Appendix C

Testing the Software Solution

Objectives

Upon completion of this appendix, you should be able to:

- Define three major types of system tests
- Develop a functional test plan based on use cases

Process Map

The Testing workflow is a broad topic. This module introduces several areas of testing, but the focus will be on functional testing. Figure C-1 shows the activity and artifact discussed in this appendix.

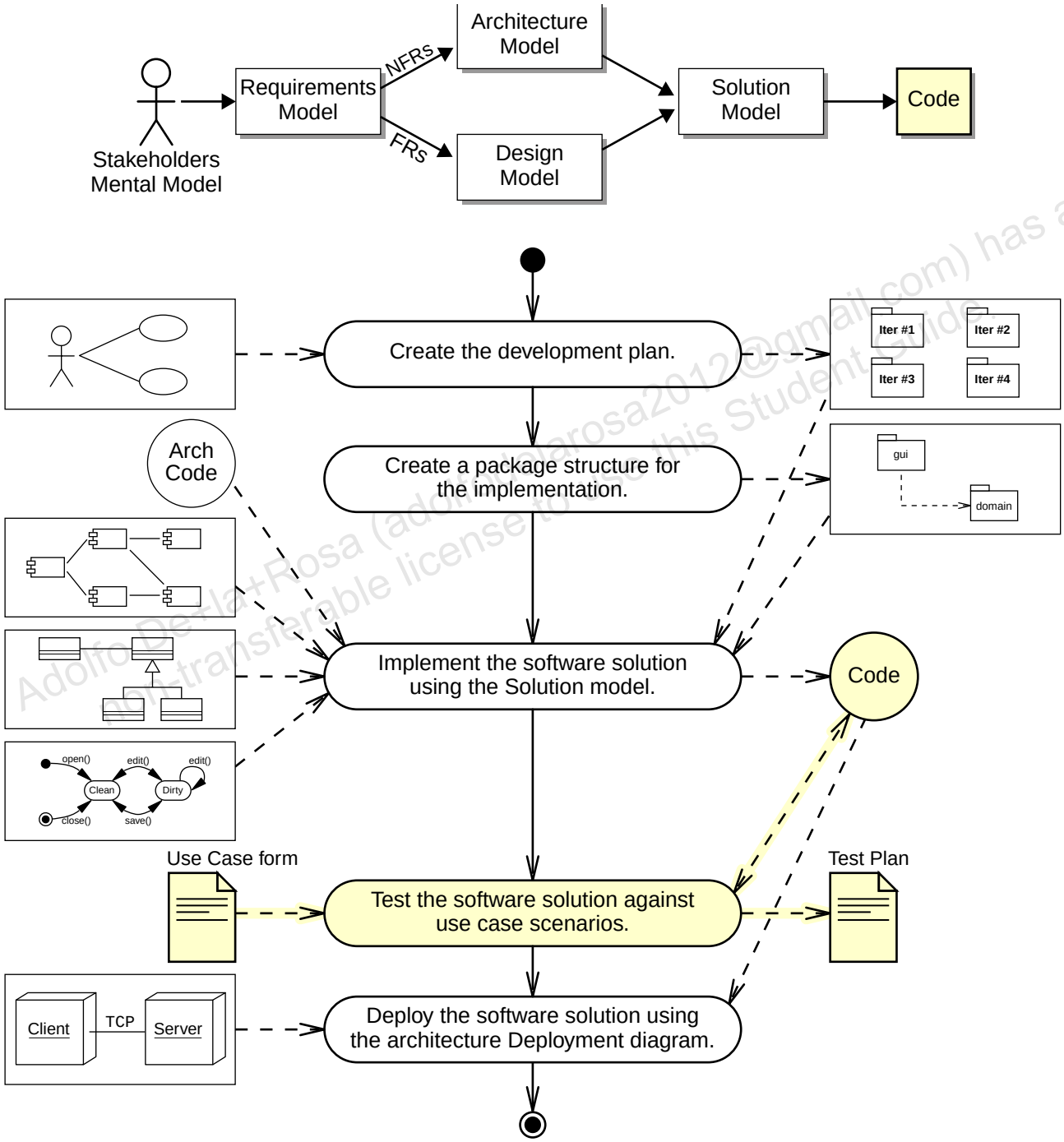

Figure C-1 Testing Workflow Process Map

Defining System Testing

The Test workflow is meant to "verify that the system correctly implements its specification." (Jacobson, Booch, and Rumbaugh page 55) Ideally, testing is performed by a Software Tester; a job role that is independent of the development team. In practice, some testing is performed by developers. The reason why it is important to have independent testers is that the developer is often *too close* to the system being built. Developers are biased by their knowledge of how the system works so it is hard for them to be objective in finding ways of breaking the system. Also, at an unconscious level, most developers do not want to see their system fail.

There are numerous tests that could be performed. The following is a noncomprehensive list:

- Unit

  Unit testing tests individual components or classes.

- Integration

  Integration testing tests the correct functioning of groups of components; in particular, how major subsystems interact.

- Functional

  Functional testing tests the system as a whole against the use cases defined during the Requirements Gathering workflow.

- Regression

  Regressing testing is about going back and retesting things to make sure errors are fixed and that new errors did not occur. Regression testing can be done at any stage of development, but it is most important at the end of an iteration or at the end of a new release. Regressing testing can include unit, integration, functional tests, and so on.

- Load

  Load testing verifies that the system can handle the number of users and frequency of requests that are specified in the nonfunctional requirements. This test requires a complete system (even at increments) deployed on the specified hardware for production to verify that the measurements are realistic.

- Boundary condition

  Boundary condition testing verifies that the components, class, subsystems, and systems can handle inputs that fall on the boundary of acceptable values. For example, if a list is designed to contain no more than ten values, then you should test the addition of the tenth *and* eleventh item to the list. Adding the tenth item should succeed, but the eleventh should fail.

- Usability

  Usability testing verifies that the UI of the system is efficient, clear, correct, and user-friendly. Usability testing should be performed as early as possible. If a UI prototype is created, then this is the ideal time to perform usability tests.

The following subsections discuss the unit, integration, and functional types of testing in more detail.

Unit Tests

> "A test written from the perspective of the programmer." (Beck page 179)

Unit testing tests individual components or classes. Here are a few important concepts:

- Unit tests verify the behavior of individual components.

  For example, you might have an entity class that has a mutator method that takes a date string and stores that as a `Date` object. Then for this example, a good unit test is to pass in several different values including valid dates, strings that look like dates but are not valid (such as "29-February-2002"), and even strings that do not look like dates (such as "xyzzy").

- Unit tests are written by developers.

  Some testing professionals would argue that developers should never write tests. However, most organizations cannot afford to devote a Software Tester to the complete life cycle of the project, so developers usually end up writing unit tests.

There are many techniques for creating unit tests. One technique is to add a `main` method in every class that needs unit testing. However, this technique is a violation of Separation of Concerns (that is, the functionality of the component and the testing of that functionality). A more robust approach is to use a testing framework.

Note – JUnit (`www.junit.org`) is an Open Source framework for testing Java technology components.

Integration Tests

> "Integration testing test multiple components that have each received prior and separate unit testing." (Whittaker page 77)

The focus of integration testing is to determine if large scale components or subsystems work together without error. Here are a few important concepts:

- Integration tests verify the behavior across multiple components, subsystems, and complete systems.

  System testing is a subset of Integration testing.

- Integration tests are usually written by testing professionals or by the development team.

- Integration tests should be built incrementally, just as the system is built incrementally.

  New integration tests are added for each increment. These new tests are added to the suite of regression tests for the complete system.

Functional Tests

> "Most functional tests are written as black box tests working off a functional specification." (Chillarege page 3)

Functional testing tests the system as a whole against the use cases defined during the Requirements Gathering phase. Here are a few important concepts:

- Functional tests verify the behavior of the (complete) system from the perspective of the user.

  Functional tests treat the system as a black box. This means that the internal behavior of the system is completely unknown, only the external inputs and outputs are seen. The use cases define this external behavior.

- Functional tests must be written by testing professionals in conjunction with the user community (if possible).

- Functional tests are based on the use cases of the system plus variations of all scenarios.

  Variations must test failures as well as successes to verify that the system handles failures properly.

Developing a Functional Test

A functional test is written based on a use case of the system. If the development team is using incremental development, then the functional tests can also be developed incrementally. This means that the functional tests for a specific use case are developed in the same iteration as the code for that use case.

The following are the fundamental steps involved in writing a functional test:

1. Identify the inputs for the test based on the inputs specified in the use case scenario.

   You need to create one functional test for every use case scenario. The best way to achieve this level of coverage is to use the use case scenarios. These scenarios will provide the inputs to the tests.

2. Specify the results of the test.

   You must identify the results of the test. There are usually two results to specify. The output of the system determines what appears on the screen or the form of printed reports. The persistent state of the system determines what has changed in the database after the test has been run.

3. Specify the conditions under which the test must be performed.

   You must specify the state of the persistent data store before the tests begin. What data must exist in order to execute the functional tests. For example, in the Hotel Reservation System, the database must include property and room data (and in some cases the DB should have existing reservations).

   Functional tests determine if the system behaves as specified in the requirements. Therefore, these tests are executed one at a time. However, these functional tests can also be used concurrently to see how well the system behaves for nonfunctional requirements. Therefore, functional test cases can be used for Load tests.

4. Write variations on this test to check failure processing, boundary conditions, and so on.

   You should attempt to define the boundary conditions of a specific use case scenario and create tests which verify that the system behaves correctly at the edges of these boundary conditions.

Identifying Functional Test Inputs

Functional test inputs include all data that must be entered into the system through the user interface.

For example, the following are the inputs for a single use case scenario for E1 Create a Reservation:

- The HotelApp is running with the Santa Cruz bed and breakfast.
- The begin and end dates of the reservation are January 31, 2002 and February 5, 2002, respectively.
- The type of room requested is double.
- The Blue room is selected from the room search.
- Jane Googol is entered into the customer search.
- A test credit card number is entered.

Identifying Functional Test Results

Functional test results include any output generated by the system as well as any persistent state change in the system.

For example, the follow are the results of the given use case scenario:

- A new reservation will exist for Jane Googol.
- The state of this reservation will be “confirmed.”
- The reservation has an association with the Blue room (and only the Blue room).
- The payment entity for this reservation records the correct credit card number.

Identifying Functional Test Conditions

Functional test conditions include the existence of specific entities in the database and the existence of stubs for external systems and their data.

For example, the following are the conditions under which the given use case scenario must execute:

- The database contains the Santa Cruz bed and breakfast property and is associated with the Blue room, which is a double.
- Jane Googol is a valid customer in the database.
- A stub exists for the external credit card authorization system with data for test credit cards.
- No other concurrent tests must interact with that credit card account.

Creating Functional Test Variants

Tests variants are based on a use case scenario and explore the *state space* of the system.

For example, the following are test variants for the given use case scenario:

- Include a variant in which there are multiple customer entities for a single name; for example, John Smith.
- Include a variant in which a room search produces no available rooms.
- Include a variant in which the credit card is not authorized and the customer presents a second credit card.

Documenting Test Cases

The following are considerations for documenting functional test cases:

- All test cases should be stored in a Test Plan document.

  The Test Plan should be separate from the SRS.
- Each functional test must trace back to a use case in the SRS document.

  It is important to know that the functional tests were created to verify that the system supports the use cases. In other words, the test designers should not create arbitrary tests that the system was not designed to perform.

  However, it is important to create boundary condition variations on existing use case scenarios.
- If the system is large, then the Test Plan document can be split into separate documents—one for each major increment of the system.

- The Test Plan must be approved by the customer.

  The Test Plan document should be owned by the client-side stakeholders; although, it might be written by the System Tester in the development team.

- Other types of tests (such as load and usability tests) are usually documented separately.

  For large systems, you should separate the test plans and results in separate documents because these documents can become quite large. It is usually best to separate these documents by type of test: functional, integration, load, usability, and so on. However, another alternative is to group the tests by system increment.

Summary

In this appendix, you were introduced to Testing workflow and how to develop functional tests. Here are a few important concepts:

- Testing is critical to the success of a software project.
- Except for unit and integration testing, no tests should be developed and performed by the development team.
- Functional tests are developed from the system's use case scenarios.
- Functional test variants are created to test boundary conditions and system robustness.