Chapter Six: Software Quality Assurance

6.1. Introduction

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.

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.

6.2. 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 conditio. **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 nonoptimal 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.

This material 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 failure 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

This course, 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.

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 6-1 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.

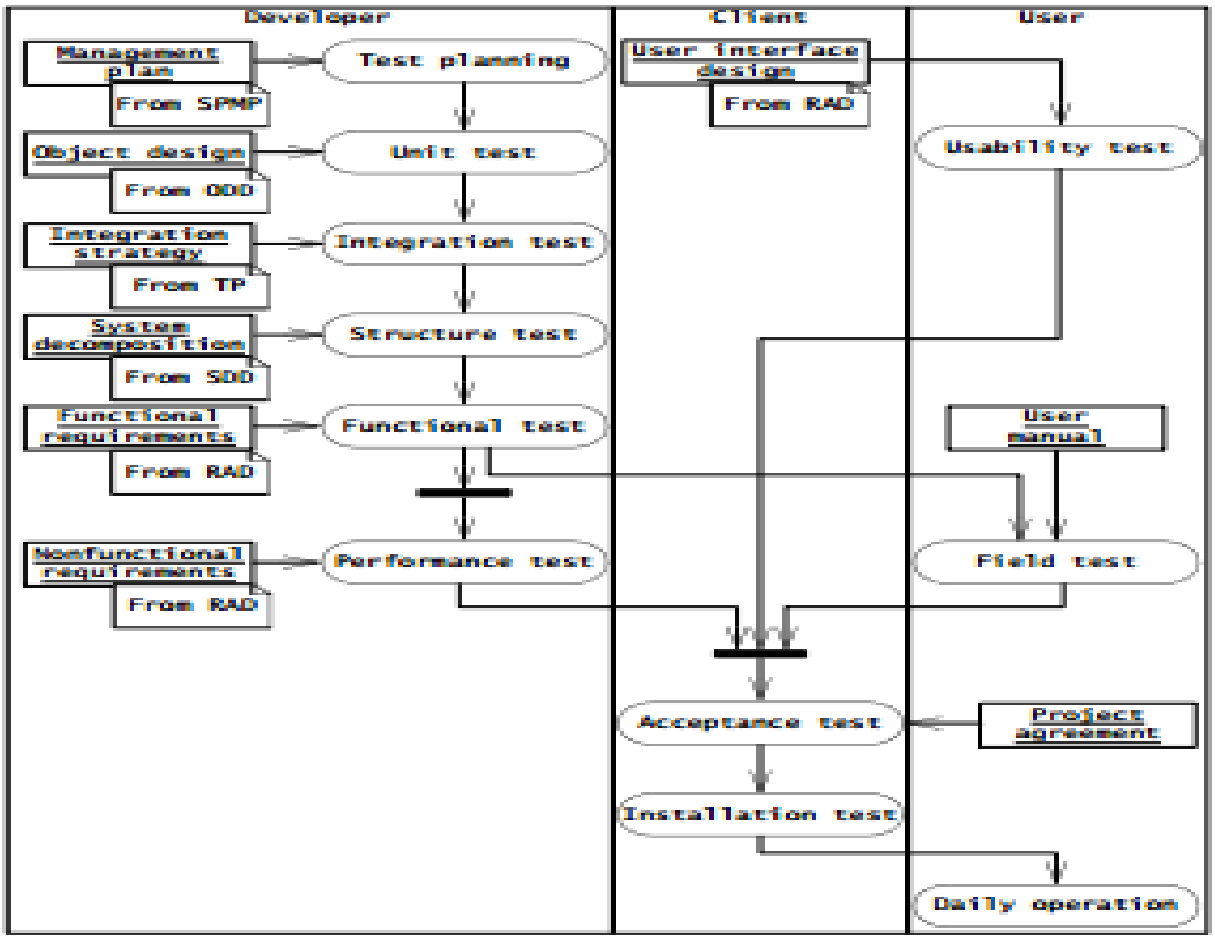

Figure 6-1 Testing activities and their related work products (UML activity diagram). Swimlanes indicate who executes the test.

6.2. Testing concepts

In this section, we present the model elements used during testing (Figure 6-2):

- 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.

- 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.

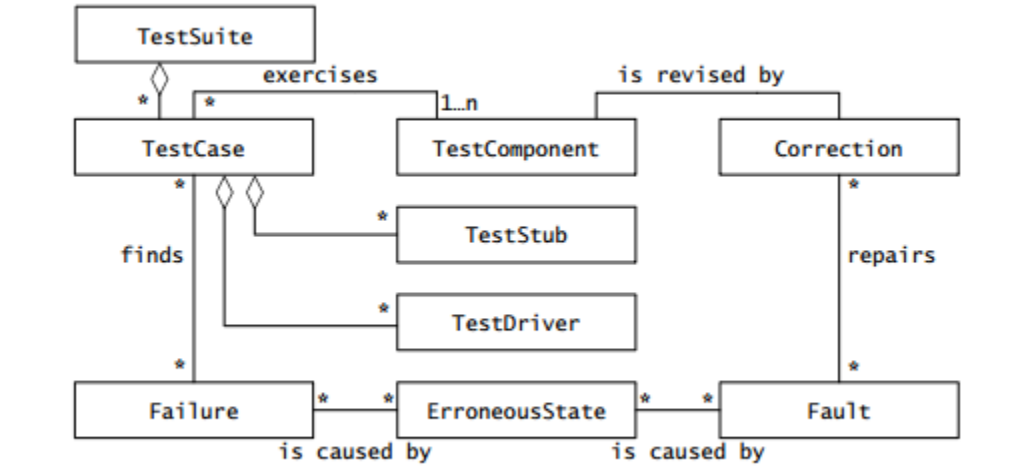

Figure 11-2 Model elements used during testing (UML class diagram).

6.3. Testing activities

This section 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
- **Usability testing**, which finds differences between what the system does and the users' expectation of what it should do
- **Unit testing**, which finds faults by isolating an individual component using test stubs and drivers and by exercising the component using test cases
- **Integration testing**, which finds faults by integrating several components together
- **System testing**, which focuses on the complete system, its functional and nonfunctional requirements, and its target environment

6.3.1. 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 re-inspected.

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. 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.

6.3.2 Usability Testing

Usability testing tests the user understands 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.

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 fullblown 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-ups3 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. 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. 6.3.3. **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.

6.3.4 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).

6.3.5 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*.

6.4. 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.

6.4.1 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.

6.4.2Documenting 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:*

- 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.

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

6.4.3 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.

6.4.4 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

- *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.

6.4.5 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, 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.