Chapter 11

Evident Evil—The Art of Testing

This chapter examines software testing. Software testing constitutes one of the major aspects of software development. It involves investigating software to discover its defects, which come in two general types. One type of defect arises when you have not developed your software according to specification. Another type of defect arises when you have developed software so that you add things to it that the specifications have not anticipated. From one perspective, your game fails to do what you expect it to do. From the other perspective, your game does things that you do not expect or want it to do. To eliminate defects, you subject your game to a variety of tests. Among these tests are those involving components, integration, and the system as a whole. Testing presents an enormous field of practice and study, but the following topics provide a good starting place:

- Identifying what counts as a defect
- Formalizing testing
- Creating documents to guide testing
- The types of domain knowledge that testing involves
- Using test reports and test templates
- Recognizing and guarding against coverage risks
- Efforts and effects of testing
- Different testing roles

Basics of Testing

Software testers explore software to discover whether it contains defects. To understand the implications of this assertion, consider what it means to find a defect. What does it mean to say that you think a program is defective? Perhaps it means that the program does not do what you *expect* it to do. But then what does it mean to expect something? One answer to this is that when you have expectations, you have been told or have intuitively arrived at an understanding that the software should behave in certain ways. When software does not behave as you expect it to, something is wrong. The software is defective.

Some types of misbehavior are obvious. Others are not. You are playing a game, and suddenly it freezes up and you have to reboot your computer manually. You try again and the same thing happens. This counts as a defect that almost anyone can recognize intuitively. On the other hand, you play the game and everything seems to be okay. However, even though you know you have landed a killer blow on Manfred, the Barbarous, his strength inexplicably increases, defying the logic, intent, and objectives that govern every other aspect of the game.

This type of defect might be something you grasp intuitively, but it is more difficult to understand precisely how to address the problem—especially if you are not a veteran game player or a programmer. Some players might try to rationalize the odd behavior. The rationalization might go along the following lines:

"Oh, well, on this level, when Manfred, the Barbarous attacks you, let him beat you to a pulp and *don't* hit him. His power will diminish to nothing, and *he'll* die. If you hit him, he'll become more powerful, and then *you'll* die. On the next level, however, if you do this, Manfred will pulverize you. This is a subtle part of the game known only to the elite players."

If you assume that the programmers specified Manfred's behavior as consistent throughout the game, you can assume that such behavior is not an intended, expected part of the game, not even for elite players. Such behavior is accidental, unexpected, and defective.

The defect that the tester searches for has been inserted into the game through the development effort and represents something that must be eliminated. The defect does not result so much from the requirements or the design (assuming they are consistent) as it does from faulty implementation. Manfred, like all characters, is supposed to inflict damage when he successfully strikes a blow. Like all characters, he is supposed to suffer a loss of health when struck. A software defect makes possible the reversal of this behavior.

As Figure 11.1 illustrates, the software tester works from the software specifications and creates a test case. The test case establishes a way that the tester can work from the specifications to test the behavior of the software within a strictly defined context. Using information derived from the specifications, the tester defines initial and resulting conditions for the test case. If the specifications stipulate that Manfred should lose health when struck, and the tester discovers that Manfred instead gains health when struck, the software fails the test case. When the software fails the test case, the tester forwards a defect report to the maintenance programmer.

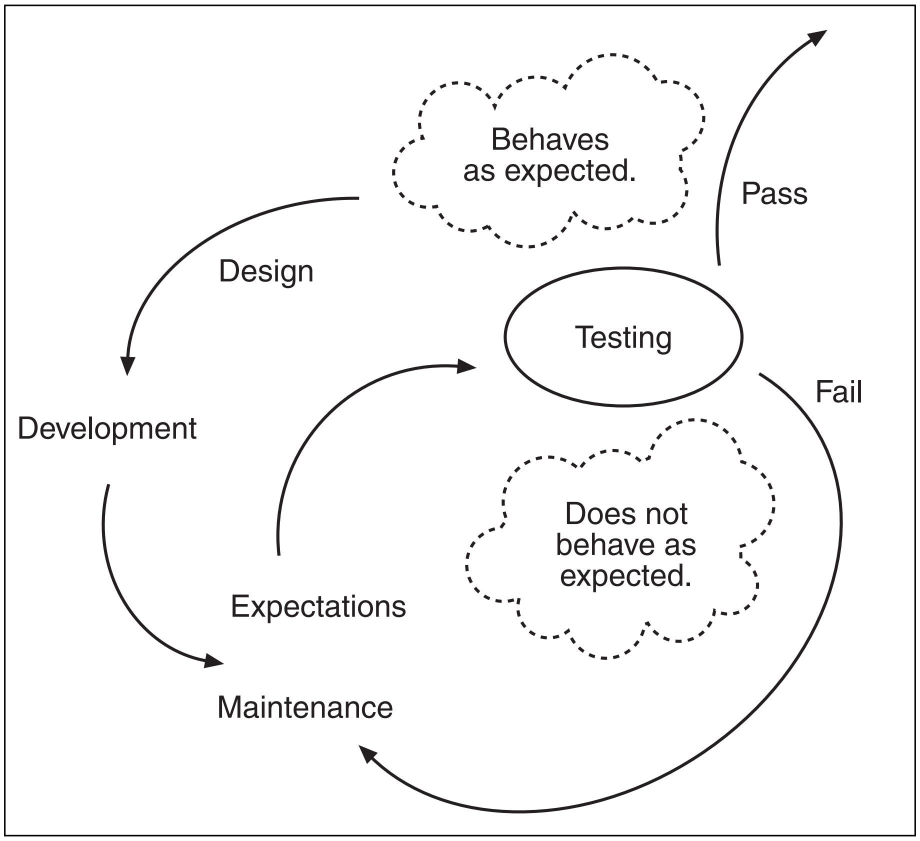


Figure 11.1
A defect is an implanted or created form of misbehavior.

Software testers differ from programmers because they detect defects instead of altering the code to eliminate them. Testers apply formalized test cases to the software, and based on the results of the test cases, they forward reports of defects to programmers. Working with the information that the testers provide, programmers eliminate the defects. Testers then verify that the programmers have eliminated the defects.

Three Concepts

Figure 11.2 illustrates three concepts that are central to testing. Although testing involves many other things, understanding what verification, validation, and exploration entail helps you grasp the basic reasoning that motivates the development of different approaches to testing.

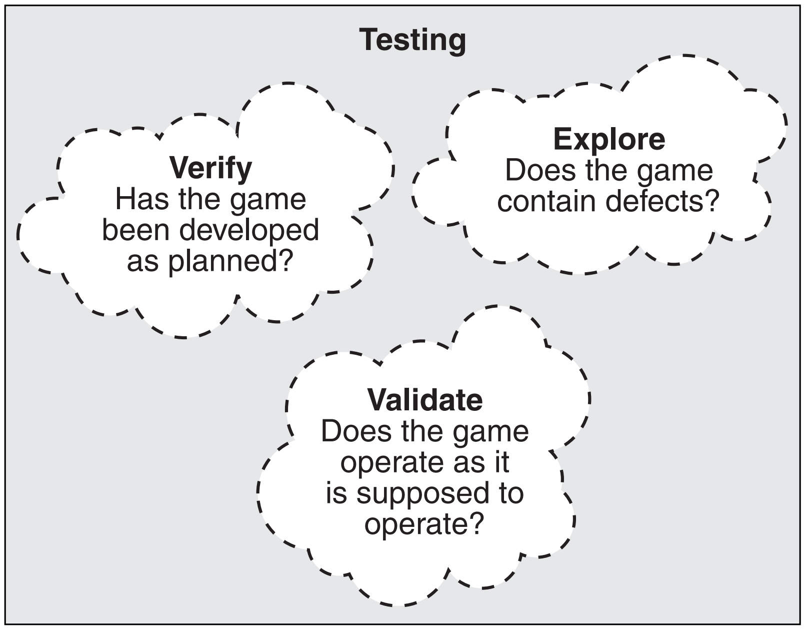

Testing
Verify
Has the game been developed as planned?
Explore
Does the game contain defects?
Validate
Does the game operate as it is supposed to operate?

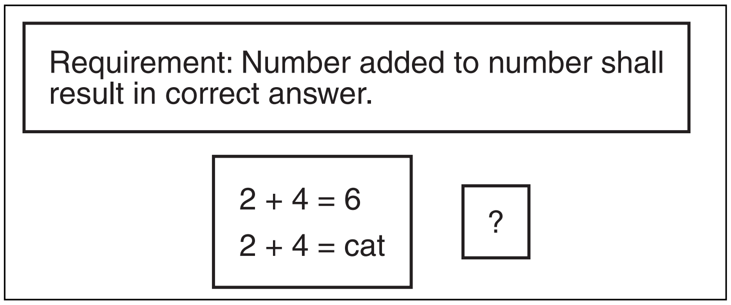
Figure 11.2
You can view testing as verification, validation, and exploration.

Verification

Before developers plan a software product—a game—they specify it in one form or another. In this book, a set of requirements specifies the game. Requirements consist of a set of sentences, a set of use cases, or both. They establish how the game is expected to behave. When you test whether the game satisfies the requirements, you verify that the developers have developed the product in accordance with the requirements.

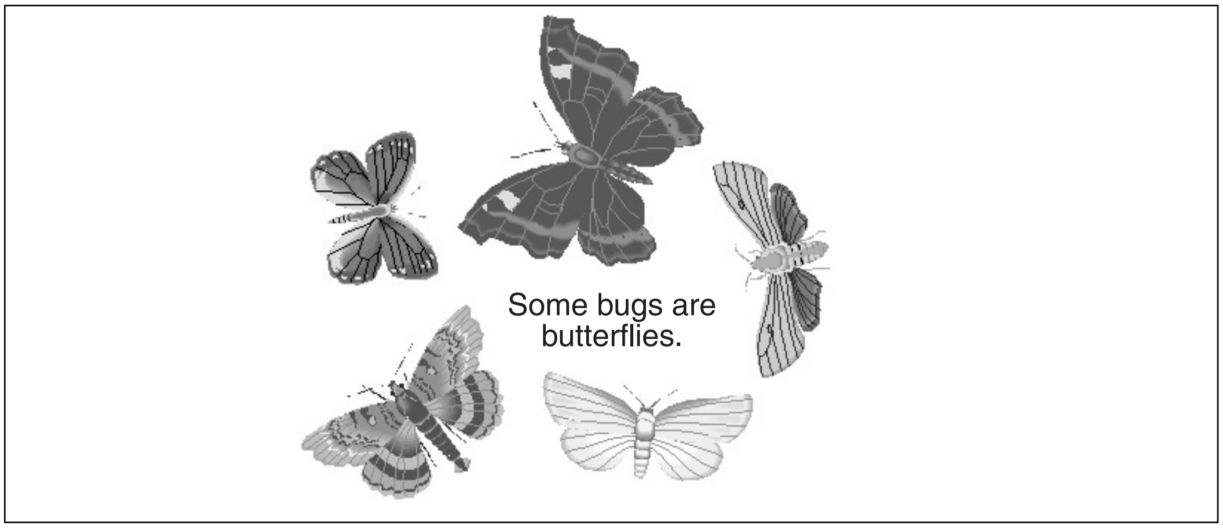
Validation

Even if developers develop a game so that it ostensibly looks like it satisfies the requirements, it still might not. Consider, for example, a requirement that two numbers shall be added and a correct answer shall result. Suppose that 2 and 4 are added, and the result is "cat." To validate this situation, the tester first must determine the appropriate answers. (In this case, assume that the requirements really mean numbers.) It might be that in a metaphysical dimension beyond the mundane, 2 and 4 really do make "cat." But the tester knows that only numbers are expected. Using this information, the tester designs a validation test. If the numbers result in numbers and if the result is correct according to standard arithmetical reasoning, the implemented software possesses validity.


Requirement: Number added to number shall result in correct answer.
2 + 4 = 6
2 + 4 = cat
?

Exploration

Testing attempts to find defects that the development process has inserted into the software product. Validity and verification offer the most direct ways to discover defects, but testers have other approaches to testing. These tests involve not so much what is supposed to be implemented (which, again, you can determine if you investigate the requirements and the design of the software) but what is *not* supposed to be implemented. When you investigate what is not supposed to be implemented, you enter into an exploratory realm of testing. In some realms of testing, bugs become butterflies.



Bugs become butterflies because exploratory testing involves finding out whether the software still behaves as specified when it is subjected to demands that it behave in ways that are not specified. Testers figure out how to break things. Breaking software involves getting it to behave in ways that are not specified. For example, in *Ankh*, a field in the character designer allows the player to enter an arbitrary name for a character. Normally, players would type a name, such as "Babba," "Rah," or "Slick." But what happens if someone decides to type a series of spaces? Discovering the answer requires exploratory testing. If the system accepts spaces, it might still perform perfectly okay. At a later moment of play, however, the player would see a blank line in the character selection list. When you click the blank line, the character that is named using the spaces appears. Is this how you want the game to behave?

Formalized Testing

The story of how you formalize testing begins with the story of how you test without formalization. When you test without formalization, you conduct what is known as *reactionary*, or find-and-fix, testing. A great deal of programming falls under this heading. A programmer types a few lines of code, tries to compile or build, and observes whether the compiler detects a syntax error.

When a game involves tens or hundreds of thousands of lines of complex code, reactionary programming no longer works. Instead, testing must take place according to an overriding plan. Testers formulate tests that cover the entire scope of the product and investigate as many logical or operational pathways through the product as possible.

Designing tests involves considering phases of development. Software begins with some type of specification. It takes form through design. It is implemented, deployed, and maintained. A type of testing corresponds to each of these phases. For example, you can conduct behavioral, or functional, testing to verify that developers have implemented the requirements. You can conduct integration testing to confirm that the modules or components constructed to satisfy the requirements can be merged together. When you conduct system testing, you can take the system and observe whether it can be installed and operated on different operating systems of computers.

As Figure 11.3 illustrates, you can encapsulate your testing activity in a relatively small number of formal tests. Many more tests with exotic names can supplement these basic forms of testing, but the core tests remain the starting point of almost any comprehensive testing effort.

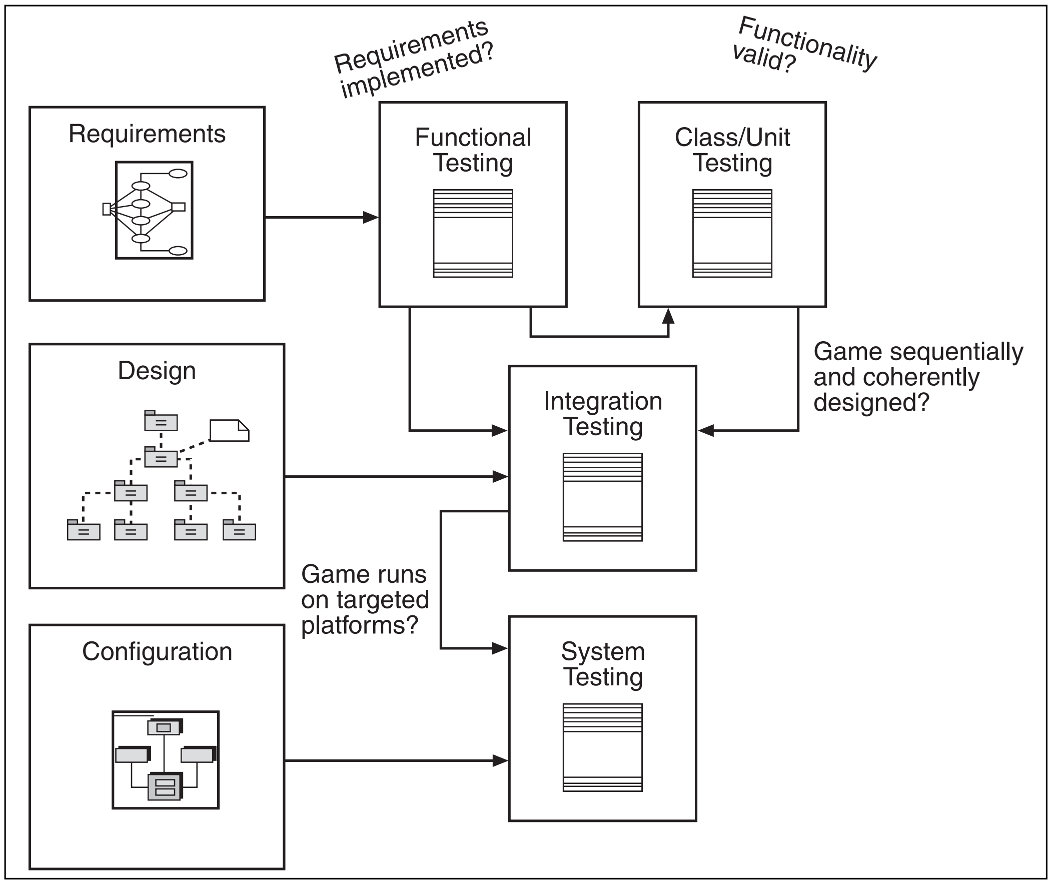


Figure 11.3
Test plans reflect development phases.

Approaches to Analysis

Games differ in at least one respect from other forms of software because they rarely lack some type of preliminary documentation. On the other hand, games resemble most other forms of software because the documentation on which they are based often lacks detail. When you formally test a game software product, you might have to spend a great deal of time analyzing the game software so that you know how to test it.

General Domain

Computer games, generally, constitute a domain of software. This domain comprises game genres, character types, learning scenarios, and a multitude of other items. As a game software tester, you apply knowledge of these items in a number of ways. Among many possibilities, for example, you formulate test scenarios, plan test strategies, and understand how much of the behavior of a game the test plan should cover. Without domain knowledge, it might not be easy to determine how much to test the scope of a game's behavior or what level of detail specific test cases require.

Domain knowledge extends in every direction and excludes nothing related to game development. Figure 11.4 illustrates a few common terms that are associated with game design.

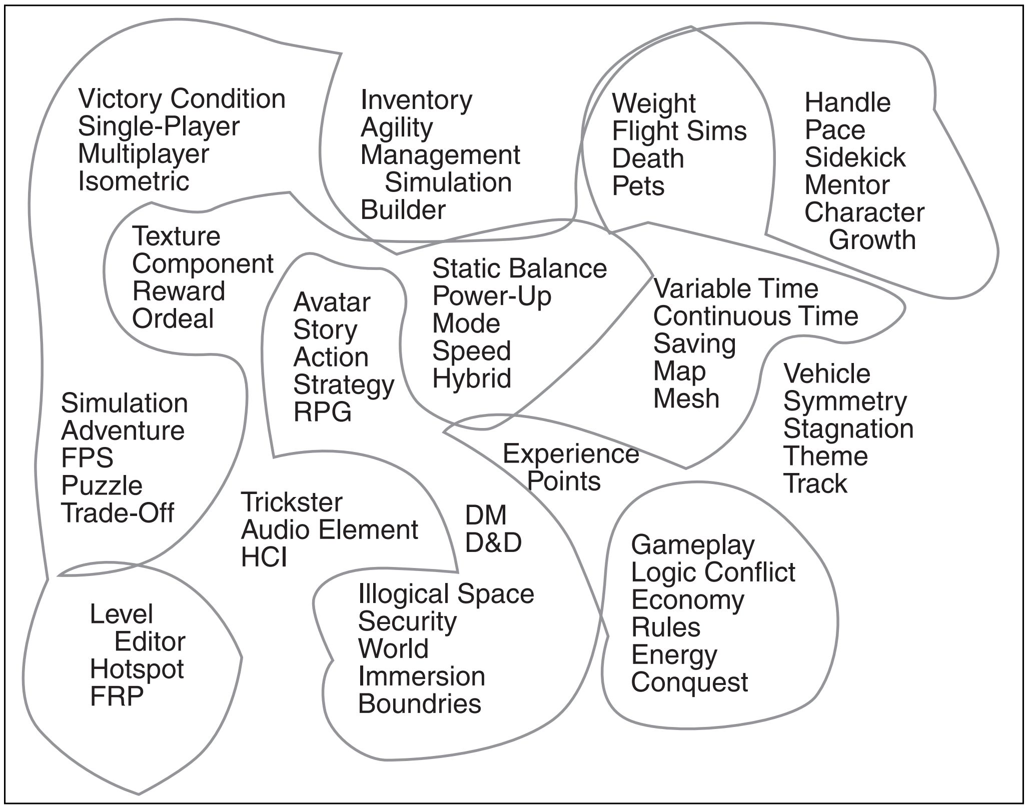


Figure 11.4
Domain knowledge furnishes a context in which to scope the testing effort.

General game development domain knowledge provides you with insights that help with the following tasks, among others:

- Understanding the genre of the game so that you can focus testing to cover the behavior that is most crucial to your game's genre
- Testing requirements specifications
- Estimating the performance standards that your game should achieve in light of benchmarks derived from knowledge of other games
- Profiling the operational characteristics of the game to determine whether they are on par with those of other games
- Being able to communicate readily with both game designers and game software developers during the testing effort

Application

You can consider application knowledge a subset of the more generalized domain of computer game knowledge. When you address application concerns, your attention shifts to a context that relates to the features of the game that express the current or forefront technologies that games embody. In recent history, among these are such items as particles, shadows, wavering motion effects, bump maps, refinements in collision detection, multiplayer voice interactions, databases, and security extensions. In addition to current and forefront technologies, games embody standard game application technologies. Among these are dialog boxes, buttons, sliders, console layouts, help options, and navigation components. Application domain knowledge enables you to test the game more effectively in some of the following ways:

- Designing test cases for verification and validation of requirements
- Understanding what parts of the game represent new techniques and deserve special attention
- Making use of standard testing artifacts to address standard game features

Architectural

Games also comprise architectural components. You can identify these components in numerous ways. From a design perspective, you work with modules. You can test modules for encapsulation, coupling, and complexity. The UML provides tools that you can use to depict architectural features. Among these are class, sequence, and package diagrams. (See Chapter 3, "A Tutorial: UML and Object-Oriented Programming," for discussion of these tools.) Testing that is related to architectural issues can involve both implemented code and design documents. Architectural domain knowledge helps you with some of the following activities:

- Understanding how to structure the integration test plan
- Knowing what test cases you can use from the design documentation as the basis of the integration and system test plans
- Establishing test criteria for test cases involving modules

Detail (Class)

Detailed design encompasses the development of functions and classes. You can consider global functions as utility objects that you can test much as you might test a class object. Classes involve interfaces, operations (which C++ programmers call member functions), attributes (also known as class variables), accessor and mutator operations, and a variety of other features that are specific to programming tasks. Knowledge of the detail domain helps you to conduct structural or white-box testing. Such testing requires that you work with the code. Among the tasks that such detail knowledge helps you perform are the following:

- Creation of test harnesses for classes and functions
- Insertion of static and other functions into classes to audit object states
- Writing standalone programs that you can use to test specific instances of objects
- Working with test tools like those from the Boost C++ testing library

Maintenance

Chapter 17, "Release Planning and Management," discusses testing as it relates to software after release. Just before the release of the game, testers face an intensive effort to guide the game through acceptance testing. Acceptance testing in general software efforts focuses on the customer. This is true to some extent in the game world, too, but it is more accurate to say that acceptance testing in the game world consists of formal measures that testers implement.

As an example of the challenges that you face when you test a game for release, consider a scenario in which your group has targeted Microsoft as a customer. Microsoft has established an extensive list of criteria that games must meet if they are to be included in its game suite. At last count, this list of criteria stretched to more than 100 pages. (For a humorous review of the list, see Mike McShaffry's book, *Game Coding Complete*. A reference appears at the end of this chapter.) Following are some paraphrases of a few of the test conditions for Microsoft:

- The application performs primary functions and remains stable during functionality testing.
- The application does not create temporary folders or place files in the wrong locations.

- The application remains stable when it processes a long file name.
- The application remains stable when it is directed to use unavailable devices.

After a game has been released, its characteristics as a software product change. For one thing, a game's life does not usually include addition of new features and functionality. Perfection of its existing features and functionality defines the focus of the programming and testing efforts exerted toward it. Generally, this phase of a software product's life is anything but trivial. On the average, 60 percent of the expense that is involved in a software product accrues after its release. Domain knowledge of maintenance efforts enables you to augment your efforts in the following ways:

- Understanding how to develop test procedures that address user-oriented issues.
- Working with a trouble tracking system.
- Verifying and validating software fixes on a revolving basis. In other words, you might have to verify first that a bug that a user reports is valid and then, following the efforts of the maintenance programmer, verify that the bug has been fixed.

The "V" Model

You can align testing activities with the software development cycle. One approach to this is the "V" model. The "V" model starts with requirements and moves downward, on the left leg of a "V," through design and detail design. Then it begins upward, on the right of a "V," with implementation of classes, functions, components, and finally the whole system. Figure 11.5 illustrates the flow of activities related to the domains discussed previously.

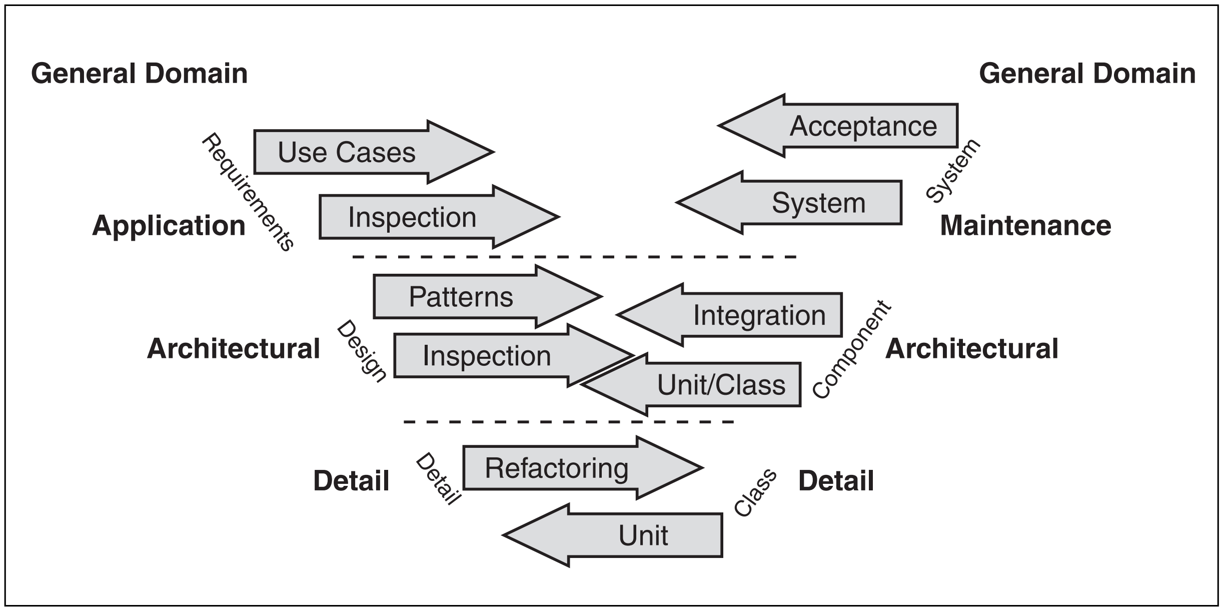


Figure 11.5
You can depict testing activities as a V that emphasizes design on one leg and development on the other.

Ways to Test

Testing involves more than playing the game. Playing the game constitutes a significant chunk of the activity that is involved in testing; however, the tests that you create often focus on implementation rather than on gameplay. Whereas you might view the experience of playing the game as the big picture, working at the software engineering level, you concentrate on an atomized view of the game that encompasses searching for defects in how the behavior has been implemented. For this reason, you might tend to view the game as a system that consists of components, modules, and performance characteristics, not as something you play.

note

Playability and *usability* testing differ from testing that focuses on classes, integration, components, and systems. Playability and usability tests directly address how players of the game interact with the game's visible features. They also address the aesthetic qualities of the game. If you work as a software engineer or software developer, such testing is definitely your concern, but you might be able to accomplish your work much more readily by structuring your test plans so that expert (or other) playability and usability testers are brought into the picture. You might view your job as structuring and supervising such tests. Others follow test cases that you set up for them.

Inspection

Inspection involves examining items that are usually placed in the category of documentation. You use inspection as your primary mode of testing when you have the luxury of beginning your testing work at the beginning of a project. This is a good way to do things, but unfortunately, many development groups do not bother to retain testers until relatively late in the project. Here are some focal points of inspections:

- **Requirements.** Participate in the development of requirements. Test the requirements to discover whether they are redundant, poorly worded, or fail to address the functionality of the game.
- **Use cases.** Assist with the development of use cases for requirements. Determine whether the use cases address requirements adequately. Verify that the trigger and ending conditions have been included. If a use case is excessively long, suggest that it be broken down into two or more use cases. You can begin creating test cases (or test procedures) using the use cases developed for requirements.
- **Design.** Inspect the software design description. If the development team employs UML diagrams, check the diagrams for accuracy and completeness.
- **Configuration.** Inspect the configuration management plan to determine whether the coding conventions, configuration policies, and other such items provide a comprehensive framework for development.

- **Documentation.** Test documents to ensure that they are identified and numbered correctly. If the versioning scheme creates confusion, report the problem.

In each case, you can test documentation and other inspected artifacts using the same tools that you use to inspect code. In other words, create a test procedure and follow it. If the procedure shows a problem, report it.

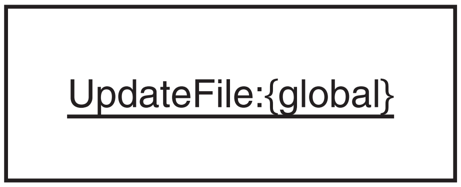
Unit or Class Testing

The traditional term for testing that involves examining the smallest components of the software product is *unit testing*. Unit testing most commonly identifies the activity of testers as they work along with the implementation effort. Unit testing can involve almost any part of the software product before it is integrated into the complete system. The focus of testing at this level is to examine behavior in isolation, before it is folded in with other behavior and rendered, as a result, complex.

Another term for unit testing is *class testing*. The terms are interchangeable because both relate to the smallest components in the software system. With object-oriented development efforts, the smallest unit is the class. In procedural development efforts, the smallest unit is the function. For the development effort for *Ankh*, almost all of the system consisted of classes. Still, in a few instances, global functions were used. Combining classes and libraries of functions characterizes many C++ development efforts because C++ does not require that programmers use only object-oriented constructions.

note

When you use UML to illustrate global functions, you can name the function as though it were an object and then comment it so that readers can understand its origin.



Unit testing involves a large set of tools, some of which are illustrated in this chapter. For now, an important point to emphasize is that unit (or class) testing offers the primary medium through which you can test functional requirements. Note the following:

- **Component test plan.** You can use the component test plan to detail most of the test procedures you intend to execute to verify and validate the implementation of the functionality that the requirements specify. With *Ankh*, the team began its component testing by examining use cases that were included in the *Ankh Software Requirements Specification.* Each use case provided a starting point for the creation of a test procedure for the requirement.

- **Black-box testing.** You can employ black box testing to test the behavior of components. A class object is a component that receives parameter information and returns a result. You can base a test case on these two points of interaction.
- **White-box testing.** State-transition diagrams illustrate the work of white-box testing. Such diagrams allow you to understand how the attributes of objects change as they perform their work. White-box testing offers a powerful analytical tool to evaluate the robustness of components at their most elemental level.

See "Using Templates" later in this chapter for more information about the tools you use to conduct unit or class testing.

Integration

Integration testing involves creating test procedures to detect defects that result when developers assemble the components created at the class level into modules. Interactions among several classes usually characterize the life of a module. When two objects exchange messages, the message that the first (client) object communicates to the second (server) object can cause problems. Among common problems are those involving improper instantiation of server objects. Improper instantiation of the server objects results in failure of the client object. The error you detect, however, might indicate only that the client fails. You must analyze the interactions of the assembled objects to determine the real source of the failure.

Planning integration testing requires that you understand the design of the software system. For this reason, you can most effectively create an integration testing plan after you have completed the software design specification. With the *Ankh* development effort, the team created a design that comprised 14 stripes (or modules). Planning the integration efforts began with developing use cases for each integration event. The use cases established behavioral scenarios that characterized expected system behavior following each integration. (See Figure 11.6.)

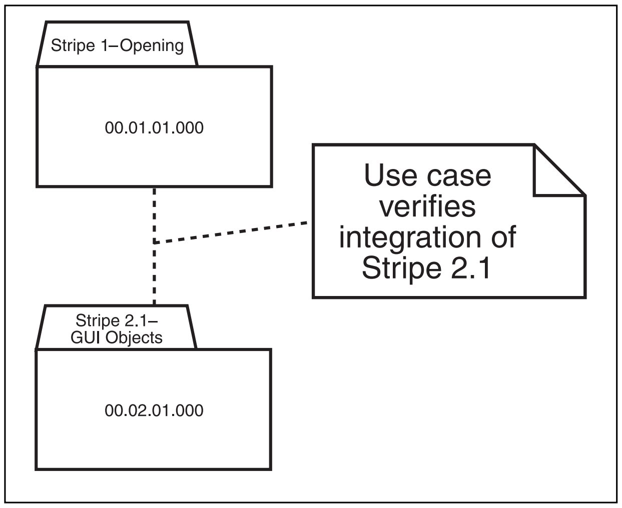


Figure 11.6
Use cases for integration provide conceptual frameworks from which to begin test planning.

Creating use cases to visualize integrations serves as a reliable way to ground a testing effort. Each integration use case provides a conceptual framework in which to understand what functionality should be newly visible as a result of the integrated stripe. You then can work from this understanding to test the specific behaviors that characterize this new situation. Using the collection of 14 integration use cases developed for the *Ankh Software Design Specification*, the *Ankh* development team easily generated the initial set of test procedures that composed the *Ankh Integration Test Plan*.

System

Consider what happens when a player installs a game. The installation succeeds if the player can begin playing the game immediately after the installation concludes. It fails if this is not the case. System testing involves examining the behavior of the software as a complete entity. Its goals consist largely of ensuring that the player knows the game only as an experience that begins and ends with what happens in the game.

The context of system testing encompasses using systems that represent those that you expect your customers to use. It involves assembling the completed software into an isolated entity. To test this entity effectively, you first should remove dependencies. To accomplish this, testers create a system test environment. Such an environment is usually a bare-bones operating system installed on a machine that you consider typical of your customers. In most development settings, the systems testing effort involves several environments that mirror those of anticipated customers.

A system test plan consists of specifications of the targeted operating systems and machines. It also provides test cases that require you to perform installations. On more detailed levels, it can establish criteria for stress tests. A stress test can involve, for instance, seeing what happens if the system on which you install your game has little available RAM. How does the game then perform? Another test involves system failure. If a power outage occurs, for example, how much data does your game lose?

As with component and integration tests, an effective approach to developing system tests involves creating use cases. You can use common installation scenarios or stress conditions as the basis of your use case development effort.

Planning Activities

When you plan test activities, you create test plans that address both different approaches to testing and different aspects of the system being tested. (See Figure 11.7.) You set these two priorities because no one type of testing suffices to test all aspects of the system. Likewise, the best way to test different aspects of the system is by using different approaches to testing. To confirm that the requirements have been met, for example, behavioral (black-box) testing allows you to set up tests based on the use cases in the software requirements specification. To investigate performance and complexity issues, you can use structural (white-box) testing. Test plans allow you to think through the approaches you want to use and establish how you want to analyze the system for testing.

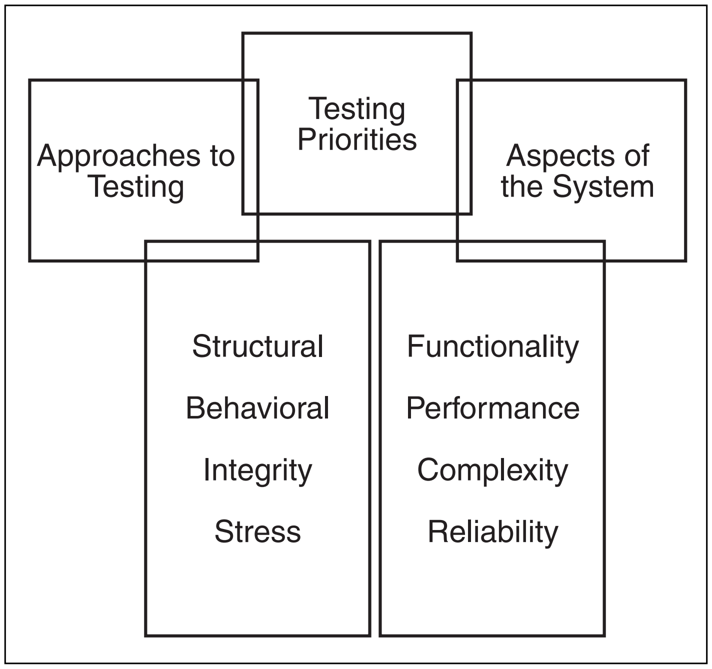


Figure 11.7
Test plans lay the groundwork for approaches to testing and different ways of analyzing the system.

Project Test Plan

You can quickly generate a project test plan if you use a template. IEEE standard 829 provides one of the most common test plan templates. (For information about IEEE standard 829, which provides one of the most common templates, refer to "Using Templates.") The project test plan provides a context in which you can plan the overall testing effort. Among the tasks it can help you complete are the following:

- **Specific plans.** Use the project test plan to determine what test plans you want to write. Figure 11.8 illustrates the plans created for *Ankh*. You might determine that other plans best suit your needs. For example, the *Ankh* development team decided not to create an acceptance test, but such a test would have been mandated had a game publisher been in the picture.
- **Testing targets.** When you work at the project scope, you can consider what aspects of the project possess the greatest importance. If you have developed a game intended to display graphically superior features, you might draft playability and usability plans that require you to subject these features to particularly harsh scrutiny. Any type of testing can be accommodated as long as you take time up front to plan your testing strategy.
- **Testing risks.** The primary risks that any testing effort faces usually encompass resources, schedule limitations, and granularity of testing. For example, if you do not test enough, the quality of the game declines. If you try to test too much, your testing effort almost inevitably fails to reach completion. If you lack testers or testing resources, even if you have planned well, you might fail to test the product adequately.

- **Testing priorities.** Priorities often reflect development cultures. If you work with a group that centers its efforts on performance, your primary testing effort might involve component test plans. Such plans often require construction of test harnesses designed to generate precise data to use for optimization. On the other hand, priorities often arise from scheduling and resource limitations, so you might find that you concentrate on the project test plan so that you can distribute limited efforts as effectively as possible.
- **Scope and intensity of testing.** A general view of the testing effort allows you to plan the degree of coverage you want to achieve. Only by segmenting, analyzing, and defining a specific testing agenda can you determine the scope and intensity of the testing effort.
- **Scheduling.** A project test plan often contains a detailed testing schedule. This almost always occurs when the testing effort resides in a group that works separately from the development group. When the development and testing efforts are merged, however, you can fold the testing activity into the project test plan. For the *Ankh* effort, testing efforts were not separate, so the project test plan referred the reader to the project development plan for information about the testing schedule.

Component/Class Test Plan

The component or class test plan often features test procedures accompanied by tables containing multitudes of test cases. In addition, you might find that this plan includes specifications for test harnesses and automated testing scripts. Such tools allow you to perform intensive structural testing at the component level. A component test plan often comprises the most exacting plan in the suite of plans you develop for a software product. It also can contain both behavior and structural test specifications. Among the tasks that this plan can help you perform are the following:

- **Designation of test cases.** For the *Ankh* development effort, the component test plan named test cases that corresponded fairly directly to the use cases named in the requirements specification. Further, the emphasis of the component test plan was on behavioral testing. Structural testing was reserved for a relatively few instances in which performance issues were estimated to be high risks.
- **Naming and specification of scripts.** Whenever an automated testing script is required, you can provide it as a supplement to a primary test procedure. You can assign each requirement a primary test procedure. The primary test procedure (which also can be referred to as a test case) can provide references to supplementary tables of test cases or separate test procedures.

- **Naming of classes to be tested.** Name the classes and the features of the classes that you want to test. To accomplish this, you identify initial and expected final states of the objects you investigate. To carry out the test, you might have to design a test harness that will set initial values and go from there.
- **Convert requirements use cases to test cases.** The component test plan provides a context by which you can convert use cases drawn from the requirements specification into test procedures (or test cases). The value of starting with use cases is that you work with material that you have already substantially refined, so most of the information you require, if not already provided, is at least clearly outlined.
- **Formalization of test reports.** Document every test so that you can record the result of the execution of the test in a convenient, self-evident fashion. One approach to this is to use a formalized test report. (See "Test Report Templates" later in this chapter.) In other instances, you can embed test cases in tables that allow you to include pass/fail results along with the cases.
- **Summarization of test results.** Summary test reports provide a view of the overall testing effort. Although it is important to establish that no test failed (or to explain why you waived a failed test), it also is important to be able to account for the percent of the overall planned testing activity you completed. You can summarize the percent of overall functionality that you estimate you covered.

Integration Test Plan

The integration test plan should provide a clear roadmap for how you intend to test the system as the developers construct it. As the developers assemble the modules (or stripes) of the system, the integration test plan should indicate how you intend to confirm that the assembly is acceptable. With the *Ankh* testing effort, the team reached this goal by creating test procedures based on the use cases that appeared in the design document. Among the tasks that an integration test plan can help you perform are the following:

- **Set specific criteria for pass/fail of integration.** When you set criteria for integration testing, consider the percent of functionality you want to see covered before you accept integration as successful. For example, you might establish that for each integration, the requirements included in the newly integrated module be tested individually to establish whether anomalies have developed or performance has declined.
- **Determine the depth of regression testing to be performed.** You can set up tables of test cases at this point to establish how much you want to test. If the testing for each integration involves following a scenario based on use cases, you should designate which use cases you want to use. These can become standard models for regression that you can use with each successive integration.

- **Establish the extent to which black-box and white-box testing are to be used.** The integration plan provides a context in which you can set general polices and practices regarding how you want to test integration. For example, you can say that you want to verify functionality only, or you can designate that you test for performance problems that might follow each integration.
- **Convert design use cases into integration test cases.** As with the component test plan, if you draw upon the use cases that you have developed during the design effort, you might find that a substantial portion of your work has already been done.

System Test Plan

You can use the system test plan to assemble the pass/fail criteria you want to assert against the overall system. Among the tasks that this plan can help you perform are the following:

- **Establish items to be installed.** You can draw on the configuration plan to determine which components, assets, and documents should be installed. Testing the installation, then, involves determining whether all the components, assets, and documents have been installed. It also involves determining whether anything other than these items has been installed.
- **Establish targeted environments**. You can designate the machines and systems you have targeted.
- **Performance.** You can designate what, specifically, you consider successful performance on the targeted system. One approach to this is to assign specific time values to tasks you think the system should be able to perform.

Acceptance Test Plan

An acceptance test plan allows you to name procedures and pass/fail criteria that represent the perspective of your customer. Among different approaches to acceptance testing are alpha and beta testing. Beta tests receive the greatest attention because they allow your customers to use your product in a friendly way. Friendliness involves providing feedback that helps you identify and repair lingering defects. Other types of acceptance testing involve exacting or legally stipulated items that you must confirm have been addressed to the satisfaction of the client. In such situations, you might find that you turn your software over to testers who work for your client. They apply a given set of tests to your software and then advise you about the results. Publishers often have a game testing system of this type in place.

Organizing Test Planning Documents

A convenient way to organize documents involves creating a project test plan and then making references to other, specialized test plans in the References section of this document. In this way, you can use a standard template (such as the IEEE 829) to provide coverage of common topics while referring readers to specialized documents for specific testing issues, such as test cases and test reports. Figure 11.8 shows a basic document hierarchy similar to the one that the *Ankh* team used.

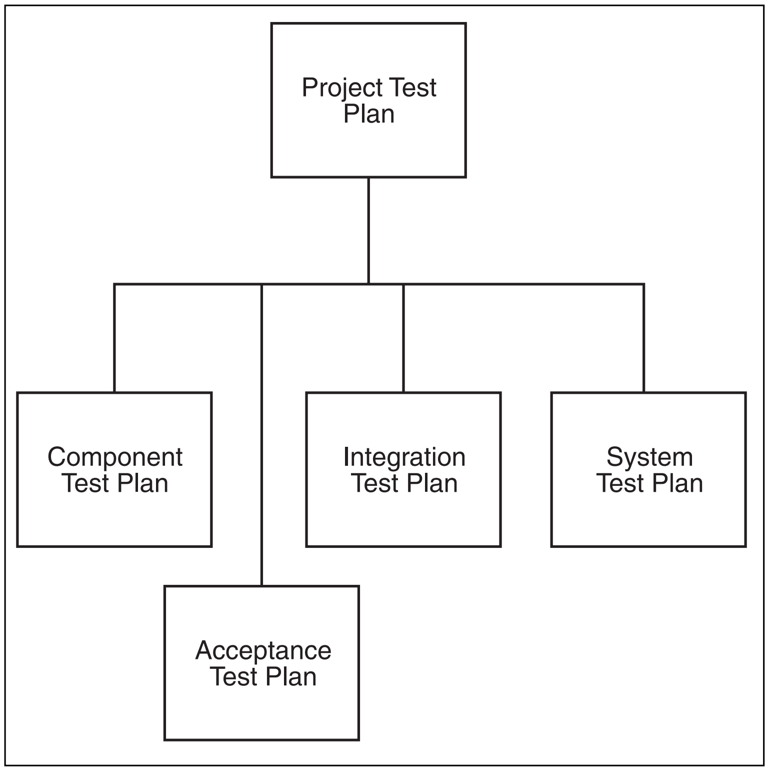


Figure 11.8
A simple document hierarchy directs readers from a project to a test-implementation view of testing activities.

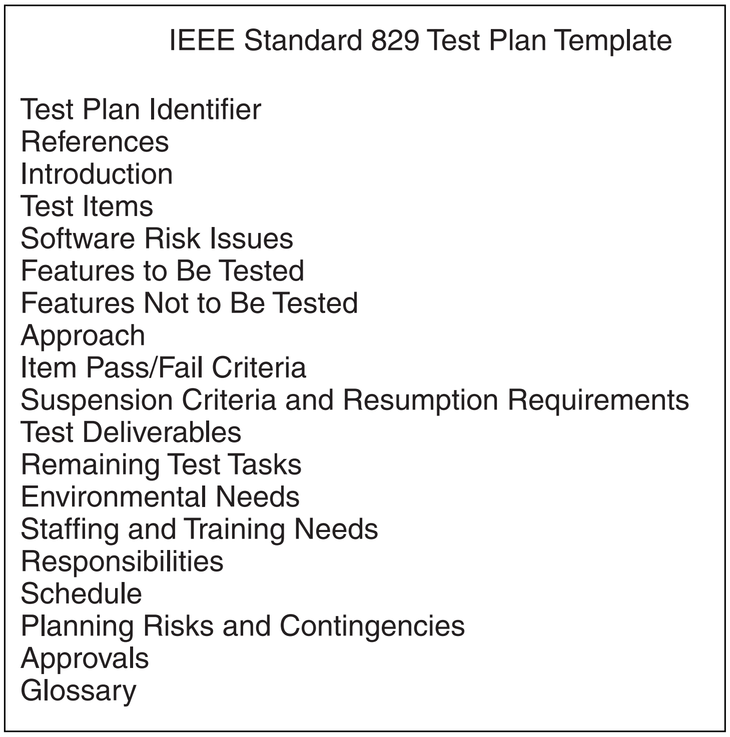
Using Templates

You can group the templates that you use for developing testing artifacts into a simple set:

- **Tables with multiple columns.** Tables provide a convenient way to store the data derived from testing. This proves especially true when you want to document test cases and must provide, in the process, specific itemizations of how you intend to combine input values.
- **Test cases.** As a part of its standard 829 commentary, the IEEE furnishes several extremely useful tabular templates for test cases. These templates provide headings you can implement in a document format, but the tabular form tends to make everything easy to understand.
- **Test reports.** As with the templates for test cases, the IEEE also furnishes sample test reports. Test reports have few sections, but often you must supplement each section with extensive tables of output values.
- **Test document outlines.** Templates for documents help you understand how to organize test cases, test reports, and tables of generated values so that you can readily access them.

IEEE 829

Standard 829 for test plans constitutes one of the best starting places for generalized test plans. Figure 11.9 provides topics from IEEE Standard 829. For a specific listing of the topics and extensive discussion of how to use them, you can consult the IEEE Web site (http://www.ieee.org). The template's superiority lies in its ability to help you focus on the types of testing you want to perform. The *Ankh* team employed the template to create the project test plan. The project test plan presented general policies and definitions. For specific information on testing, the plan referred the reader to other documents. Figure 11.8 illustrates how the primary test plan based on IEEE 829 relates to the other three test plans.



IEEE Standard 829 Test Plan Template

Test Plan Identifier
References
Introduction
Test Items
Software Risk Issues
Features to Be Tested
Features Not to Be Tested
Approach
Item Pass/Fail Criteria
Suspension Criteria and Resumption Requirements
Test Deliverables
Remaining Test Tasks
Environmental Needs
Staffing and Training Needs
Responsibilities
Schedule
Planning Risks and Contingencies
Approvals
Glossary

Figure 11.9
Project test plans can use the IEEE Standard 829 as a starting point.

Among the most important topics are those that relate to how you want to approach the overall test strategy. Thinking these topics through from the start helps you determine how you can anticipate risks involving coverage. Ultimately, every test strategy must dictate that not everything can be tested. You can test only what you have the time and resources to test. Because of this, you must give consideration to how to best compensate for limited testing. The template helps you do just this when it asks you to name things like the features to be tested, the approaches to be taken, the items you intend to designate for testing, and the risks you anticipate.

Table 11.1 discusses the topics that IEEE Standard 829 includes. (The CD includes the test plans developed for the *Ankh* development project. Appendix D, "Software Engineering and Game Design Documentation," provides instructions about how to access the documents.)

Table 11.1 IEEE Standard 829 Topics

Topic	Discussion
Test Plan Identifier	The identifier should allow you to identify both the document type and the version of the document. The identifiers that the *Ankh* team used consisted of abbreviated document titles and a number designating the level of release. The identifiers were as follows: AnkhSWTP1_0.doc (project test plan, first version), AnkhSWCTP1_0.doc (component/class test plan, first version), AnkhSWITP1_0.doc (integration test plan, first version), and AnkhSWSTP1_0.doc (system test plan, first version).
References	The References heading in the project test plan for *Ankh* included the names of the documents that contained specific testing information. This approach to documentation reduced the amount of general information included in the subordinate test plans (the integration, system, and component test plans). Subordinate test plans included a reference to the project test plan. Reducing redundant information decreased maintenance work.
Introduction	The introduction explains the overall test plan. If you want to use a minimum of narrative, you can use a diagram representing documents and their associated purposes.
Test Items	Test items designate identifiable components, classes, operations, logical entities, or other testable entities. Identifying test items allows you to determine the types of test plans and test suites that you want to develop. For example, if you work in an object-oriented context, you name classes, operations, and modules as test items.
Software Risk Issues	Testing risks can go in many directions. Two key elements are how much time you have to test and the scope and complexity of the system you are responsible for testing. See "Assessing Risks," later in this chapter, for more information.
Features to Be Tested	Those who play the game experience the features of the game. Software requirements specify the functionality that supports the features of the software. The features of the game, then, concern how it is played. Use cases provide an excellent way to name and describe features. You group features by placing them in general subheadings. After grouping them, you can assess their importance in relation to the testing effort.
Features Not to Be Tested	Features that you do not intend to test consist of those that your team planned but decided to exclude from the current release. Among these are functionality that might be implemented but left unused, documents, "trusted" components, and nonfunctional requirements. The more formally you test your game, the more you must clearly indicate what you have and have not tested and the extent to which you have covered the features you have tested.

(continued on next page)

Table 11.1 IEEE Standard 829 Topics (continued)

Topic	Discussion
Approach	When you describe your approach, you describe what might be viewed as the terrain of your testing activity. In other words, you might explain that the testing effort encompasses class, module, and system testing, and that module testing is specified in the integration test plan. You might explain that regression testing forms, in your approach, an important element of integration testing. In addition, you might indicate that your approach addresses multiple platforms, strives to generate valuable metrics, and will adhere to a strictly planned (documented) course. See "Ways to Test," earlier in this chapter, for specific information on this topic.
Item Pass/Fail Criteria	Pass and fail criteria apply both to individual tests and the overall test effort. This section allows you to establish policies, such as that the test scenario of every test case must be completed and reported before the test suite containing the test case can be said to be complete. Further, you can say that every test case that a procedure contains must pass before the test procedure can pass. On the other hand, you might designate percentages of acceptable coverage. Generally, a test case does not pass until it has shown that the software flawlessly furnishes the behavior set as the pass criteria for the test case.
Suspension Criteria and Resumption Requirements	This section provides a space in which to list such reasons for suspension. One commonly occurring reason for suspending testing is that a given type of testing no longer generates useful data. For example, if you are testing a given field and have tested boundary conditions and 75 percent of the numbers you could enter into the field, you can suspend testing. Another commonly occurring reason for suspending testing is that you have detected a defect in the software that shows that any results that follow will be unreliable. Until maintenance programmers eliminate the defect, further testing is a waste of time.
Test Deliverables	This section allows you to state specific deliverables, such as test cases, test reports, and test plans. When you complete this section, name items such as logs, tables, summary reports, reviews, and documents that should accompany your test documentation effort.
Remaining Test Tasks	You purposely exclude some test items, and other test items fall outside the scope of your testing effort. Just because such items fall outside the scope of your testing effort does not imply that anyone should assume they should not be tested. For this reason, after you have completed the actions named in the test plan, you describe testing tasks that might still need to be tested.

Topic	Discussion
Environmental Needs	This section names the tools you use for testing. The tools include software and hardware. For example, in most industrial settings, it is considered essential to establish a separate and distinct test environment from the ground up for testing purposes. When you describe environmental needs for testing, you identify everything you need to establish and sustain the testing effort.
Staffing and Training Needs	Staffing and training needs can relate to tools required to perform testing. How this works out depends on the culture of your organization. You can include a schedule that names those who are to perform testing.
Responsibilities	Defining roles facilitates the development effort. You can set policies about how you want the testing effort to proceed. Even if roles are mixed, you can establish ground rules about what responsibilities people have when they work as testers.
Schedule	For the *Ankh* effort, a separate test team was not established. Because of this, the testing schedule was folded into the project plan. When setting up the project plan, the project manager sets up specific testing tasks to be assigned to testers. Many industry experts contend that software products benefit if the managers assign testers and developers to separate groups. There is great virtue to this approach. In a setting in which separate teams exist, the test manager should create a distinct test plan. The testing schedule should provide information about when testing is to begin, what amount of time is to be allotted for testing for each component or stripe, and when testing is to be completed.
Planning Risks and Contingencies	Risks include what happens when you do not have enough time to test everything that you plan to test. Another issue is what to do if you lack people to perform testing. A risk is anything in the testing effort that endangers the completion of the game development effort. If the testing effort can endanger the schedule, you should list it.
Approvals	Name the people who should approve the test plan. This includes the project manager and the test manager, but you can designate many other people, too.
Glossary	Include a glossary of the terms that apply to your project. Even if you think that things like "black-box testing" are clear to everyone, ensure that disagreements or misunderstandings do not occur. Definitions ensure that everyone has a common understanding of basic terms.

Test Case Template

The IEEE defines a test case as a specific action, usually isolated from others, that you take to evaluate the performance characteristics of a software system. The view taken in this book is that the expression "text case" can be used interchangeably with "test procedure." Still, it is important to note that some testers use the expressions to designate different things. Accordingly, a *test case* is said to be a specific action performed to evaluate a specific software property. On the other hand, a *test procedure* is said to consist of a set of test cases. Again, the approach to testing that this book presents merges the two. Many development organizations commonly do the same thing.

To expeditiously formulate a test case (or a test procedure), you can use the IEEE standard 829 template. To accomplish this, you can begin by creating a table for the test case using a word processor (Word, FrameMaker, and so on) or a tool such as SmartDraw. If you use this approach, you expend the greatest effort when you create the first test table. After that, the effort involves copying, pasting, and modifying. If you want to assert a specific set of policies for creating test cases, you can include instructions in the first table you create. Figure 11.10 illustrates one of the templates that the *Ankh* tests used.

Test identifier: Provide the abbreviation for the stripe, an underscore, "ITC," an underscore, and an integer identifying the version of the test procedure of case. Example: S1_ITC_001
Requirements adressed: List the requirements that the stripe addressed.
Prerequisite conditions: List conditions for operation of stripe. This is usually stripes that should already be implemented, libraries, or other such conditions.
Test input: Describe what you do as a tester. Mouse clicks or other user actions. Use a table if necessary.
Expected test results: Start with the results named in the use case.
Criteria for evaluating results: Start with the scenario given in the use case. System responds as prompted and allows user to complete use case actions without defects.
Instructions for conducting procedure: Copy items from the use case.
Features to be tested: Name the features or provide a table or other representation.
Requirements traceability: Name dependencies.
Rationale for decisions: Name any reasoning that qualifies the pass/fail criteria.

Figure 11.10
IEEE standard 829 provides a template that you can use to create test cases.

To use the template effectively, you can create two or more columns instead of just one. If you use several columns, you can more effectively detail the information that you want to use to conduct the tests. Table 11.2 discusses the information that's included in the test plan.

Table 11.2 Test Case Headings*

Heading	Discussion
Test identifier	The unique identifier that you use for test cases varies with the approach that you adopt to testing. For the *Ankh* testing effort, because most of the test cases were general, the team used a simple type of identifier. In more intensive testing environments, this scheme can be extended to accommodate large numbers of test cases or test cases that identify different types of testing scenarios. The numbering scheme for the *Ankh* test effort was as follows: The letter *R* or *S* (for requirement or stripe) and the number of the test case, an underscore, letters designating the type of test (ITC for integration, CTC for classes, STC for system), an underscore, and an integer identifying the version or segment of the test case. *Example: S001_ITC_01*.
Requirements addressed	List the requirements that the test case addresses. This activity extends in two directions. For black-box test cases (also known as functional or behavioral testing), a test case might address only one requirement. For the *Ankh* effort, because each black-box text of the requirements originated with the use case included in the requirements document, it made sense to address only one requirement at a time. For test cases concerning stripes or modules, you should include the requirements that cover the functionality of the stripe or module. For the *Ankh* effort, the integration test plan listed test cases derived from the software design specification. The software design specification included use cases for each stripe. Determining the correctness of system responses with respect to stripes was the central focus of integration testing.
Prerequisite conditions	Prerequisites can be specified in a number of ways. If you are testing classes, for example, the values of attributes when an object of a class is instantiated constitute testable preconditions. The test case can specifically list the values that define the preconditions. On a broader basis, when you are dealing with modules in the integration test plan, you might designate dependencies on other modules. This approach also provides a context in which regression tests can be performed.

(continued on next page)

Table 11.2 Test Case Headings* (continued)

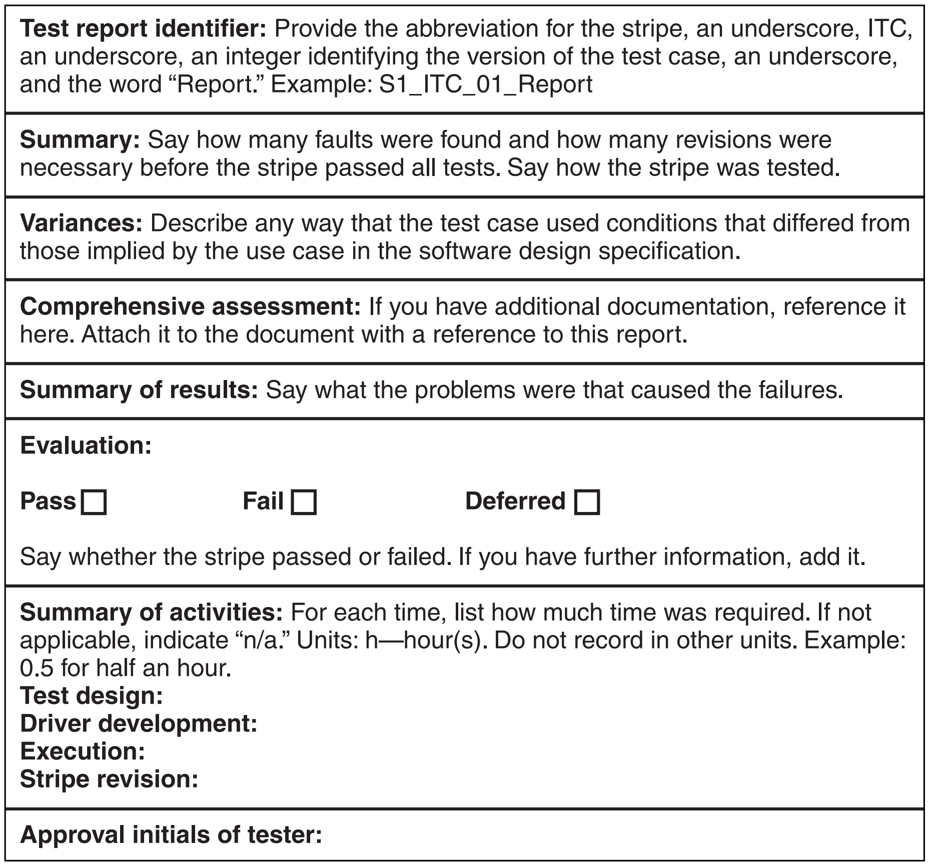
Heading	Discussion
Test input	Detailed test cases require that you provide tables of values you want to use for testing. For the *Ankh* effort, the development team worked at several levels. At the most general level, the team designated input as general as the action. In other cases, the team evaluated operations through stress and similar tests involving specified data. For example, trying letters, numbers, nonletter characters, and blank spaces in the dialog box fields involved creating tables that listed such values.
Expected test results	For the black-box tests of the requirements, the use cases included in the requirements specification made it easy to determine expected results. Having at hand such ready-made scenarios for testing constitutes a major reason for including use cases in the requirements specification. With brief rewording of the scenarios and insertion of specific test data, the use cases almost always completely anticipate the categories of information states in the test case. If you are testing classes, you can define expected test results at specific levels. For example, you can use sequence, collaboration, or state diagrams to determine the values you expect to see when messaging occurs. In basic terms, for example, given that you have created an instance of a class, what value do you expect when you use a given operation with a given set of parameter values?
Criteria for evaluating results	When you evaluate the results of testing, you do so using criteria such as acceptable variations and performance profiles. In other words, you try to provide a generalized context in which you can judge the significance of the results you have obtained through testing. That a given operation returns zero means nothing unless you have established a context in which you can tell whether such a return value properly expresses the conditions set for the operation. In other cases, it is possible to speak fairly broadly. For example, given that a use case from the design specification establishes the context of performance, it might be reasonable to write that evaluation of a stripe cab be based on system responses. The *Ankh* testing scenario designated that the system should perform as prompted and allow the user to successfully complete the action named in the test scenario.

Heading	Discussion
Instructions for conducting procedure	If you create use cases for both requirements and design efforts, you can begin writing instruction sets for test cases by copying the scenario from the use case. If you do not use this approach, then you can formulate a numbered list of steps you intend to follow as you conduct the test. If you are using a test harness, you should provide enough information to allow readers of the test case to repeat the actions you name. It is helpful, for instance, to tell the user of the test case to copy code into the harness (if this is what you do) and how to proceed from there.
Features to be tested	A requirement designates the functionality that is to be implemented to support a feature, so to establish the feature to which a test of a requirement relates involves inspecting the game for specific ways that the implementation of the requirement affects the player's experience of the game. If you employ a use case approach to requirements engineering, the use cases in the requirements specification provide the best starting place for identifying features to be tested.
Requirements traceability	Requirements tend to be mutually dependent, so you should trace dependencies. Requirements often address modules or stripes, so you should trace them to modules and stripes. On the other hand, when given classes embody the functionality a given requirement has designated, then you should trace the class to the requirement.
Rationale for decisions	Most test results fall into four categories, as follows: *pass, fail, deferred,* and *cancelled.* Regardless of the result you reach, it is beneficial to make a statement that helps the reader know how the test case can be used in different ways to accord with different circumstances. As an extreme example, you might set up tests for the numbers 1 through 100. You could say that if you test 0,1, 100, and 101 (boundary conditions), the other tests are not needed.

*See Figure 11.10 for a sample test case.

Test Report Templates

Test report templates provide you with an expeditious way to record test results. Figure 11.11 illustrates a test report template that is suitable for reporting the results of test cases or test procedures. The test report template guides you as you create tables or other artifacts for recording the results of testing. As with test cases and test procedures, the best approach to organize test reports involves using a table.



Test report identifier: Provide the abbreviation for the stripe, an underscore, ITC, an underscore, an integer identifying the version of the test case, an underscore, and the word "Report." Example: S1_ITC_01_Report

Summary: Say how many faults were found and how many revisions were necessary before the stripe passed all tests. Say how the stripe was tested.

Variances: Describe any way that the test case used conditions that differed from those implied by the use case in the software design specification.

Comprehensive assessment: If you have additional documentation, reference it here. Attach it to the document with a reference to this report.

Summary of results: Say what the problems were that caused the failures.

Evaluation:

Pass ☐ **Fail** ☐ **Deferred** ☐

Say whether the stripe passed or failed. If you have further information, add it.

Summary of activities: For each time, list how much time was required. If not applicable, indicate "n/a." Units: h—hour(s). Do not record in other units. Example: 0.5 for half an hour.
Test design:
Driver development:
Execution:
Stripe revision:

Approval initials of tester:

Figure 11.11
A test report template guides you through either recording test results or developing tables and other artifacts to record test results.

Effectively developing testing reports requires examining how much information you must record if you are to document the results of testing. For the vast majority of tests involving fields and buttons, you might find that it is best to use a single table to combine test procedures or cases with reports. In other instances, the test reports might require that you document a succession of results that are more easily understood if examined in isolation. In this case, a table similar to the one shown in Figure 11.11 might be best. Table 11.3 discusses the information included in the test report.

Table 11.3 Test Case Headings

Heading	Discussion
Test report identifier	For the *Ankh* effort, the identifier involved the abbreviation for the stripe, an underscore, a three-letter designation of the type of test (ITC, for integration test case, for example), an underscore, an integer identifying the version of the test case, an underscore, and the word "Report." *Example: S1_ITC_001_Report*
Summary	The summary should provide an overview of the test. You can write a summary. To make things easy, you can check boxes for Open, Completed, Failed, or Complete-Passed.
Variances	Variances can describe differences in initial test conditions, procedures, and results. While it is important to conduct consistently repeatable tests, you often need to make allowances for unanticipated aspects of the planned testing routines. Chief among such variations are those that arise due to limitations of time. For example, if you stipulated 100 test cases but due to scheduling limitations decided to test a sample of this, you can note this as a variation.
Comprehensive assessment	You can refer readers to a table of summary values, or you can provide a simple description of the results. The purpose of the report is to provide developers with information about how the system failed (or passed) the test you applied to it. In contrast to the Summary section, this section calls for specific testing data. You can provide, for example, specific data to help with further analysis of the defect.
Summary of results	Say what the problems were that caused the failure. If you set up a table in this section, you can provide metrics on how many times you conducted the test before it rendered a passing result and how much effort you expended to remove the defect.
Evaluation	The easiest approach to this might be to create check boxes.
Summary of activities	As the template featured in Figure 11.11 indicates, important metrics center on duration and effort. The information that you call for when you summarize activities depends on what you or your organization views as important. Key metrics, such as number of testers, duration, and effort, are usually beneficial.
Test design	If you develop a standard set of testing tools, you can create a set of check boxes that enables you to indicate quickly which standard design you applied. See "Tables for Testing" later in this chapter for a list of table types that you can employ to guide test design.
Driver development	If you apply a standard test harness to a multitude of test cases, you can provide a reference to the files that contain the driver. Readers of the report can then view this material to see how it was implemented.

(continued on next page)

Table 11.3 Test Case Headings (continued)

Heading	Discussion
Stripe (module) revision	This topic proved useful in the *Ankh* development effort because the team conducted development and testing in the context of specific stripes. It was important, then, always to trace activities to stripes.
Approval initials of tester	It is important to identify those who are involved in a given test scenario. This information proves useful if the defect is still current and if you are gathering historical data.

Tables for Testing

Tables are one of the most used type of testing artifact. To set up tables for your testing effort, you might find that it is best to take some time at the start of your project to create table templates. To create table templates, you can review your test plans and derive from them the types of tables you will need. Experienced testers often develop an extensive set of table types that address specific testing needs. In her book *Introducing Software Testing,* Louise Tamres provides an extensive review of tables she has developed. (The complete reference appears at the end of this chapter.) Among the tables that Tamres discusses are the following:

- **Decision.** A decision table allows you to organize information along the lines typical of logical tables. Such tables are useful when you have multiple conditions that correspond to multiple actions.
- **Keystroke/mouse action.** This table tracks keystrokes or mouse actions against fields and buttons. Among other tasks, this table provides a convenient way to track the minute test cases that are involved in testing user interfaces.
- **Test procedure summary.** This table enables you to coordinate test procedures when you have a set of test procedures that consist of many test cases or when you have test cases that involve procedures or a set of subordinate test cases.
- **Boundary condition tables.** This table provides a way to ensure that you cover the relatively few values you apply to a test scenario involving boundary conditions.
- **Cross reference.** This can be a general utility table for tracking test procedures and test cases. For example, suppose that you face a set of dependent operations. You might use one dialog box in the GUI to enter values and then, later, access another dialog box to retrieve the values. You might find it necessary to cross reference the results of the two sets of interactions.
- **Test management.** Test management involves tracking the performance of tests and the results. You might want to create a table that summarizes the results of dozens of test procedures.

- **Test cases, procedures, and reports.** With the *Ankh* development effort, the team used tables for test cases, test procedures, and testing reports because these things provided a convenient way to reduce the amount of writing involved in creating testing material.
- **State transition.** You can use tables to track the changes in objects. If you first create a UML state transition diagram when you use such tables, you can readily visualize the attributes that the table contains.
- **Test requirements.** Summarizing requirements in a table ensures that you can track your testing progress.
- **Failed and passed reports.** Summarizing testing reports, generally, makes it easier to gather metrics and present summaries.

note

Companies like IBM (Rational) and Borland manufacture testing tools that automatically generate tables. For cottage industry endeavors, you can create tables using a spreadsheet, such as Excel, Lotus, or Quattro Pro. Use spreadsheets if you anticipate extensive work with metrics or if you want to access data for automated test scripts. If you want things to be easy to use, the table features of word processors, such as Corel WordPerfect, Microsoft Word, or Adobe FrameMaker, are the best. A final option is a general-purpose application, such as SmartDraw.

Limitations of space make it impossible to provide extensive discussion of how to use tables, but a simple scenario can illustrate a few key ideas. (For comprehensive treatment of the topic, see the book by Tamres listed at the end of the chapter.) As an example of how the *Ankh* test effort used a table, consider the dialog box in Figure 11.12.

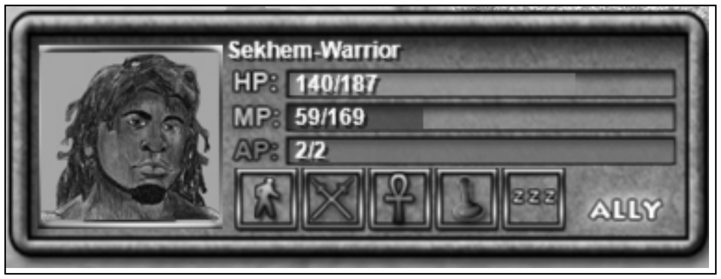


Figure 11.12
A table helps you control the testing of dialog boxes.

The dialog box shown in Figure 11.12 is the central control dialog box that the player sees during most of the game. To designate an action for a turn, the player clicks one of the buttons. The dialog box features three sliders that indicate health and other points. Testing the dialog box requires the tester to investigate the behavior of each of its features.

To set up a testing table that facilitates this effort, you create a matrix that shows the test cases that guide you as you subject each of the features of the dialog box (such as buttons and fields) to the possible inputs (mouse clicks, keystrokes, scroll actions, and so on).

As Figure 11.13 illustrates, after you have created a table to track the keystrokes and mouse actions against sliders, fields, and buttons, you can easily work through each test case that the table names and record the results.

Cursor Location / Key/Mouse Action	Left Mouse	Right Mouse	Scroller	LM + Key	RM + Key	SRL + Key	Non Act Key	Middle Mouse	g	w	s	a	z	Esc	Expected Result	Exceptional Result	Pass/Fail
Walk button	+	–	–	–	–	–	–	–	+	–	–	–	–	–	Squares appear around character.	None	P
Weapon button	+	–	–	–	–	–	–	–	–	+		–	–	–	Opens Weapon Selection dialog box. See TC D-10.	None	P
Status button	+	–	–	–	–	–	–	–	–	–	+	–	–	–	Opens Status dialog box. Display only.	None	P
Scroll button	+	–	–	–	–	–	–	–				+		–	Shows Inventory dialog box. See TC D-11.	None	P
Sleep button	+	–	–	–	–	–	–	–					+	–	No action.	None	P
HP field	–	–	–	–	–	–	–	–	–	–	–	–	–	–	No action.	None	P
MP field	–	–	–	–	–	–	–	–	–	–	–	–	–	–	No action.	None	P
AP field	–	–	–	–	–	–	–	–	–	–	–	–	–	–	No action.	None	P

Figure 11.13
Track involved test case scenarios using a table.

The table in Figure 11.13 is set up so that the column on the left names buttons and sliders (fields), and the top row identifies actions. The plus and minus signs signify that the tester should try the action, but only those actions that a plus sign marks should elicit a response. The two columns containing result information allow you to know the specified result or, if the specified result does not occur, to note what did happen. If you take an action and the expected result occurs, the test case or procedure passes. P indicates pass.

This table satisfies the basic criteria for setting up a test case. When you use such a table, you should exercise caution to ensure that you distinguish each test case. In this instance, either clicking on the Walk button or pressing the G key results in the action described. (Note that the table should indicate if the Shift key is used with any other keys.) If the test fails for either input, this table requires that you employ the Exceptional Result column to explain how the test failed. Perhaps a better approach would be to set up the keys in the Cursor Location column. This table decreases the number of rows but can introduce some confusion.

Testing Activities

When you perform testing activities, one maxim is that you should test as many things as possible as much as possible. What and how much you test depend to a great extent on how you plan your efforts and what tools you employ. Tools consist of documentation, strategies, and test programs. The following few sections concentrate on strategies and programs that you can create.

Test Cases and Procedures

Test cases and test procedures are documented instances of testing. The IEEE distinguishes between test procedures and test cases, but the practice in this chapter is to use the two expressions interchangeably. The IEEE defines a *test case* as a specific, isolated instance of testing. A *procedure* is defined as a collection of test cases. In practice, however, a procedure can consist of test cases; likewise, a test case can be a procedure. Testers also employ the expressions *test scenario* and *test script* in much the same way that they do test case and test procedure. Context determines meaning.

Although the specific terms that apply to testing might be open to interpretation, a few recommendations drawn from the IEEE should be followed fairly strictly:

- **Identified start state.** Every procedure or test must commence with a clearly defined set of conditions. In other words, when you design a test, you identify the state of the system from which the test is to be initiated.
- **Procedures that can be repeated.** A second stipulation is that you describe the test that you perform so that it can be repeated. To achieve this goal, you can use a script of actions you perform to complete the test. You can supplement your test procedure with data tables and other artifacts. If you find yourself plodding through directories and files to find test scripts, data, result information, and other such items that you have not documented as part of your test scenario, it is unlikely that you have created a test that you or others are likely to be able to repeat. Documenting a test procedure amounts to writing a set of instructions that others should be able to use.
- **Identified end state.** A test determines whether or not the system properly transitions from one state to another. Because transitions constitute primary criteria for a test, as part of your test, you must identify criteria you can use to establish that transitions result in specified changes. Just as you designate the state of the system at the start of the test, you designate the state of the system at the conclusion of the

test. The way you designate the concluding state depends on the type of test you are performing. If you work at a behavioral level of testing, describing what the game does might suffice. On the other hand, if you are working at a structural level, you might need to know the specific values stored in object attributes.

Every test in some way tests a change in the system. You must be able to know, identify, and evaluate these changes.

Converting Use Cases to Test Cases

Converting use cases to test cases constitutes a central activity of testing in object-oriented development efforts. This occurs because the use cases you develop during the requirements and design phase of your development effort can anticipate almost every behavioral scenario of the software product. What results is a substantial body of material from which you can draw most of the information you require to create test plans and develop test procedures or test cases. (Page limitations make it impossible to discuss this important activity in detail in this chapter. For a full description of how to convert a use case to a test case, see the section titled "Use Case Confirmation" in Chapter 14, "Practice, Practice, Practice.")

Using Outlines

An alternative to converting use cases to test procedures is developing test outlines. This approach to testing predates object-oriented programming, but it remains an instrumental part of many testing efforts. One reason for this is that many testers enter the development process only after the product has been substantially completed and the available documentation traces the design of the product in a rudimentary, incomplete way.

When this type of situation arrives, the tester can initiate a testing process based on tracing the activities that the product is supposed to perform. An outline allows you to begin a general level and gradually refine the outlines as you collect more information about the product. As you go, you can begin supplementing the outline with use cases, test cases, testing tables, and other artifacts.

Test Suites

A test suite is a group of test procedures that you package together to investigate what you consider to be an area of behavior that the test procedures commonly address. A test suite, for example, can address a module that provides GUI services. The test cases in the suite investigate classes, operations, assets, and other items that you have determined to support GUI capabilities. Ideally, you can use the tests that you develop for one component in a test suite for others in the suite.

Black-Box Testing

Black-box testing is also called *behavioral* or *functional* testing. To perform black-box testing, you create a test procedure that establishes a set of start conditions for a given scenario of change you expect to take place. To investigate this change, you establish a clearly defined set of results. You then write a procedure that allows you to operate the system so that it goes through the changes you have anticipated. The test succeeds if you can repeatedly perform the same operations, beginning from the same set of conditions, and end up with the same results.

Black-box testing is most frequently applied to verification and validation of requirements. The approach employed for *Ankh* involved converting the use cases from the requirements specification into test procedures. Each use case provided a scenario of use. The scenario of use began with trigger conditions and ended in a given state change. The test procedure formalized this scenario. The testing, then, investigated only the behavior of the system. The way that the system structurally affected the changes that were investigated was left to other types of testing.

White-Box Testing

White-box testing is also known as *structural* testing. It has some of the features of black-box testing (beginning conditions, test scenario, end conditions), but with white-box testing, the granularity of the test activity becomes much finer. The objective involves investigating the specific changes of state that objects undergo as the system transitions from the beginning conditions to the end conditions that your test describes. This type of transition is known as a *state transition*. UML tools provide ways to trace and depict such transitions.

To execute white-box test procedures, you develop several types of artifacts. Among these are main function programs that you employ to instantiate and explore class objects, operations you add to a class so that you can derive information from objects of its type as they go through state transitions, and complex test harnesses that allow you to investigate the interactions of objects from different classes.

You can refer to the code that you write to conduct such tests as test *harnesses*. Harnesses allow you to execute operations in isolation. Because a test harness executes a given body of code in isolation, you can develop it any way you want without fear of corrupting the entire program. Reducing the amount of system interaction limits the complexity of the material you are testing.

Harnesses with Static Class Operations

Some testers use static operations and attributes to report object counts and states. You can use a static attribute as a gauge for the activities of all instances of a given class.

Therefore, if you include static functions in your classes, you can maintain tallies of object counts and the cumulative values of attributes among class objects. In most cases, when you output values, you should output to a log file. The following program shows only the basics:

```
namespace test{
class CThing
{
private:
        string thingName;
        int thingPower;
        int id;
        static int testTotal;
        static int testPower;
        static int testName;

public:

CThing(int power):id(0),thingPower(power)
{
        testTotal++;
        id = testTotal;
        testPower = power;
}

static void ShowAttributes()
{
        cout << "\nTotal - " << testTotal << endl;
        cout << "\Power - " << testPower << endl;
}

~CThing()
{
        cout << "Destructor: " << testTotal << endl;
        testTotal--;
        cout << "Count: " << testTotal;
}
};

int CThing::testTotal = 0;
int CThing::testPower = 0;
} //end test nsp

int main()
{
       using namespace test;
       cout << "\n" << "Harness";
       CThing* t1 = new CThing(0);
       CThing::ShowAttributes();
       CThing* t2 = new CThing(10);
       CThing::ShowAttributes();
```

```
        delete t2;
        getch();
        return 0;
}
```

Class Test Harnesses

To create a class test harness, you can develop a template class as your primary test harness. The advantage of using a template class for a test harness is that you can establish temporary messaging systems that simulate those that occur in the game. To make a class harness system work effectively, you should explore the design of the game to discover whether class hierarchies afford you opportunities to make use of existing code features as you develop your test harness. Using abstract classes in your template definition makes it easier to manipulate complex message exchanges. The following bit of code creates two classes, one of which is a template. The template class enables you to immediately put the tested class to work:

```
/*
——————————————————————————————
The template declaration in this program sets up a class
that can receive generic data types.
The calls are cascaded, as in
ctrObjD.getFirstOfItems().getDataMName();
———————————————————————————————
*/

template <class SDataS>
class CTestContainer
{
private:
        SDataS firstOfItems;

public:
        CTestContainer(SDataS setVal):firstOfItems(setVal)
        {}

        ~CTestContainer(){}
        SDataS GetFirstOfItems()
        {
                return firstOfItems;
        }

        void SetFirstOfItems(SDataS setVal)
        {
                firstOfItems = setVal;
        }
};//end of CTestContainer Class
```

```
class CGameClass
{
private:
      string boxName;
      long boxNumber;

public:
      CGameClass(string bName, long bNum):
                                        boxName(bName),
                                        boxNumber(bNum)
      {}
      ~CGameClass(){}

      string GetDataName()
      {
            return boxName;
      }

      long GetDataNumber()
      {
            return boxNumber;
      }
};//end CGameClass class

int main()
{
CGameClass dMObj("Helper", 45);

/*
The CTestContainer constructor uses the CGameClass type.
Having done this, it is then possible to access
operations associated with the object.
*/

CTestContainer<CGameClass> ctrObjD(dMObj);
cout << ctrObjD.GetFirstOfItems().GetDataName() << "\n";
cout << ctrObjD.GetFirstOfItems().GetDataNumber() << "\n";
cout << "\nEnd of program.";
getch();
return 0;
}
```

Module Testing Harnesses

One of the biggest tasks that you face as a tester involves testing modules—classes that work together to provide an identifiable service to your game. (The *Ankh* development team referred to modules as stripes.) Being able to test modules requires that you have an architectural view of the system that enables you to determine if a set of classes behaves correctly as a service provider.

With the *Ankh* development effort, stripe testing began with the isolation of the classes that composed specific stripes. To develop a module test, the team first had to establish the behavioral scope of the test using a design use case. Then the team was able to use a sequence diagram to scope out how to test specific class interactions for the stripe. (See Figure 11.14.)

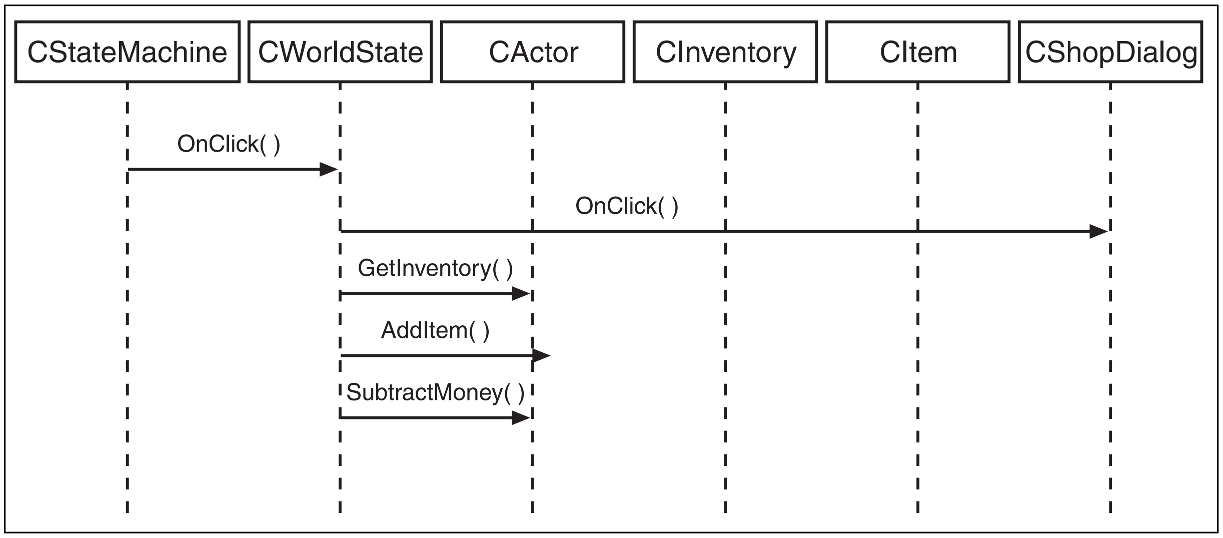


Figure 11.14
A sequence diagram is a starting point for module testing.

Although Figure 11.14 shows only six classes and a set of five message exchanges, it still provides an excellent start for testing. If you start from such a beginning, you can construct a test harness that allows you to create instances of these classes and then proceed from there to specific types of tests. When you develop harnesses for testing modules, your objective is to create a context that resembles the entire system but is simpler. Such test harnesses require planning.

Assessing Risks

Risk assessment is an area of quality assurance, but testing can contribute to risk assessment in important ways. This chapter emphasizes the way that risk assessment can involve determining the extent to which your tests cover the complete scope of the game and provide a suitable level of testing granularity. In the large world of quality assurance, many considerations that go beyond this relatively limited horizon come into consideration. For example, quality assurance efforts endeavor to determine how maintainable software is. In other words, given that something is bound to go wrong at some point, how much effort (cost) will be required to fix the problem?

In the smaller world of test-specific work, two quality assurances that come into play are testability and complexity. *Testability* is the extent to which you can reliably test the software. *Complexity* is sometimes said to be a measure of how many logical paths you can take as you operate a software program. If a program possesses high complexity, you have to expend a greater effort to test it. Given that you have limited testing resources, if the complexity is great and you are to competently test the software, you might have to tell the project manager that the duration of the project must be extended to accommodate the testing effort. (This is likely to meet with objections.)

Coverage

Consider the class diagram that Figure 11.15 illustrates. In this diagram, class A communicates with class B. After it has received a return value from its communication with class B, class A communicates with an abstract object of class C. Depending on the return value communicated to A from B, the abstract object of class C can be instantiated as an object from class D, E, F, or G.

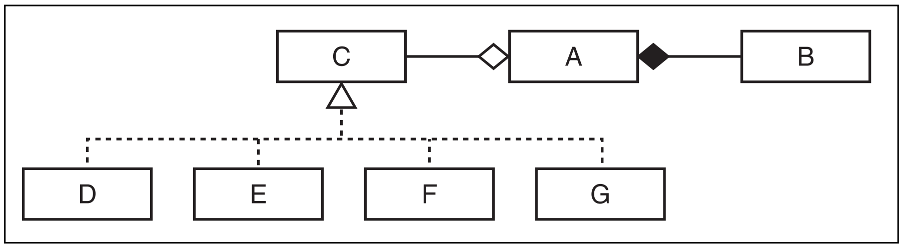


Figure 11.15
A system of classes sustains a multitude of interactions.

As a tester, designing a set of tests that comprehensively covers the interactions of the system classes involves investigating each of the paths that the system generates. As a starting point, you might construct a table to trace the logical branches that the system of classes sustains. Figure 11.16 illustrates a rudimentary venture in this direction. The table consists of a set of test procedures. The test procedures contain a multitude of test cases (which the table does not document specifically).

In Figure 11.16, all paths begin with class A. Class A communicates with class B. Then class B returns a value that allows A to communicate with class C, a factory class. Depending on the value that C receives, it provides an instance of the specialized class (D, E, F, G).

Class / Case/Procedure	A	B	C	D	E	F	G
t1	A–D	A–B	A–C	C–D			
t2	A–E	A–B	A–C		C–E		
t3	A–F	A–B	A–C			C–F	
t4	A–F	A–B	A–C				C–G

Figure 11.16
A table traces the interactions of a system of classes.

To test this system of classes, you can develop a structural (white-box) test. You face the task of creating a test harness that can support at least the manipulation of an object of class A. If you are to completely cover the interactions of this class, you must script the test procedure so that you can evaluate whether classes D, E, F, and G have been instantiated correctly. To accomplish this, you must have initial data to provide to class A and a table of result data by which to evaluate the instances of classes D, E, F, and G. These are the basic paths. Your coverage is determined by your analysis of the interactions of the classes and the number of test cases you develop to test the interactions. Further, the picture grows in detail when you test the specific attributes of each of the evaluated classes to determine whether they possess acceptable values at all points during the transaction.

The example presented so far involves white-box testing, which requires you to investigate the code. You can use another approach, however. Assume that the complexities that are exposed in Figure 11.16 are accounted for by clicks on the screen. Classes D, E, F, and G are really characters that appear when you click given icons. In this instance, you can create a black-box test. Generally, even if you decide to engage in white-box testing, you should employ black-box testing first. That's because it is essential to test the system to determine whether it has met the specifications set by the requirements. If the system should create a character when you click a given icon, you should establish this first through a black-box test. Generally, you should achieve as much coverage of requirements as possible using white-box testing before you begin developing black-box tests.

Complexity

Due to limitations of space and scope, a concept that is not dealt with in detail in this chapter is *cyclomatic complexity*. This approach to complexity constitutes one of the leading ways that software engineers investigate software complexity. Cyclomatic complexity measures a software system's structural complexity. To establish a software system's cyclomatic complexity, software engineers measure the number of ways it is possible to navigate a given body of code. The paths through a body of code are usually referred to as *logical branch points* that the system contains.

For the purposes of the discussion here, the definition of complexity can be scaled back to the basic idea that a game can be tested in terms of the number of ways a player might be expected to use it. The requirements and design of the game software determine the number of ways that a player interacts with a game. If developers competently engineer and implement the software, player interactions will be limited to those that are specified. Unanticipated or unspecified interactions that result in unanticipated or unspecified system changes reveal defects. It is the job of testing to discover such defects.

Considerations of complexity underlie the confidence with which testers can anticipate unanticipated interactions. To achieve a high degree of confidence, testers must design test efforts along two general lines. Along the first line, they create large matrix-based testing plans in which they schematize all the possible interactive pathways. They test each of the possible interactions. Along the second line, they try to arrive at statistical determinations of the paths through the software that are most important. By placing emphasis on testing these paths, testers make effective use of their time and can say, in the end, that they have exerted the greatest testing effort on the areas of greatest risk. This is one goal of orthogonal defect categorization.

Orthogonal Defect Categories

The story of testing risks continues when you consider what happens when you create so many test cases that you do not have time to execute them all. Good testers plan their tests and lay them out in a test plan. However, consider a situation in which a tester plans 1,000 test cases and then has time to execute only the first 30. Assuming the initial set of tests comprehensively covers the product, if the tester just started at the top of the list and worked down, only approximately 3 percent of the product would undergo adequate testing.

To avoid such risk situations, you use orthogonal approaches to categorize test cases. To understand the notion of orthogonal categorization of test data, consider, for example, what happens when you test combinations of values generated from a dialog box. In almost any situation that involves two or more input fields in a dialog box, the combinatorial product tends to create a large number of test cases. (See Figure 11.17.)

Consider a simplified version of the type of dialog box interaction represented in Figure 11.17. The player can, at different points, configure a character so that it is identified with a distinct class, possesses a primary skill, and wields a particular weapon. Figure 11.18 presents a tabular representation of these attributes. At arbitrary moments during the life of the game, the player can access a dialog box and select any combination of the items listed in Figure 11.18. After the player completes this activity, the resulting character has a distinct appearance, possesses a given skill, and wields a given weapon.

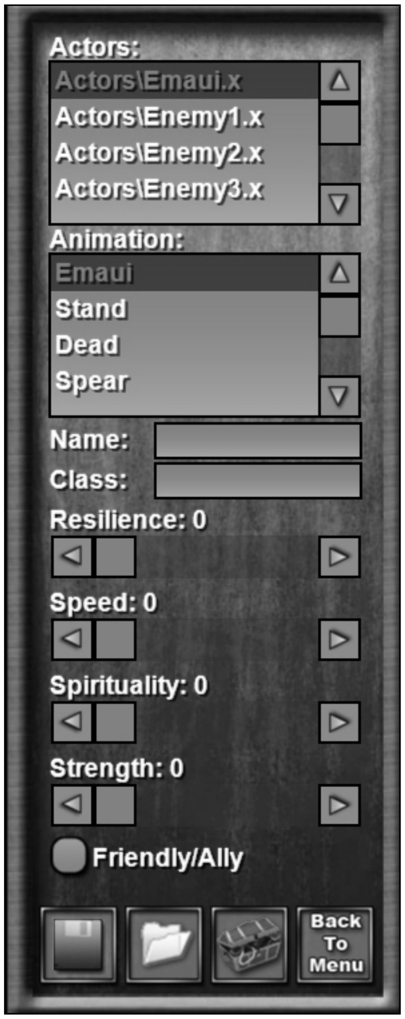


Figure 11.17
Multiple fields create large combinatorial products.

Name	Skills	Weapon
Sekhem	Healing	Greek Dagger
Sati	Meditation	Golden Ankh
Uheset	Burning Eye	Staff of Isis
Khefta	Healing	Staff of Osiris
Neru	Touch of Osiris	Twisted Render
Emaui	Boils	Singer of Boneshatter
Suten-Heh	Sores	Hunting Bow
	Healing	

Figure 11.18
Tabular representation of data combinations allows orthogonal classifications.

To create the test for the dialog box, you begin with the state of the character. You can set the state of the character using default values. You then designate as the test input the values obtained from the selected items. The game design document provides information that allows you to determine expected results. To assess the results, you examine the state of the character after you close the dialog box. At this point, the state values that are assigned to the character should be precisely those obtained from the dialog box selections.

When you test this dialog box, the work is extremely cumbersome even with a relatively limited number of fields unless you automate the testing. Figure 11.19 illustrates the typical combinatorial procedure. To create test cases, you begin with the first item in the Name column, Sekhem, you proceed to the next column, Skills, to the first item, Healing, and then you test these two items in combination with each of the items in the Weapon column. Having completed this work, you return to the Name column and again select Sekhem. You then move to the Skills column and select the second item, Meditation. Then you test these two items in combination with each of the items in the Weapon column. The work is tedious.

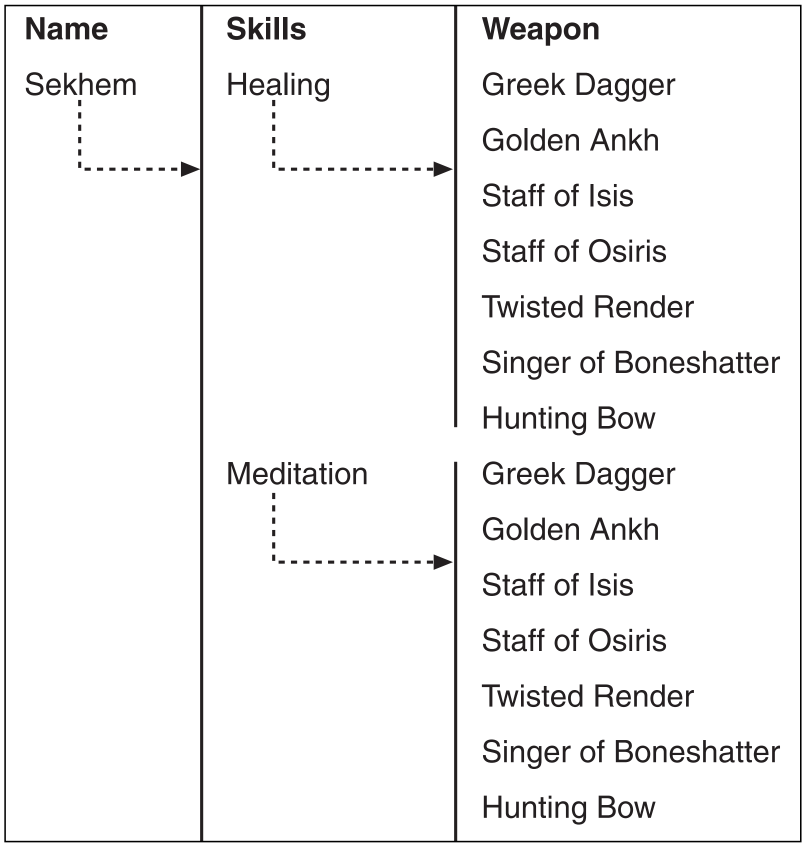
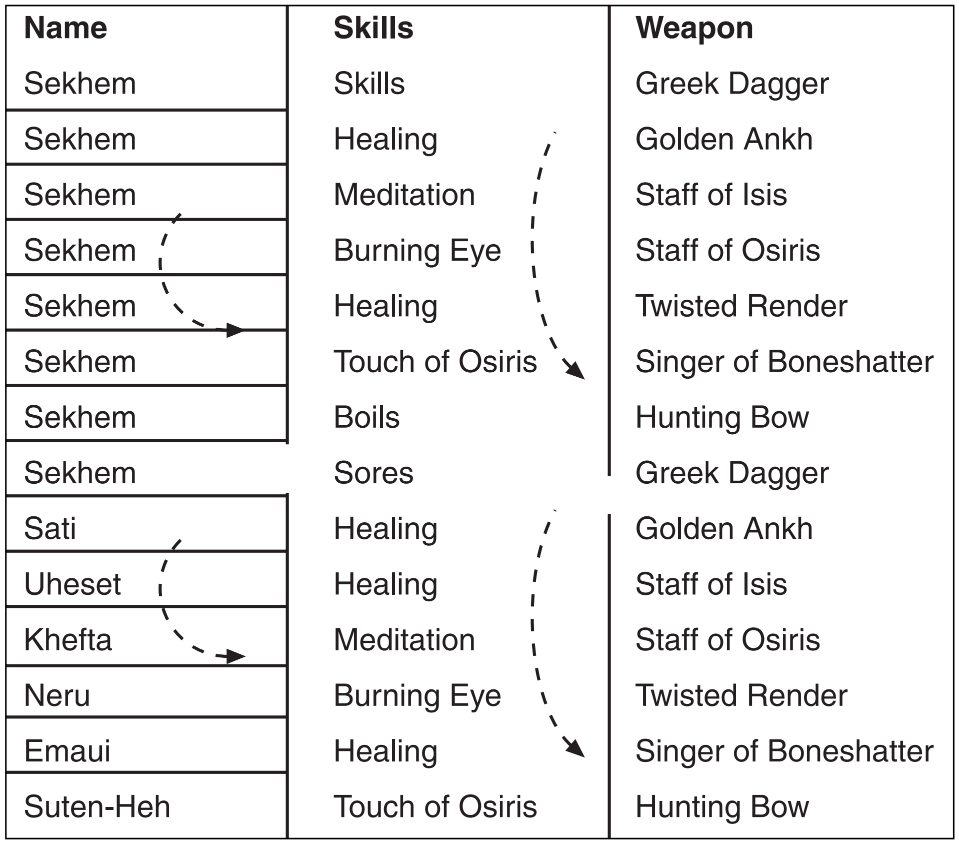


Figure 11.19
When you create test cases, you anticipate all possible combinations.

You can determine the possible combinations (permutations) of the values displayed in the table in Figure 11.18 if you multiply the number of values in each row (7 × 8 × 7). The result is the number of possible test cases (392). When you determine the number of possible test cases using simple combinatorial procedures, you create quite a few redundant test cases.

When you employ an orthogonal approach to testing, you try to discover a few paths through the combinatorial options that can responsibly represent all paths. Considering again the values shown in Figure 11.18, the simplest approach emerges when you test only one combination, such as Sekhem-Healing-Greek Dagger. You can say that you have tested the dialog box scenario with minimum coverage (and substantial risk). The problem with this approach is that it is so minimal that it is nearly irresponsible. A better approach involves obtaining a healthy mixture of test cases while avoiding having to work through an endless enumeration of combinatorial possibilities. Figure 11.20 illustrates one approach.



Name	Skills	Weapon
Sekhem	Skills	Greek Dagger
Sekhem	Healing	Golden Ankh
Sekhem	Meditation	Staff of Isis
Sekhem	Burning Eye	Staff of Osiris
Sekhem	Healing	Twisted Render
Sekhem	Touch of Osiris	Singer of Boneshatter
Sekhem	Boils	Hunting Bow
Sekhem	Sores	Greek Dagger
Sati	Healing	Golden Ankh
Uheset	Healing	Staff of Isis
Khefta	Meditation	Staff of Osiris
Neru	Burning Eye	Twisted Render
Emaui	Healing	Singer of Boneshatter
Suten-Heh	Touch of Osiris	Hunting Bow

Figure 11.20
Taking a moment to create, vary, and distribute combinations expedites testing in a responsible way.

The approach shown in Figure 11.20 is not as scientific as it might be, but it serves to show what can be viewed as a practical approach to orthogonal classification. The scheme involves moving down each column in a way that distributes coverage in each instance across all possible items but also reduces the number of permutations to a minimum.

Impact

Impact concerns how much testing activities affect the development schedule. Testing activities affect the testing effort in two ways. The first way entails causing the development process to take longer than it would without testing. The second way entails decreasing or increasing the reliability of the product. Generally, to a certain extent, the longer you test a software product, the greater its reliability. On the other hand, it is also the case that the longer you test, the fewer defects you discover. After a certain point, you arrive at a crossover point. Determining when you will reach this point heavily impacts the extent to which you can say that you have achieved a high degree of coverage and reduced risk to a minimum.

On the implementation level, each test requires a certain amount of time. The more specifically you plan a test procedure and its constituent test cases, the more you can say precisely how long a test will take to perform. Add to this that experienced testers usually declare that you should make test cases as specific as possible, so that you can perform any given test case as quickly as possible. When you use this approach to test development, you can more flexibly adapt to schedule pressures.

Figure 11.21 illustrates the danger that arises if you plan only a few extensive tests. If you spend a great deal of time setting up four large tests rather than 12 to 16 short ones, your testing effort becomes vulnerable in terms of *effort* and *effect*. Respecting effort, you create a risk because, for example, if you have time to complete only the first two tests, you will cover only 50 percent of the system. The effect this has on the quality of the system is significant. In the second situation, which involves the creation of numerous test cases that do not involve a great deal of preparation, you can adjust your effort if you face time restraints so that you will have time to test all parts of the system in some way. Testers sometimes contend that it is better to test a little bit of a lot rather that a lot of a little.

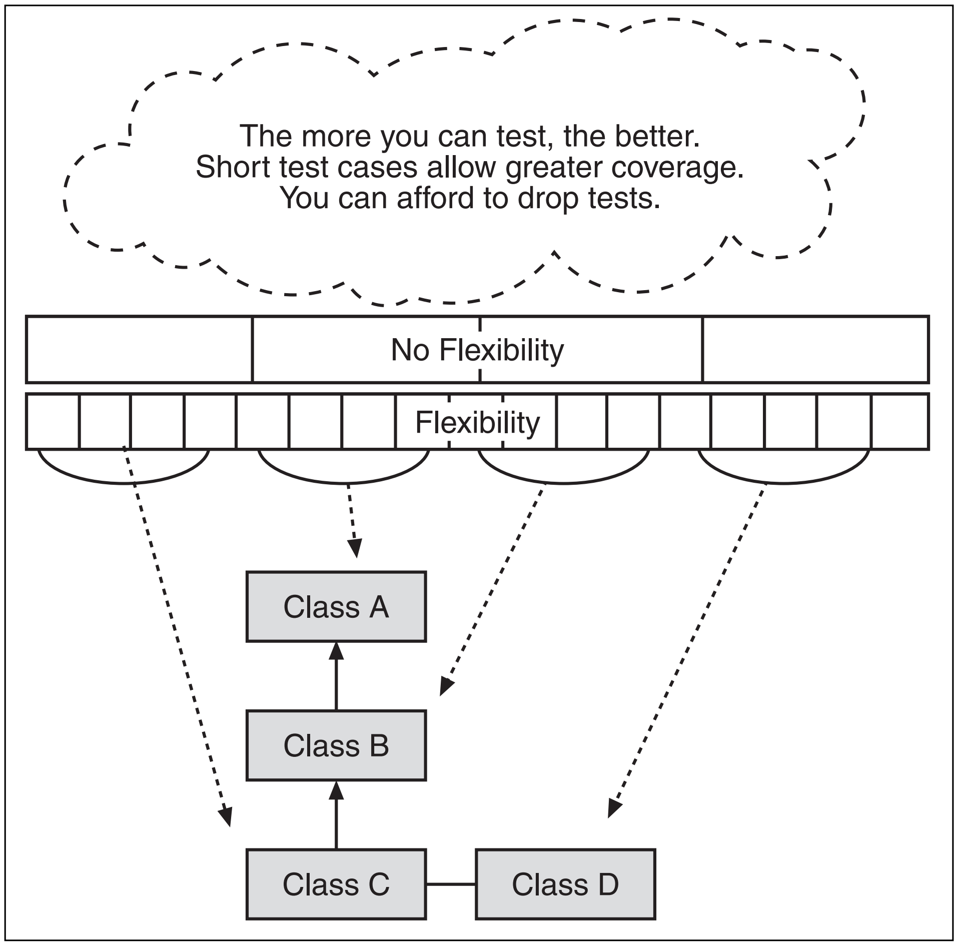

The more you can test, the better.
Short test cases allow greater coverage.
You can afford to drop tests.
No Flexibility
Flexibility
Class A
Class B
Class C
Class D

Figure 11.21
Test a little of a lot rather than a lot of a little.

Testing Roles

Testing roles differ from organization to organization. Generally, experts of the old school contend that it is best to have separate test and development teams. Those who have adopted practices from the Agile Process tend to regard testing and development as mixed roles. The position taken in this book is that testing is a distinct type of software development work, so the more you can spend time learning the special skills that apply to it, the better. Specialization over the centuries has been shown to be a key to quality.

Along similar lines, any given tester is likely to take up different testing roles at different times during the project. These roles, to an extent, take on definition according to the tools and techniques that are developed and acquired during the performance of the work. Some of more common roles are as follows:

- **Integration.** When you perform this work, you are generally the person on the team who is taking the high-level view. Assembling pieces enables you to discern problems of performance that others, working on specific components, might not notice.
- **System.** When you work with the system, you are thrown into the customer realm because you are seeing the product as a whole, one that has to install and operate correctly. Issues of size, completeness, and operability surface.
- **Manager.** If you have worked long enough in a testing capacity and have a good sense of how to develop test plans and schedule work, you are qualified to assume a management role. As a manager, key tasks involve understanding the scope of the game, assessing the appropriate level of test coverage, and assigning enough people to the work to ensure that it is completed.
- **Structural.** This is a general term for testing that penetrates into the system and involves continuous dissection and inspection of minute aspects of the code. An experienced developer who wants to assume a lead test position serves best in this position.
- **Player.** Expert players need not be software testers or programmers, but they need to know their business as game players. Such testers are excellent resources at all phases of the development effort as long as you structure their activity so that the information they provide can be used effectively.

Conclusion

This chapter has investigated a few of the many topics that arise when you test software. Testing software is, along with design and implementation, one of the major activities involved in creating an engineered software product. To understand the intent of testing, reviewing the concepts of validation, verification, and exploration proves helpful. Much of the purpose of testing is to verify and validate the implementation effort in light of the functionality that is specified in the requirements. In addition to ensuring that the software product meets specifications, testing has an exploratory role. The role of exploration involves combing the software to discover if random defects have been added to the product. Exploration differs from validation and verification because it searches for behavior that was not planned.

Planning a testing effort involves creating documents that guide you through your testing activities. The IEEE standard 829 provides an excellent guide for creating such documents. Generally, it is a good idea to start with a high-level test plan for the whole project and then to derive from this high-level plan several low-level plans that cover components, integration, system, and acceptance testing.

Many tools and techniques of testing comprise a successful testing effort. Tables you can create using spreadsheets or word processors are among the most humble but most useful tools for software testers. You can customize tables to fill specific testing functions. You can also use them to organize cases that test the behavior of dialog boxes. Finally, you can use tables to cross reference or logically relate test cases. Table uses are multitudinous and can expedite almost any testing operation.

Testing tends to go in two general directions: behavioral and structural. Behavior testing, also known as black-box testing, seeks to establish only that the software behaves as specified. Such testing can be conducted largely from a user's perspective. Structural testing, also known as white-box testing, involves investigating the inner workings of operations, classes, and modules. To pursue such testing, it is often necessary to construct test harnesses, which come in many forms. Among commonly used harnesses are those that make use of such language features as static operations and template classes.

When you formulate specific test cases or test procedures, try to test a little of everything rather than a lot of a few things. If you can design a multitude of tests that you can perform quickly, even if time limitations restrict the extent of your testing effort, you still stand a chance of being able to test a wide variety of system features. This is better than testing only a few features in depth.

For further reading on testing, consult the following sources:

Beizer, Boris. *Software Testing Techniques.* New York: Van Nostrand Reinhold, 1983.

Black, Rex. *Critical Testing Processes: Plan, Prepare, Perform, Perfect.* Boston: Addison-Wesley, 2004.

McGregor, John D. and David A. Sykes. *A Practical Guide to Testing Object-Oriented Software.* Boston: Addison-Wesley, 2004.

Tamres, Louise. *Introducing Software Testing.* Boston: Addison-Wesley, 2002.