**Unit Test Planning & Designing**

**TEST PLAN DEFINITION**

A Software Test Plan is a document describing the testing scope and activities. It is the basis for formally testing any software/product in a project. Following are some of the definitions:

* **test plan**: A document describing the scope, approach, resources and schedule of intended test activities. It identifies amongst others test items, the features to be tested, the testing tasks, who will do each task, degree of tester independence, the test environment, the test design techniques and entry and exit criteria to be used, and the rationale for their choice, and any risks requiring contingency planning. It is a record of the test planning process.
* **master test plan**: A test plan that typically addresses multiple test levels. It is a single high-level test plan for a project/product that unifies all other test plans.
* **phase test plan**: A test plan that typically addresses one test phase.
* **Testing Level Specific Test Plans:** Plans for each level of testing.
  + Unit Test Plan
  + Integration Test Plan
  + System Test Plan
  + Acceptance Test Plan
* **Testing Type Specific Test Plans:**Plans for major types of testing like Performance Test Plan and Security Test Plan.

**TEST PLAN TEMPLATE**

The format and content of a software test plan vary depending on the processes, standards, and test management tools being implemented. However, the following format, which is based on IEEE standard for software test documentation, provides a summary of what a test plan can/should contain.

**Test Plan Identifier:**

* Provide a unique identifier for the document.

**Introduction:**

* Provide an overview of the test plan.
* Specify the goals/objectives.
* Specify any constraints.

**References**:

* List the related documents, with links to them if available, including the following:
  + Project Plan
  + Configuration Management Plan

**Test Items:**

* List the test items (software/products) and their versions.

**Features to be tested:**

* List the features of the software/product to be tested.
* Provide references to the Requirements and/or Design specifications of the features to be tested

**Features Not to Be Tested**:

* List the features of the software/product which will not be tested.
* Specify the reasons these features won’t be tested.

**Approach**:

* Mention the overall approach to testing.
* Specify the testing levels [if it’s a Master Test Plan], the testing types, and the testing methods [Manual/Automated; White Box/Black Box/Gray Box]

**Item Pass/Fail Criteria:**

* Specify the criteria that will be used to determine whether each test item (software/product) has passed or failed testing.

**Suspension Criteria and Resumption Requirements:**

* Specify criteria to be used to suspend the testing activity.
* Specify testing activities which must be redone when testing is resumed.

**Test Deliverables**:

* List test deliverables, and links to them if available, including the following:
  + Test Plan
  + Test Cases
  + Test Scripts
  + Defect/Enhancement Logs
  + Test Reports

**Test Environment:**

* Specify the properties of test environment: hardware, software, network etc.
* List any testing or related tools.

**Estimate:**

* Provide a summary of test estimates (cost or effort) and/or provide a link to the detailed estimation.

**Schedule:**

* Provide a summary of the schedule, specifying key test milestones, and/or provide a link to the detailed schedule.

**Staffing and Training Needs:**

* Specify staffing needs by role and required skills.
* Identify training that is necessary to provide those skills, if not already acquired.

**Responsibilities:**

* List the responsibilities of each team/role/individual.

**Risks:**

* List the risks that have been identified.
* Specify the mitigation plan and the contingency plan for each risk.

**Assumptions and Dependencies:**

* List the assumptions that have been made during the preparation of this plan.
* List the dependencies.

**Approvals:**

* Specify the names and roles of all persons who must approve the plan.
* Provide space for signatures and dates. (If the document is to be printed.)

**TEST PLAN GUIDELINES**

* Make the plan concise. Avoid redundancy. If something is not needed in a section that has been mentioned in the template, delete that section in the test plan.
* Be specific. For example, when you specify an operating system as a property of a test environment, mention the OS Edition/Version as well, not just the OS Name.
* Make use of lists and tables wherever possible. Avoid lengthy paragraphs.
* Have the test plan reviewed a number of times prior to baselining it or sending it for approval. The quality of the test plan speaks volumes about the quality of the testing that a member(s) of the team is going to perform.
* Update the plan as and when necessary. An out-dated and unused document is worse than not having the document in the first place.

**The class as a testable unit**

If the object-oriented paradigm is being used to develop software systems it will need to select the component to be considered for unit test. The choices consist of either the individual method as a unit or the class as a whole. Each of these choices requires special consideration on the part of the testers when designing and running the unit tests, and when retesting needs to be done. For example, in the case of the method as the selected unit to test, it may call other methods within its own class to support its functionality. Additional code, in the form of a test harness, must be built to represent the called methods within the class. Building such a test harness for each individual method often requires developing code equivalent to that already existing in the class itself (all of its other methods). This is costly; however, the tester needs to consider that testing each individual method in this way helps to ensure that all statements/branches have been executed at least once, and that the basic functionality of the method is correct. This is especially important for safety critical methods.

In spite of the potential advantages of testing each method individually, many developers/testers consider the class to be the component of choice for unit testing. The process of testing classes as units is sometimes called component test. A class encapsulates multiple interacting methods operating on common data, so what we are testing is the intraclass interaction of the methods. When testing on the class level it is possible to detect not only traditional types of defects, for example, those due to control or data flow errors, but also defects due to the nature of object-oriented systems, for example, defects due to encapsulation, inheritance, and polymorphism errors.

If the class is the selected component, testers may need to address special issues related to the testing and retesting of these components. Some of these issues are as follows:

**Issue 1: Adequately Testing Classes**

The potentially high costs for testing each individual method in a class will be particularly apparent when there are many methods in a class; the numbers can reach as high as 20 to 30. If the class is selected as the unit to test, it is possible to reduce these costs since in many cases the methods in a single class serve as drivers and stubs for one another. This has the effect of lowering the complexity of the test harness that needs to be developed. However, in some cases driver classes that represent outside classes using the methods of the class under test will have to be developed. In addition, if it is decided that the class is the smallest component to test, testers must decide if they are able to adequately cover all necessary features of each method in class testing.

Many different sequences and combination of calls are possible for even a simple class. Exhaustively testing every possible sequence is usually not practical. The tester must select those sequences he/she believes will reveal the most defects in the class. Finally, a tester might use a combination of approaches, testing some of the critical methods on an individual basis as units, and then testing the class as a whole.

**Issue 2: Observation of Object States and State Changes**

Methods may not return a specific value to a caller. They may instead change the state of an object. The state of an object is represented by a specific set of values for its attributes or state variables. State-based testing is very useful for testing objects. Methods will often modify the state of an object, and the tester must ensure that each state transition is proper. The test designer can prepare a state table (using state diagrams developed for the requirements specification) that specifies states that object can assume, and then in the table indicate sequence of messages and parameters that will cause the object to enter each state. When the tests are run the tester can enter results in this same type of table.

Testers should have the option of going back to the designers and requesting changes that make a class more testable. In any case, test planners should insure that code is available to display state variables. Test plans should provide resources for developing this type of code.

**Issue 3: The Retesting of Classes—I**

One of the most beneficial features of object-oriented development is encapsulation. This is a technique that can be used to hide information. A program unit, in this case a class, can be built with a well-defined public interface that proclaims its services (available methods) to client classes. The implementation of the services is private. Clients who use the services are unaware of implementation details. As long as the interface is unchanged, making changes to the implementation should not affect the client classes. A tester of object-oriented code would therefore conclude that only the class with implementation changes to its methods needs to be retested. Client classes using unchanged interfaces need not be retested.

This is not necessarily correct. In an object-oriented system, if a developer changes a class implementation that class needs to be retested as well as all the classes that depend on it. If a superclass, for example, is changed, then it is necessary to retest all of its subclasses. In addition, when a new subclass is added (or modified), we must also retest the methods inherited from each of its ancestor superclasses. The new (or changed) subclass introduces an unexpected form of dependency because there now exists a new context for the inherited components.

**Issue 4: The Retesting of Classes—II**

Classes are usually a part of a class hierarchy where there are existing inheritance relationships. Subclasses inherit methods from their superclasses. Very often a tester may assume that once a method in a superclass has been tested, it does not need retested in a subclass that inherits it.

However, in some cases the method is used in a different context by the subclass and will need to be retested. In addition, there may be an overriding of methods where a subclass may replace an inherited method with a locally defined method. Not only will the new locally defined method have to be retested, but designing a new set of test cases may be necessary. This is because the two methods (inherited and new) may be structurally different.

Suppose the shape superclass has a subclass, triangle, and triangle has a subclass, equilateral triangle. Also suppose that the method *display* in shape needs to call the method *color* for its operation. Equilateral triangle could have a local definition for the method *display.* That method could in turn use a local definition for *color* which has been defined in triangle. This local definition of the *color* method in triangle has been tested to work with the inherited *display* method in shape, but not with the locally defined *display* in equilateral triangle. This is a new context that must be retested. A set of new test cases should be developed.

The tester must carefully examine all the relationships between members of a class to detect such occurrences.

**The Test Harness**

In addition to developing the test cases, supporting code must be developed to exercise each unit and to connect it to the outside world. Since the tester is considering a stand-alone function/procedure/class, rather than a complete system, code will be needed to call the target unit, and also to represent modules that are called by the target unit. This code called the test harness, is developed especially for test and is in addition to the code that composes the system under development. The test harness is shown in the following figure and it is defined as follows:



**The auxiliary code developed to support testing of units and components is called a test harness. The harness consists of drivers that call the target code and stubs that represent modules it calls.**

The development of drivers and stubs requires testing resources. The drivers and stubs must be tested themselves to insure they are working properly and that they are reusable for subsequent releases of the software. Drivers and stubs can be developed at several levels of functionality. For example, a driver could have the following options and combinations of options:

**(i)** call the target unit;

**(ii)** do 1, and pass inputs parameters from a table;

**(iii)** do 1, 2, and display parameters;

**(iv)** do 1, 2, 3 and display results (output parameters).

The stubs could also exhibit different levels of functionality. For example a stub could:

**(i)** display a message that it has been called by the target unit;

**(ii)** do 1, and display any input parameters passed from the target unit;

**(iii)** do 1, 2, and pass back a result from a table;

**(iv)** do 1, 2, 3, and display result from table.

Drivers and stubs as shown in the figure are developed as procedures and functions for traditional imperative-language based systems. For object-oriented systems, developing drivers and stubs often means the design and implementation of special classes to perform the required testing tasks. The test harness itself may be a hierarchy of classes. For example, in the figure the driver for a procedural system may be designed as a single procedure or main module to call the unit under test; however, in an object-oriented system it may consist of several test classes to emulate all the classes that call for services in the class under test.

The test planner must realize that, the higher the degree of functionally for the harness, the more resources it will require to design, implement, and test. Developers/testers will have to decide depending on the nature of the code under test, just how complex the test harness needs to be. Test harnesses for individual classes tend to be more complex than those needed for individual procedures and functions since the items being tested are more complex and there are more interactions to consider.

**Running the Unit Tests and Recording Results**

Unit tests can begin when

1. the units becomes available from the developers (an estimation of availability is part of the test plan)
2. the test cases have been designed and reviewed
3. the test harness, and any other supplemental supporting tools, are available. The testers then proceed to run the tests and record results.

The status of the test efforts for a unit, and a summary of the test results, could be recorded in a simple format such as shown in the following table:

|  |
| --- |
| Unit test Worksheet |
| Unit Name: |
| Unit Identifier: |
| Tester: |
| Date: |
| Test case ID Status(run/not run) Summary of results Pass/Fail |

These forms can be included in the test summary report, and are of value at the weekly status meetings that are often used to monitor test progress. It is very important for the tester at any level of testing to carefully record, review, and check test results. The tester must determine from the results whether the unit has passed or failed the test. If the test is failed, the nature of the problem should be recorded in what is sometimes called a test incident report. Differences from expected behavior should be described in detail. This gives clues to the developers to help them locate any faults. During testing the tester may determine that additional tests are required. For example, a tester may observe that a particular coverage goal has not been achieved. The test set will have to be augmented and the test plan documents should reflect these changes.

When a unit fails a test there may be several reasons for the failure. The most likely reason for the failure is a fault in the unit implementation (the code). Other likely causes that need to be carefully investigated by the tester are the following:

* a fault in the test case specification (the input or the output was not specified correctly)
* a fault in test procedure execution (the test should be rerun)
* a fault in the test environment (perhaps a database was not set up properly)
* a fault in the unit design (the code correctly adheres to the design specification, but the latter is incorrect).

The causes of the failure should be recorded in a test summary report, which is a summary of testing activities for all the units covered by the unit test plan. Ideally, when a unit has been completely tested and finally passes all of the required tests it is ready for integration. Under some circumstances a unit may be given a conditional acceptance for integration test. This may occur when the unit fails some tests, but the impact of the failure is not significant with respect to its ability to function in a subsystem, and the availability of a unit is critical for integration test to proceed on schedule. This a risky procedure and testers should evaluate the risks involved. Units with a conditional pass must eventually be repaired. When testing of the units is complete, a test summary report should be prepared. This is a valuable document for the groups responsible for integration and system tests. It is also a valuable component of the project history. Its value lies in the useful data it provides for test process improvement and defect prevention. Finally, the tester should insure that the test cases, test procedures, and test harnesses are preserved for future reuse.

**Designing Integration Tests**

Integration tests for procedural software can be designed using a black or white box approach. Both are recommended. Some unit tests can be reused. Since many errors occur at module interfaces, test designers need to focus on exercising all input/output parameter pairs, and all calling relationships. The tester needs to insure the parameters are of the correct type and in the correct order. The author has had the personal experience of spending many hours trying to locate a fault that was due to an incorrect ordering of parameters in the calling routine. The tester must also insure that once the parameters are passed to a routine they are used correctly.

For example, in the following figure, Procedure\_b is being integrated with Procedure\_a. Procedure\_a calls Procedure\_b with two input parameters in3, in4. Procedure\_b uses those parameters and then returns a value for the output parameter out1. Terms such as *lhs* and *rhs* could be any variable or expression. The parameters could be involved in a number of *def* and/or *use* data flow patterns. The actual usage patterns of the parameters must be checked at integration time. Dataflow–based (def-use paths) and control flow (branch coverage) test data generation methods are useful here to insure that the input parameters, in3, in4, are used properly in Procedure\_b.

other modules

|  |
| --- |
| Procedure\_a(in1, in2, out2) |
| … |
| in3 = rhs |
| in4 = rhs |
| … |
| call Procedure\_b(in3, in4, out1) |
| … |
| lhs = out1 |
| … |
| out2 = rhs |

in3 out1

In4

|  |
| --- |
| Procedure\_a(in3, in4, out1) |
| … |
| if (in3 …) |
| … |
| end if |
|  |
|  |
| lhs = in4 |
| … |
| out1 = rhs |

Other modules

Example: integration of two procedures

Again data flow methods (def-use pairs) could also be used to check that the proper sequence of data flow operations is being carried out to generate the correct value for out1 that flows back to Procedure\_a.

Black box tests are useful in this example for checking the behavior of the pair of procedures. For this example test input values for the input parameters in1 and in2 should be provided, and the outcome in out2 should be examined for correctness. For conventional systems, input/output parameters and calling relationships will appear in a structure chart built during detailed design. Testers must insure that test cases are designed so that all modules in the structure chart are called at least once, and all called modules are called by every caller. Coverage requirements for the internal logic of each of the integrated units should be achieved during unit tests. Some black box tests used for module integration may be reusable from unit testing.

Sources for development of black box or functional tests at the integration level are the requirements documents and the user manual. Testers need to work with requirements analysts to insure that the requirements are testable, accurate, and complete. Black box tests should be developed to insure proper functionally and ability to handle subsystem stress.

For example, in a transaction-based subsystem the testers want to determine the limits in number of transactions that can be handled. The tester also wants to observe subsystem actions when excessive amounts of transactions are generated. Performance issues such as the time requirements for a transaction should also be subjected to test. These will be repeated when the software is assembled as a whole and is undergoing system test.

**Integration Test Planning**

Integration test must be planned. Planning can begin when high-level design is complete so that the system architecture is defined. Other documents relevant to integration test planning are the requirements document, the user manual, and usage scenarios. These documents contain structure charts, state charts, data dictionaries, cross-reference tables, module interface descriptions, data flow descriptions, messages and event descriptions, all necessary to plan integration tests. The strategy for integration should be defined. For procedural-oriented system the order of integration of the units of the units should be defined. This depends on the strategy selected. Consider the fact that the testing objectives are to assemble components into subsystems and to demonstrate that the subsystem functions properly with the integration test cases. For object-oriented systems a working definition of a cluster or similar construct must be described, and relevant test cases must be specified. In addition, testing resources and schedules for integration should be included in the test plan.

One of the goals of integration test is to build working subsystems, and then combine these into the system as a whole. When planning for integration test the planner selects subsystems to build based upon the requirements and user needs. Very often subsystems selected for integration are prioritized. Those that represent key features, critical features, and/or user-oriented functions may be given the highest priority. Developers may want to show clients that certain key subsystems have been assembled and are minimally functional.

**System Test: The Different Types**

When integration tests are completed, a software system has been assembled and its major subsystems have been tested. At this point the developers/testers begin to test it as a whole. System test planning should begin at the requirements phase with the development of a master test plan sand requirements-based (black box) tests. System test planning is a complicated task. There are many components of the plan that need to be prepared such as test approaches, costs, schedules, test cases, and test procedures.

System testing itself requires a large amount of resources. The goal is to ensure that the system performs according to its requirements. System test evaluates both functional behavior and quality requirements such as reliability, usability, performance and security. This phase of testing is especially useful for detecting external hardware and software interface defects, for example, those causing race conditions, deadlocks, problems with interrupts and exception handling, and ineffective memory usage. After system test the software will be turned over to users for evaluation during acceptance test or alpha/beta test. It has to be insured that the quality of the software has been measured and evaluated before users/clients are invited to use the system. In fact system test serves as a good rehearsal scenario for acceptance test.

Because system test often requires many resources, special laboratory equipment, and long test times, it is usually performed by a team of testers. The best scenario is for the team to be part of an independent testing group. The team must do their best to find any weak areas in the software; therefore, it is best that no developers are directly involved.

The several types of system tests are as follows:

• Functional testing

• Performance testing

• Stress testing

• Configuration testing

• Security testing

• Recovery testing

Not all software systems need to undergo all the types of system testing. Test planners need to decide on the type of tests applicable to a particular software system. Decisions depend on the characteristics of the system and the available test resources.

**Regression Testing**

Regression testing is not a level of testing, but it is the retesting of software that occurs when changes are made to ensure that the new version of the software has retained the capabilities of the old version and that no new defects have been introduced due to the changes. Regression testing can occur at any level of test, for example, when unit tests are run the unit may pass a number of these tests until one of the tests does reveal a defect. The unit is repaired and then retested with all the old test cases to ensure that the changes have not affected its functionality. Regression tests are especially important when multiple software releases are developed. Users want new capabilities in the latest releases, but still expect the older capabilities to remain in place. This is where regression testing plays a role. Test cases, test procedures, and other test-related items from previous releases should be available so that these tests can be run with the new versions of the software. Automated testing tools support testers with this very time-consuming task.

**Alpha, Beta, and Acceptance Tests**

The clients along with test planners design the actual test cases that will be run during acceptance test. After the software has passed all the system tests and defect repairs have been made, the users take a more active role in the testing process. Developers/testers must keep in mind that the software is being developed to satisfy the users requirements, and no matter how elegant its design it will not be accepted by the users unless it helps them to achieve their goals as specified in the requirements. Alpha, beta, and acceptance tests allow users to evaluate the software in terms of their expectations and goals.

When software is being developed for a specific client, acceptance tests are carried out after system testing. The acceptance tests must be planned carefully with input from the client/users. Acceptance test cases are based on requirements. The user manual is an additional source for test cases. System test cases may be reused. The software must run under real-world conditions on operational hardware and software. The software-under-test should be stressed. For continuous systems the software should be run at least through a 25-hour test cycle. Conditions should be typical for a working day. Typical inputs and illegal inputs should be used and all major functions should be exercised. If the entire suite of tests cannot be run for any reason, then the full set of tests needs to be rerun from the start.

Acceptance tests are a very important milestone for the developers. At this time the clients will determine if the software meets their requirements. Contractual obligations can be satisfied if the client is satisfied with the software. Development organizations will often receive their final payment when acceptance tests have been passed.

Acceptance tests must be rehearsed by the developers/testers. There should be no signs of unprofessional behavior or lack of preparation. Clients should be provided with documents and other material to help them participate in the acceptance testing process, and to evaluate the results. After acceptance testing the client will point out to the developers which requirement have/have not been satisfied. Some requirements may be deleted, modified, or added due to changing needs. If the client has been involved in prototype evaluations then the changes may be less extensive.

If the client is satisfied that the software is usable and reliable, and they give their approval, then the next step is to install the system at the client’s site. If the client’s site conditions are different from that of the developers, the developers must set up the system so that it can interface with client software and hardware. Retesting may have to be done to insure that the software works as required in the client’s environment. This is called installation test.

If the software has been developed for the mass market (shrink wrapped software), then testing it for individual clients/users is not practical or even possible in most cases. Very often this type of software undergoes two stages of acceptance test. The first is called alpha test. This test takes place at the developer’s site. A cross-section of potential users and members of the developer’s organization are invited to use the software. Developers observe the users and note problems. Beta test sends the software to a cross-section of users who install it and use it under real world working conditions. The users send records of problems with the software to the development organization where the defects are repaired sometimes in time for the current release. In many cases the repairs are delayed until the next release.