed vertically, and the test cases are listed horizontally. The test case name is recorded on the matrix along with the number.

In the current task, the functional test cases are documented and transformed into reusable test scripts with test data created. To aid in the development of the script of the test cases, the GUI-based Function Test Matrix template in Table 16.1 can be used to document function test cases that are GUI-based (see "GUI-Based Functional Test Matrix," for more details).

Table 16.1: Function/GUI Test Script

Function (Create a New Customer Order)							
Case No.	***Req. No.***	***Test Objective***	***Case Steps***	***Expected Results***	***(P/F)***	***Tester***	***Date***
Menu Bar							
15	67	Create a valid new customer order	Select File/Create Order from the menu bar	Edit-Order Window appears	Passed	Jones	7/21/2004
Edit-Order Window							

Table 16.1: Function/GUI Test Script

Function (Create a New Customer Order)							
Case No.	***Req. No.***	***Test Objective***	***Case Steps***	***Expected Results***	***(P/F)***	***Tester***	***Date***
			1. Enter order number	Order validated	Passed	Jones	7/21/2004
			1. Enter customer number	Customer validated	Passed	Jones	7/21/2004
			1. Enter model number	Model validated	Passed	Jones	7/21/2004
			1. Enter product number	Product validated	Passed	Jones	7/21/2004
			1. Enter quantity	Quantity validated date, invoice number, and total price generated	Passed	Jones	7/21/2004
			1. Select OK	Customer is created successfully	Passed	Jones	7/21/2004

Consider the script in Table 16.1, which uses the template to create a new customer order. The use of this template shows the function, the case number within the test case (a variation of a specific test), the requirement identification cross-reference, the test objective, the case steps, the expected results, the pass/fail status, the tester name, and the date the test was performed. Within a function, the current GUI component is also documented. In Table 16.1, a new customer order is created by first invoking the menu bar to select the function, followed by the Edit-Order Window to enter the order number, customer number, model number, product number, and quantity.

Task 2: Script the Manual/Automated System Fragment Tests

In a previous task, the system fragment tests (Section 15) were designed. They are sample subsets of full system tests, which can be performed during each spiral loop.

In this task, the system fragment tests can be scripted using the GUI-based Function Test Matrix discussed in the previous task. The test objective description is probably more broad than the Function/GUI tests, as they involve more global testing issues such as performance, security, usability, documentation, procedure, and so on.

Step 2: Review/Approve Test Development

Task 1: Schedule/Prepare for Review

The test development review should be scheduled well in advance of the actual review and the participants should have the latest copy of the test design.

As with any interview or review, it should contain certain elements. The first is defining what will be discussed, or "talking about what we are going to talk about." The second is discussing the details, or "talking about it." The third is summarization, or "talking about what we talked about." The final element is timeliness. The reviewer should state up front the estimated duration of the review and set the ground rule that if time expires before completing all items on the agenda, a follow-on review will be scheduled.

The purpose of this task is for development and the project sponsor to agree and accept the test development. If there are any suggested changes to the test development during the review, they should be incorporated into the test development.

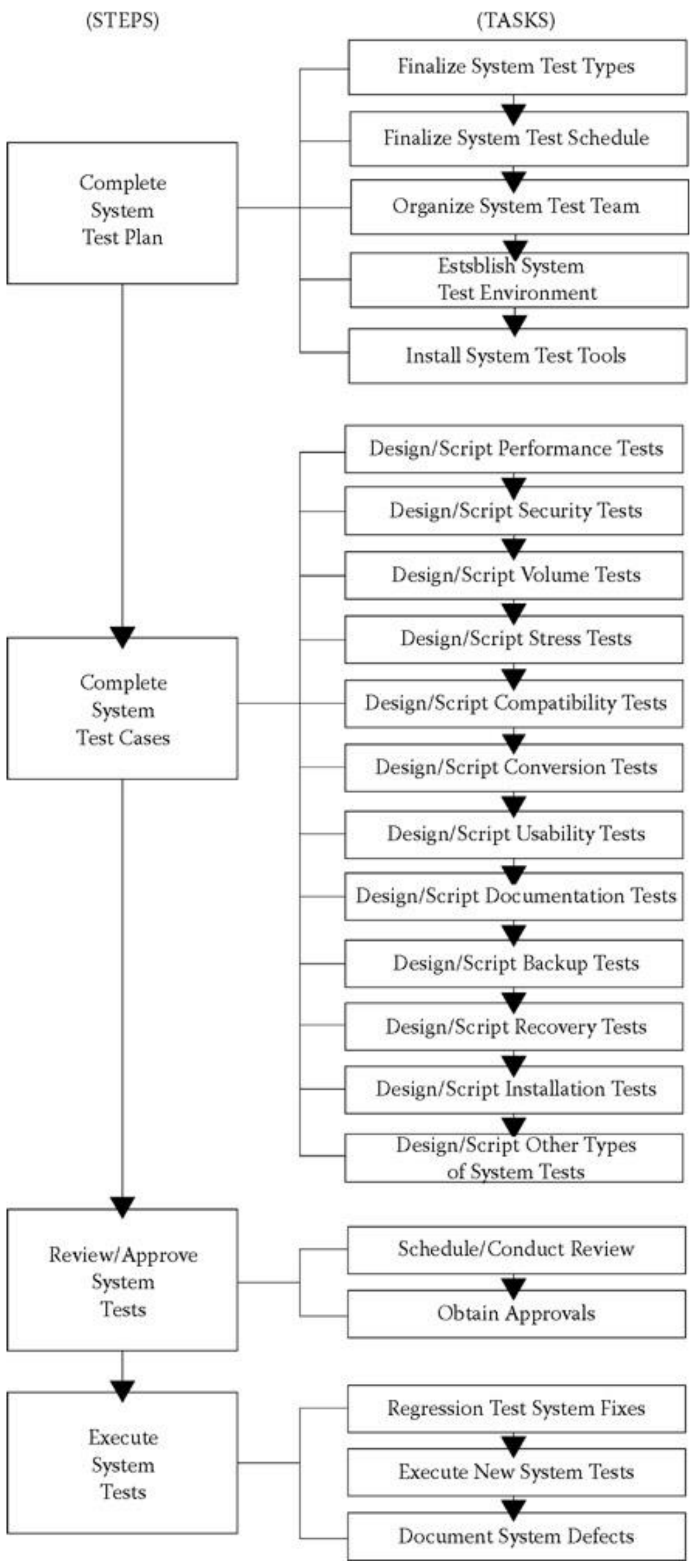
Task 2: Obtain Approvals

Approval is critical in a testing effort, because it helps provide the necessary agreements among the testing, development, and the sponsor. The best approach is with a formal sign-off procedure of a test development. If this is the case, use the management approval sign-off forms. However, if a formal agreement procedure is not in place, send a memo to each key participant, including at least the project manager, development manager, and sponsor. In the document, attach the latest test development, and point out that all their feedback comments have been incorporated and that if you do not hear from them, it is assumed that they agree with the development. Finally, indicate that in a spiral development environment, the test development will evolve with each iteration but that you will include them in any modification.

Section 20: Conduct the System Test (Act)

System testing evaluates the functionality and performance of the whole application and consists of a variety of tests including the following: performance, usability, stress, documentation, security, volume, recovery, and so on. Figure 20.1 describes how to extend fragment system testing. It includes discussions of how to prepare for the system tests, design and script them, execute them, and report anomalies discovered during the test.

Figure 20.1: Conduct system test (steps/tasks)


(STEPS)
(TASKS)
Complete System Test Plan
Finalize System Test Types
Finalize System Test Schedule
Organize System Test Team
Estsblish System Test Environment
Install System Test Tools
Complete System Test Cases
Design/Script Performance Tests
Design/Script Security Tests
Design/Script Volume Tests
Design/Script Stress Tests
Design/Script Compatibility Tests
Design/Script Conversion Tests
Design/Script Usability Tests
Design/Script Documentation Tests
Design/Script Backup Tests
Design/Script Recovery Tests
Design/Script Installation Tests
Design/Script Other Types of System Tests
Review/Approve System Tests
Schedule/Conduct Review
Obtain Approvals
Execute System Tests
Regression Test System Fixes
Execute New System Tests
Document System Defects

Step 1: Complete System Test Plan

Task 1: Finalize the System Test Types

In a previous task, a set of system fragment tests was selected and executed during each spiral. The purpose of the current task is to finalize the system test types that will be performed during system testing.

You will recall that systems testing consists of one or more tests that are based on the original objectives of the system, which were defined during the project interview. The purpose of this task is to select the system tests to be performed, not to implement the tests. Our initial list consisted of the following system test types:

- Performance
- Security
- Volume
- Stress
- Compatibility
- Conversion
- Usability
- Documentation
- Backup
- Recovery
- Installation

The sequence of system test-type execution should also be defined in this task. For example, related tests such as performance, stress, and volume might be clustered together and performed early during system testing. Security, backup, and recovery are also logical groupings, and so on. Finally, the system tests that can be automated with a testing tool need to be finalized. Automated tests provide three benefits: repeatability, leverage, and increased functionality. Repeatability enables automated tests to be executed more than once, consistently. Leverage comes from repeatability, from tests previously captured and tests that can be programmed with the tool, which might not have been possible without automation. As applications evolve, more and more functionality is added. With automation, the functional coverage is maintained with the test library.

Task 2: Finalize System Test Schedule

In this task, the system test schedule should be finalized; this includes the testing steps (and perhaps tasks), target start and target end dates, and responsibilities. It should also describe how it will be reviewed, tracked, and approved. A sample system test schedule is shown in Table 20.1.

Table 20.1: Final System Test Schedule

Test Step	*Begin Date*	*End Date*	*Responsible Staff Member*
General Setup			
Organize the system test team	12/1/2004	12/7/2004	Smith, test manager
Establish the system test environment	12/1/2004	12/7/2004	Smith, test manager
Establish the system test tools	12/1/2004	12/10/2004	Jones, tester
Performance Testing			
Design/script the tests	12/11/2004	12/15/2004	Jones, tester
Test review	12/16/2004	12/16/2004	Smith, test manager
Execute the tests	12/17/2004	12/22/2004	Jones, tester
Retest system defects	12/23/2004	12/25/2004	Jones, tester
Stress Testing			
Design/script the tests	12/26/2004	12/30/2004	Jones, tester
Test review	12/31/2004	12/31/2004	Smith, test manager
Execute the tests	1/1/2004	1/6/2004	Jones, tester
Retest system defects	1/7/2004	1/9/2004	Jones, tester
Volume Testing			
Design/script the tests	1/10/2004	1/14/2004	Jones, tester
Test review	1/15/2004	1/15/2004	Smith, test manager
Execute the tests	1/16/2004	1/21/2004	Jones, tester
Retest system defects	1/22/2004	1/24/2004	Jones, tester
Security Testing			
Design/script the tests	1/25/2004	1/29/2004	Jones, tester
Test review	1/30/2004	1/31/2004	Smith, test manager
Execute the tests	2/1/2004	2/6/2004	Jones, tester
Retest system defects	2/7/2004	2/9/202004	Jones, tester
Backup Testing			
Design/script the tests	2/10/2004	2/14/2004	Jones, tester
Test review	2/15/2004	2/15/2004	Smith, test manager
Execute the tests	2/16/2004	1/21/2004	Jones, tester
Retest system defects	2/22/2004	2/24/2004	Jones, tester
Recovery Testing			
Design/script the tests	2/25/2004	2/29/2004	Jones, tester
Test review	2/30/2004	2/31/2004	Smith, test manager
Execute the tests	3/1/2004	3/6/2004	Jones, tester
Retest system defects	3/7/2004	3/9/2004	Jones, tester
Compatibility Testing			
Design/script the tests	3/10/2004	3/14/2004	Jones, tester

Table 20.1: Final System Test Schedule

Test Step	***Begin Date***	***End Date***	***Responsible Staff Member***
Test review	3/15/2004	3/15/2004	Smith, test manager
Execute the tests	3/16/2004	3/21/2004	Jones, tester
Retest system defects	3/22/2004	3/24/2004	Jones, tester
Conversion Testing			
Design/script the tests	4/10/2004	4/14/2004	Jones, tester
Test review	4/15/2004	4/15/2004	Smith, test manager
Execute the tests	4/16/2004	4/21/2004	Jones, tester
Retest system defects	4/22/2004	4/24/2004	Jones, tester
Usability Testing			
Design/script the tests	5/10/2004	5/14/2004	Jones, tester
Test review	5/15/2004	5/15/2004	Smith, test manager
Execute the tests	5/16/2004	5/21/2004	Jones, tester
Retest system defects	5/22/2004	5/24/2004	Jones, tester
Documentation Testing			
Design/script the tests	6/10/2004	6/14/2004	Jones, tester
Test review	6/15/2004	6/15/2004	Smith, test manager
Execute the tests	6/16/2004	6/21/2004	Jones, tester
Retest system defects	6/22/2004	6/24/2004	Jones, tester
Installation Testing			
Design/script the tests	7/10/2004	7/14/2004	Jones, tester
Test review	7/15/2004	7/15/2004	Smith, test manager
Execute the tests	7/16/2004	7/21/2004	Jones, tester
Retest system defects	7/22/2004	7/24/2004	Jones, tester

Task 3: Organize the System Test Team

With all testing types, the system test team needs to be organized. The system test team is responsible for designing and executing the tests, evaluating the results and reporting any defects to development, and using the defect-tracking system. When development corrects defects, the test team retests the defects to verify the correction.

The system test team is led by a test manager whose responsibilities include the following:

- Organizing the test team
- Establishing the test environment
- Organizing the testing policies, procedures, and standards
- Assurance test readiness
- Working the test plan and controlling the project
- Tracking test costs

- Ensuring test documentation is accurate and timely
- Managing the team members

Task 4: Establish the System Test Environment
During this task, the system test environment is also finalized. The purpose of the test environment is to provide a physical framework for the testing activity. The test environment needs are established and reviewed before implementation.

The main components of the test environment include the physical test facility, technologies, and tools. The test facility component includes the physical setup. The technologies component includes the hardware platforms, physical network and all its components, operating system software, and other software. The tools component includes any specialized testing software, such as automated test tools, testing libraries, and support software.

The testing facility and workplace need to be established. These may range from an individual workplace configuration to a formal testing laboratory. In any event, it is important that the testers be together and near the development team. This facilitates communication and the sense of a common goal. The system testing tools need to be installed.

The hardware and software technologies need to be set up. This includes the installation of test hardware and software and coordination with vendors, users, and information technology personnel. It may be necessary to test the hardware and coordinate with hardware vendors. Communication networks need to be installed and tested.

Task 5: Install the System Test Tools
During this task, the system test tools are installed and verified for readiness. A trial run of tool test cases and scripts should be performed to verify that the test tools are ready for the actual acceptance test. Some other tool readiness considerations include the following:

- Test team tool training
- Tool compatibility with operating environment
- Ample disk space for the tools
- Maximizing the tool potentials
- Vendor tool help hotline
- Test procedures modified to accommodate tools
- Installing the latest tool changes
- Verifying the vendor contractual provisions

Step 2: Complete System Test Cases
During this step, the system test cases are designed and scripted. The conceptual system test cases are transformed into reusable test scripts with test data created.

To aid in developing the script test cases, the GUI-based Function Test Matrix template can be used to document system-level test cases, with the "function" heading replaced with the system test name.

Task 1: Design/Script the Performance Tests

The objective of performance testing is to measure the system against predefined objectives. The required performance levels are compared against the actual performance levels and discrepancies are documented.

Performance testing is a combination of black-box and white-box testing. From a black-box point of view, the performance analyst does not have to know the internal workings of the system. Real workloads or benchmarks are used to compare one system version with another for performance improvements or degradation. From a white-box point of view, the performance analyst needs to know the internal workings of the system and define specific system resources to investigate, such as instructions, modules, and tasks.

Some of the performance information of interest includes the following:

- CPU utilization
- IO utilization
- Number of IOs per instruction
- Channel utilization
- Main storage memory utilization
- Secondary storage memory utilization
- Percentage of execution time per module
- Percentage of time a module is waiting for IO completion
- Percentage of time module spent in main storage
- Instruction trace paths over time
- Number of times control is passed from one module to another
- Number of waits encountered for each group of instructions
- Number of pages-in and pages-out for each group of instructions
- System response time, for example, last key until first key time
- System throughput, that is, number of transactions per time unit
- Unit performance timings for all major functions

Baseline performance measurements should first be taken on all major functions in a noncontention mode, for example, unit measurements of functions when a single task is in operation. This can be easily done with a simple stopwatch, as was done earlier for each spiral. The next set of measurements should be made in a system-contended mode in which multiple tasks are operating, and queuing results in demands on common resources such as CPU, memory, storage, channel, network, and so on. Contended system execution time and resource utilization performance measurements are performed by monitoring the system to identify potential areas of inefficiency.

There are two approaches to gathering system execution time and resource utilization. With the first approach, samples are taken while the system is executing in its typical environment with the use of external probes, performance monitors, or a stopwatch. With the other approach, probes are inserted into the system code, for example, calls to a performance monitor program that gathers the performance information. The following is a discussion of each approach, followed by a discussion of test drivers, which are support techniques used to generate data for the performance study.

Monitoring Approach
This approach involves monitoring a system by determining its status at periodic time intervals, and is controlled by an elapsed time facility in the testing tool or operating system. Samples taken during each time interval indicate the status of the performance criteria during the interval. The smaller the time interval, the more precise the sampling accuracy.

Statistics gathered by the monitoring are collected and summarized in performance.

Probe Approach
This approach involves inserting probes or program instructions into the system programs at various locations. To determine, for example, the CPU time necessary to execute a sequence of statements, a problem execution results in a call to the data collection routine that records the CPU clock at that instant. A second probe execution results in a second call to the data collection routine. Subtracting the first CPU time from the second yields the net CPU time used. Reports can be produced showing execution time breakdowns by statement, module, and statement type.

The value of these approaches is their use as performance requirements validation tools. However, formally defined performance requirements must be stated, and the system should be designed so that the performance requirements can be traced to specific system modules.

Test Drivers
In many cases test drivers and test harnesses are required to make system performance measurements. A test driver provides the facilities needed to execute a system, for example, inputs. The input data files for the system are loaded with data values representing the test situation to yield recorded data to evaluate against the expected results. Data are generated in an external form and presented to the system.

Performance test cases need to be defined, using one or more of the test templates located in the appendices, and test scripts need to be built. Before any performance test is conducted, however, the performance analyst must make sure that the target system is relatively bug-free. Otherwise, a lot of time will be spent documenting and fixing defects rather than analyzing the performance. The following are the five recommended steps for any performance study:

1. Document the performance objectives; for example, exactly what the measurable performance criteria are must be verified.
2. Define the test driver or source of inputs to drive the system.
3. Define the performance methods or tools that will be used.
4. Define how the performance study will be conducted; for example, what is the baseline, what are the variations, how can it be verified as repeatable, and how does one know when the study is complete?
5. Define the reporting process, for example, techniques and tools.

Task 2: Design/Script the Security Tests

The objective of security testing is to evaluate the presence and appropriate functioning of the security of the application to ensure the integrity and confidentiality of the data. Security tests should be designed to demonstrate how resources are protected.

A Security Design Strategy

A security strategy for designing security test cases is to focus on the following four security components: the assets, threats, exposures, and controls. In this manner, matrices and checklists will suggest ideas for security test cases.

Assets are the tangible and intangible resources of an entity. The evaluation approach is to list what should be protected. It is also useful to examine the attributes of assets, such as amount, value, use, and characteristics. Two useful analysis techniques are asset value and exploitation analysis. Asset value analysis determines how the value differs among users and potential attackers. Asset exploitation analysis examines different ways to use an asset for illicit gain.

Threats are events with the potential to cause loss or harm. The evaluation approach is to list the sources of potential threats. It is important to distinguish among accidental, intentional, and natural threats, and threat frequencies.

Exposures are forms of possible loss or harm. The evaluation approach is to list what might happen to assets if a threat is realized. Exposures include disclosure violations, erroneous decision, and fraud. Exposure analysis focuses on identifying areas in which exposure is the greatest.

Security functions or controls are measures that protect against loss or harm. The evaluation approach is to list the security functions and tasks, and focus on controls embodied in specific system functions or procedures. Security functions assess the protection against human errors and casual attempts to misuse the system. Some functional security questions include the following:

- Do the control features work properly?
- Are invalid and improbable parameters detected and properly handled?
- Are invalid or out-of-sequence commands detected and properly handled?
- Are errors and file accesses properly recorded?
- Do procedures for changing security tables work?
- Is it possible to log in without a password?
- Are valid passwords accepted and invalid passwords rejected?
- Does the system respond properly to multiple invalid passwords?
- Does the system-initialed authentication function properly?
- Are there security features for remote access?

It is important to assess the performance of the security mechanisms as well as the functions themselves. Some questions and issues concerning security performance include the following:

- *Availability*—What portion of time is the application or control available to perform critical security functions? Security controls usually require higher availability than other portions of the system.
- *Survivability*—How well does the system withstand major failures or natural disasters? This includes the support of emergency operations during failure, backup operations afterward, and recovery actions to return to regular operation.
- *Accuracy*—How accurate is the security control? Accuracy encompasses the number, frequency, and significance of errors.
- *Response time*—Are response times acceptable? Slow response times can tempt users to bypass security controls. Response time can also be critical for control management, for example, the dynamic modification of security tables.
- *Throughput*—Does the security control support required use capacities? Capacity includes the peak and average loading of users and service requests.

A useful performance test is stress testing, which involves large numbers of users and requests to attain operational stress conditions. Stress testing is used to attempt to exhaust limits for such resources as buffers, queues, tables, and ports. This form of testing is useful in evaluating protection against service denial threats.

Task 3: Design/Script the Volume Tests

The objective of volume testing is to subject the system to heavy volumes of data to find out if it can handle the volume. This test is often confused with stress testing. Stress testing subjects the system to heavy loads or stresses in terms of rates, such as throughputs over a short time period. Volume testing is data oriented, and its purpose is to show that the system can handle the volume of data specified in its objectives.

Some examples of volume testing are as follows:

- Relative data comparison is made when processing date-sensitive transactions.
- A compiler is fed an extremely large source program to compile.
- A linkage editor is fed a program containing thousands of modules.
- An electronic-circuit simulator is given a circuit containing thousands of components.
- An operation system's job queue is filled to maximum capacity.
- Enough data is created to cause a system to span files.
- A test-formatting system is fed a massive document format.
- The Internet is flooded with huge e-mail messages and files.

Task 4: Design/Script the Stress Tests

The objective of stress testing is to investigate the behavior of the system under conditions that overload its resources. Of particular interest is the impact that this has on the system processing time. Stress testing is boundary testing. For example, test with the maximum number of terminals active and then add more terminals than specified in the requirements under different limit combinations. Some of the resources subjected to heavy loads by stress testing include the following:

- Buffers
- Controllers
- Display terminals
- Interrupt handlers
- Memory
- Networks
- Printers
- Spoolers
- Storage devices
- Transaction queues
- Transaction schedulers
- User of the system

Stress testing studies the system's response to peak bursts of activity in short periods of time and attempts to find defects in a system. It is often confused with volume testing, in which the system's capability of handling large amounts of data is the objective.

Stress testing should be performed early in development because it often uncovers major design flaws that can have an impact on many areas. If stress testing is not performed early, subtle defects, which might have been more apparent earlier in development, may be difficult to uncover.
The following are the suggested steps for stress testing:

1. Perform simple multitask tests.

2. After the simple stress defects are corrected, stress the system to breaking point.
3. Perform the stress tests repeatedly for every spiral.

Some stress-testing examples include the following:

- Word-processing response time for a fixed entry rate, such as 120 words per minute
- Introducing a heavy volume of data in a very short period of time
- Varying loads for interactive, real-time process control
- Simultaneous introduction of a large number of transactions
- Thousands of users signing on to the Internet within a minute

Task 5: Design/Script the Compatibility Tests

The objective of compatibility testing (sometimes called *cohabitation testing*) is to test the compatibility of the application with other applications or systems. This is a test that is often overlooked until the system is put into production. Defects are often subtle and difficult to uncover in this test. An example is when the system works perfectly in the testing laboratory in a controlled environment, but does not work when it coexists with other applications. An example of compatibility is when two systems share the same data or data files or reside in the same memory at the same time. The system may satisfy the system requirements, but not work in a shared environment; it may also interfere with other systems.

The following is a compatibility (cohabitation) testing strategy:

1. Update the compatibility objectives to note how the application has actually been developed and the actual environments in which it is to perform. Modify the objectives for any changes in the cohabiting systems or the configuration resources.
2. Update the compatibility test cases to make sure they are comprehensive. Make sure that the test cases in the other systems that can affect the target system are comprehensive. And ensure maximum coverage of instances in which one system could affect another.
3. Perform the compatibility tests and carefully monitor the results to ensure the expected results. Use a baseline approach, which is the system's operating characteristics before the incorporation of the target system into the shared environment. The baseline needs to be accurate and incorporate not only the functioning but also the operational performance to ensure that it is not degraded in a cohabitation setting.
4. Document the results of the compatibility tests and note any deviations in the target system or the other cohabitation systems.
5. Regression test the compatibility tests after the defects have been resolved, and record the tests in the retest matrix.

Task 6: Design/Script the Conversion Tests

The objective of conversion testing is to verify the conversion of existing data and load a new database. The most common conversion problem is between two versions of the same system. A new version may have a different data format, but must include the data from the old system. Ample time needs to be set aside to carefully think of all the conversion issues that may arise.

Some key factors that need to be considered when designing conversion tests include the following:

- *Auditability*—There needs to be a plan to perform before-and-after comparisons and analysis of the converted data to ensure it was converted successfully. Techniques to ensure auditability include file reports, comparison programs, and regression testing. Regression testing checks to verify that the converted data does not change the business requirements or cause the system to behave differently.
- *Database verification*—Prior to conversion, the new database needs to be reviewed to verify that it is designed properly, satisfies the business needs, and that the support center and database administrators are trained to support it.
- *Data cleanup*—Before the data is converted to the new system, the old data needs to be examined to verify that inaccuracies or discrepancies in the data are removed.
- *Recovery plan*—Roll-back procedures need to be in place before any conversion is attempted to restore the system to its previous state and undo the conversions.
- *Synchronization*—It must be verified that the conversion process does not interfere with normal operations. Sensitive data, such as customer data, may be changing dynamically during conversions. One way to achieve this is to perform conversions during nonoperational hours.

Task 7: Design/Script the Usability Tests

The objective of usability testing is to determine how well the user will be able to use and understand the application. This includes the system functions, publications, help text, and procedures to ensure that the user comfortably interacts with the system. Usability testing should be performed as early as possible during development and should be designed into the system. Late usability testing might be impossible, because it is locked in and often requires a major redesign of the system to correct serious usability problems. This may make it economically infeasible.

Some of the usability problems the tester should look for include the following:

- Overly complex functions or instructions
- Difficult installation procedures
- Poor error messages, for example, “syntax error”
- Syntax difficult to understand and use
- Nonstandardized GUI interfaces
- User forced to remember too much information
- Difficult log-in procedures
- Help text not context sensitive or not detailed enough

- Poor linkage to other systems
- Unclear defaults
- Interface too simple or too complex
- Inconsistency of syntax, format, and definitions
- User not provided with clear acknowledgment of all inputs

Task 8: Design/Script the Documentation Tests

The objective of documentation testing is to verify that the user documentation is accurate and ensure that the manual procedures work correctly. Documentation testing has several advantages, including improving the usability of the system, reliability, maintainability, and installability. In these cases, testing the document will help uncover deficiencies in the system or make the system more usable.

Documentation testing also reduces customer support costs; when customers can figure out answers to their questions by reading the documentation, they are not forced to call the help desk. The tester verifies the technical accuracy of the documentation to ensure that it agrees with and describes the system accurately. He or she needs to assume the user's point of view and carry out the steps described in the documentation.

Some tips and suggestions for the documentation tester include the following:

- Use documentation as a source of many test cases.
- Use the system exactly as the documentation describes it should be used.
- Test every hint or suggestion.
- Incorporate defects into the defect-tracking database.
- Test every online help hypertext link.
- Test every statement of fact, and do not take anything for granted.
- Work like a technical editor rather than a passive reviewer.
- Perform a general review of the whole document first and then a detailed review.
- Check all the error messages.
- Test every example provided in the document.
- Make sure all index entries have documentation text.
- Make sure documentation covers all key user functions.
- Make sure the reading style is not too technical.
- Look for areas that are weaker than others and need more explanation.

Task 9: Design/Script the Backup Tests

The objective of backup testing is to verify the ability of the system to back up its data in the event of a software or hardware failure. This test is complementary to recovery testing and should be part of recovery test planning.

Some backup testing considerations include the following:

- Backing up files and comparing the backup with the original
- Archiving files and data
- Complete system backup procedures
- Checkpoint backups
- Backup performance system degradation
- Effect of backup on manual processes
- Detection of “triggers” to backup system
- Security procedures during backup
- Maintaining transaction logs during backup procedures

Task 10: Design/Script the Recovery Tests

The objective of recovery testing is to verify the system's ability to recover from a software or hardware failure. This test verifies the contingency features of the system for handling interruptions and returning to specific points in the application's processing cycle. The key questions for designing recovery tests are as follows:

- Have the potentials for disasters and system failures, and their respective damages, been identified? Fire-drill brainstorming sessions can be an effective method of defining disaster scenarios.
- Do the prevention and recovery procedures provide for adequate responses to failures? The plan procedures should be tested with technical reviews by subject matter experts and the system users.
- Will the recovery procedures work properly when really needed? Simulated disasters need to be created with the actual system verifying the recovery procedures. This should involve the system users, the support organization, vendors, and so on.

Some recovery testing examples include the following:

- Complete restoration of files that were backed up either during routine maintenance or error recovery
- Partial restoration of file backup to the last checkpoint
- Execution of recovery programs
- Archive retrieval of selected files and data
- Restoration when power supply is the problem
- Verification of manual recovery procedures
- Recovery by switching to parallel systems
- Restoration performance system degradation
- Security procedures during recovery
- Ability to recover transaction logs

Task 11: Design/Script the Installation Tests

The objective of installation testing is to verify the ability to install the system successfully. Customers have to install the product on their systems. Installation is often the developers' last activity and often receives the least amount of attention during development. Yet, it is the first activity that the customer performs when using the new system. Therefore, clear and concise installation procedures are among the most important parts of the system documentation.

Reinstallation procedures need to be included to be able to reverse the installation process and validate the previous environmental condition. Also, the installation procedures need to document how the user can tune the system options and upgrade from a previous version.

Some key installation questions the tester needs to consider include the following:

- Who is the user installer? For example, what technical capabilities are assumed?
- Is the installation process documented thoroughly with specific and concise installation steps?
- For which environments are the installation procedures supposed to work, for example, platforms, software, hardware, networks, or versions?
- Will the installation change the user's current environmental setup, for example, config.sys, and so on?
- How does the installer know the system has been installed correctly? For example, is there an installation test procedure in place?

Task 12: Design/Script other System Test Types

In addition to the foregoing system tests, the following system tests may also be required:

- *API testing*—Verify the system uses APIs correctly, for example, operating system calls.
- *Communication testing*—Verify the system's communications and networks.
- *Configuration testing*—Verify that the system works correctly in different system configurations, for example, software, hardware, and networks.
- *Database testing*—Verify the database integrity, business rules, access, and refresh capabilities.
- *Degraded system testing*—Verify that the system performs properly under less than optimum conditions, for example, line connections down, and the like.
- *Disaster recovery testing*—Verify that the system recovery processes work correctly.
- *Embedded system test*—Verify systems that operate on low-level devices, such as video chips.
- *Facility testing*—Verify that each stated requirement facility is met.
- *Field testing*—Verify that the system works correctly in the real environment.
- *Middleware testing*—Verify that the middleware software works correctly, for example, the common interfaces and accessibility among clients and servers.

- *Multimedia testing*—Verify the multimedia system features, which use video, graphics, and sound.
- *Online help testing*—Verify that the system's online help features work properly.
- *Operability testing*—Verify system will work correctly in the actual business environment.
- *Package testing*—Verify that the installed software package works correctly.
- *Parallel testing*—Verify that the system behaves the same in the old and new versions.
- *Port testing*—Verify that the system works correctly on different operating systems and computers.
- *Procedure testing*—Verify that nonautomated procedures work properly, for example, operation, DBA, and the like.
- *Production testing*—Verify that the system will work correctly during actual ongoing production and not just in the test laboratory environment.
- *Real-time testing*—Verify systems in which time issues are critical and there are response time requirements.
- *Reliability testing*—Verify that the system works correctly within predefined expected failure duration, for example, mean time to failure (MTF).
- *Serviceability testing*—Verify that service facilities of the system work properly, for example, mean time to debug a defect and maintenance procedures.
- *SQL testing*—Verify the queries, data retrievals, and updates.
- *Storage testing*—Verify that the system storage requirements are met, for example, sizes of spill files and amount of main or secondary storage used.

Step 3: Review/Approve System Tests
Task 1: Schedule/Conduct the Review
The system test plan review should be scheduled well in advance of the actual review, and the participants should have the latest copy of the test plan.

As with any interview or review, certain elements must be present. The first is defining what will be discussed; the second is discussing the details; and the third is summarization. The final element is timeliness. The reviewer should state up front the estimated duration of the review and set the ground rule that if time expires before completing all items on the agenda, a follow-on review will be scheduled.

The purpose of this task is for development and the project sponsor to agree and accept the system test plan. If there are any suggested changes to the test plan during the review, they should be incorporated into the test plan.

Task 2: Obtain Approvals
Approval is critical in a testing effort because it helps testing, development, and the sponsor agree. The best approach is with a formal sign-off procedure of a system test plan. If this is the case, use

the management approval sign-off forms. However, if a formal agreement procedure is not in place, send a memo to each key participant including at least the project manager, development manager, and sponsor. In the document, attach the latest test plan and point out that all their feedback comments have been incorporated and that if you do not hear from them, it is assumed that they agree with the plan. Finally, indicate that in a spiral development environment, the system test plan will evolve with each iteration but that you will include them in any modification.

Step 4: Execute the System Tests

Task 1: Regression Test the System Fixes

The purpose of this task is to retest the system tests that discovered defects in the previous system test cycle for this build. The technique used is regression testing. Regression testing is a technique that detects spurious errors caused by software modifications or corrections.
A set of test cases must be maintained and available throughout the entire life of the software. The test cases should be complete enough so that all the software's functional capabilities are thoroughly tested. The question arises as to how to locate those test cases to test defects discovered during the previous test spiral. An excellent mechanism is the retest matrix.

As described earlier, a retest matrix relates test cases to functions (or program units). A check entry in the matrix indicates that the test case is to be retested when the function (or program unit) has been modified due to enhancements or corrections. The absence of an entry indicates that the test does not need to be retested. The retest matrix can be built before the first testing spiral, but needs to be maintained during subsequent spirals. As functions (or program units) are modified during a development spiral, existing or new test cases need to be created and checked in the retest matrix in preparation for the next test spiral. Over time with subsequent spirals, some functions (or program units) may be stable, with no recent modifications. Selective removal of check entries should be considered, and undertaken between testing spirals.

Task 2: Execute the New System Tests

The purpose of this task is to execute new system tests that were created at the end of the previous system test cycle. In the previous spiral, the testing team updated the function/GUI, system fragment, and acceptance tests in preparation for the current testing spiral. During this task, those tests are executed.

Task 3: Document the System Defects

During system test execution, the results of the testing must be reported in the defect-tracking database. These defects are typically related to individual tests that have been conducted. However, variations to the formal test cases often uncover other defects. The objective of this task is to produce a complete record of the defects.

If the execution step has been recorded properly, the defects have already been recorded on the defect-tracking database. If the defects are already recorded, the objective of this step becomes to collect and consolidate the defect information.

Tools can be used to consolidate and record defects depending on the test execution methods. If the defects are recorded on paper, the consolidation involves collecting and organizing the papers. If the defects are recorded electronically, search features can easily locate duplicate defects.

Section 21: Conduct Acceptance Testing

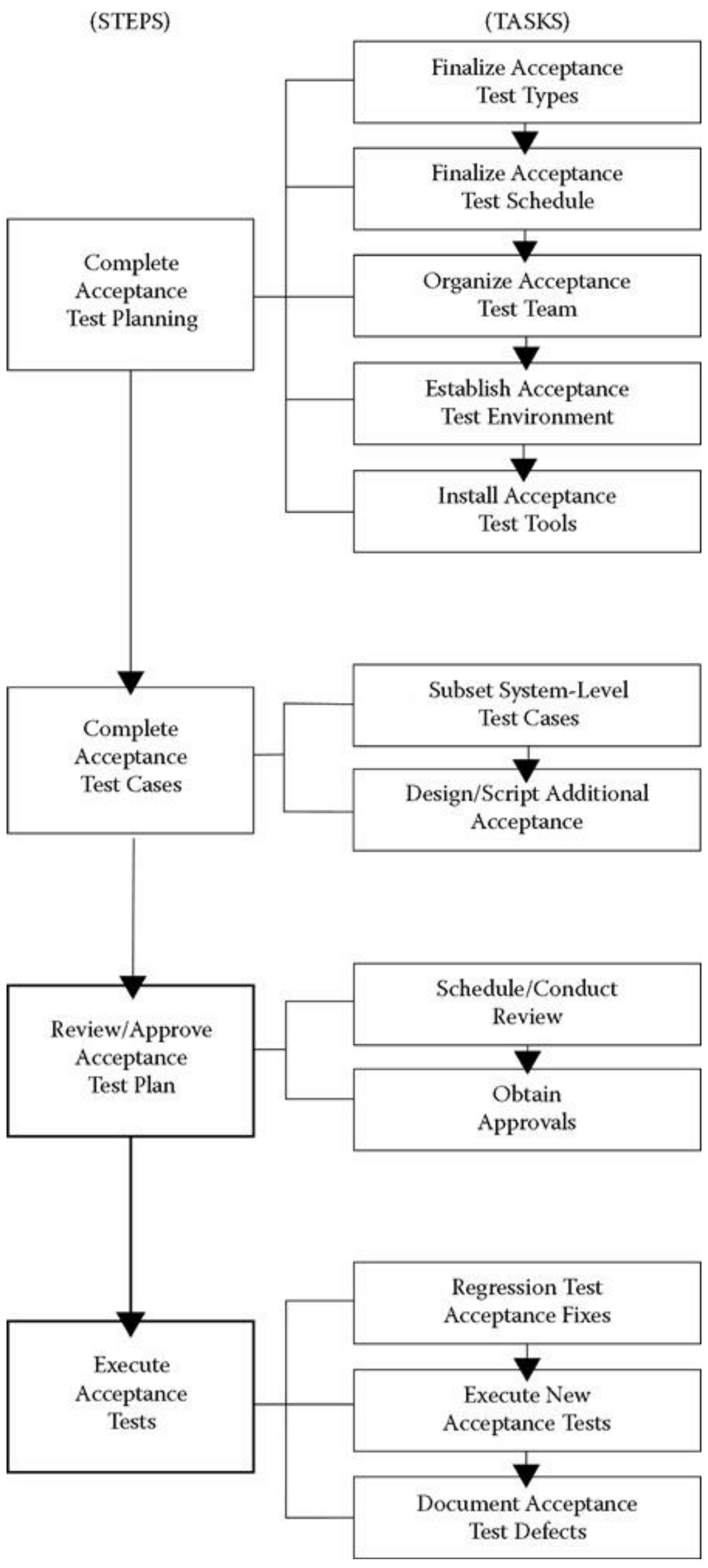
Acceptance testing is a user-run test that demonstrates the application's ability to meet the original business objectives and system requirements, and usually consists of a subset of system tests (see Figure 21.1). It includes discussions on how to prepare for the acceptance tests, design and script them, execute them, and report anomalies discovered during the test.
Figure 21.1: Conduct acceptance testing (steps/tasks)

Step 1: Complete Acceptance Test Planning

Task 1: Finalize the Acceptance Test Types

In this task, the initial acceptance testing type list is refined, and the actual tests to be performed are selected.

Acceptance testing is an optional user-run test that demonstrates the ability of the application to meet the user's requirements. The motivation for this test is to demonstrate rather than be destructive, that is, to show that the system works. Less emphasis is placed on the technical issues and more on the question of whether the system is a good business fit for the end user. Users usually perform the test. However, the users sometimes define “special tests,” such as intensive stress or volume tests, to stretch the limits of the system even beyond what was tested during the system test.


(STEPS)
(TASKS)
Complete Acceptance Test Planning
Finalize Acceptance Test Types
Finalize Acceptance Test Schedule
Organize Acceptance Test Team
Establish Acceptance Test Environment
Install Acceptance Test Tools
Complete Acceptance Test Cases
Subset System-Level Test Cases
Design/Script Additional Acceptance
Review/Approve Acceptance Test Plan
Schedule/Conduct Review
Obtain Approvals
Execute Acceptance Tests
Regression Test Acceptance Fixes
Execute New Acceptance Tests
Document Acceptance Test Defects

Task 2: Finalize the Acceptance Test Schedule

In this task, the acceptance test schedule should be finalized. It includes the testing steps (and perhaps tasks), target begin dates and target end dates, and responsibilities. It should also describe how it will be reviewed, tracked, and approved. For acceptance testing, the test team usually consists of user representatives. However, the team test environment and test tool are probably the same as those used during system testing. A sample acceptance test schedule is shown in Table 21.1.

Table 21.1: Acceptance Test Schedule

Test Step	Begin Date	End Date	*Responsible Staff Member*
General Setup			
Organize the acceptance test team	8/1/2004	8/7/2004	Smith, test manager
Establish the acceptance test environment	8/8/2004	8/9/2004	Smith, test manager
Establish the acceptance test tools	8/10/2004	8/10/2004	Jones, tester
Acceptance Testing			
Design/script the tests	12/11/2004	12/15/2004	Jones, Baker (user), testers
Test review	12/16/2004	12/16/2004	Smith, test manager
Execute the tests	12/17/2004	12/22/2004	Jones, Baker (user), tester
Retest acceptance defects	12/23/2004	12/25/2004	Jones, Baker (user), tester

Task 3: Organize the Acceptance Test Team

The acceptance test team is responsible for designing and executing the tests, evaluating the test results, and reporting any defects to development, using the defect-tracking system. When development corrects defects, the test team retests the defects to validate the correction. The acceptance test team typically has representation from the user community, because this is their final opportunity to accept the system.

The acceptance test team is led by a test manager whose responsibilities include the following:

- Organizing the test team
- Establishing the test environment
- Organizing the testing policies, procedures, and standards
- Ensuring test readiness
- Working the test plan and controlling the project
- Tracking test costs
- Ensuring test documentation is accurate and timely
- Managing the team members

Task 4: Establish the Acceptance Test Environment

During this task, the acceptance test environment is finalized. Typically, the test environment for acceptance testing is the same as that for system testing. The purpose of the test environment is to provide the physical framework necessary for the testing activity. For this task, the test environment needs are established and reviewed before implementation.

The Business usually performs the user acceptance tests. Thus, it is important that the details of the acceptance test environment be communicated to them.

Task 5: Install Acceptance Test Tools
During this task, the acceptance test tools are installed and verified for readiness. A trial run of sample tool test cases and scripts should be performed to verify that the test tools are ready for the actual acceptance test. Typically, the acceptance testing tools are the same as the system level testing tools, but this needs to be confirmed between the Business and the QA department. Some other tool readiness considerations include the following:

- Test team tool training
- Tool compatibility with operating environment
- Ample disk space for the tools
- Maximizing the tool potentials
- Vendor tool help hotline
- Test procedures modified to accommodate tools
- Installing the latest tool changes
- Verifying the vendor contractual provisions

Step 3: Review/Approve Acceptance Test Plan
Task 1: Schedule/Conduct the Review
The acceptance test plan review should be scheduled well in advance of the actual review, and the participants should have the latest copy of the test plan.

As with any interview or review, it should contain certain elements. The first defines what will be discussed; the second discusses the details; the third summarizes; and the final element is timeliness. The reviewer should state up front the estimated duration of the review and set the ground rule that if the allotted time expires before completing all items on the agenda, a follow-on review will be scheduled.

The purpose of this task is for development and the project sponsor to agree and accept the system test plan. If there are any suggested changes to the test plan during the review, they should be incorporated into the test plan.

Task 2: Obtain Approvals
Approval is critical in a testing effort because it helps provide the necessary agreements among testing, development, and the sponsor. The best approach is with a formal sign-off procedure of an acceptance test plan. If this is the case, use the management approval sign-off forms. However, if

a formal agreement procedure is not in place, send a memo to each key participant, including at least the project manager, development manager, and sponsor. Attach to the document the latest test plan, and point out that all feedback comments have been incorporated and that if you do not hear from them, it is assumed they agree with the plan. Finally, indicate that in a spiral development environment, the system test plan will evolve with each iteration but that you will include them in any modification.

Step 4: Execute the Acceptance Tests

Task 1: Regression Test the Acceptance Fixes

The purpose of this task is to retest the tests that discovered defects in the previous acceptance test cycle for this build. The technique used is regression testing. Regression testing detects spurious errors caused by software modifications or corrections.

A set of test cases must be maintained and made available throughout the entire life of the software. The test cases should be complete enough so that all the software's functional capabilities are thoroughly tested. The question arises as to how to locate those test cases to test defects discovered during the previous test spiral. An excellent mechanism is the retest matrix.

As described earlier, a retest matrix relates test cases to functions (or program units). A check entry in the matrix indicates that the test case is to be retested when the function (or program unit) has been modified due to enhancements or corrections. The absence of an entry indicates that the test does not need to be retested. The retest matrix can be built before the first testing spiral, but needs to be maintained during subsequent spirals. As functions (or program units) are modified during a development spiral, existing or new test cases need to be created and checked in the retest matrix in preparation for the next test spiral. Over time with subsequent spirals, some functions (or program units) may be stable with no recent modifications. Selective removal of their check entries should be considered, and undertaken between testing spirals.

Task 2: Execute the New Acceptance Tests

The purpose of this task is to execute new tests that were created at the end of the previous acceptance test cycle. In the previous spiral, the testing team updated the function/GUI, system fragment, and acceptance tests in preparation for the current testing spiral. During this task, those tests are executed.

Task 3: Document the Acceptance Defects

During acceptance test execution, the results of the testing must be reported in the defect-tracking database. These defects are typically related to individual tests that have been conducted. However, variations to the formal test cases often uncover other defects. The objective of this task is to produce a complete record of the defects. If the execution step has been recorded properly, the defects have already been recorded on the defect-tracking database. If the defects are already recorded, the objective of this step becomes to collect and consolidate the defect information.

Tools can be used to consolidate and record defects, depending on the test execution methods. If the defects are recorded on paper, the consolidation involves collecting and organizing the papers. If the defects are recorded electronically, search features can easily locate duplicate defects.

Section 22: Summarize/Report Test Results

"Project Completion Checklist," can be used to confirm that all the key activities have been completed for the project.

Step 1: Perform Data Reduction

Task 1: Ensure All Tests Were Executed/Resolved

During this task, the test plans and logs are examined by the test team to verify that all tests were executed (see Figure 22.1). The team can usually do this by ensuring that all the tests are recorded on the activity log and examining the log to confirm that the tests have been completed. When there are defects that are still open and not resolved, they need to be prioritized and deployment workarounds need to be established.

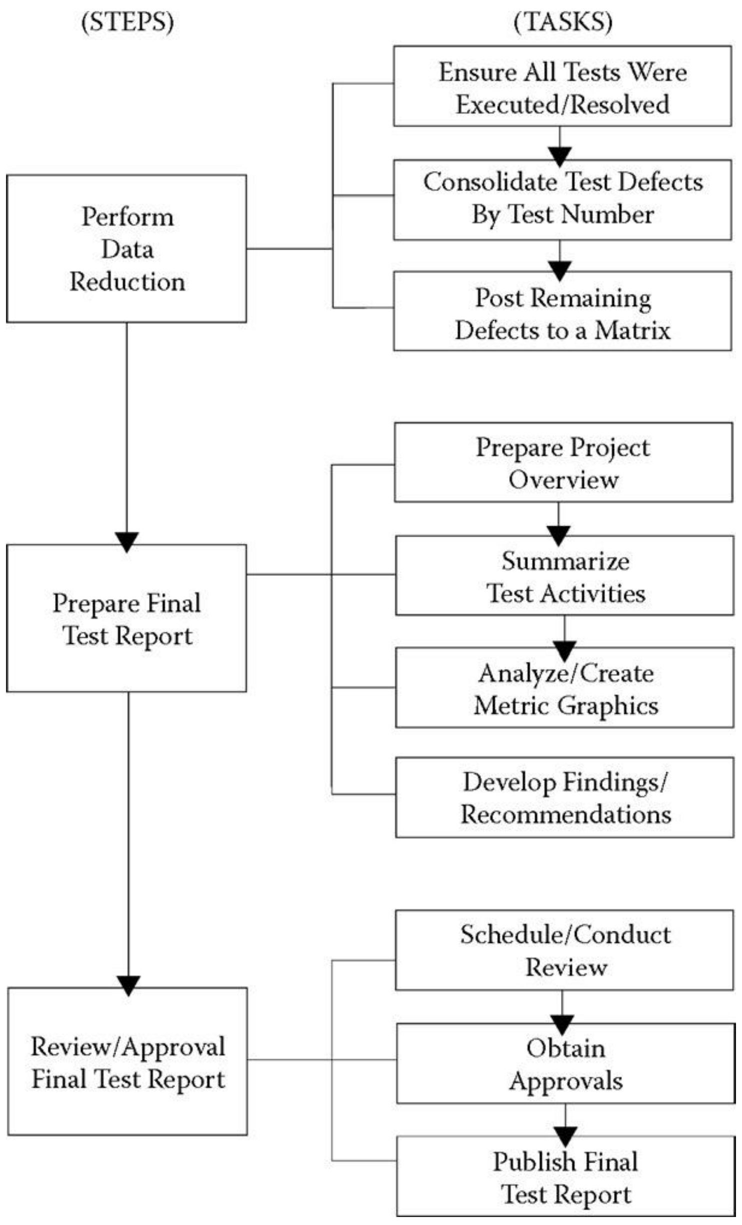
Task 2: Consolidate Test Defects by Test Number

During this task, the team examines the recorded test defects. If the tests have been properly performed, it is logical to assume that, unless a defect test document was reported, the correct or expected result was received. If that defect were not corrected, it would have been posted to the test defect log. The team can assume that all items are working except those recorded on the test log as having no corrective action or unsatisfactory corrective action. The test number should consolidate these defects so that they can be posted to the appropriate matrix.

Task 3: Post Remaining Defects to a Matrix

During this task, the uncorrected or unsatisfactorily corrected defects should be posted to a special function test matrix. The matrix indicates which test-by-test number tested which function. The defect is recorded in the intersection between the test and the functions for which that test occurred. All uncorrected defects should be posted to the function/test matrix intersection.

Figure 22.1: Summarize/report spiral test results


(STEPS)
(TASKS)
Perform Data Reduction
Ensure All Tests Were Executed/Resolved
Consolidate Test Defects By Test Number
Post Remaining Defects to a Matrix
Prepare Final Test Report
Prepare Project Overview
Summarize Test Activities
Analyze/Create Metric Graphics
Develop Findings/ Recommendations
Review/Approval Final Test Report
Schedule/Conduct Review
Obtain Approvals
Publish Final Test Report

Step 2: Prepare Final Test Report

The objective of the final spiral test report is to describe the results of the testing, including not only what works and what does not, from above, but the test team's evaluation regarding performance of the application when it is placed into production.

For some projects, informal reports are the practice, whereas in others, very formal reports are required. The following is a compromise between the two extremes to provide essential information not requiring an inordinate amount of preparation (see “Spiral Testing Summary Report”; also see “Final Test Summary Report,” which can be used as a final report of the test project with key findings).

Task 1: Prepare the Project Overview

An objective of this task is to document an overview of the project in paragraph format. Some pertinent information contained in the introduction includes the project name, project objectives, the type of system, the target audience, the organizational units that participated in the project, why the system was developed, what subsystems are involved, the major and subfunctions of the system, and what functions are out of scope and will not be implemented.

Task 2: Summarize the Test Activities

The objective of this task is to describe the test activities for the project including such information as the following:

- *Test team*—The composition of the test team, for example, test manager, test leader, and testers, and the contribution of each, such as test planning, test design, test development, and test execution.
- *Test environment*—Physical test facility, technology, testing tools, software, hardware, networks, testing libraries, and support software.
- *Types of tests*—Spiral (how many spirals), system testing (types of tests and how many), and acceptance testing (types of tests and how many).
- *Test schedule (major milestones)*—External and internal. External milestones are those events external to the project but that may have a direct impact on it. Internal milestones are the events within the project that can be controlled to some extent.
- *Test tools*—The testing tools used and their purpose, for example, path analysis, regression testing, load testing, and so on.

Task 3: Analyze/Create Metric Graphics

During this task, the defect and test management metrics measured during the project are gathered and analyzed. Defect tracking should be automated for greater productivity. Reports are run, and metric totals and trends are analyzed. This analysis will be instrumental in determining the quality of the system and its acceptability for use, and also will be useful for future testing endeavors. The final test report should include a series of metric graphics. The suggested graphics follow.

Defects by Function

Table 22.1 shows the number and percentage of defects discovered for each function or group. This analysis will flag the functions that have the most defects. Typically, such functions had poor requirements or design. In the following example, the reports had 43 percent of the total defects, which suggests an area that should be examined for maintainability after it is released for production.

Table 21.1: Defects Documented by Function

Function	*Number of Defects*	*Percentage of Total*
Order Processing		
Create new order	11	6
Fulfill order	5	3
Edit order	15	8
Delete order	9	5
Subtotal	40	22
Customer Processing		
Create new customer	6	3
Edit customer	0	0
Delete customer	10	6
Subtotal	16	9
Financial Processing		
Receive customer payment	0	0
Deposit payment	5	3
Pay vendor	9	5
Write a check	4	2
Display register	6	3
Subtotal	24	13
Inventory Processing		
Acquire vendor products	3	2
Maintain stock	7	4
Handle back orders	9	5
Audit inventory	0	0
Adjust product price	6	3
Subtotal	25	14
Reports		
Create order report	23	13
Create account receivable report	19	11
Create account payable report	35	19
Subtotal	77	43
Grand totals	182	100

Defects by Tester

Table 22.2 shows the number and percentage of defects discovered for each tester during the project. This analysis flags those testers who documented fewer than the expected number of defects. These statistics, however, should be used with care. A tester may have recorded fewer defects because the functional area tested may have relatively fewer defects, for example, tester Baker in Table 22.2. On the other hand, a tester who records a higher percentage of defects could be more productive, for example, tester Brown.

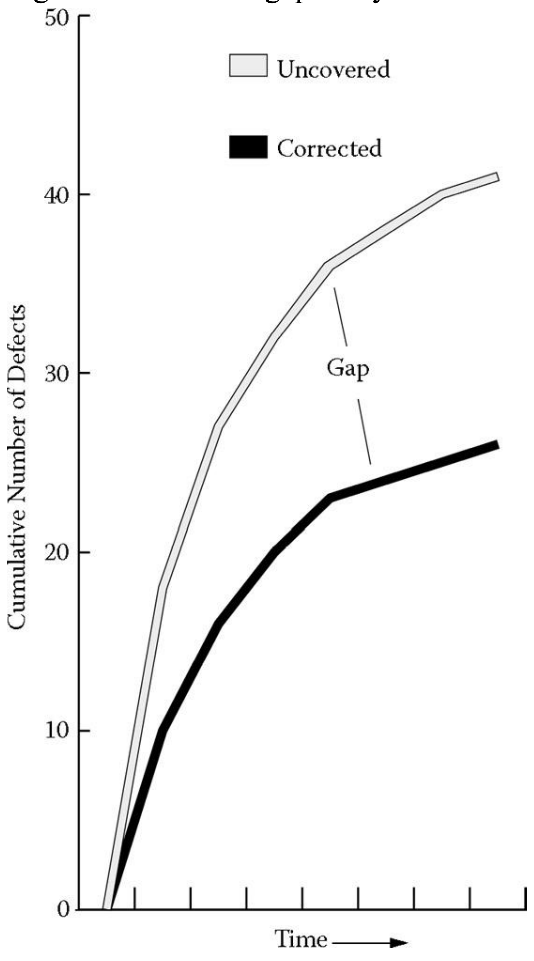
Table 22.2: Defects Documented by Function

Tester	***Number of Defects***	***Percent of Total***
Jones	51	28
Baker	19	11
Brown	112	61
Grand totals	182	100

Defect Gap Analysis

Figure 22.2 shows the gap between the number of defects that has been uncovered and the number that has been corrected during the entire project. At project completion, these curves should coincide, indicating that the majority of the defects uncovered have been corrected and the system is ready for production.

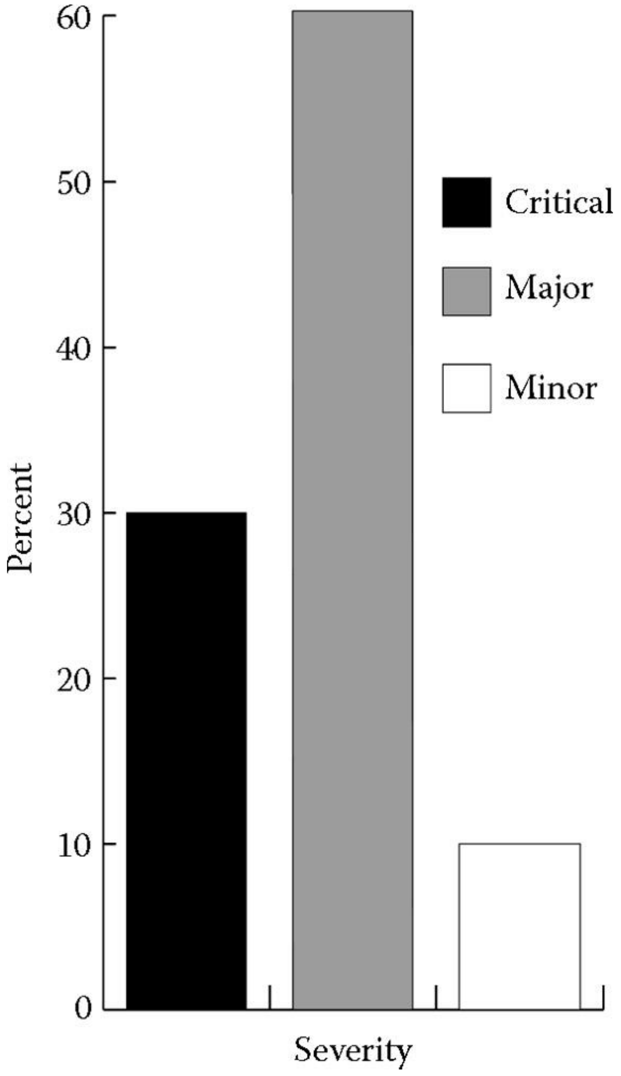
Figure 22.2: Defect gap analysis



Defect Severity Status

Figure 22.3 shows the distribution of the three severity categories for the entire project, for example, critical, major, and minor. A large percentage of defects in the critical category indicates that a problem existed with the design or architecture of the application that should be examined for maintainability after it is released for production.

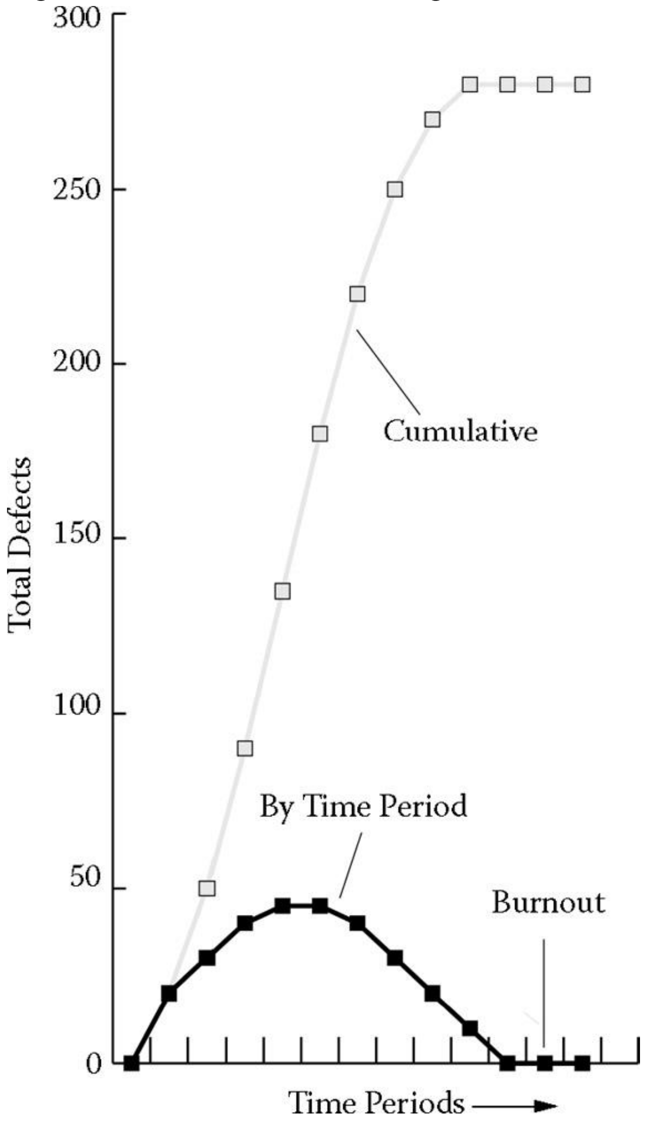
Figure 22.3: Defect severity status



Test Burnout Tracking

Figure 22.4 indicates the rate of uncovering defects for the entire project and is a valuable test completion indicator. The cumulative (e.g., running total) number of defects and defects by time period help predict when fewer and fewer defects are being discovered. This is indicated when the cumulative curve "bends" and the defects by time period approach zero.

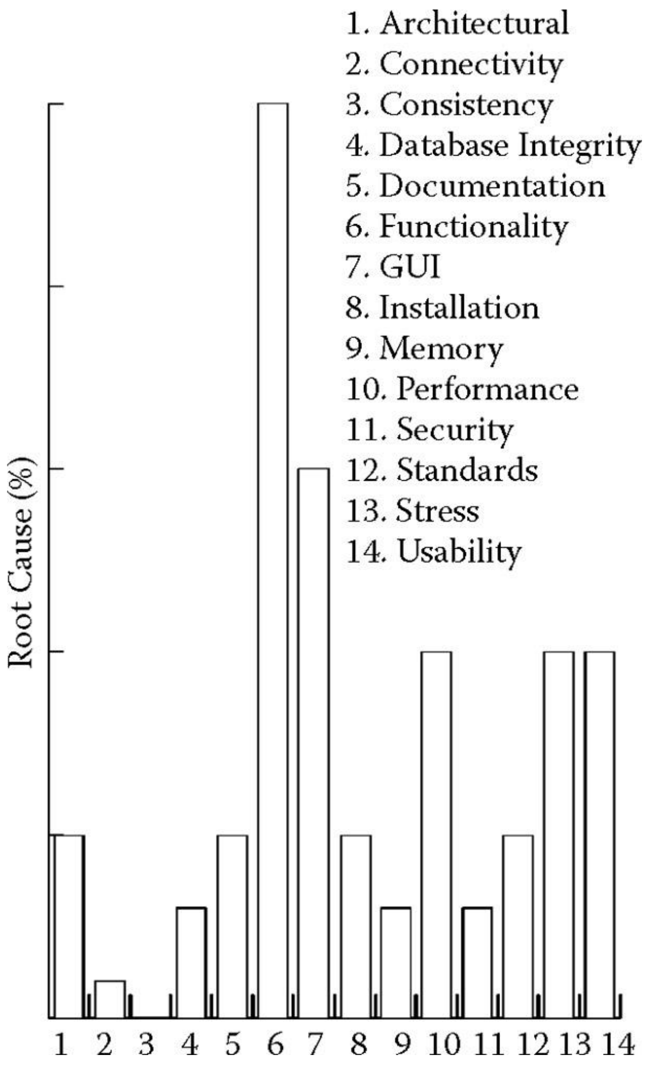
Figure 22.4: Test burnout tracking



Root Cause Analysis

Figure 22.5 shows the source of the defects, for example, architectural, functional, usability, and so on. If the majority of the defects are architectural, the entire system will be affected, and a great deal of redesign and rework will be required. High-percentage categories should be examined for maintainability after they are released for production.

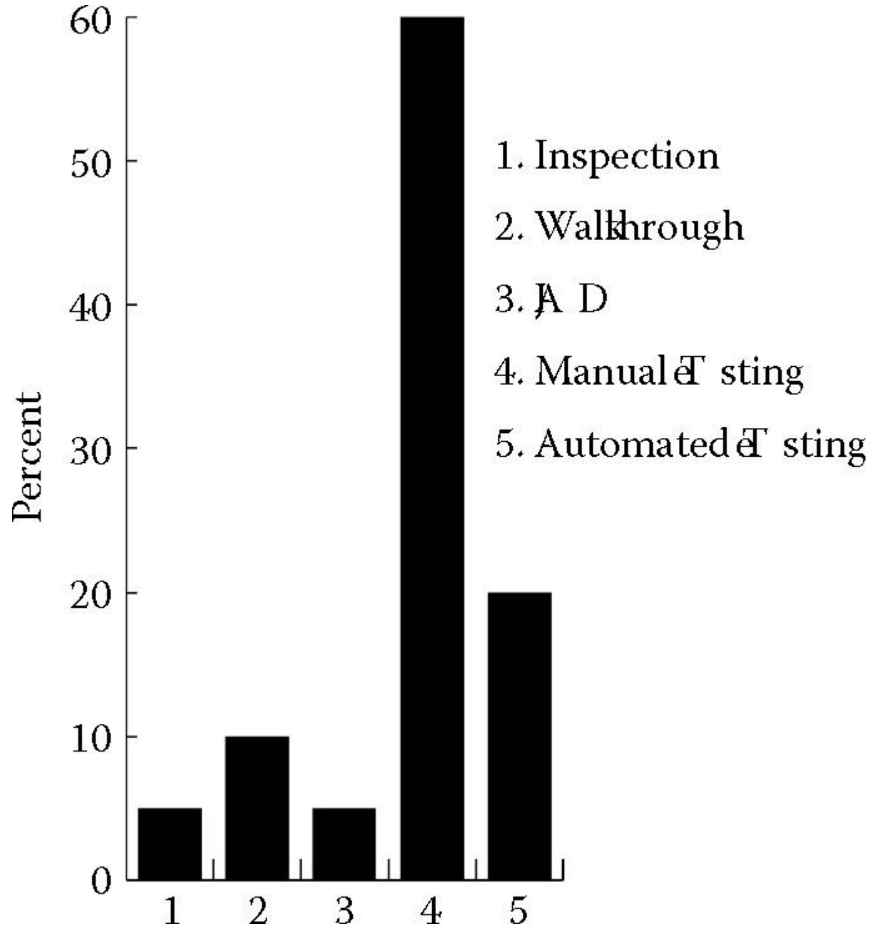
Figure 22.5: Root cause analysis



Defects by How Found

Figure 22.6 shows how the defects were discovered, for example, by external customers, manual testing, and the like. If a very low percentage of defects were discovered through inspections, walkthroughs, or JADs, this would indicate that there may be too much emphasis on testing and too little on the review process.

Figure 22.6: Defects by how found

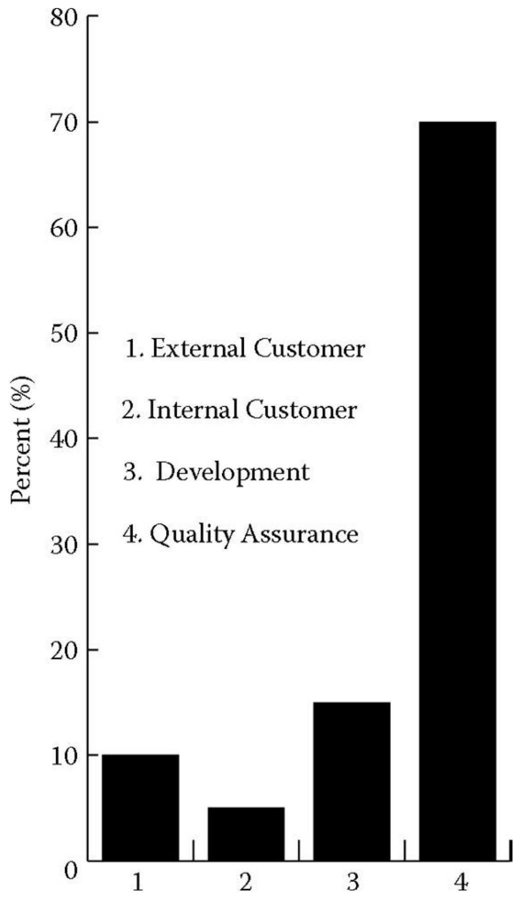


The percentage differences between manual and automated testing also illustrate the contribution of automated testing to the process.

Defects by Who Found

Figure 22.7 shows who discovered the defects, for example, external customers, development, quality assurance testing, and so on. For most projects, quality assurance testing will discover most of the defects. However, if external or internal customers discovered the majority of the defects, this would indicate that quality assurance testing was lacking.

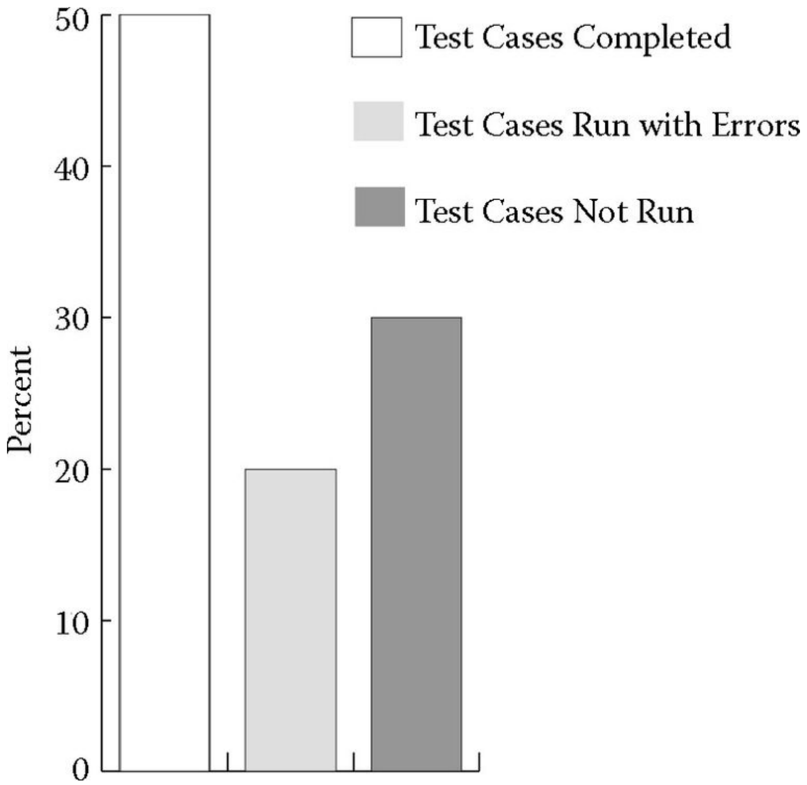
Figure 22.7: Defects by who found



Functions Tested and Not Tested

Figure 22.8 shows the final status of testing and verifies that all or most defects have been corrected and the system is ready for production. At the end of the project, all test cases should have been completed and the percentage of test cases run with errors and not run should be zero. Exceptions should be evaluated by management and documented.

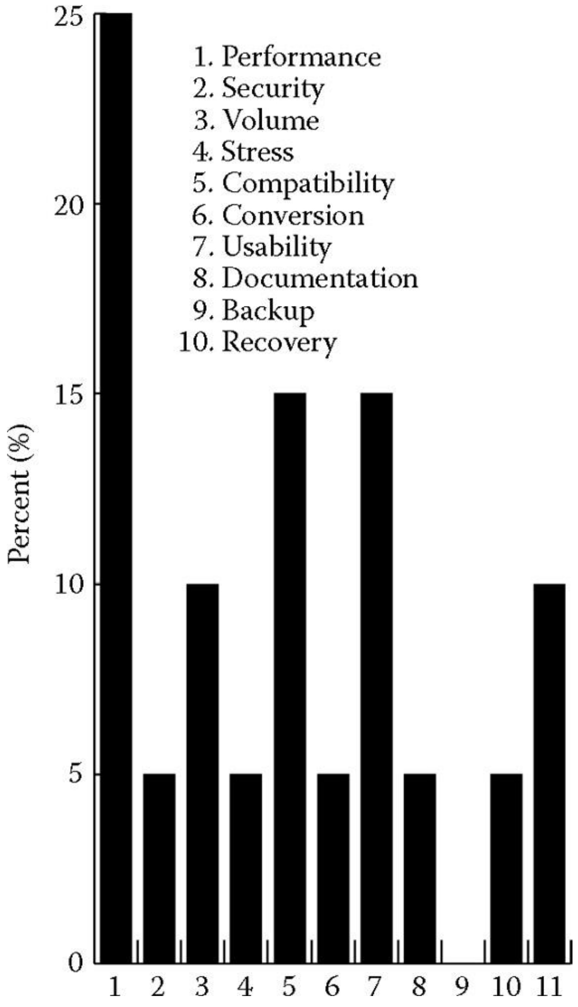
Figure 22.8: Functions tested/not tested



System Testing Defect Types

Systems testing consists of one or more tests that are based on the original objectives of the system. Figure 22.9 shows a distribution of defects by system testing type. In the example, performance testing had the most defects, followed by compatibility and usability. An unusually high percentage of performance tests indicates a poorly designed system.

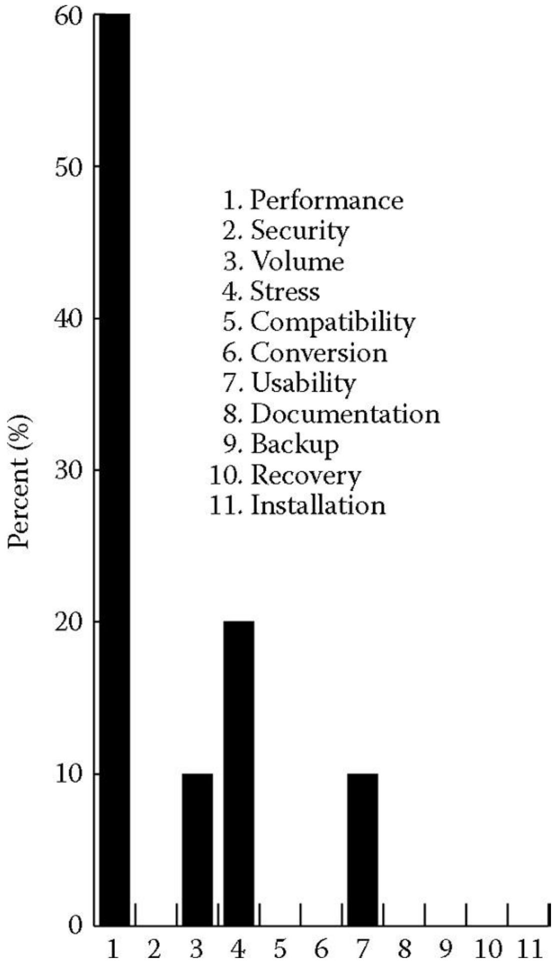
Figure 22.9: System testing by root cause



Acceptance Testing Defect Types
Acceptance testing is an optional user-run test that demonstrates the ability of the application to meet the user's requirements. The motivation for this test is to positive rather than negative, for example, to show that the system works. Less emphasis is placed on the technical issues, and more is placed on the question of whether the system is a good business fit for the end user.

There should not be many defects discovered during acceptance testing, as most of them should have been corrected during system testing. In Figure 22.10, performance testing still had the most defects, followed by stress and volume testing.

Figure 22.10: Acceptance testing by root cause



Task 4: Develop Findings/Recommendations

A finding is a discrepancy between what is and what should be. A recommendation is a suggestion on how to correct a problem or improve a system. Findings and recommendations from the test team constitute most of the test report.

The objective of this task is to develop the findings and recommendations from the testing process and document "lessons learned." Previously, data reduction has identified the findings, but they must be put in a format suitable for use by the project team and management.

The test team should make the recommendations to correct a situation. The project team should also confirm that the findings are correct and the recommendations reasonable. Each finding and recommendation can be documented in the Finding/ Recommendation matrix depicted in Table 22.3.

Table 21.3: Defects Documented by Function

Finding Description[a]	***Business Function***[b]	***Impact***[c]	***Impact on Other Systems***[d]	***Costs to Correct***[e]	***Recommendation***[f]
Not enough testers were initially assigned to the project	N/A	Caused the testing process to lag behind the original schedule	N/A	Contracted five additional testers from a contract agency	Perform more resource planning in future projects
Defect tracking was not monitored adequately by development	N/A	Number of outstanding defects grew significantly	N/A	Authorized overtime for development	QA needs to stress the importance of defect tracking on a daily basis in future projects
Automated testing tools did contribute significantly to regression testing	N/A	Increased testing productivity	N/A	N/A	Utilize testing tools as much as possible
Excessive number of defects in one functional area	Reports	Caused a lot of developer rework time	N/A	Excessive developer overtime	Perform more technical design reviews early in the project
Functional area not compatible with other systems	Order Processing	Rework costs	Had to redesign the database	Contracted an Oracle database DBA	Perform more database design reviews early in the project
30 percent of defects had critical severity	N/A	Significantly impacted the development and testing effort	N/A	Hired additional development programmers	Perform more technical reviews early in the project and tighten up on the sign-off procedures
Function/GUI had the most defects	N/A	Required a lot of rework	N/A	Testers authorized overtime	Perform more technical reviews early in the project and tighten up on the sign-off procedures
Two test cases could not be completed because performance load test tool did not work properly	Stress testing order entry with 1000 terminals	Cannot guarantee system will perform adequately under extreme load conditions	N/A	Delay system delivery until new testing tool acquired (2 months delay at $85,000 loss in revenue, $10,000 for tool)	Loss of revenue overshadows risk. Ship system but acquire performance test tool and complete stress test

[a] **This includes a description of the problem found from the defect information recorded in the defect-tracking database. It could also include test team, test procedures, or test environment findings and recommendations**

Table 21.3: Defects Documented by Function

Finding Description[a]	*Business Function*[b]	*Impact*[c]	*Impact on Other Systems*[d]	*Costs to Correct*[e]	*Recommendation*[f]

[b] **Describes the business function that was involved and affected.**
[c] **Describes the effect the finding will have on the operational system. The impact should be described only as major (the defect would cause the application system to produce incorrect results) or minor (the system is incorrect, but the results will be correct).**
[d] **Describes where the finding will affect application systems other than the one being tested. If the finding affects other development teams, they should be involved in the decision on whether to correct the problem.**
[e] **Management must know both the costs and the benefits before it can make a decision on whether to install the system without the problem being corrected.**
[f] **Describes the recommendation from the test team on what action to take.**

Step 3: Review/Approve the Final Test Report

Task 1: Schedule/Conduct the Review

The test summary report review should be scheduled well in advance of the actual review, and the participants should have the latest copy of the test plan.

As with any interview or review, there are certain common elements. The first is defining what will be discussed; the second is discussing the details; the third is summarization; and the final element is timeliness. The reviewer should state up front the estimated duration of the review and set the ground rule that if time expires before completing all items on the agenda, a follow-on review will be scheduled.

The purpose of this task is for development and the project sponsor to agree and accept the test report. If there are any suggested changes to the report during the review, they should be incorporated.

Task 2: Obtain Approvals

Approval is critical in a testing effort, because it helps provide the necessary agreement among testing, development, and the sponsor. The best approach is with a formal sign-off procedure of a test plan. If this is the case, use the management approval sign-off forms. However, if a formal agreement procedure is not in place, send a memo to each key participant, including at least the project manager, development manager, and sponsor. In the document, attach the latest test plan and point out that all their feedback comments have been incorporated and that if you do not hear from them, it is assumed that they agree with the plan. Finally, indicate that in a spiral development environment, the test plan will evolve with each iteration but that you will include them in any modification.

Task 3: Publish the Final Test Report
The test report is finalized with the suggestions from the review and distributed to the appropriate parties. The purpose has short- and long-term objectives.

The short-term objective is to provide information to the software user to determine if the system is ready for production. It also provides information about outstanding issues, including testing not completed or outstanding problems, and recommendations.

The long-term objectives are to provide information to the project regarding how it was managed and developed from a quality point of view. The project can use the report to trace problems if the system malfunctions in production, for example, defect-prone functions that had the most errors and the ones that were not corrected. The project and organization also have the opportunity to learn from the current project. A determination of which development, project management, and testing procedures worked, and which did not work or need improvement, can be invaluable for future projects.