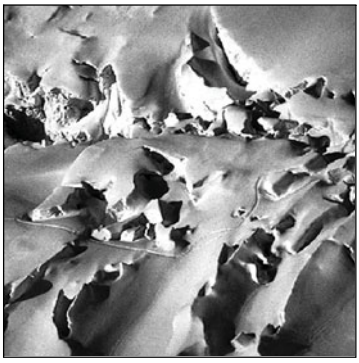
11



Testing

The software is done.
We are just trying to get it to work.
—Statement made in a Joint STARS E-8A FSD
Executive Program Review

Testing is the process of finding differences between the expected behavior specified by system models and the observed behavior of the implemented system. Unit testing finds differences between a specification of an object and its realization as a component. Structural testing finds differences between the system design model and a subset of integrated subsystems. Functional testing finds differences between the use case model and the system. Finally, performance testing finds differences between nonfunctional requirements and actual system performance. When differences are found, developers identify the defect causing the observed failure and modify the system to correct it. In other cases, the system model is identified as the cause of the difference, and the system model is updated to reflect the system.

From a modeling point of view, testing is the attempt to show that the implementation of the system is inconsistent with the system models. The goal of testing is to design tests that exercise defects in the system and to reveal problems. This activity is contrary to all other activities we described in previous chapters: analysis, design, implementation, communication, and negotiation are constructive activities. Testing, however, is aimed at breaking the system. Consequently, testing is usually accomplished by developers that were not involved with the construction of the system.

In this chapter, we first emphasize the importance of testing. We provide a bird's-eye view of the testing activities, we describe in more detail the concepts of fault, erroneous state, failure, and test, and then we describe the testing activities that result in the plan, design, and execution of tests. We introduce UML profiles as an extension mechanism to describe model-based testing. We conclude this chapter by discussing management issues related to testing.

11.1 Introduction: Testing The Space Shuttle

Testing is the process of analyzing a system or system component to detect the differences between specified (required) and observed (existing) behavior. Unfortunately, it is impossible to completely test a nontrivial system. First, testing is not decidable. Second, testing must be performed under time and budget constraints. As a result, systems are often deployed without being completely tested, leading to faults discovered by end users.

The first launch of the Space Shuttle Columbia in 1981, for example, was canceled because of a problem that was not detected during development. The problem was traced to a change made by a programmer two years earlier, who erroneously reset a delay factor from 50 to 80 milliseconds. This added a probability of 1/67 that any space shuttle launch would fail. Unfortunately, in spite of thousands of hours of testing after the change was made, the fault was not discovered during the testing phase. During the actual launch, the fault caused a synchronization problem with the shuttle's five on-board computers that led to the decision to abort the launch. The following is an excerpt of an article by Richard Feynman that describes the challenges of testing the Space Shuttle.

> In a total of about 250,000 seconds of operation, the engines have failed seriously perhaps 16 times. Engineering pays close attention to these failings and tries to remedy them as quickly as possible. It does this by test studies on special rigs experimentally designed for the flaws in question, by careful inspection of the engine for suggestive clues (like cracks), and by considerable study and analysis. . . .
>
> The usual way that such engines are tested (for military or civilian aircraft) may be called the component system, or bottom-up test. First it is necessary to thoroughly understand the properties and limitations of the materials to be used (for turbine blades, for example), and tests are begun in experimental rigs to determine those. With this knowledge larger component parts (such as bearings) are designed and tested individually. As deficiencies and design errors are noted they are corrected and verified with further testing. Since one tests only one component at a time, these tests and modifications are not overly expensive. Finally one works up to the final design of the entire engine, to the necessary specifications. There is a good chance, by this time that the engine will generally succeed, or that any failures are easily isolated and analyzed because the failure modes, limitations of materials, etc., are so well understood. There is a very good chance that the modifications to the engine to get around the final difficulties are not very hard to make, for most of the serious problems have already been discovered and dealt with in the earlier, less expensive, stages of the process.
>
> The Space Shuttle Main Engine was handled in a different manner, top down, we might say. The engine was designed and put together all at once with relatively little detailed preliminary study of the material and components. Then when troubles are found in the bearings, turbine blades, coolant pipes, etc., it is more expensive and difficult to discover the causes and make changes. For example, cracks have been found in the turbine blades of the high-pressure oxygen turbopump. Are they caused by flaws in the material, the effect of the oxygen atmosphere on the properties of the material, the thermal stresses of start-up or shutdown, the vibration and stresses of steady running, or mainly at some resonance at certain speeds, etc.? How long can we run from crack initiation to crack failure, and how does this depend on power level? Using the completed engine as a test bed to resolve such questions is extremely expensive. One does not wish to lose an entire engine in order to find out where and how failure occurs.

Yet, an accurate knowledge of this information is essential to acquire a confidence in the engine reliability in use. Without detailed understanding, confidence can not be attained. A further disadvantage of the top-down method is that, if an understanding of a fault is obtained, a simple fix, such as a new shape for the turbine housing, may be impossible to implement without a redesign of the entire engine.

The Space Shuttle Main Engine is a very remarkable machine. It has a greater ratio of thrust to weight than any previous engine. It is built at the edge of, or outside of, previous engineering experience. Therefore, as expected, many different kinds of flaws and difficulties have turned up. Because, unfortunately, it was built in the top-down manner, they are difficult to find and fix. The design aim of a lifetime of 55 missions' equivalent firings (27,000 seconds of operation, either in a mission of 500 seconds, or on a test stand) has not been obtained. The engine now requires very frequent maintenance and replacement of important parts, such as turbopumps, bearings, sheet metal housings, etc. The high-pressure fuel turbopump had to be replaced every three or four mission equivalents (although that may have been fixed, now) and the high-pressure oxygen turbopump every five or six. This is at most ten percent of the original specification.

Feynman's article[1] gives us an idea of the problems associated with testing complex systems. Even though the space shuttle is an extremely complex hardware and software system, the testing challenges are the same for any complex system.

Testing is often viewed as a job that can be done by beginners. Managers would assign the new members to the testing team, because the experienced people detested testing or are needed for the more important jobs of analysis and design. Unfortunately, such an attitude leads to many problems. To test a system effectively, a tester must have a detailed understanding of the whole system, ranging from the requirements to system design decisions and implementation issues. A tester must also be knowledgeable of testing techniques and apply these techniques effectively and efficiently to meet time, budget, and quality constraints.

Section 11.2 takes a bird's-eye view of testing. Section 11.3 defines in more detail the model elements related to testing, including faults, their manifestation, and their relationship to testing. Section 11.4 describes the testing activities found in the development process, including unit testing, which focuses on finding faults in a single component, and integration and system testing, which focus on finding faults in combination of components and in the complete system, respectively. We also discuss testing activities that focus on nonfunctional requirements, such as usability, performance, and stress tests. Section 11.4 concludes with the activities of field testing and installation testing. Section 11.5 discusses management issues related to testing.

1. Feynman [Feynman, 1988] wrote this article while he was a member of the Presidential Commission investigating the explosion of the Space Shuttle Challenger in January 1985. The cause of the accident was traced to an erosion of the O-rings in the solid rocket boosters. In addition to the testing problems of the main shuttle engine and the solid rocket boosters, the article mentions the phenomenon of gradually changing critical testing acceptance criteria and problems resulting from miscommunication between management and developers typically found in hierarchical organizations.

11.2 An Overview of Testing

Reliability is a measure of success with which the observed behavior of a system conforms to the specification of its behavior. **Software reliability** is the probability that a software system will not cause system failure for a specified time under specified conditions [IEEE Std. 982.2-1988]. **Failure** is any deviation of the observed behavior from the specified behavior. An **erroneous state** (also called an *error*) means the system is in a state such that further processing by the system will lead to a failure, which then causes the system to deviate from its intended behavior. A **fault**, also called "defect" or "bug," is the mechanical or algorithmic cause of an erroneous state. The goal of testing is to maximize the number of discovered faults, which then allows developers to correct them and increase the reliability of the system.

We define **testing** as the systematic attempt to *find faults in a planned way* in the implemented software. Contrast this definition with another common one: "testing is the process of demonstrating that *faults are not present*." The distinction between these two definitions is important. Our definition does not mean that we simply demonstrate that the program does what it is intended to do. The explicit goal of testing is to demonstrate the presence of faults and non-optimal behavior. Our definition implies that the developers are willing to dismantle things. Moreover, for the most part, demonstrating that faults are not present is not possible in systems of any realistic size.

Most activities of the development process are constructive: during analysis, design, and implementation, objects and relationships are identified, refined, and mapped onto a computer environment. Testing requires a different thinking, in that developers try to detect faults in the system, that is, differences between the reality of the system and the requirements. Many developers find this difficult to do. One reason is the way we use the word "success" during testing. Many project managers call a test case "successful" if it does not find a fault; that is, they use the second definition of testing during development. However, because "successful" denotes an achievement, and "unsuccessful" means something undesirable, these words should not be used in this fashion during testing.

In this chapter, we treat testing as an activity based on the falsification of system models, which is based on Popper's falsification of scientific theories [Popper, 1992]. According to Popper, when testing a scientific hypothesis, the goal is to design experiments that falsify the underlying theory. If the experiments are unable to break the theory, our confidence in the theory is strengthened and the theory is adopted (until it is eventually falsified). Similarly, in software testing, the goal is to identify faults in the software system (to falsify the theory). If none of the tests have been able to falsify software system behavior with respect to the requirements, it is ready for delivery. In other words, a software system is released when the falsification attempts (tests) show a certain level of confidence that the software system does what it is supposed to do.

There are many techniques for increasing the reliability of a software system:

- **Fault avoidance** techniques try to detect faults statically, that is, without relying on the execution of any of the system models, in particular the code model. Fault avoidance tries to prevent the insertion of faults into the system before it is released. Fault

avoidance includes development methodologies, configuration management, and verification.

- **Fault detection** techniques, such as debugging and testing, are uncontrolled and controlled experiments, respectively, used during the development process to identify erroneous states and find the underlying faults before releasing the system. Fault detection techniques assist in finding faults in systems, but do not try to recover from the failures caused by them. In general, fault detection techniques are applied during development, but in some cases they are also used after the release of the system. The blackboxes in an airplane to log the last few minutes of a flight is an example of a fault detection technique.
- **Fault tolerance** techniques assume that a system can be released with faults and that system failures can be dealt with by recovering from them at runtime. For example, modular redundant systems assign more than one component with the same task, then compare the results from the redundant components. The space shuttle has five onboard computers running two different pieces of software to accomplish the same task.

In this chapter, we focus on fault detection techniques, including reviews and testing. A **review** is the manual inspection of parts or all aspects of the system without actually executing the system. There are two types of reviews: walkthrough and inspection. In a code **walkthrough**, the developer informally presents the API (Application Programmer Interface), the code, and associated documentation of the component to the review team. The review team makes comments on the mapping of the analysis and object design to the code using use cases and scenarios from the analysis phase. An **inspection** is similar to a walkthrough, but the presentation of the component is formal. In fact, in a code inspection, the developer is not allowed to present the artifacts (models, code, and documentation). This is done by the review team, which is responsible for checking the interface and code of the component against the requirements. It also checks the algorithms for efficiency with respect to the nonfunctional requirements. Finally, it checks comments about the code and compares them with the code itself to find inaccurate and incomplete comments. The developer is only present in case the review needs clarifications about the definition and use of data structures or algorithms. Code reviews have proven to be effective at detecting faults. In some experiments, up to 85 percent of all identified faults were found in code reviews [Fagan, 1976], [Jones, 1977], [Porter et al., 1997].

Debugging assumes that faults can be found by starting from an unplanned failure. The developer moves the system through a succession of states, ultimately arriving at and identifying the erroneous state. Once this state has been identified, the algorithmic or mechanical fault causing this state must be determined. There are two types of debugging: The goal of correctness debugging is to find any deviation between observed and specified functional requirements. Performance debugging addresses the deviation between observed and specified nonfunctional requirements, such as response time.

Testing is a fault detection technique that tries to create failures or erroneous states in a planned way. This allows the developer to detect failures in the system before it is released to the customer. Note that this definition of testing implies that a successful test is a test that identifies faults. We will use this definition throughout the development phases. Another often-used definition of testing is that "it demonstrates that faults are not present." We will use this definition only after the development of the system when we try to demonstrate that the delivered system fulfills the functional and nonfunctional requirements.

If we used this second definition all the time, we would tend to select test data that have a low probability of causing the program to fail. If, on the other hand, the goal is to demonstrate that a program has faults, we tend to look for test data with a higher probability of finding faults. The characteristic of a good test model is that it contains test cases that identify faults. Tests should include a broad range of input values, including invalid inputs and boundary cases, otherwise, faults may not be detected. Unfortunately, such an approach requires extremely lengthy testing times for even small systems.

Figure 11-1 depicts an overview of testing activities:

- **Test planning** allocates resources and schedules the testing. This activity should occur early in the development phase so that sufficient time and skill is dedicated to testing. For example, developers can design test cases as soon as the models they validate become stable.
- **Usability testing** tries to find faults in the user interface design of the system. Often, systems fail to accomplish their intended purpose simply because their users are confused by the user interface and unwillingly introduce erroneous data.
- **Unit testing** tries to find faults in participating objects and/or subsystems with respect to the use cases from the use case model.
- **Integration testing** is the activity of finding faults by testing individual components in combination. **Structural testing** is the culmination of integration testing involving all components of the system. Integration tests and structural tests exploit knowledge from the SDD (System Design Document) using an integration strategy described in the Test Plan (TP).
- **System testing** tests all the components together, seen as a single system to identify faults with respect to the scenarios from the problem statement and the requirements and design goals identified in the analysis and system design, respectively:
  - **Functional testing** tests the requirements from the RAD and the user manual.
  - **Performance testing** checks the nonfunctional requirements and additional design goals from the SDD. Functional and performance testing are done by developers.
  - **Acceptance testing** and **installation testing** check the system against the project agreement and is done by the client, if necessary, with help by the developers.

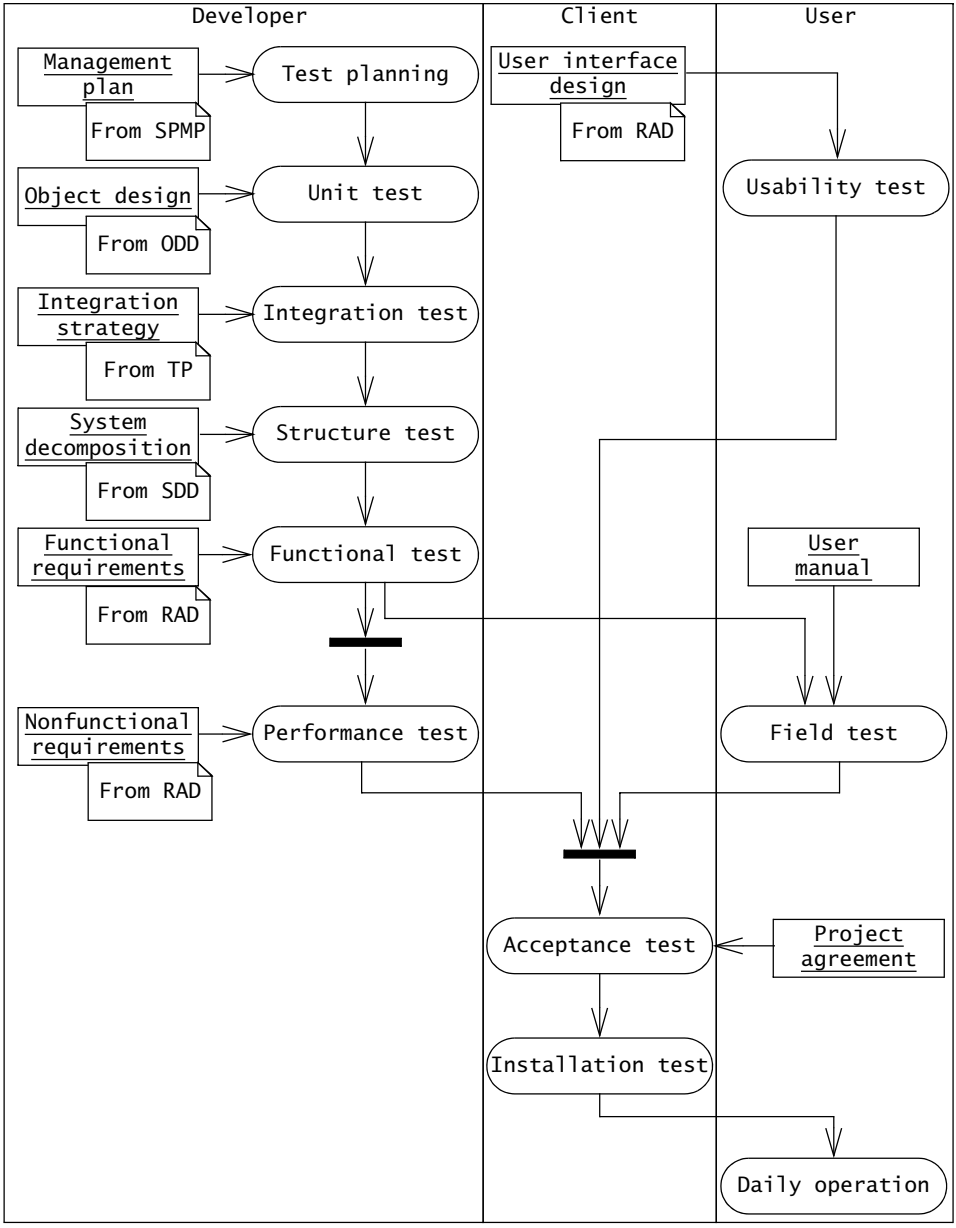


Figure 11-1 Testing activities and their related work products (UML activity diagram). Swimlanes indicate who executes the test.

11.3 Testing Concepts

In this section, we present the model elements used during testing (Figure 11-2):

- A **test component** is a part of the system that can be isolated for testing. A component can be an object, a group of objects, or one or more subsystems.
- A **fault**, also called *bug* or *defect*, is a design or coding mistake that may cause abnormal component behavior.
- An **erroneous state** is a manifestation of a fault during the execution of the system. An erroneous state is caused by one or more faults and can lead to a failure.
- A **failure** is a deviation between the specification and the actual behavior. A failure is triggered by one or more erroneous states. Not all erroneous states trigger a failure.[2]
- A **test case** is a set of inputs and expected results that exercises a test component with the purpose of causing failures and detecting faults.
- A **test stub** is a partial implementation of components on which the tested component depends. A **test driver** is a partial implementation of a component that depends on the test component. Test stubs and drivers enable components to be isolated from the rest of the system for testing.

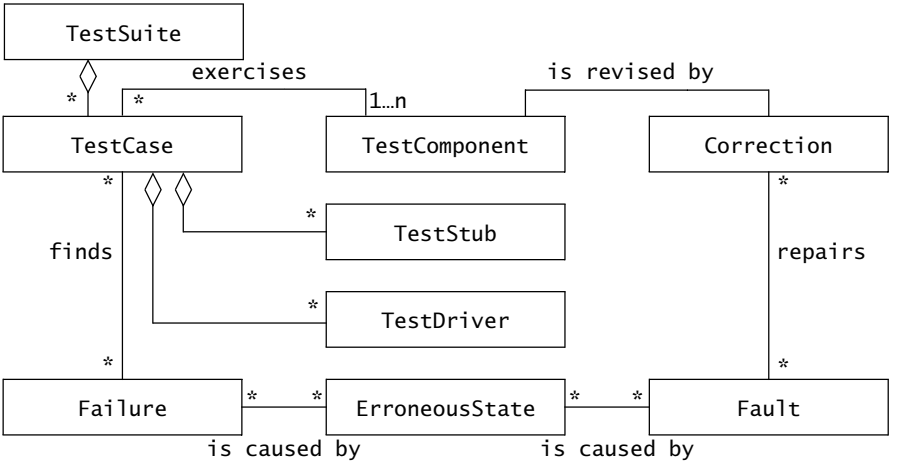


Figure 11-2 Model elements used during testing (UML class diagram).

2. Note that, outside the testing community, developers often do not distinguish between faults, failures, and erroneous states, and instead, refer to all three concepts as "errors."

- A **correction** is a change to a component. The purpose of a correction is to repair a fault. Note that a correction can introduce new faults.

11.3.1 Faults, Erroneous States, and Failures

With the initial understanding of the terms from the definitions in Section 11.3, let's take a look at Figure 11-3. What do you see? Figure 11-3 shows a pair of tracks that are not aligned with each other. If we envision a train running over the tracks, it would crash (fail). However, the figure actually does not present a failure, nor an erroneous state, nor a fault. It does not show a failure, because the expected behavior has not been specified, nor is there any observed behavior. Figure 11-3 also does not show an erroneous state, because that would mean that the system is in a state that further processing will lead to a failure. We only see tracks here; no moving train is shown. To speak about erroneous state, failure, or fault, we need to compare the desired behavior (described in the use case in the RAD) with the observed behavior (described by the test case). Assume that we have a use case with a train moving from the upper left track to the lower right track (Figure 11-4).

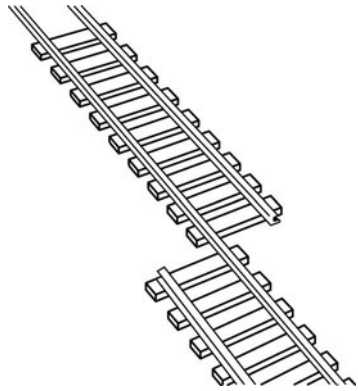


Figure 11-3 An example of a fault. The desired behavior is that the train remain on the tracks.

Use case name	`DriveTrain`
Participating actor	`TrainOperator`
Entry condition	`TrainOperator` pushes the "StartTrain" button at the control panel.
Flow of events	1. The train starts moving on track 1. 2. The train transitions to track 2.
Exit condition	The train is running on track 2.

Figure 11-4 Use case `DriveTrain` specifying the expected behavior of the train.

We can then proceed to derive a test case that moves the train from the state described in the entry condition of the use case to a state where it will crash, namely when it is leaving the upper track (Figure 11-5).

Test-case identifier	`DriveTrain`
Test location	`http://www12.in.tum.de/TrainSystem/test-cases/test1`
Feature to be tested	Continuous operation of engine for 5 seconds
Feature Pass/Fail Criteria	The test passes if the train drives for 5 seconds and covers the length of at least two tracks.
Means of control	1. The `StartTrain()` method is called via a test driver `StartTrain` (contained in the same directory as the `DriveTrain` test).
Data	2. Direction of trip and duration are read from a input file `http://www12.in.tum.de/TrainSystem/test-cases/input`. 3. If debug is set to TRUE, then the test case will output the system messages "Enter Track n, Exit Track n" for each n, where n is the number of the current track.
Test Procedure	The test is started by double-clicking the test case at the specified location. The test will run without further intervention until completion. The test should take no more than 7 seconds.
Special requirements	The test stub `Engine` is needed for the test execution.

Figure 11-5 Test case `DriveTrain` for the use case described in Figure 11-4.

In other words, when executing this test case, we can demonstrate that the system contains a fault. Note that the current state shown in Figure 11-6 is erroneous, but does not show a failure.

The misalignment of the tracks can be a result of bad communication between the development teams (each track had to be positioned by one team) or because of a wrong implementation of the specification by one of the teams (Figure 11-7). Both of these are examples of algorithmic faults. You are probably already familiar with many other algorithmic faults that are introduced during the implementation phase. For example, "Exiting a loop too soon," "exiting a loop too late," "testing for the wrong condition," "forgetting to initialize a variable" are all implementation-specific algorithmic faults. Algorithmic faults can also occur during analysis and system design. Stress and overload problems, for example, are object design specific algorithmic faults that lead to failure when data structures are filled beyond their specified capacity. Throughput and performance failures are possible when a system does not perform at the speed specified by the nonfunctional requirements.

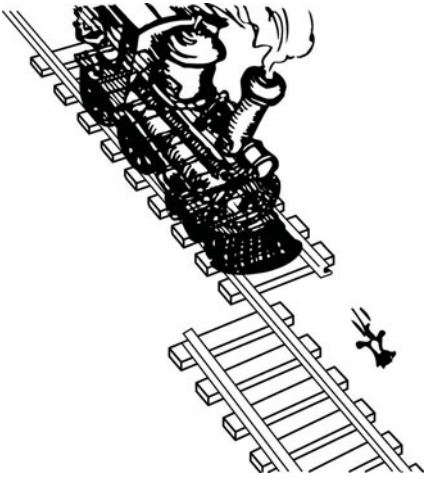


Figure 11-6 An example of an erroneous state.

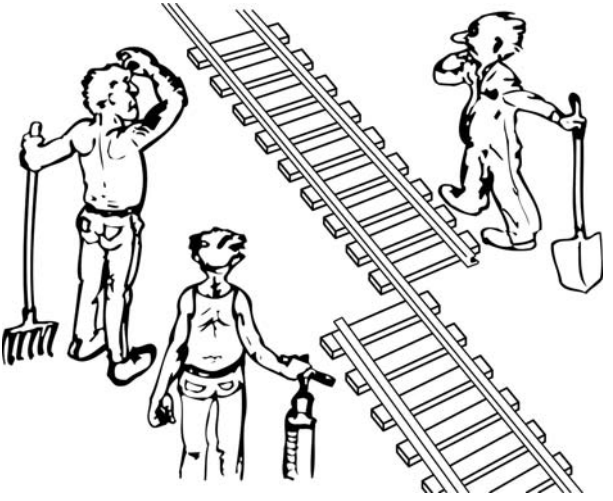


Figure 11-7 A fault can have an algorithmic cause.

Even if the tracks are implemented according to the specification in the RAD, they could still end up misaligned during daily operation, for example, if an earthquake happens that moves the underlying soil (Figure 11-8).

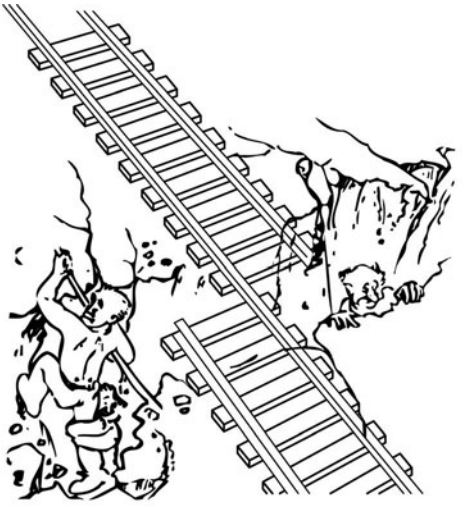


Figure 11-8 A fault can have a mechanical cause, such as an earthquake.

A fault in the virtual machine of a software system is another example of a mechanical fault: even if the developers have implemented correctly, that is, they have mapped the object model correctly onto the code, the observed behavior can still deviate from the specified behavior. In concurrent engineering projects, for example, where hardware is developed in parallel with software, we cannot always make the assumption that the virtual machine executes as specified. Other examples of mechanical faults are power failures. Note the relativity of the terms "fault" and "failure" with respect to a particular system component: the failure in one system component (the power system) is the mechanical fault that can lead to failure in another system component (the software system).

11.3.2 Test Cases

A **test case** is a set of input data and expected results that exercises a component with the purpose of causing failures and detecting faults. A test case has five attributes: `name`, `location`, `input`, `oracle`, and `log` (Table 11-1). The `name` of the test case allows the tester to distinguish between different test cases. A heuristic for naming test cases is to derive the name from the requirement it is testing or from the component being tested. For example, if you are testing a use case `Deposit()`, you might want to call the test case `Test_Deposit`. If a test case involves two components A and B, a good name would be `Test_AB`. The `location` attribute describes where the test case can be found. It should be either the path name or the URL to the executable of the test program and its inputs.

Table 11-1 Attributes of the class `TestCase`.

Attributes	Description
`name`	Name of test case
`location`	Full path name of executable
`input`	Input data or commands
`oracle`	Expected test results against which the output of the test is compared
`log`	Output produced by the test

`Input` describes the set of input data or commands to be entered by the actor of the test case (which can be the tester or a test driver). The expected behavior of the test case is the sequence of output data or commands that a correct execution of the test should yield. The expected behavior is described by the `oracle` attribute. The `log` is a set of time-stamped correlations of the observed behavior with the expected behavior for various test runs.

Once test cases are identified and described, relationships among test cases are identified. Aggregation and the `precede` associations are used to describe the relationships between the test cases. Aggregation is used when a test case can be decomposed into a set of subtests. Two test cases are related via the `precede` association when one test case must precede another test case.

Figure 11-9 shows a test model where `TestA` must precede `TestB` and `TestC`. For example, `TestA` consists of `TestA1` and `TestA2`, meaning that once `TestA1` and `TestA2` are tested, `TestA` is tested; there is no separate test for `TestA`. A good test model has as few associations as possible, because tests that are not associated with each other can be executed independently from each other. This allows a tester to speed up testing, if the necessary testing resources are available. In Figure 11-9, `TestB` and `TestC` can be tested in parallel, because there is no relation between them.

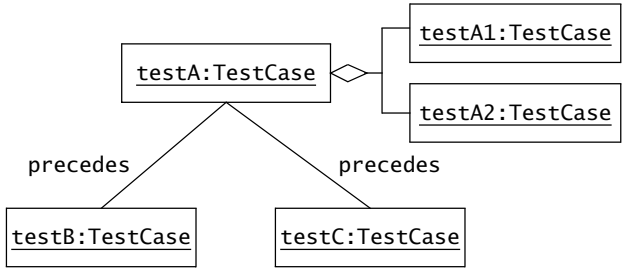


Figure 11-9 Test model with test cases. `TestA` consists of two tests, `TestA1` and `TestA2`. `TestB` and `TestC` can be tested independently, but only after `TestA` has been performed.

Test cases are classified into blackbox tests and whitebox tests, depending on which aspect of the system model is tested. **Blackbox tests** focus on the input/output behavior of the component. Blackbox tests do not deal with the internal aspects of the component, nor with the behavior or the structure of the components. **Whitebox tests** focus on the internal structure of the component. A whitebox test makes sure that, independently from the particular input/output behavior, every state in the dynamic model of the object and every interaction among the objects is tested. As a result, whitebox testing goes beyond blackbox testing. In fact, most of the whitebox tests require input data that could not be derived from a description of the functional requirements alone. Unit testing combines both testing techniques: blackbox testing to test the functionality of the component, and whitebox testing to test structural and dynamic aspects of the component.

11.3.3 Test Stubs and Drivers

Executing test cases on single components or combinations of components requires the tested component to be isolated from the rest of the system. Test drivers and test stubs are used to substitute for missing parts of the system. A **test driver** simulates the part of the system that calls the component under test. A test driver passes the test inputs identified in the test case analysis to the component and displays the results.

A **test stub** simulates a component that is called by the tested component. The test stub must provide the same API as the method of the simulated component and must return a value compliant with the return result type of the method's type signature. Note that the interface of all components must be baselined. If the interface of a component changes, the corresponding test drivers and stubs must change as well.

The implementation of test stubs is a nontrivial task. It is not sufficient to write a test stub that simply prints a message stating that the test stub was called. In most situations, when component A calls component B, A is expecting B to perform some work, which is then returned as a set of result parameters. If the test stub does not simulate this behavior, A will fail, not because of a fault in A, but because the test stub does not simulate B correctly.

Even providing a return value is not always sufficient. For example, if a test stub always returns the same value, it might not return the value expected by the calling component in a particular scenario. This can produce confusing results and even lead to the failure of the calling component, even though it is correctly implemented. Often, there is a trade-off between implementing accurate test stubs and substituting the test stubs by the actual component. For many components, drivers and stubs are often written after the component is completed, and for components that are behind schedule, stubs are often not written at all.

To ensure that stubs and drivers are developed and available when needed, several development methods stipulate that drivers be developed for every component. This results in lower effort because it provides developers the opportunity to find problems with the interface specification of the component under test before it is completely implemented.

11.3.4 Corrections

Once tests have been executed and failures have been detected, developers change the component to eliminate the suspected faults. A **correction** is a change to a component whose purpose is to repair a fault. Corrections can range from a simple modification to a single component, to a complete redesign of a data structure or a subsystem. In all cases, the likelihood that the developer introduces new faults into the revised component is high. Several techniques can be used to minimize the occurrence of such faults:

- *Problem tracking* includes the documentation of each failure, erroneous state, and fault detected, its correction, and the revisions of the components involved in the change. Together with configuration management, problem tracking enables developers to narrow the search for new faults. We describe problem tracking in more detail in Chapter 13, *Configuration Management*.
- *Regression testing* includes the reexecution of all prior tests after a change. This ensures that functionality which worked before the correction has not been affected. Regression testing is important in object-oriented methods, which call for an iterative development process. This requires testing to be initiated earlier and for test suites to be maintained after each iteration. Regression testing unfortunately is costly, especially when part of the tests is not automated. We describe regression testing in more detail in Section 11.4.4.
- *Rationale maintenance* includes the documentation of the rationale for the change and its relationship with the rationale of the revised component. Rationale maintenance enables developers to avoid introducing new faults by inspecting the assumptions that were used to build the component. We describe rationale maintenance in Chapter 12, *Rationale Management*.

Next, let us describe in more detail the testing activities that lead to the creation of test cases, their execution, and the development of corrections.

11.4 Testing Activities

In this section, we describe the technical activities of testing. These include

- **Component inspection**, which finds faults in an individual component through the manual inspection of its source code (Section 11.4.1)
- **Usability testing**, which finds differences between what the system does and the users' expectation of what it should do (Section 11.4.2)
- **Unit testing**, which finds faults by isolating an individual component using test stubs and drivers and by exercising the component using test cases (Section 11.4.3)
- **Integration testing**, which finds faults by integrating several components together (Section 11.4.4)

- **System testing**, which focuses on the complete system, its functional and nonfunctional requirements, and its target environment (Section 11.4.5).

11.4.1 Component Inspection

Inspections find faults in a component by reviewing its source code in a formal meeting. Inspections can be conducted before or after the unit test. The first structured inspection process was Michael Fagan's inspection method [Fagan, 1976]. The inspection is conducted by a team of developers, including the author of the component, a moderator who facilitates the process, and one or more reviewers who find faults in the component. Fagan's inspection method consists of five steps:

- *Overview.* The author of the component briefly presents the purpose and scope of the component and the goals of the inspection.
- *Preparation*. The reviewers become familiar with the implementation of the component.
- *Inspection meeting*. A reader paraphrases the source code of the component, and the inspection team raises issues with the component. A moderator keeps the meeting on track.
- *Rework*. The author revises the component.
- *Follow-up*. The moderator checks the quality of the rework and may determine the component that needs to be reinspected.

The critical steps in this process are the preparation phase and the inspection meeting. During the preparation phase, the reviewers become familiar with the source code; they do not yet focus on finding faults. During the inspection meeting, the reader paraphrases the source code, that is, he reads each source code statement and explains what the statement should do. The reviewers then raise issues if they think there is a fault. Most of the time is spent debating whether or not a fault is present, but solutions to repair the fault are not explored at this point. During the overview phase of the inspection, the author states the objectives of the inspection. In addition to finding faults, reviewers may also be asked to look for deviations from coding standards or for inefficiencies.

Fagan's inspections are usually perceived as time-consuming because of the length of the preparation and inspection meeting phase. The effectiveness of a review also depends on the preparation of the reviewers. David Parnas proposed a revised inspection process, the active design review, which eliminates the inspection meeting of all inspection team members [Parnas & Weiss, 1985]. Instead, reviewers are asked to find faults during the preparation phase. At the end of the preparation phase, each reviewer fills out a questionnaire testing his or her understanding of the component. The author then meets individually with each reviewer to collect feedback on the component.

Both Fagan's inspections and the active design reviews have been shown to be usually more effective than testing in uncovering faults. Both testing and inspections are used in safety-critical projects, as they tend to find different types of faults.

11.4.2 Usability Testing

Usability testing tests the user's understanding of the system. Usability testing does not compare the system against a specification. Instead, it focuses on finding differences between the system and the users' expectation of what it should do. As it is difficult to define a formal model of the user against which to test, usability testing takes an empirical approach: participants representative of the user population find problems by manipulating the user interface or a simulation thereof. Usability tests are also concerned with user interface details, such as the look and feel of the user interface, the geometrical layout of the screens, sequence of interactions, and the hardware. For example, in case of a wearable computer, a usability test would test the ability of the user to issue commands to the system while lying in an awkward position, as in the case of a mechanic looking at a screen under a car while checking a muffler.

The technique for conducting usability tests is based on the classical approach for conducting a controlled experiment. Developers first formulate a set of test objectives, describing what they hope to learn in the test. These can include, for example, evaluating specific dimensions or geometrical layout of the user interface, evaluating the impact of response time on user efficiency, or evaluating whether the online help documentation is sufficient for novice users. The test objectives are then evaluated in a series of experiments in which participants are trained to accomplish predefined tasks (e.g., exercising the user interface feature under investigation). Developers observe the participants and collect data measuring user performance (e.g., time to accomplish a task, error rate) and preferences (e.g, opinions and thought processes) to identify specific problems with the system or collect ideas for improving it [Rubin, 1994].

There are two important differences between controlled experiments and usability tests. Whereas the classical experimental method is designed to refute a hypothesis, the goal of usability tests is to obtain qualitative information on how to fix usability problems and how to improve the system. The other difference is the rigor with which the experiments are performed. It has been shown that even a series of quick focused tests starting as early as requirements elicitation is extremely helpful. Nielsen uses the term *discount usability engineering* to refer to simplified usability tests that can be accomplished at a fraction of the time and cost of a full-blown study, noting that a few usability tests are better than none at all [Nielsen & Mack, 1994]. Examples of discount usability tests include using paper scenario mock-ups (as opposed to a videotaped scenario), relying on handwritten notes as opposed to analyzing audio tape transcripts, or using fewer subjects to elicit suggestions and uncover major defects (as opposed to achieving statistical significance and using quantitative measures).

There are three types of usability tests:

- **Scenario test**. During this test, one or more users are presented with a visionary scenario of the system. Developers identify how quickly users are able to understand the scenario, how accurately it represents their model of work, and how positively they react to the description of the new system. The selected scenarios should be as realistic and detailed as possible. A scenario test allows rapid and frequent feedback from the user. Scenario tests can be realized as paper mock-ups[3] or with a simple prototyping environment, which is often easier to learn than the programming environment used for development. The advantage of scenario tests is that they are cheap to realize and to repeat. The disadvantages are that the user cannot interact directly with the system and that the data are fixed.
- **Prototype test**. During this type of test, the end users are presented with a piece of software that implements key aspects of the system. A **vertical prototype** completely implements a use case through the system. Vertical prototypes are used to evaluate core requirements, for example, response time of the system or user behavior under stress. A **horizontal prototype** implements a single layer in the system; an example is a **user interface prototype**, which presents an interface for most use cases (without providing much or any functionality). User interface prototypes are used to evaluate issues such as alternative user interface concepts or window layouts. A **Wizard of Oz prototype** is a user interface prototype in which a human operator behind the scenes pulls the levers [Kelly, 1984]. Wizard of Oz prototypes are used for testing natural language applications, when the speech recognition or the natural language parsing subsystems are incomplete. A human operator intercepts user queries and rephrases them in terms that the system understands, without the test user being aware of the operator. The advantages of prototype tests are that they provide a realistic view of the system to the user and that prototypes can be instrumented to collect detailed data. However, prototypes require more effort to build than test scenarios.
- **Product test**. This test is similar to the prototype test except that a functional version of the system is used in place of the prototype. A product test can only be conducted after most of the system is developed. It also requires that the system be easily modifiable such that the results of the usability test can be taken into account.

In all three types of tests, the basic elements of usability testing include [Rubin, 1994]

- development of test objectives

3. Using storyboards, a technique from the feature animation industry, consists of sketching a sequence of pictures of the screen at different points in the scenario. The pictures of each scenario are then lined up chronologically against a wall on a board (hence the term "storyboard"). Developers and users walk around the room when reviewing and discussing the scenarios. Given a reasonably sized room, participants can deal with several hundreds of sketches.

- a representative sample of end users
- the actual or simulated work environment
- controlled, extensive interrogation, and probing of the users by the person performing the usability test
- collection and analysis of quantitative and qualitative results
- recommendations on how to improve the system.

Typical test objectives in a usability test address the comparison of two user interaction styles, the identification of the best and the worst features in a scenario or a prototype, the main stumbling blocks, the identification of useful features for novice and expert users, when help is needed, and what type of training information is required.

11.4.3 Unit Testing

Unit testing focuses on the building blocks of the software system, that is, objects and subsystems. There are three motivations behind focusing on these building blocks. First, unit testing reduces the complexity of overall test activities, allowing us to focus on smaller units of the system. Second, unit testing makes it easier to pinpoint and correct faults, given that few components are involved in the test. Third, unit testing allows parallelism in the testing activities; that is, each component can be tested independently of the others.

The specific candidates for unit testing are chosen from the object model and the system decomposition. In principle, all the objects developed during the development process should be tested, which is often not feasible because of time and budget constraints. The minimal set of objects to be tested should be the participating objects in use cases. Subsystems should be tested as components only after each of the classes within that subsystem have been tested individually.

Existing subsystems, which were reused or purchased, should be treated as components with unknown internal structure. This applies in particular to commercially available subsystems, where the internal structure is not known or available to the developer.

Many unit testing techniques have been devised. Below, we describe the most important ones: equivalence testing, boundary testing, path testing, and state-based testing.

Equivalence testing

This blackbox testing technique minimizes the number of test cases. The possible inputs are partitioned into equivalence classes, and a test case is selected for each class. The assumption of equivalence testing is that systems usually behave in similar ways for all members of a class. To test the behavior associated with an equivalence class, we only need to test one member of the class. Equivalence testing consists of two steps: identification of the equivalence classes and selection of the test inputs. The following criteria are used in determining the equivalence classes.

- *Coverage*. Every possible input belongs to one of the equivalence classes.

- *Disjointedness*. No input belongs to more than one equivalence class.
- *Representation*. If the execution demonstrates an erroneous state when a particular member of a equivalence class is used as input, then the same erroneous state can be detected by using any other member of the class as input.

For each equivalence class, at least two pieces of data are selected: a typical input, which exercises the common case, and an invalid input, which exercises the exception handling capabilities of the component. After all equivalence classes have been identified, a test input for each class has to be identified that covers the equivalence class. If there is a possibility that not all the elements of the equivalence class are covered by the test input, the equivalence class must be split into smaller equivalence classes, and test inputs must be identified for each of the new classes.

For example, consider a method that returns the number of days in a month, given the month and year (see Figure 11-10). The month and year are specified as integers. By convention, `1` represents the month of January, `2` the month of February, and so on. The range of valid inputs for the year is `0` to `maxInt`.

```
class MyGregorianCalendar {
...
    public static int getNumDaysInMonth(int month, int year) {…}
...
}
```

Figure 11-10 Interface for a method computing the number of days in a given month (in Java). The `getNumDaysInMonth()` method takes two parameters, a month and a year, both specified as integers.

We find three equivalence classes for the month parameter: months with 31 days (i.e., `1`, `3`, `5`, `7`, `8`, `10`, `12`), months with 30 days (i.e., `4`, `6`, `9`, `11`), and February, which can have 28 or 29 days. Nonpositive integers and integers larger than `12` are invalid values for the month parameter. Similarly, we find two equivalence classes for the year: leap years and non–leap years. By specification, negative integers are invalid values for the year. First we select one valid value for each equivalence class (e.g., February, June, July, 1901, and 1904). Given that the return value of the `getNumDaysInMonth()` method depends on both parameters, we combine these values to test for interaction, resulting in the six equivalence classes displayed in Table 11-2.

Boundary testing

This special case of equivalence testing focuses on the conditions at the boundary of the equivalence classes. Rather than selecting any element in the equivalence class, boundary testing requires that the elements be selected from the "edges" of the equivalence class. The assumption

Table 11-2 Equivalence classes and selected valid inputs for testing the `getNumDaysInMonth()` method.

Equivalence class	Value for month input	Value for year input
Months with 31 days, non–leap years	`7 (July)`	`1901`
Months with 31 days, leap years	`7 (July)`	`1904`
Months with 30 days, non–leap years	`6 (June)`	`1901`
Month with 30 days, leap year	`6 (June)`	`1904`
Month with 28 or 29 days, non–leap year	`2 (February)`	`1901`
Month with 28 or 29 days, leap year	`2 (February)`	`1904`

behind boundary testing is that developers often overlook special cases at the boundary of the equivalence classes (e.g., 0, empty strings, year 2000).

In our example, the month of February presents several boundary cases. In general, years that are multiples of 4 are leap years. Years that are multiples of 100, however, are not leap years, unless they are also multiple of 400. For example, 2000 was a leap year, whereas 1900 was not. Both year 1900 and 2000 are good boundary cases we should test. Other boundary cases include the months 0 and 13, which are at the boundaries of the invalid equivalence class. Table 11-3 displays the additional boundary cases we selected for the `getNumDaysInMonth()` method.

A disadvantage of equivalence and boundary testing is that these techniques do not explore combinations of test input data. In many cases, a program fails because a combination of certain values causes the erroneous fault. Cause-effect testing addresses this problem by establishing logical relationships between input and outputs or inputs and transformations. The inputs are called causes, the outputs or transformations are effects. The technique is based on the premise that the input/output behavior can be transformed into a Boolean function. For details on this technique and another technique called "error guessing," we refer you to the literature on testing (for example [Myers, 1979]).

Table 11-3 Additional boundary cases selected for the `getNumDaysInMonth()` method.

Equivalence class	Value for month input	Value for year input
Leap years divisible by 400	`2 (February)`	`2000`
Non–leap years divisible by 100	`2 (February)`	`1900`
Nonpositive invalid months	`0`	`1291`
Positive invalid months	`13`	`1315`

Path testing

This whitebox testing technique identifies faults in the implementation of the component. The assumption behind path testing is that, by exercising all possible paths through the code at least once, most faults will trigger failures. The identification of paths requires knowledge of the source code and data structures. The starting point for path testing is the flow graph. A flow graph consists of nodes representing executable blocks and edges representing flow of control. A flow graph is constructed from the code of a component by mapping decision statements (e.g., `if` statements, `while` loops) to nodes. Statements between each decision (e.g., `then` block, `else` block) are mapped to other nodes. Associations between each node represent the precedence relationships. Figure 11-11 depicts an example of a *faulty* implementation of the `getNumDaysInMonth()` method. Figure 11-12 depicts the equivalent flow graph as a UML

```
public class MonthOutOfBounds extends Exception {…};
public class YearOutOfBounds extends Exception {…};

class MyGregorianCalendar {
    public static boolean isLeapYear(int year) {
        boolean leap;
        if ((year%4) == 0){
            leap = true;
        } else {
            leap = false;
        }
        return leap;
    }
    public static int getNumDaysInMonth(int month, int year)
            throws MonthOutOfBounds, YearOutOfBounds {
        int numDays;
        if (year < 1) {
            throw new YearOutOfBounds(year);
        }
        if (month == 1 || month == 3 || month == 5 || month == 7 ||
                month == 10 || month == 12) {
            numDays = 32;
        } else if (month == 4 || month == 6 || month == 9 || month == 11) {
            numDays = 30;
        } else if (month == 2) {
            if (isLeapYear(year)) {
                numDays = 29;
            } else {
                numDays = 28;
            }
        } else {
            throw new MonthOutOfBounds(month);
        }
        return numDays;
    }
}
```

Figure 11-11 An example of a (faulty) implementation of the `getNumDaysInMonth()` method (Java).

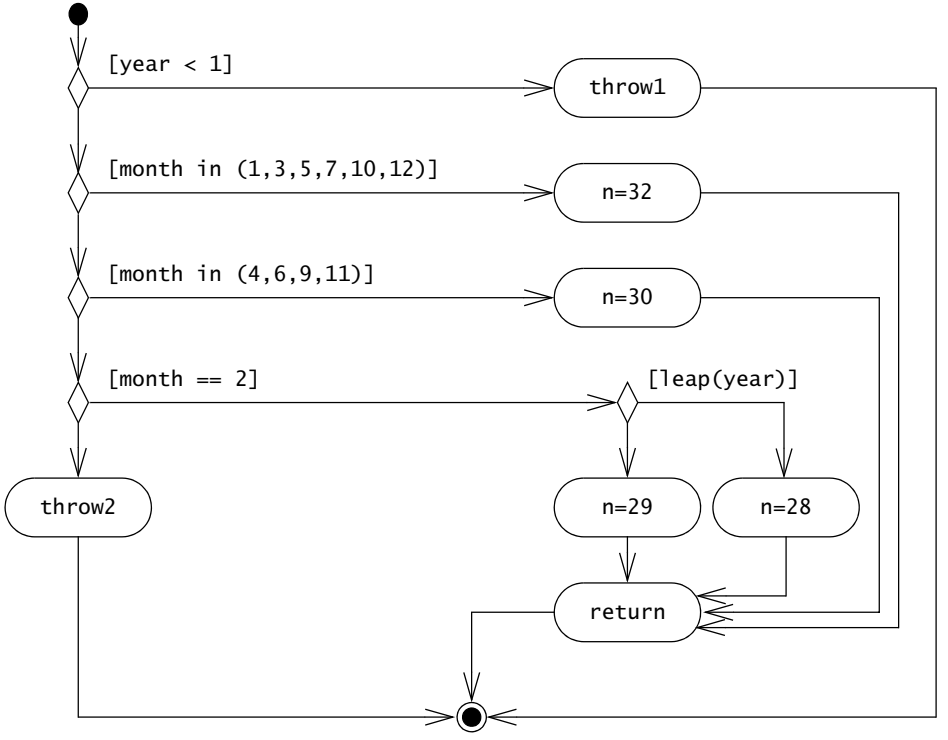


Figure 11-12 Equivalent flow graph for the (faulty) implementation of the `getNumDaysInMonth()` method of Figure 11-11 (UML activity diagram).

activity diagram. In this example, we model decisions with UML branches, blocks with UML actions, and control flow with UML transitions.

Complete path testing consists of designing test cases such that each edge in the activity diagram is traversed at least once. This is done by examining the condition associated with each branch point and selecting an input for the `true` branch and another input for the `false` branch. For example, examining the first branch point in Figure 11-12, we select two inputs: `year=0` (such that `year < 1` is true) and `year=1901` (such that `year < 1` is false). We then repeat the process for the second branch and select the inputs `month=1` and `month=2`. The input `(year=0, month=1)` produces the path `{throw1}`. The input `(year=1901, month=1)` produces a second path `{n=32 return}`, which uncovers one of the faults in the `getNumDaysInMonth()` method. By repeating this process for each node, we generate the test cases depicted in Table 11-4.

We can similarly construct the activity diagram for the method `isLeapYear()` and derive test cases to exercise the single branch point of this method (Figure 11-13). Note that the test case `(year = 1901, month = 2)` of the `getNumDaysInMonth()` method already exercises one of

Table 11-4 Test cases and their corresponding path for the activity diagram depicted in Figure 11-12.

Test case	Path
(year = 0, month = 1)	{throw1}
(year = 1901, month = 1)	{n=32 return}
(year = 1901, month = 2)	{n=28 return}
(year = 1904, month = 2)	{n=29 return}
(year = 1901, month = 4)	{n=30 return}
(year = 1901, month = 0)	{throw2}

the paths of the `isLeapYear()` method. By systematically constructing tests to cover all the paths of all methods, we can deal with the complexity associated with a large number of methods.

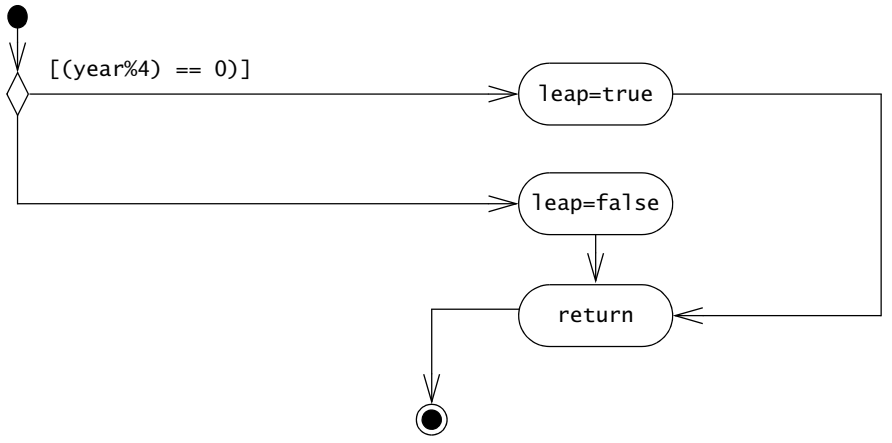
Using graph theory, it can be shown that the minimum number of tests necessary to cover all edges is equal to the number of independent paths through the flow graph [McCabe, 1976]. This is defined as the *cyclomatic complexity* `CC` of the flow graph, which is

```
CC = number of edges - number of nodes + 2
```

where the number of nodes is the number of branches and actions, and the number of edges is the number of transitions in the activity diagram. The cyclomatic complexity of the `getNumDaysInMonth()` method is 6, which is also the number of test cases we found in Table 11-4. Similarly, the cyclomatic complexity of the `isLeapYear()` method and the number of derived test cases is 2.

By comparing the test cases we derived from the equivalence classes (Table 11-2) and boundary cases (Table 11-3) with the test cases we derived from the flow graph (Table 11-4 and Figure 11-13), several differences can be noted. In both cases, we test the method extensively for computations involving the month of February. However, because the implementation of `isLeapYear()` does not take into account years divisible by 100, path testing did not generate any test case for this equivalence class.

In general, path testing and whitebox methods can detect only faults resulting from exercising a path in the program, such as the faulty `numDays=32` statement. Whitebox testing methods cannot detect omissions, such as the failure to handle the non–leap year 1900. Path testing is also heavily based on the control structure of the program; faults associated with violating invariants of data structures, such as accessing an array out of bounds, are not explicitly addressed. However, no testing method short of exhaustive testing can guarantee the discovery of all faults. In our example, neither equivalence testing or path testing uncovered the fault associated with the month of August.



Test case	Path
year = 1901, month = 2	*leap=false return*
year = 1904, month=2	leap=true return

Figure 11-13 Equivalent flow graph for the (faulty) `isLeapYear()` method implementation of Figure 11-11 (UML activity diagram) and derived tests. The test in *italic* is redundant with a test we derived for the `getNumDaysInMonth()` method.

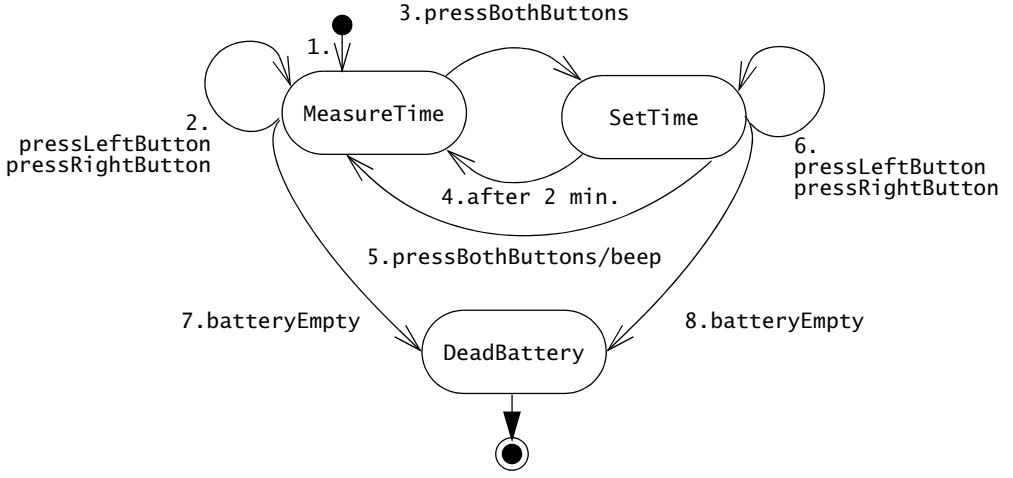
State-based testing

This testing technique was recently developed for object-oriented systems [Turner & Robson, 1993]. Most testing techniques focus on selecting a number of test inputs for a given state of the system, exercising a component or a system, and comparing the observed outputs with an oracle. State-based testing, however, compares the resulting state of the system with the expected state. In the context of a class, state-based testing consists of deriving test cases from the UML state machine diagram for the class. For each state, a representative set of stimuli is derived for each transition (similar to equivalence testing). The attributes of the class are then instrumented and tested after each stimuli has been applied to ensure that the class has reached the specified state.

For example, Figure 11-14 depicts a state machine diagram and its associated tests for the `2Bwatch` we described in Chapter 2, *Modeling with UML*. It specifies which stimuli change the watch from the high-level state `MeasureTime` to the high-level state `SetTime`. It does not show the low-level states of the watch when the date and time change, either because of actions of the user or because of time passing. The test inputs in Figure 11-14 were generated such that each transition is traversed at least once. After each input, instrumentation code checks if the watch is in the predicted state and reports a failure otherwise. Note that some transitions (e.g., transition 3) are traversed several times, as it is necessary to put the watch back into the `SetTime` state

(e.g., to test transitions 4, 5, and 6). Only the first eight stimuli are displayed. The test inputs for the `DeadBattery` state were not generated.

Currently, state-based testing presents several difficulties. Because the state of a class is encapsulated, test cases must include sequences for putting classes in the desired state before given transitions can be tested. State-based testing also requires the instrumentation of class attributes. Although state-based testing is currently not part of the state of the practice, it promises to become an effective testing technique for object-oriented systems as soon as proper automation is provided.



Stimuli	Transition tested	Predicted resulting state
Empty set	1. *Initial transition*	`MeasureTime`
Press left button	2.	`MeasureTime`
Press both buttons simultaneously	3.	`SetTime`
Wait 2 minutes	4. *Timeout*	`MeasureTime`
Press both buttons simultaneously	3. *Put the system into the SetTime state to test the next transition.*	`SetTime`
Press both buttons simultaneously	5.	`SetTime->MeasureTime`
Press both buttons simultaneously	3. *Put the system into the SetTime state to test the next transition.*	`SetTime`
Press left button	6. Loop back onto `MeasureTime`	`MeasureTime`

Figure 11-14 UML state machine diagram and resulting tests for `2Bwatch SetTime` use case. Only the first eight stimuli are shown.

Polymorphism testing

Polymorphism introduces a new challenge in testing because it enables messages to be bound to different methods based on the class of the target. Although this enables developers to reuse code across a larger number of classes, it also introduces more cases to test. All possible bindings should be identified and tested [Binder, 2000].

Consider the `NetworkInterface` Strategy design pattern that we introduced in Chapter 8, *Object Design: Reusing Pattern Solutions* (see Figure 11-15). The Strategy design pattern uses polymorphism to shield the context (i.e., the `NetworkConnection` class) from the concrete strategy (i.e., the `Ethernet`, `WaveLAN`, and `UMTS` classes). For example, the `NetworkConnection.send()` method calls the `NetworkInterface.send()` method to send bytes across the current `NetworkInterface`, regardless of the actual concrete strategy. This means that, at run time, the `NetworkInterface.send()` method invocation can be bound to one of three methods, `Ethernet.send()`, `WaveLAN.send()`, `UMTS.send()`.

When applying the path testing technique to an operation that uses polymorphism, we need to consider all dynamic bindings, one for each message that could be sent. In `NetworkConnect.send()` in the left column of Figure 11-16, we invoke the `NetworkInterface.send()` operation, which can be bound to either `Ethernet.send()`, `WaveLAN.send()`, or the `UMTS.send()` methods, depending on the class of the `nif` object. To deal with this situation explicitly, we expand the original source code by replacing each invocation of `NetworkInterface.send()` with a nested `if else` statement that tests for all subclasses of`NetworkInterface` (right column of Figure 11-16). Depending on the class, `nif` is cast into the appropriate concrete class, and the associated method is invoked.

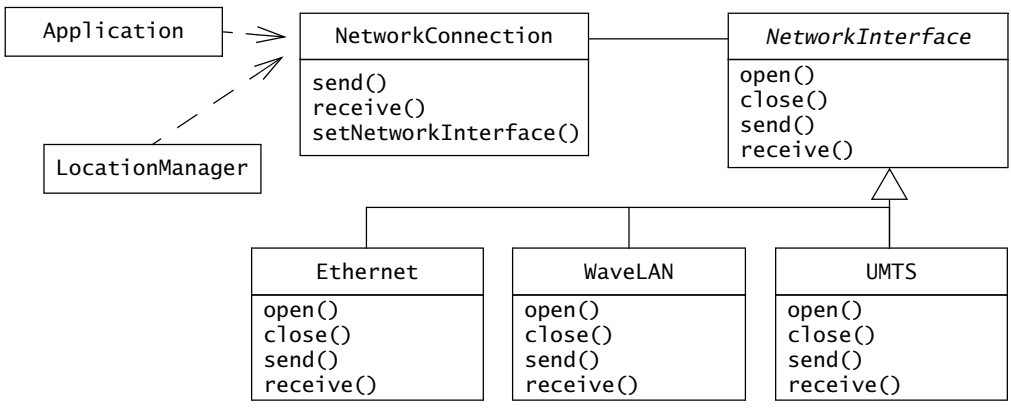


Figure 11-15 A Strategy design pattern for encapsulating multiple implementations of a `NetworkInterface` (UML class diagram).

Note that in some situations, the number of paths can be reduced by eliminating redundancy. For the sake of this example, we simply adopt a mechanical approach.

Once the source code is expanded, we extract the flow graph (Figure 11-17) and generate test cases covering all paths. This results in test cases that exercise the `send()` method of all three concrete network interfaces.

When many interfaces and abstract classes are involved, generating the flow graph for a method of medium complexity can result in an explosion of paths. This illustrates, on the one hand, how object-oriented code using polymorphism can result in compact and extensible components, and on the other hand, how the number of test cases increases when trying to achieve any acceptable path coverage.

```
public class NetworkConnection {
//...
private NetworkInterface nif;
void send(byte msg[]) {
    queue.concat(msg);
    if (nif.isReady()) {
        nif.send(queue);
        queue.setLength(0);
    }
}
}
```

```
public class NetworkConnection {
//...
private NetworkInterface nif;
void send(byte msg[]) {
    queue.concat(msg);
    boolean ready = false;
    if (nif instanceof Ethernet) {
        Ethernet eNif = (Ethernet)nif;
        ready = eNif.isReady();
    } else if (nif instanceof WaveLAN) {
        WaveLAN wNif = (WaveLAN)nif;
        ready = wNif.isReady();
    } else if (nif instanceof UMTS) {
        UMTS uNif = (UMTS)nif;
        ready = uNif.isReady();
    }
    if (ready) {
        if (nif instanceof Ethernet) {
            Ethernet eNif = (Ethernet)nif;
            eNif.send(queue);
        } else if (nif instanceof WaveLAN){
            WaveLAN wNif = (WaveLAN)nif;
            wNif.send(queue);
        } else if (nif instanceof UMTS){
            UMTS uNif = (UMTS)nif;
            uNif.send(queue);
        }
        queue.setLength(0);
    }
}
}
```

Figure 11-16 Java source code for the `NetworkConnection.send()` message (left) and equivalent Java source code without polymorphism (right). The source code on the right is used for generating test cases.

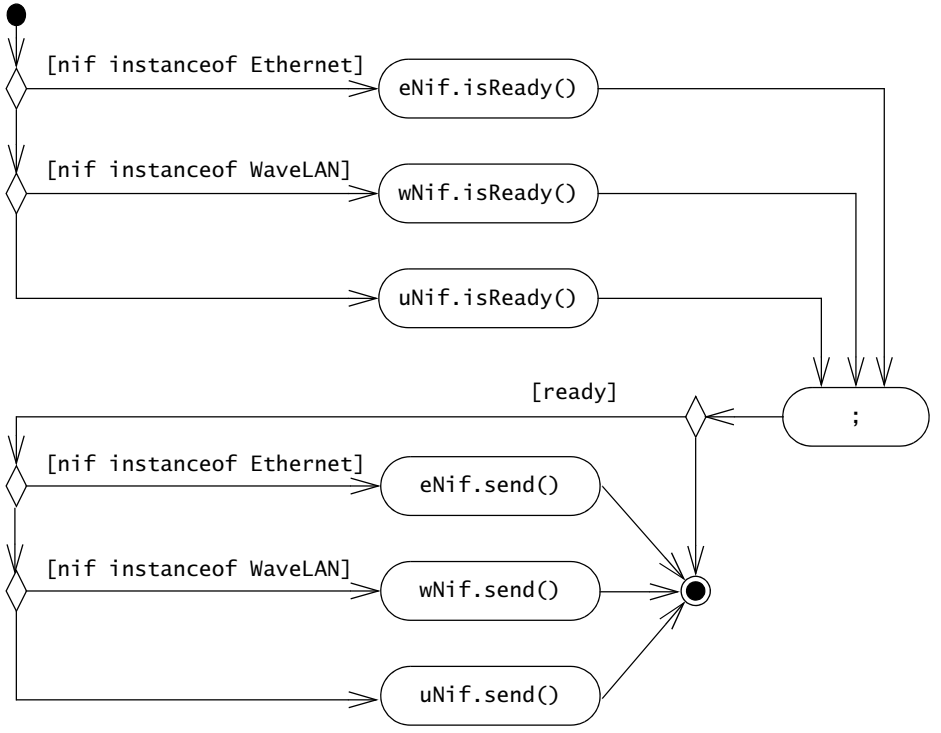


Figure 11-17 Equivalent flow graph for the expanded source code of the `NetworkConnection.send()` method of Figure 11-16 (UML activity diagram).

11.4.4 Integration Testing

Unit testing focuses on individual components. The developer discovers faults using equivalence testing, boundary testing, path testing, and other methods. Once faults in each component have been removed and the test cases do not reveal any new fault, components are ready to be integrated into larger subsystems. At this point, components are still likely to contain faults, as test stubs and drivers used during unit testing are only approximations of the components they simulate. Moreover, unit testing does not reveal faults associated with the component interfaces resulting from invalid assumptions when calling these interfaces.

Integration testing detects faults that have not been detected during unit testing by focusing on small groups of components. Two or more components are integrated and tested, and when no new faults are revealed, additional components are added to the group. If two components are tested together, we call this a *double test*. Testing three components together is a *triple test*, and a test with four components is called a *quadruple test*. This procedure allows the testing of increasingly more complex parts of the system while keeping the location of potential

faults relatively small (i.e., the most recently added component is usually the one that triggers the most recently discovered faults).

Developing test stubs and drivers for a systematic integration test is time consuming. For that reason, Extreme Programming, for example, stipulates that drivers be written before components are developed [Beck & Andres, 2005]. The order in which components are tested, however, can influence the total effort required by the integration test. A careful ordering of components can reduce the resources needed for the overall integration test. In the next sections, we discuss **horizontal integration testing strategies**, in which components are integrated according to layers, and **vertical integration testing strategies**, in which components are integrated according to functions.

Horizontal integration testing strategies

Several approaches have been devised to implement a horizontal integration testing strategy: big bang testing, bottom-up testing, top-down testing, and sandwich testing. Each of these strategies was originally devised by assuming that the system decomposition is hierarchical and that each of the components belong to hierarchical layers ordered with respect to the "Call" association. These strategies, however, can be easily adapted to nonhierarchical system decompositions. Figure 11-18 shows a hierarchical system decomposition that we use for discussing these strategies.

The **big bang testing** strategy assumes that all components are first tested individually and then tested together as a single system. The advantage is that no additional test stubs or drivers are needed. Although this strategy sounds simple, big bang testing is expensive: if a test uncovers a failure, it is impossible to distinguish failures in the interface from failures within a component. Moreover, it is difficult to pinpoint the specific component (or combination of components) responsible for the failure, as all components in the system are potentially exercised. This results in integration strategies that integrate only a few components at the time.

The **bottom-up testing** strategy first tests each component of the bottom layer individually, and then integrates them with components of the next layer up. This is repeated until all components from all layers are combined. Test drivers are used to simulate the components of higher layers that have not yet been integrated. Note that no test stubs are necessary during bottom-up testing.

The **top-down testing** strategy unit tests the components of the top layer first, and then integrates the components of the next layer down. When all components of the new layer have been tested together, the next layer is selected. Again, the tests incrementally add one component at a time. This is repeated until all layers are combined and involved in the test. Test stubs are used to simulate the components of lower layers that have not yet been integrated. Note that test drivers are not needed during top-down testing.

The advantage of bottom-up testing is that interface faults can be more easily found: when the developers substitute a test driver for a higher-level component, they have a clear model of how the lower-level component works and of the assumptions embedded in its interface. If the

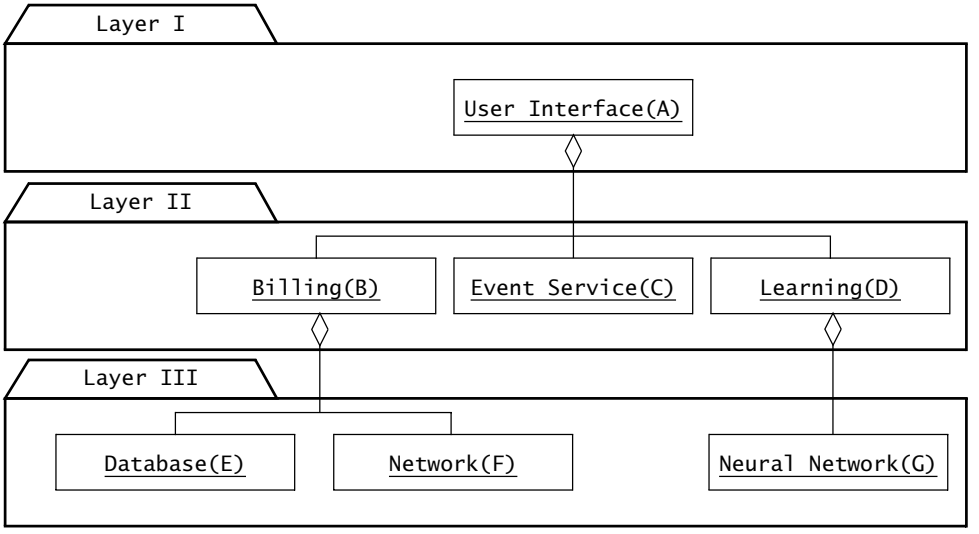


Figure 11-18 Example of a hierarchal system decomposition with three layers (UML class diagram, layers represented by packages).

higher-level component violates assumptions made in the lower-level component, developers are more likely to find them quickly. The disadvantage of bottom-up testing is that it tests the most important subsystems, namely the components of the user interface, last. Faults found in the top layer may often lead to changes in the subsystem decomposition or in the subsystem interfaces of lower layers, invalidating previous tests.

The advantage of top-down testing is that it starts with user interface components. The same set of tests, derived from the requirements, can be used in testing the increasingly more complex set of subsystems. The disadvantage of top-down testing is that the development of test stubs is time-consuming and prone to error. A large number of stubs is usually required for testing nontrivial systems, especially when the lowest level of the system decomposition implements many methods.

Figures 11-19 and 11-20 illustrate the possible combinations of subsystems that can be used during integration testing. Using a bottom-up strategy, subsystems E, F, and G are united tested first, then the triple test B-E-F and the double test D-G are executed, and so on. Using a top-down strategy, subsystem A is unit tested, then double tests A-B, A-C, and A-D are executed, then the quad test A-B-C-D is executed, and so on. Both strategies cover the same number of subsystem dependencies, but exercise them in different order.

The **sandwich testing** strategy combines the top-down and bottom-up strategies, attempting to make use of the best of both. During sandwich testing, the tester must be able to reformulate or map the subsystem decomposition into three layers, a target layer (“the meat”), a

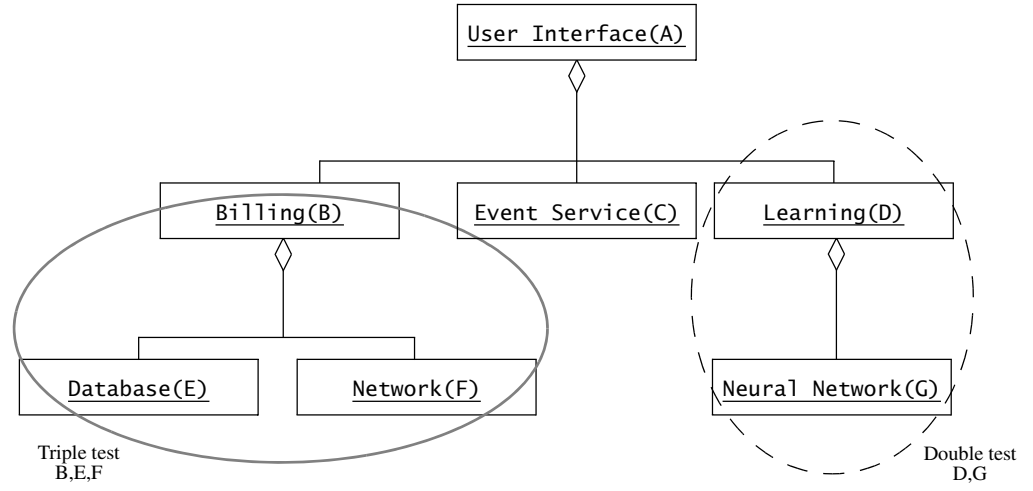


Figure 11-19 Bottom-up test strategy. After unit testing subsystems E, F, and G, the bottom up integration test proceeds with the triple test B-E-F and the double test D-G.

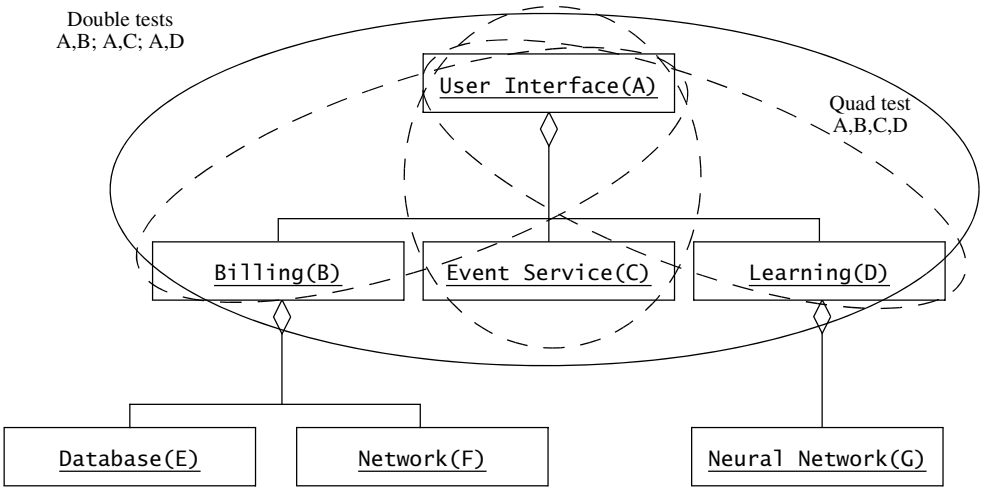


Figure 11-20 Top-down test strategy. After unit testing subsystem A, the integration test proceeds with the double tests A-B, A-C, and A-D, followed by the quad test A-B-C-D.

layer above the target layer ("the top slice of bread"), and a layer below the target layer ("the bottom slice of bread"). Using the target layer as the focus of attention, top-down testing and bottom-up testing can now be done in parallel. Top-down integration testing is done by testing the top layer incrementally with the components of the target layer, and bottom-up testing is used for testing the bottom layer incremental with the components of the target layer. As a result, test stubs and drivers need not be written for the top and bottom layers, because they use the actual components from the target layer.

Note that this also allows early testing of the user interface components. There is one problem with sandwich testing: it does not thoroughly test the individual components of the target layer before integration. For example, the sandwich test shown in Figure 11-21 does not unit test component C of the target layer.

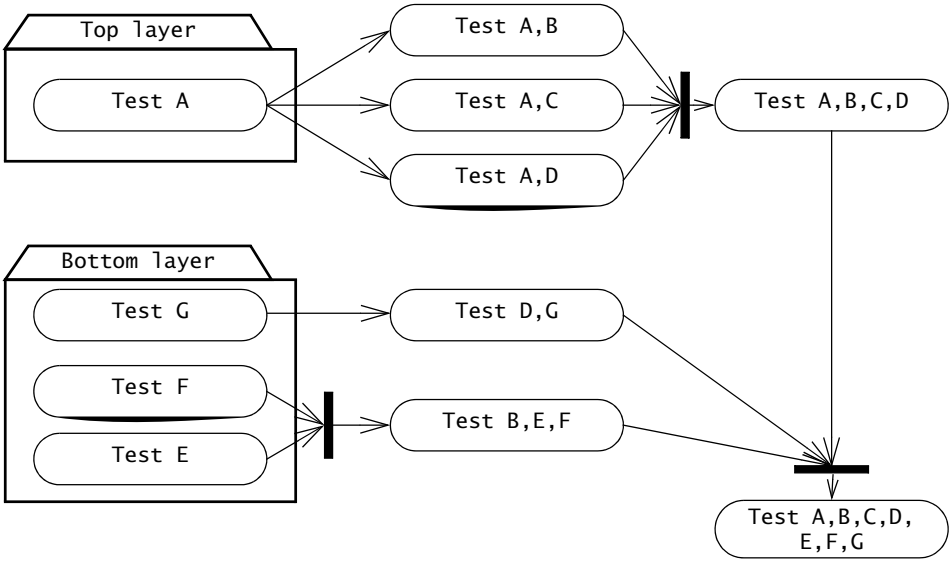


Figure 11-21 Sandwich testing strategy (UML activity diagram). None of the components in the target layer (i.e., B, C, D) are unit tested.

The **modified sandwich testing** strategy tests the three layers individually before combining them in incremental tests with one another. The individual layer tests consists of a group of three tests:

- a top layer test with stubs for the target layer
- a target layer test with drivers and stubs replacing the top and bottom layers
- a bottom layer test with a driver for the target layer.

The combined layer tests consist of two tests:

- The top layer accesses the target layer. This test can reuse the target layer tests from the individual layer tests, replacing the drivers with components from the top layer.
- The bottom layer is accessed by the target layer. This test can reuse the target layer tests from the individual layer tests, replacing the stub with components from the bottom layer.

The advantage of modified sandwich testing is that many testing activities can be performed in parallel, as indicated by the activity diagrams of Figures 11-21 and 11-22. The disadvantage of modified sandwich testing is the need for additional test stubs and drivers. Overall, modified sandwich testing leads to a significantly shorter overall testing time than top-down or bottom-up testing.

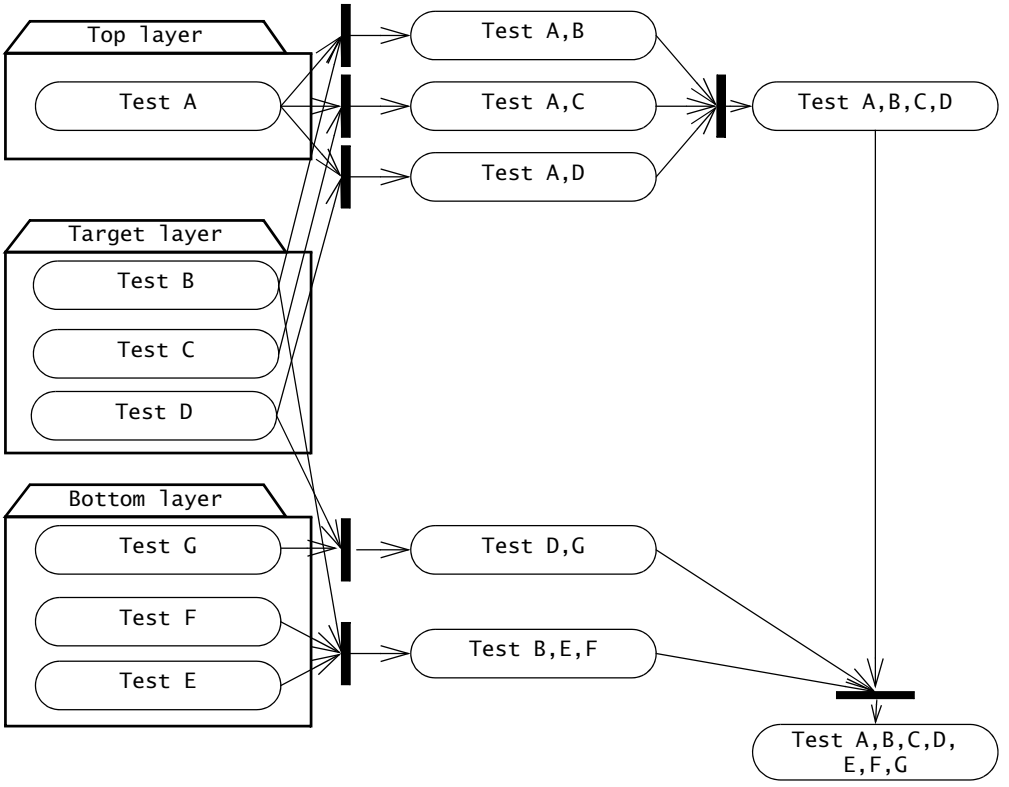


Figure 11-22 An example of modified sandwich testing strategy (UML activity diagrams). The components of the target layer are unit tested before they are integrated with the top and bottom layers.

Vertical integration testing strategies

In the previous section, we discussed horizontal integration testing strategies, in which components are integrated into layers, following the subsystem decomposition. As development responsibilities also follow the subsystem decomposition, horizontal integration is straightforward to manage, as tests verify the interfaces that have been negotiated between teams. The main drawback, however, is that an operational system that can be a release candidate, is only available very late during development.

Vertical integration testing strategies, in contrast, focus on early integration. For a given use case, the needed parts of each component, such the user interface, business logic, middleware, and storage, are identified and developed in parallel and integration tested. Note that this is different than the vertical prototypes for usability testing discussed in Section 11.4.2, as vertical prototypes are not release candidates. A system build with a vertical integration strategy produces release candidates.

For example, Extreme Programming [Beck & Andres, 2005], uses a vertical integration strategy in terms of a user stories. A user story is a single functional requirement formulated by the customer that is realized and integration tested during an iteration. At the end of an iteration, a release candidate is produced and demonstrated to the customer. The drawback of vertical integration testing, however, is that the system design is evolved incrementally, often resulting in reopening major system design decisions.

We discuss solutions to the early integration challenges when introducing continuous integration in Chapter 13, *Configuration Management*.

11.4.5 System Testing

Unit and integration testing focus on finding faults in individual components and the interfaces between the components. Once components have been integrated, **system testing** ensures that the complete system complies with the functional and nonfunctional requirements. Note that vertical integration testing is a special case of system testing: the former focuses only on a new slice of functionality, whereas the system testing focuses on the complete system.

During system testing, several activities are performed:

- **Functional testing**. Test of functional requirements (from RAD)
- **Performance testing**. Test of nonfunctional requirements (from SDD)
- **Pilot testing**. Tests of common functionality among a selected group of end users in the target environment
- **Acceptance testing.** Usability, functional, and performance tests performed by the customer in the development environment against acceptance criteria (from Project Agreement)
- **Installation testing**. Usability, functional, and performance tests performed by the customer in the target environment. If the system is only installed at a small selected set of customers it is called a *beta test*.

Functional testing

Functional testing, also called **requirements testing**, finds differences between the functional requirements and the system. Functional testing is a blackbox technique: test cases are derived from the use case model. In systems with complex functional requirements, it is usually not possible to test all use cases for all valid and invalid inputs. The goal of the tester is to select those tests that are relevant to the user and have a high probability of uncovering a failure. Note that functional testing is different from usability testing (described in Chapter 4, *Requirements Elicitation*), which also focuses on the use case model. Functional testing finds differences between the use case model and the observed system behavior, whereas usability testing finds differences between the use case model and the user's expectation of the system.

To identify functional tests, we inspect the use case model and identify use case instances that are likely to cause failures. This is done using blackbox techniques similar to equivalence testing and boundary testing (see Section 11.4.3). Test cases should exercise both common and exceptional use cases. For example, consider the use case model for a subway ticket distributor (see Figure 11-23). The common case functionality is modeled by the `PurchaseTicket` use case, describing the steps necessary for a `Passenger` to successfully purchase a ticket. The `TimeOut`, `Cancel`, `OutOfOrder`, and `NoChange` use cases describe various exceptional conditions resulting from the state of the distributor or actions by the `Passenger`.

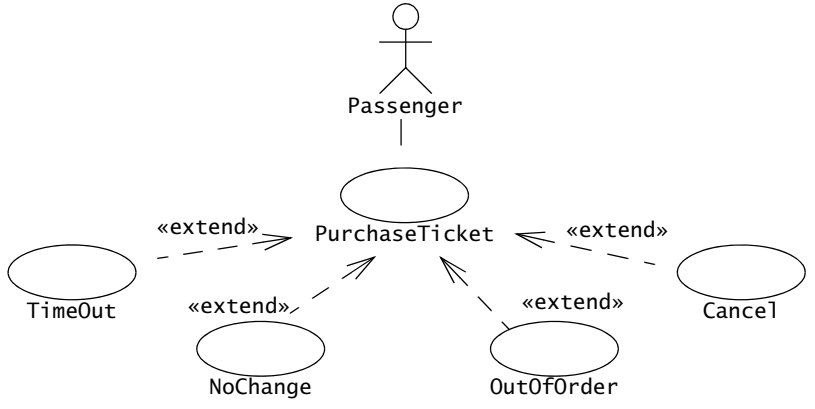


Figure 11-23 An example of use case model for a subway ticket distributor (UML use case diagram).

Figure 11-24 depicts the `PurchaseTicket` use case describing the normal interaction between the `Passenger` actor and the `Distributor`. We notice that three features of the `Distributor` are likely to fail and should be tested:

1. The Passenger may press multiple zone buttons before inserting money, in which case the Distributor should display the amount of the last zone.
2. The Passenger may select another zone button after beginning to insert money, in which case the Distributor should return all money inserted by the Passenger.
3. The Passenger may insert more money than needed, in which case the Distributor should return the correct change.

Use case name	PurchaseTicket
Entry condition	The Passenger is standing in front of ticket Distributor. The Passenger has sufficient money to purchase ticket.
Flow of events	1. The Passenger selects the number of zones to be traveled. If the Passenger presses multiple zone buttons, only the last button pressed is considered by the Distributor. 2. The Distributor displays the amount due. 3. The Passenger inserts money. 4. If the Passenger selects a new zone before inserting sufficient money, the Distributor returns all the coins and bills inserted by the Passenger. 5. If the Passenger inserted more money than the amount due, the Distributor returns excess change. 6. The Distributor issues ticket. 7. The Passenger picks up the change and the ticket.
Exit condition	The Passenger has the selected ticket.

Figure 11-24 An example of use case from the ticket distributor use case model PurchaseTicket.

Figure 11-25 depicts the test case PurchaseTicket_CommonCase, which exercises these three features. Note that the flow of events describes both the inputs to the system (stimuli that the Passenger sends to the Distributor) and desired outputs (correct responses from the Distributor). Similar test cases can also be derived for the exceptional use cases NoChange, OutOfOrder, TimeOut, and Cancel.

Test cases, such as PurchaseTicket_CommonCase, are derived for all use cases, including use cases representing exceptional behavior. Test cases are associated with the use cases from which they are derived, making it easier to update the test cases when use cases are modified.

Performance testing

Performance testing finds differences between the design goals selected during system design and the system. Because the design goals are derived from the nonfunctional requirements, the test cases can be derived from the SDD or from the RAD. The following tests are performed during performance testing:

Test case name	`PurchaseTicket_CommonCase`
Entry condition	The `Passenger` standing in front of ticket `Distributor`. The `Passenger` has two $5 bills and three dimes.
Flow of events	1. The `Passenger` presses in succession the zone buttons 2, 4, 1, and 2. 2. The `Distributor` should display in succession $1.25, $2.25, $0.75, and $1.25. 3. The `Passenger` inserts a $5 bill. 4. The `Distributor` returns three $1 bills and three quarters and issues a 2-zone ticket. 5. The `Passenger` repeats steps 1–4 using his second $5 bill. 6. The `Passenger` repeats steps 1–3 using four quarters and three dimes. The `Distributor` issues a 2-zone ticket and returns a nickel. 7. The `Passenger` selects zone 1 and inserts a dollar bill. The `Distributor` issues a 1-zone ticket and returns a quarter. 8. The `Passenger` selects zone 4 and inserts two $1 bills and a quarter. The `Distributor` issues a 4-zone ticket. 9. The `Passenger` selects zone 4. The `Distributor` displays $2.25. The `Passenger` inserts a $1 bill and a nickel, and selects zone 2. The `Distributor` returns the $1 bill and the nickel and displays $1.25.
Exit condition	The `Passenger` has three 2-zone tickets, one 1-zone ticket, and one 4-zone ticket.

Figure 11-25 An example of test case derived from the `PurchaseTicket` use case.

- *Stress testing* checks if the system can respond to many simultaneous requests. For example, if an information system for car dealers is required to interface with 6000 dealers, the stress test evaluates how the system performs with more than 6000 simultaneous users.
- *Volume testing* attempts to find faults associated with large amounts of data, such as static limits imposed by the data structure, or high-complexity algorithms, or high disk fragmentation.
- *Security testing* attempts to find security faults in the system. There are few systematic methods for finding security faults. Usually this test is accomplished by "tiger teams" who attempt to break into the system, using their experience and knowledge of typical security flaws.
- *Timing testing* attempts to find behaviors that violate timing constraints described by the nonfunctional requirements.
- *Recovery tests* evaluates the ability of the system to recover from erroneous states, such as the unavailability of resources, a hardware failure, or a network failure.

After all the functional and performance tests have been performed, and no failures have been detected during these tests, the system is said to be validated.

Pilot testing

During the **pilot test**, also called the **field test**, the system is installed and used by a selected set of users. Users exercise the system as if it had been permanently installed. No explicit guidelines or test scenarios are given to the users. Pilot tests are useful when a system is built without a specific set of requirements or without a specific customer in mind. In this case, a group of people is invited to use the system for a limited time and to give their feedback to the developers.

An *alpha test* is a pilot test with users exercising the system in the development environment. In a *beta test*, the pilot test is performed by a limited number of end users in the target environment; that is, the difference between usability tests and alpha or beta tests is that the behavior of the end user is not observed and recorded. As a result, beta tests do not test usability requirements as thoroughly as usability tests do. For interactive systems where ease of use is a requirement, the usability test therefore cannot be replaced with a beta test.

The Internet has made the distribution of software very easy. As a result, beta tests are more and more common. In fact, some companies now use it as the main method for system testing their software. Because the downloading process is the responsibility of the end user, not the developers, the cost of distributing the experimental software has decreased sharply. Consequently, a restricted number of beta testers is also a matter of the past. The new beta test paradigm offers the software to anybody who is interested in testing it. In fact, some companies charge their users for beta testing their software!

Acceptance testing

There are three ways the client evaluates a system during **acceptance testing**. In a *benchmark test*, the client prepares a set of test cases that represent typical conditions under which the system should operate. Benchmark tests can be performed with actual users or by a special test team exercising the system functions, but it is important that the testers be familiar with the functional and nonfunctional requirements so they can evaluate the system.

Another kind of system acceptance testing is used in reengineering projects, when the new system replaces an existing system. In *competitor testing*, the new system is tested against an existing system or competitor product. In *shadow testing*, a form of comparison testing, the new and the legacy systems are run in parallel and their outputs are compared.

After acceptance testing, the client reports to the project manager which requirements are not satisfied. Acceptance testing also gives the opportunity for a dialog between the developers and client about conditions that have changed and which requirements must be added, modified, or deleted because of the changes. If requirements must be changed, the changes should be reported in the minutes to the client acceptance review and should form the basis for another iteration of the software life-cycle process. If the customer is satisfied, the system is accepted, possibly contingent on a list of changes recorded in the minutes of the acceptance test.

Installation testing

After the system is accepted, it is installed in the target environment. A good system testing plan allows the easy reconfiguration of the system from the development environment to the target environment. The desired outcome of the **installation test** is that the installed system correctly addresses all requirements.

In most cases, the installation test repeats the test cases executed during function and performance testing in the target environment. Some requirements cannot be executed in the development environment because they require target-specific resources. To test these requirements, additional test cases have to be designed and performed as part of the installation test. Once the customer is satisfied with the results of the installation test, system testing is complete, and the system is formally delivered and ready for operation.

11.5 Managing Testing

In previous sections, we showed how different testing techniques are used to maximize the number of faults discovered. In this section, we describe how to manage testing activities to minimize the resources needed. Many testing activities occur near the end of the project, when resources are running low and delivery pressure increases. Often, trade-offs lie between the faults to be repaired before delivery and those that can be repaired in a subsequent revision of the system. In the end, however, developers should detect and repair a sufficient number of faults such that the system meets functional and nonfunctional requirements to an extent acceptable to the client.

First, we describe the planning of test activities (Section 11.5.1). Next, we describe the test plan, which documents the activities of testing (Section 11.5.2). Next, we describe the roles assigned during testing (Section 11.5.3). Next, we discuss the topics of regression testing(Section 11.5.4), automated testing (Section 11.5.5), and model-based testing (Section 11.5.6).

11.5.1 Planning Testing

Developers can reduce the cost of testing and the elapsed time necessary for its completion through careful planning. Two key elements are to start the selection of test cases early and to parallelize tests.

Developers responsible for testing can design test cases as soon as the models they validate become stable. Functional tests can be developed when the use cases are completed. Unit tests of subsystems can be developed when their interfaces is defined. Similarly, test stubs and drivers can be developed when component interfaces are stable. Developing tests early enables the execution of tests to start as soon as components become available. Moreover, given that developing tests requires a close examination of the models under validation, developers can find faults in the models even before the system is constructed. Note, however, that developing

tests early on introduces a maintenance problem: test cases, drivers, and stubs need to be updated whenever the system models change.

The second key element in shortening testing time is to parallelize testing activities. All component tests can be conducted in parallel; double tests for components in which no faults were discovered can be initiated while other components are repaired. For example, the quad test A-B-C-D in Figure 11-26 can be performed as soon as double tests A-B, A-C, and A-D have not resulted in any failures. These double tests, in turn, can be performed as soon as unit test A is completed. The quad test A-B-C-D can be performed in parallel with the double test D-G and the triple test B-E-F, even if tests E, F, or G uncover failures and delay the rest of the tests.

Testing represents a substantial part of the overall project resources. A typical guideline for projects following a Unified Process life cycle is to allocate 25 percent of project resources to testing (see Section 15.4.2; [Royce, 1998]). However, this number can go up depending onsafety and reliability requirements on the system. Hence, it is critical that test planning start early, as early as the use case model is stable.

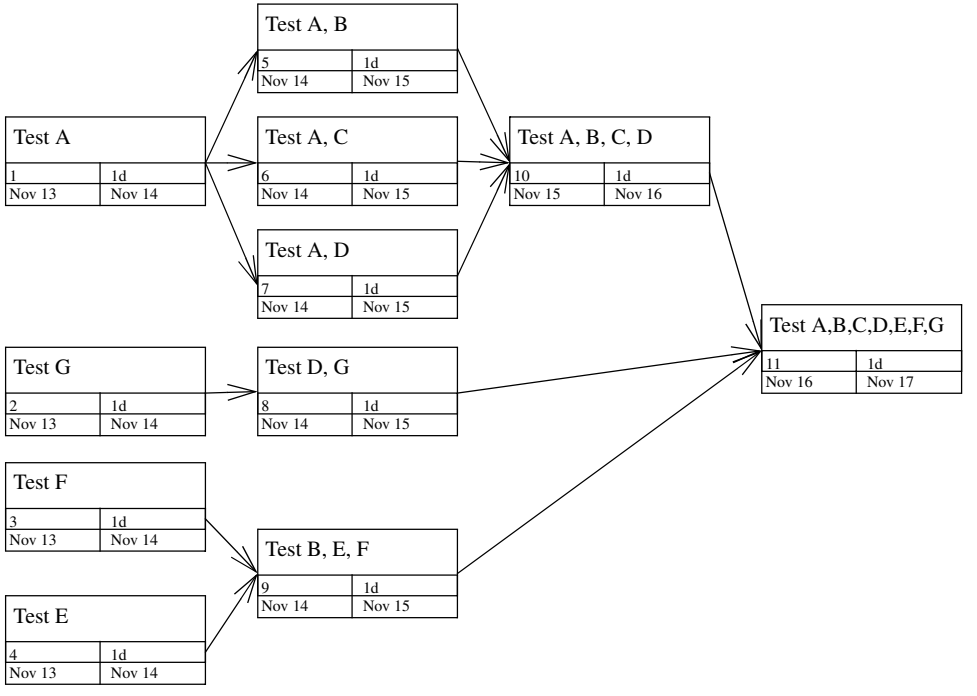


Figure 11-26 Example of a PERT chart for a schedule of the sandwich tests shown in Figure 11-21. The PERT chart notation is defined in Section 3.3.4.

11.5.2 Documenting Testing

Testing activities are documented in four types of documents, the *Test Plan*, the *Test Case Specifications*, the *Test Incident Reports*, and the *Test Summary Report*:[4]

- The *Test Plan* focuses on the managerial aspects of testing. It documents the scope, approach, resources, and schedule of testing activities. The requirements and the components to be tested are identified in this document.
- Each test is documented by a *Test Case Specification*. This document contains the inputs, drivers, stubs, and expected outputs of the tests, as well as the tasks to be performed.
- Each execution of each test is documented by a *Test Incident Report*. The actual results of the tests and differences from the expected output are recorded.
- The *Test Report Summary* document lists all the failures discovered during the tests that need to be investigated. From the *Test Report Summary*, the developers analyze and prioritize each failure and plan for changes in the system and in the models. These changes in turn can trigger new test cases and new test executions.

The *Test Plan* (TP) and the *Test Case Specifications* (TCS) are written early in the process, as soon as the test planning and each test case are completed. These documents are under configuration management and updated as the system models change. Figure 11-27 is an outline for a *Test Plan*.

Test Plan

1. Introduction
2. Relationship to other documents
3. System overview
4. Features to be tested/not to be tested
5. Pass/Fail criteria
6. Approach
7. Suspension and resumption
8. Testing materials (hardware/software requirements)
9. Test cases
10. Testing schedule

Figure 11-27 Outline of a Test Plan.

4. Documents described in this section are based on the IEEE 829 standard on testing documentation. Note that we omitted certain sections and documents (e.g., the Test Item Transmittal Report) for the sake of simplicity. Refer to the standard for a complete description of these documents [IEEE Std. 829-2008].

Section 1 of the test plan describes the objectives and extent of the tests. The goal is to provide a framework that can be used by managers and testers to plan and execute the necessary tests in a timely and cost-effective manner.

Section 2 explains the relationship of the test plan to the other documents produced during the development effort such as the RAD, SDD, and ODD (Object Design Document). It explains how all the tests are related to the functional and nonfunctional requirements, as well as to the system design stated in the respective documents. If necessary, this section introduces a naming scheme to establish the correspondence between requirements and tests.

Section 3, focusing on the structural aspects of testing, provides an overview of the system in terms of the components that are tested during the unit test. The granularity of components and their dependencies are defined in this section.

Section 4, focusing on the functional aspects of testing, identifies all features and combinations of features to be tested. It also describes all those features that are not to be tested and the reasons for not testing them.

Section 5 specifies generic pass/fail criteria for the tests covered in this plan. They are supplemented by pass/fail criteria in the test design specification. Note that "fail" in the IEEE standard terminology means "successful test" in our terminology.

Section 6 describes the general approach to the testing process. It discusses the reasons for the selected integration testing strategy. Different strategies are often needed to test different parts of the system. A UML class diagram can be used to illustrate the dependencies between the individual tests and their involvement in the integration tests.

Section 7 specifies the criteria for suspending the testing on the test items associated with the plan. It also specifies the test activities that must be repeated when testing is resumed.

Section 8 identifies the resources that are needed for testing. This should include the physical characteristics of the facilities, including the hardware, software, special test tools, and other resources needed (office space, etc.) to support the tests.

Section 9, the core of the test plan, lists the test cases that are used during testing. Each test case is described in detail in a separate *Test Case Specification* document. Each execution of these tests will be documented in a *Test Incident Report* document. We describe these documents in more details later in this section.

Section 10 of the test plan covers responsibilities, staffing and training needs, risks and contingencies, and the test schedule.

Figure 11-28 is an outline of a *Test Case Specification*.

The Test Case Specification identifier is the name of the test case, used to distinguish it from other test cases. Conventions such as naming the test cases from the features or the component being tested allow developers to more easily refer to test cases. Section 2 of the TCS lists the components under test and the features being exercised. Section 3 lists the inputs required for the test cases. Section 4 lists the expected output. This output is computed manually or with a competing system (such as a legacy system being replaced). Section 5 lists the hardware and software platform needed to execute the test, including any test drivers or stubs.

Test Case Specification

1. Test case specification identifier
2. Test items
3. Input specifications
4. Output specifications
5. Environmental needs
6. Special procedural requirements
7. Intercase dependencies

Figure 11-28 Outline of a Test Specification.

Section 6 lists any constraints needed to execute the test such as timing, load, or operator intervention. Section 7 lists the dependencies with other test cases.

The *Test Incident Report* lists the actual test results and the failures that were experienced. The description of the results must include which features were demonstrated and whether the features have been met. If a failure has been experienced, the test incident report should contain sufficient information to allow the failure to be reproduced. Failures from all *Test Incident Reports* are collected and listed in the *Test Summary Report* and then further analyzed and prioritized by the developers.

Note that the IEEE standard [IEEE Std. 829-2008] for software test documentation uses a slightly different outline that is more appropriate for large organizations and systems. Section 10, for example, is covered by several sections in the standard (responsibilities, staffing and training needs, schedule, risks, and contingencies).

11.5.3 Assigning Responsibilities

Testing requires developers to find faults in components of the system. This is best done when the testing is performed by a developer who was not involved in the development of the component under test, one who is less reticent to break the component being tested and who is more likely to find ambiguities in the component specification.

For stringent quality requirements, a separate team dedicated to quality control is solely responsible for testing. The testing team is provided with the system models, the source code, and the system for developing and executing test cases. *Test Incident Reports* and *Test Report Summaries* are then sent back to the subsystem teams for analysis and possible revision of the system. The revised system is then retested by the testing team, not only to check if the original failures have been addressed, but also to ensure that no new faults have been inserted in the system.

For systems that do not have stringent quality requirements, subsystem teams can double as a testing team for components developed by other subsystem teams. The architecture team can define standards for test procedures, drivers, and stubs, and can perform as the integration test team. The same test documents can be used for communication among subsystem teams.

One of the main problems of usability tests is with enrolling participants. Several obstacles are faced by project managers in selecting real end users [Grudin, 1990]:

- The project manager is usually afraid that users will bypass established technical support organizations and call the developers directly, once they know how to get to them. Once this line of communication is established, developers might be sidetracked too often from doing their assigned jobs.
- Sales personnel do not want developers to talk to "their" clients. Sales people are afraid that developers may offend the client or create dissatisfaction with the current generation of products (which still must be sold).
- The end users do not have time.
- The end users dislike being studied. For example, an automotive mechanic might think that an augmented reality system will put him out of work.

Debriefing the participants is the key to coming to understanding how to improve the usability of the system being tested. Even though the usability test uncovers and exposes problems, it is often the debriefing session that illustrates why these problems have occurred in the first place. It is important to write recommendations on how to improve the tested components as fast as possible after the usability test is finished, so they can be used by the developers to implement any necessary changes in the system models of the tested component.

11.5.4 Regression Testing

Object-oriented development is an iterative process. Developers modify, integrate, and retest components often, as new features are implemented or improved. When modifying a component, developers design new unit tests exercising the new feature under consideration. They may also retest the component by updating and rerunning previous unit tests. Once the modified component passes the unit tests, developers can be reasonably confident about the changes within the component. However, they should not assume that the rest of the system will work with the modified component, even if the system has previously been tested. The modification can introduce side effects or reveal previously hidden faults in other components. The changes can exercise different assumptions about the unchanged components, leading to erroneous states. Integration tests that are rerun on the system to produce such failures are called **regression tests**.

The most robust and straightforward technique for regression testing is to accumulate all integration tests and rerun them whenever new components are integrated into the system. This requires developers to keep all tests up-to-date, to evolve them as the subsystem interfaces change, and to add new integration tests as new services or new subsystems are added. As regression testing can become time consuming, different techniques have been developed for selecting specific regression tests. Such techniques include [Binder, 2000]:

- *Retest dependent components*. Components that depend on the modified component are the most likely to fail in a regression test. Selecting these tests will maximize the likelihood of finding faults when rerunning all tests is not feasible.
- *Retest risky use cases*. Often, ensuring that the most catastrophic faults are identified is more critical than identifying the largest number of faults. By focusing first on use cases that present the highest risk, developers can minimize the likelihood of catastrophic failures.
- *Retest frequent use cases*. When users are exposed to successive releases of the same system, they expect that features that worked before continue to work in the new release. To maximize the likelihood of this perception, developers focus on the use cases that are most often used by the users.

In all cases, regression testing leads to running many tests many times. Hence, regression testing is feasible only when an automated testing infrastructure is in place, enabling developers to automatically set up, initialize, and execute tests and compare their results with a predefined oracle. We discuss automated testing in the next section.

11.5.5 Automating Testing

Manual testing involves a tester to feed predefined inputs into the system using the user interface, a command line console, or a debugger. The tester then compares the outputs generated by the system with the expected oracle. Manual testing can be costly and error prone when many tests are involved or when the system generates a large volume of outputs. When requirements change and the system evolves rapidly, testing should be repeatable. This makes these drawbacks worse, as it is difficult to guarantee that the same test is executed under the same conditions every time.

The repeatability of test execution can be achieved with automation. Although all aspects of testing can be automated (including test case and oracle generation), the main focus of test automation has been on execution. For system tests, test cases are specified in terms of the sequence and timing of inputs and an expected output trace. The test harness can then execute a number of test cases and compare the system output with the expected output trace. For unit and integration tests, developers specify a test as a test driver that exercises one or more methods of the classes under tests.

The benefit of automating test execution is that tests are repeatable. Once a fault is corrected as a result of a failure, the test that uncovered the failure can be repeated to ensure that the failure does not occur anymore. Moreover, other tests can be run to ensure (to a limited extent) that no new faults have been introduced. Moreover, when tests are repeated many times, for example, in the case of refactoring (see Section 10.3.2), the cost of testing is decreased substantially. However, note that developing a test harness and test cases is an investment. If tests are run only once or twice, manual testing may be a better alternative.

An example of an automated test infrastructure is JUnit, a framework for writing and automating the execution of unit tests for Java classes [JUnit, 2009]. The JUnit test framework is made out of a small number of tightly integrated classes (Figure 11-29). Developers write new test cases by subclassing the TestCase class. The setUp() and tearDown() methods of the concrete test case initialize and clean up the testing environment, respectively. The runTest() method includes the actual test code that exercises the class under test and compares the results with an expected condition. The test success or failure is then recorded in an instance of TestResult. TestCases can be organized into TestSuites, which will invoke sequentially each of its tests. TestSuites can also be included in other TestSuites, thereby enabling developers to group unit tests into increasingly larger test suites.

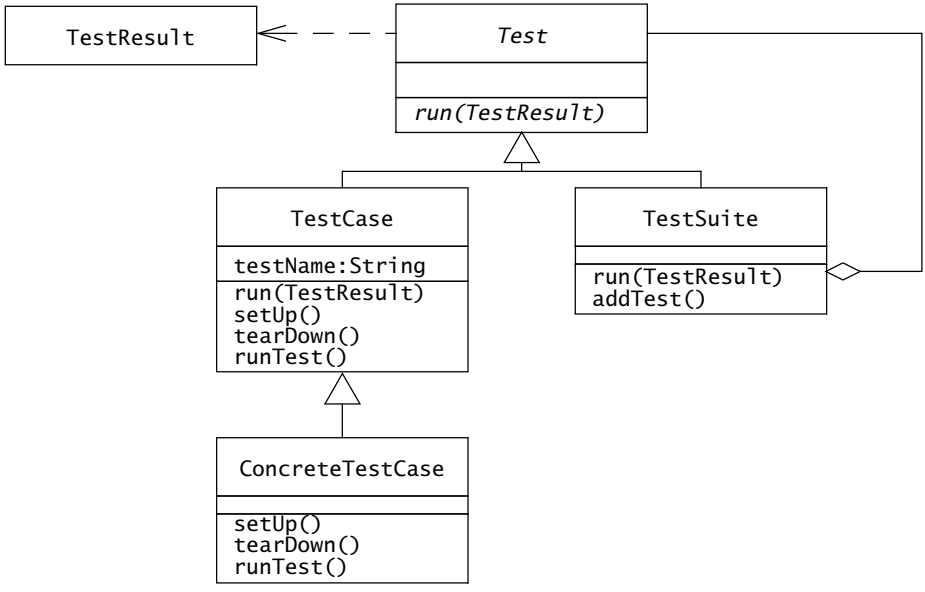


Figure 11-29 JUnit test framework (UML class diagram).

Typically, when using JUnit, each TestCase instance exercises one method of the class under test. To minimize the proliferation of TestCase classes, all test methods exercising the same class (and requiring the same test environment initialized by the setUp() method) are grouped in the same ConcreteTestCase class. The actual method that is invoked by runTest() can then be configured when creating instances of TestCases. This enables developers to organize and selectively invoke large number of tests.

11.5.6 Model-Based Testing

Testing (manual or automated) requires an infrastructure for executing tests, instrumenting the system under test, and collecting and assessing test results. This infrastructure is called the **test harness** or **test system**. The test system is made of software and hardware components that interact with various actors, which can then be modeled using UML. In Chapters 2-10, we have shown how the system under development and the development organization can be modeled in UML. Similarly, we can model the test system in UML. To be able to do this, we need to extend UML with new entity objects for modeling the test system.

UML profiles provide a way for extending UML. A **UML profile** is a collection of new stereotypes, new interfaces, or new constraints, thus providing new concepts specialized to an application domain or a solution domain.

U2TP (UML 2 Testing Profile, [OMG, 2005]) is an example of a UML profile, which extends UML for modeling testing. Modeling the test system in U2TP provides the same advantages as when modeling the system under development: test cases are modeled in a standard notation understood by all participants, test cases can be automatically generated from test models, test cases execution and results can be automatically collected and recorded.

U2TP extends UML with the following concepts:

- The **system under test** (stereotype «`sut`»), which may be the complete system under development, or only a part of it, such as a subsystem or a single class.
- A **test case** (stereotype «`testCase`») is a specification of behavior realizing one or more test objectives. A test cases specifies the sequence of interactions among the system under test and the test components. The interactions are either stimuli on the system under test or observations gathered from the system under test or from test components. A test case is represented as a sequence diagram or state machine. Test cases return an enumerated type called **verdict**, denoting if the test run passed, failed, was inconclusive, or an error in the test case itself was detected. In U2TP terminology, an error is caused by a fault in the test system, while a failure is caused by a fault in the system under test.
- A **test objective** (stereotype «`testObjective`») describes in English the goal of one or several test cases. A test objective is typically a requirement or a part of a requirement that is being verified. For example, the test objective of the displayTicketPrices test case is to verify that the correct price is displayed after selected a zone button on the ticket distributor.
- **Test components** (stereotype «`testComponent`»), such as test stubs and utilities needed for executing a test case. Examples of test components include simulated hardware, simulated user behavior, or components that inject faults.
- **Test contexts** (stereotype «`testContext`»), which include the set of test cases, the configuration of test components and system under test needed for every test case, and a test control for sequencing the test cases.

- An **arbiter** (interface `Arbiter`), which collects the local test results into an aggregated result.
- A **scheduler** (interface `Scheduler`), which creates and coordinate the execution of the test cases among test components and system under test.

Figure 11-30 depicts an example of a test system in U2TP for the `TicketDistributor` of Figures 11-23–11-25. The test context `PurchaseTicketSuite` groups all the test cases for the `PurchaseTicket` use case. The system under test is the `TicketDistributor` software. To make it easier to control and instrument the system to assess the success or failure of tests, we simulate the ticket distributor display with a `DisplaySimulator` test component.

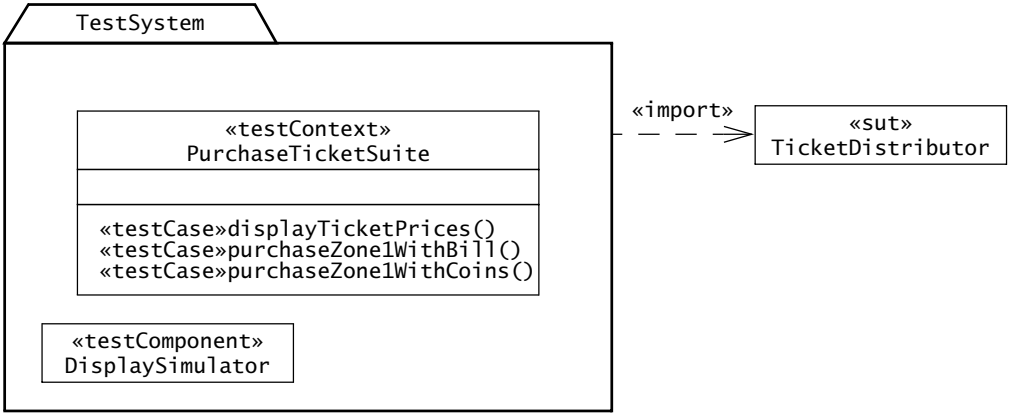


Figure 11-30 Example of a test system for the `TicketDistributor` (U2TP).

For example, Figure 11-31 depicts the expected interactions of the `displayTicketPrices()` test case resulting to a `pass` verdict. `selectZone1()`, `selectZone2()`, and `selectZone4()` are stimuli on the system under test. `getDisplay()` are observations to assess if individual test steps were successful. Note that only the expected interactions are displayed. Any unexpected interactions, missing interactions, or observations that do not match the oracles, lead to a `failed` verdict. U2TP also provides mechanisms, not discussed here, to explicitly model interactions that lead to an `inconclusive` or a `failed` verdict.

The `displayTicketPrices()` test case of Figure 11-31 explicitly models the mapping between zones and ticket prices. In a realistic system, this approach would not be sustainable, as many test cases are repeated for boundary values and with samples of different equivalence classes. To address this challenge, U2TP provides the concepts of `DataPool`, `DataPartition`, and `DataSelector`, to represent test data samples, equivalence classes, and data selection strategies, respectively. These allow to parameterize test cases with different sets of values, keeping the specification of test cases concise and reusable.

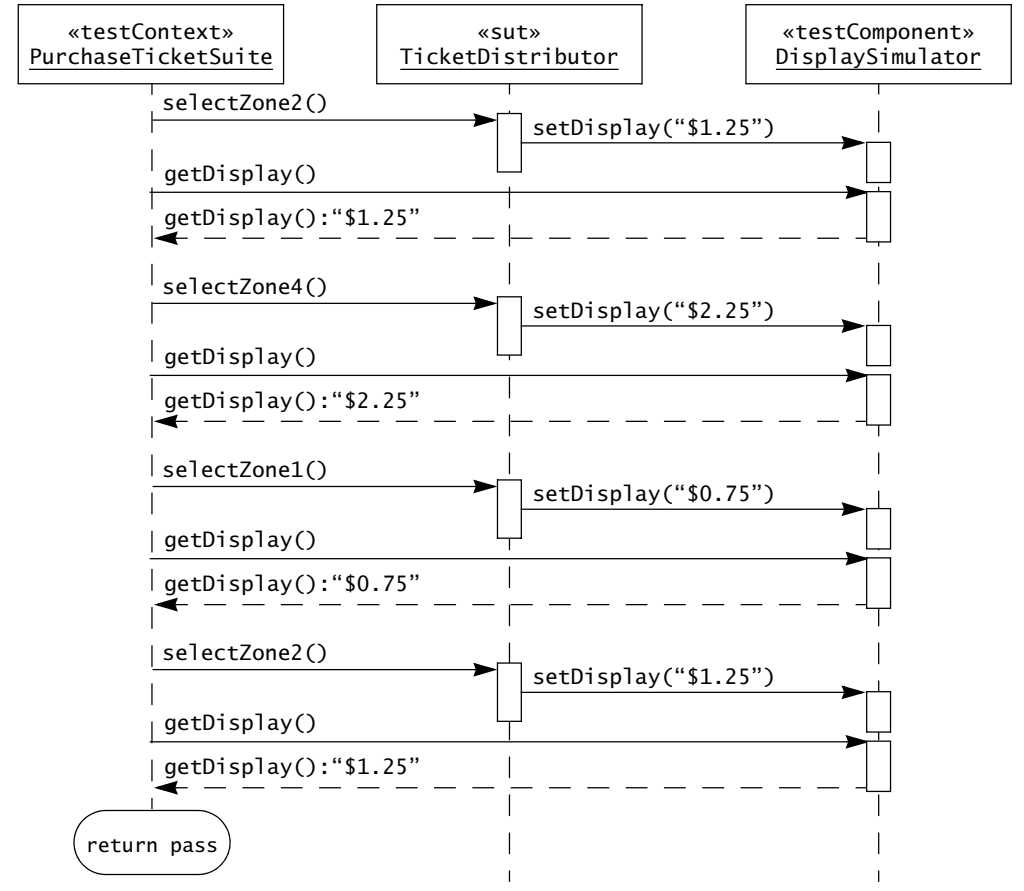


Figure 11-31 Example of test behavior for the test case `displayTicketPrices()` (U2TP). Only expected interactions leading to a `pass` verdict are represented.

11.6 Further Readings

Historically, the term "bug" was first used to denote a fault by Grace Hopper when a moth interfered with a computer relay, causing the program to stop [Hopper, 1981]. The term has been used since to denote design and coding faults caused by developers.

Fagan showed that code inspections can be more effective than testing for finding faults in a given amount of time [Fagan, 1976]. Many replicated experiments confirmed Fagan's finding. For that reason, inspections and peer reviews are stipulated by several standards, including ISO 9000. However, with the exception of critical systems, code inspections are not widely used because they are often perceived, ironically, as too time consuming.

Many books have been written about testing. However, progress in this discipline is slow and few new ideas have yielded results comparable to code inspections. *The Art of Software Testing*, although several decades old, remains a classic in the testing literature and relevant for today's systems [Myers, 1979].

The introduction of object-oriented programming techniques opened the door for increased modularity and reuse. However, polymorphism also increased drastically the number of paths to be tested. *Testing Object-Oriented Systems* contains the most comprehensive treatment of testing issues and techniques for object-oriented systems [Binder, 2000].

A common misconception is that usability testing requires large budgets and sophisticated know-how. *Handbook of Usability Testing* [Rubin, 1994] and *Usability Inspection Methods* [Nielsen & Mack, 1994] provide practical guidance and show how even limited usability tests can dramatically improve a system. *Usability Engineering Lifecycle* [Mayhew, 1999] integrates usability testing into object-oriented software engineering life cycle.

Developing reliable systems goes beyond testing. As discussed in the introduction of this chapter, alternative techniques, such as fault avoidance and fault tolerance, can complement testing to produce a highly reliable system. An excellent coverage of the topic is provided in [Siewiorek & Swarz, 1992].

U2TP is a response by a consortium to a request for proposals from OMG to develop a UML profile for testing. U2TP has been finalized and is now an official OMG standard [OMG, 2005]. Baker provides a practical introduction on model-based testing using U2TP [Baker et al., 2008].

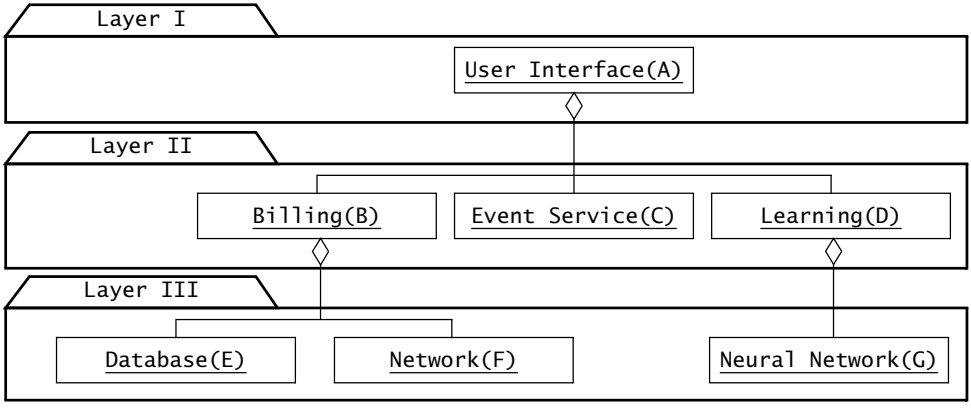
11.7 Exercises

11-1 Correct the faults in the `isLeapYear()` and `getNumDaysInMonth()` methods of Figure 11-11 and generate test cases using the path testing method. Are the test cases you found different than those of Table 11-4 and Figure 11-13? Why? Would the test cases you found uncover the faults you corrected?

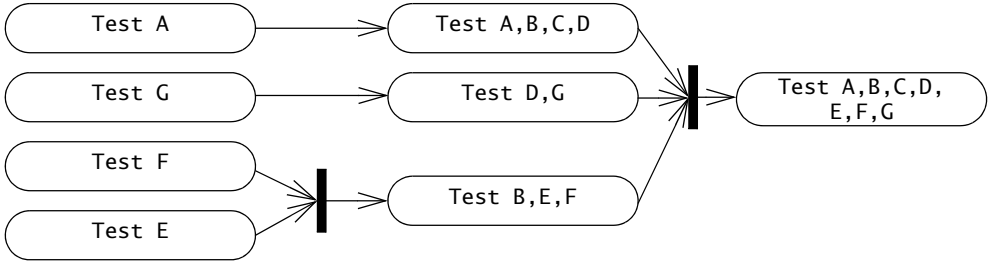
11-2 Generate equivalent Java code for the state machine diagram for the `SetTime` use case of `2Bwatch` (Figure 11-14). Use equivalence testing, boundary testing, and path testing to create test cases for the code you have just generated. How do these test cases compare with those generated using state-based testing?

11-3 Build the state machine diagram corresponding to the `PurchaseTicket` use case of Figure 11-24. Generate test cases based on the state machine diagram using the state-based testing technique. Discuss the number of test cases and differences with the test case of Figure 11-25.

11-4 Given the subsystem decomposition



comment on the testing plan used by the project manager:



What decisions were made? What are the advantages and disadvantages of this test plan?

11-5 You are responsible for the integration testing of a system that encrypts network traffic. This system includes a key generator subsystem that uses random numbers. During integration testing, you use a stub implementation of the key generator that produces a predictable result. However, for the release version of the system, you want to substitute the stub implementation with the random implementation, so that the generated keys are not predictable to an outsider. Implement a test infrastructure using one of the design patterns described in Chapter 8, *Object Design: Reusing Pattern Solutions* to enable the exchange of these two key generator implementations at run time. Justify your choice.

11-6 Use path testing to generate test cases for all the methods of the `NetworkConnection` class depicted in Figure 11-15 and in Figure 8-11. Expand first the source code to remove any polymorphism. How many test cases did you generate using path testing? How many test cases would you generate if the source code is not expanded?

11-7 Apply the software engineering and testing terminology from this chapter to the following terms used in Feynman's article mentioned in the introduction:

- What is a "crack"?
- What is "crack initiation"?
- What is "high engine reliability"?
- What is a "design aim"?
- What is a "mission equivalent"?
- What does "10 percent of the original specification" mean?
- How is Feynman using the term "verification," when he says that "As deficiencies and design errors are noted they are corrected and verified with further testing"?

References

[Beck & Andres, 2005] K. Beck & C. Andres, *Extreme Programming Explained: Embrace Change*, 2nd ed., Addison-Wesley, Reading, MA, 2005.

[Baker et al., 2008] P. Baker, Z. R. Dai, & R. Grabowski, *Model-Driven Testing: Using the UML Testing Profile*, Springer, Berlin, 2008.

[Binder, 2000] R. V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley, Reading, MA, 2000.

[Fagan, 1976] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, Vol. 15, No. 3, 1976.

[Feynman, 1988] R. P. Feynman, "Personal observations on the reliability of the Shuttle," Rogers Commission, *The Presidential Commission on the Space Shuttle Challenger Accident Report*, Washington, DC, June 1986.

[Grudin, 1990] J. Grudin, "Obstacles to user involvement in interface design in large product development organizations," *Proceedings of IFIP INTERACT'90 Third International Conference on Human-Computer Interaction*, Cambridge, U.K., August 1990.

[Hopper, 1981] G. M. Hopper, "The First Bug," *Annals of the History of Computing* 3, pp. 285–6, 1981.

[IEEE Std. 829-2008] *IEEE Standard for Software Test Documentation*, IEEE Standards Board, July 2008.

[IEEE Std. 982.2-1988] *IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software*, IEEE Standards Board, June 1988.

[Jones, 1977] T. C. Jones, "Programmer quality and programmer productivity," IBM TR-02.764, 1977.

[JUnit, 2009] JUnit, http://www.junit.org/.

[Kelly, 1984] J. F. Kelly, "An iterative design methodology for user-friendly natural language office information applications," *ACM Transactions on Information Systems*, Vol. 2, No. 1, January 1984.

[Mayhew, 1999] D. J. Mayhew, *The Usability Engineering Lifecycle: A Practitioner's Handbook for User Interface Design*, Morgan Kaufmann, 1999.

[McCabe, 1976] T. McCabe, "A software complexity measure," *IEEE Transactions on Software Engineering*, Vol. 2, No. 12, December 1976.

[Myers, 1979] G. J. Myers, *The Art of Software Testing*, Wiley, New York, 1979.

[Nielsen & Mack, 1994] J. Nielsen & R. L. Mack (eds.), *Usability Inspection Methods*, Wiley, New York, 1994.

[OMG, 2005] Object Management Group *UML Testing Profile Version 1.0*. http://www.omg.org/ 2005.

[Parnas & Weiss, 1985] D. L. Parnas & D. M. Weiss, "Active design reviews: principles and practice," *Proceedings of the Eighth International Conference on Software Engineering*, London, U.K., pp 132–136, August 1985.

[Popper, 1992] K. Popper, *Objective Knowledge: An Evolutionary Approach*, Clarendon, Oxford, 1992.

[Porter et al., 1997] A. A. Porter, H. Siy, C.A. Toman, & L.G. Votta, "An experiment to assess the cost-benefits of code inspections in large scale software development," *IEEE Transactions on Software Engineering*, Vol. 23, No. 6, pp. 329–346, June 1997.

[Royce, 1998] W. Royce, *Software Project Management: A Unified Framework*, Addison-Wesley, Reading, MA, 1998.

[Rubin, 1994] J. Rubin, *Handbook of Usability Testing*, Wiley, New York, 1994.

[Siewiorek & Swarz, 1992] D. P. Siewiorek & R. S. Swarz, *Reliable Computer Systems: Design and Evaluation*, 2nd ed., Digital, Burlington, MA, 1992.

[Turner & Robson, 1993] C. D. Turner & D. J. Robson, "The state-based testing of object-oriented programs," *Conference on Software Maintenance*, pp. 302–310, September 1993.