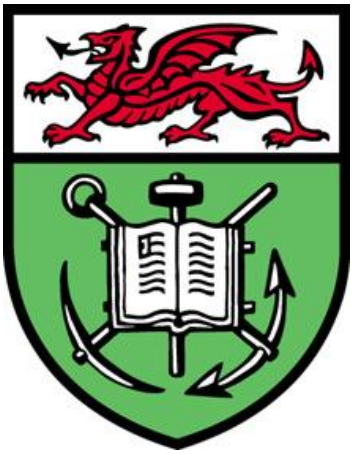
University of Wales Swansea Computer Science Department

Levels of Testing

Advanced topics in computer science



Written by: Patrick Oladimeji
Supervised by: Prof. Dr. Holger Schlingloff & Dr. Markus Roggenbach
1/12/2007

Contents

1. INTRODUCTION

Historically, software testing has been known to consume about 50% of development cost and span about 50% of the development time. This clearly points towards the fact that software testing is a very important aspect of software development. Despite this level of importance, software testing still remains a subject area we have relatively little knowledge of compared to other aspects of software engineering and software development [1].

This lack of knowledge is easily tied to the complexity of software testing. For example in order to completely test a relatively simple program, the number of test cases needed might be in the tens of thousands.

Another problem often faced while testing is with the quality of testing produced. This is due to a general misconception of the meaning of testing. Many programmers believe testing to be the process of showing that a program contains no errors or that testing is the process of showing that a program correctly performs its intended functions. The problem with the first definition is that it is virtually impossible to prove that a program contains no errors. The problem with the other definition is that if you set out to show that a program correctly performs its intended functions then you are in danger of not discovering hidden malicious or unwanted features in the program which clearly mean the program is incorrect.

A more appropriate view of testing would be to move from the notion of showing that a program works to a notion of finding errors in a program. This is far more practical as the assumption (that a program contains errors) is true for almost any program.

2. LEVELS OF TESTING

To enhance the quality of software testing, and to produce a more unified testing methodology applicable across several projects, the testing process could be abstracted to different levels. This classification into different levels introduces some parallelism in the testing process as multiple tests could be performed simultaneously. Although these levels are best suited for the waterfall model, (since the levels correspond directly to the different stages in the waterfall model) the level of abstraction still poses to be useful across other software development models. We shall be looking at:

- Unit testing
- Integration testing
- System testing
- Acceptance testing
- Regression testing

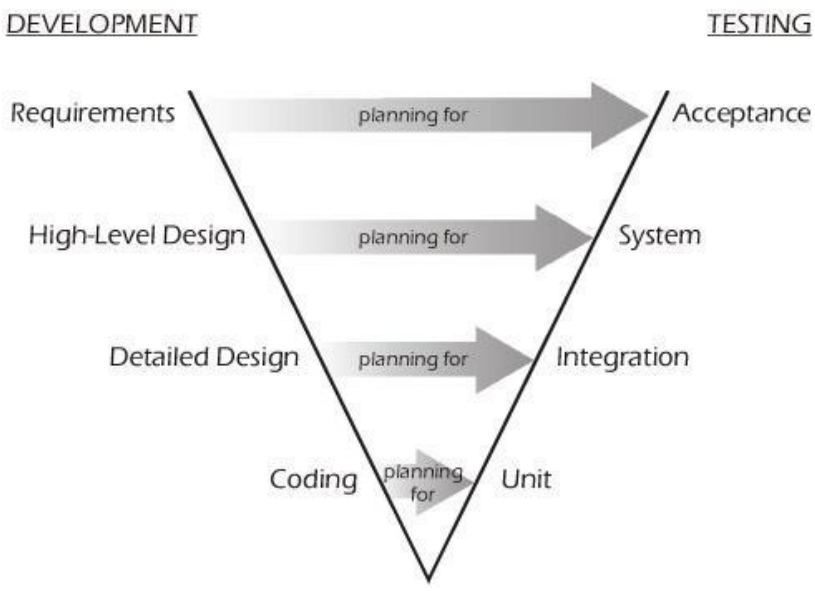


Figure 2-1: The "V" model of software testing

The levels of testing (with the exception of Regression Testing which happens everytime the system is modified) have a hierarchical structure which builds up from the bottom up – where higher levels assume successful and satisfactory completion of lower level tests. Without this structured approach, a big-bang testing approach (which combines the different components at once and tries to test the combination as a system) would be a very daunting task [2]. It is normally required for unit testing to be done before integration testing which is done before system testing and acceptance testing.

Each level of test is characterised by an environment i.e. a type of people, hardware, software, data and interface. For example the people involved in unit testing are mainly programmers, while testers are mainly those involved in system testing. We also find that unit and integration testing are generally done in the programmer's IDE (integrated development environment) while system testing is done on a system test machine or a simulated environment [3].

These environment variables vary from project to project as does the number of levels of test. Apart from the five levels of testing identified above, other testing levels that are considered include alpha testing, beta testing, compatibility testing, stress testing and security testing.

In general before testing begins, the Test Manager should compile a Master Test Plan which should contain a detailed structure of how the other test plans would be executed. It is left to the Test Manager to determine how many levels of test would be required for a given project. Hence, it is necessary for the Test Manager to have a thorough understanding of

the requirements specification of the system. This enables him to avoid overlapping test levels and also avoid gaps between test levels. Some of the factors that help in deciding how many levels of tests are needed for a project include budget for the project, complexity of the project, time scope for the project and staffing.

2.1 Formality across testing levels

The general structure of formal testing (as suggested by the IEEE 829-1998) should consist of the following processes:

- Test planning
- Test set gathering
- Test result verification

Test planning generally entails planning time management, resource allocation (who performs the tests) and enumerating the features that will be tested. This process usually produces a document called the Level Test Plan document.

The test set gathering process requires generating appropriate test cases and carrying out the test plan laid out in the test planning stage. This stage usually produces the Test Case Specification document.

The final stage is concerned with running the test cases, checking for termination and comparing the test set result with the expected result. This stage usually produces the Test Log document which contains a record of what test cases were run, the order in which they were run, who ran the test cases and whether each test passed or failed.

The level of formality and documentation necessary during testing varies from project to project. The customer involved in the project is an important factor in the matter. For example an internal project (e.g. software produced for use within the producing company) would probably need a lot less formality (protocols) and documentation compared to an external project where specific standards are required.

In general, testing documentation are necessary because they could be used to verify the adequacy of a given test suite. They could also be used to repeat test cases for the sake of regression testing. As a general rule of thumb, testing documentation should be *reviewable*, *repeatable* and *archivable* [2].

The rest of this paper proceeds by expanding on each of the level of test enumerated above, giving examples to better explain the concept at each level.

2.2 UNIT TESTING

A unit is the smallest piece of software that can be tested. This usually means the software can be compiled, linked or loaded into memory. A typical example in a procedural programming language would be a function/procedure or a group of these contained in a source file. In an object-oriented programming language, this typically refers to simple classes and interfaces.

An example unit testing session would involve a number of calls to the unit (function, procedure or method) under test where each call might be preceded by a setup code that generates appropriate method parameters wherever required and another call performed after the test to check whether the test was successful or unsuccessful.

Unit testing is the lowest testing level. It helps to refine the testing process so that reasonable system reliability can be expected when testing is performed on the next hierarchical levels. Testing at the unit level helps to expose bugs that might appear to be hidden if a big-bang testing approach was used and unit testing omitted. When these stealth bugs are not uncovered and corrected they could cause unexpected system faults or crashes when running the system.

Unit testing is usually seen as a *white-box* testing technique. This is because its main focus is on the implementation of the unit being tested i.e. the class, interface, function or method under test. Unit testing seeks to find if the implementation satisfies the functional specification. It is not unusual to have system level requirements tested at the unit level for example a system might specify in its functional requirements to have a detailed complex algorithm. The most efficient way to test this algorithm would be at the unit level [2].

Since unit testing focuses on implementation and requires thorough understanding of the systems functional specification, it is usually performed by the developers. What then happens is that the programmer writes test cases after s/he has implemented the program. The obvious problem with this is that the test cases are unconsciously written to suit the programmer's implementation rather than the initial functional specification.

2.2.1 Buddy Testing

A better approach that has been encouraged is the use of a process called "*Buddy Testing*" [3]. Buddy testing takes a team approach to code implementation and unit testing. Developers working on the same project are paired up so that programmer A codes to implementation while programmer B performs the unit tests and vice-versa. In this case the test cases are written by the programmer in charge of testing and this process is totally

independent of the implementation. As a matter of fact, the test cases are written before implementation begins (at the unit test planning stage when the detailed functional specifications are ready).

The advantages of the team approach over the single worker approach are

- The testing process is more objective and productive since the test cases are written by a second party before implementation and the process is unbiased;
- The actual implementation models the initial program specification better since the programmer has the test cases ready before he starts coding;
- It provides a bit of staff cross-training on the project. This is because the tester requires a good level of understanding for the module being implemented. This way if programmer A is unavailable for any reason then programmer B has a good knowledge of the code implementation.

Unit testing must be performed to such a level as to provide sufficient confidence in the quality level of the system under test. This allows for tests at higher levels (integration and system tests) to focus on system behaviours that are at a higher level of abstraction.

Sufficient confidence in a unit can be usually obtained by measuring “code coverage”. Code coverage is usually determined by:

- Statement coverage – which tests that each statement in a unit source file is executed at least once;

- Decision coverage – which tests that each Boolean value of a conditional statement occurs at least once.

2.2.2 Automating unit testing

Unit testing is usually an automated process and performed within the programmers IDE. Many plug-ins exist to allow for unit testing from within an IDE. A good example of a framework that allows automated unit testing is JUNIT (a unit testing framework for java). xUnit [20] is a more general framework which supports other languages like C#, ASP, C++, Delphi and Python to name a few.

Test case generation could also be automated to some extent. JUB, JUnit test case Builder is a test case generator framework for use with java and JUNIT [5]. Recent research by Andrews J. H et al. [4] has proposed a framework for generating unit test cases. Results show that their system – called RUTE-J (a randomised unit testing engine for java) can be an effective and efficient method of testing. It works by allowing programmers select optimal ranges for test case parameters and the engine constructs appropriate test case arguments from this set.

Stots et al. [6] also developed a system for systematically generating complete test class for JUnit. Their system, JAX (JUnit Axioms) [6] is based on Gutag's [7] algebraic semantics of abstract data types.

2.3 INTEGRATION TESTING

The beginning of integration testing assumes a reliable and complete set of unit tests. At this stage we begin to combine the different tested units or components to form a working subsystem. Despite the fact that a unit has been through a successful unit test, it might still behave unpredictably when interacting with other components in the system. Hence the

objective of integration testing is to ensure correct interaction and interfacing between the units in a software system as defined in the detailed design specifications.

The integration test planning process begins as soon as the system's detailed design specifications are ready. Integration testing is mostly done by the developers and is usually performed in the development environment.

The need for integration testing must not be overlooked in the testing process. According to Li et al. [8], approximately 40% of software errors are revealed during integration testing.

Moreover, many systems are now built from existing libraries of code. It is now common practice to purchase a library of components (or download an open source project) for use in other larger projects. All that is required from the programmer is the ability to write wrapper code in order to glue the components together and expose the required functionality of the given library. Even though these libraries are thoroughly tested, it is still necessary to test for interoperability between these libraries and other components of the system since the libraries were not originally built for the system.

Even when external libraries are not being used, different units get implemented by different programmers. Different programmers tend to have different programming style and consequently the difference in implementation might cause unexpected results when the components start to interface.

2.3.1 Forms of Integration Testing

Integration testing can take many different forms. These include:

- "Big-bang" integration
- Bottom-up integration

- Top-down integration
- Sandwich integration

The above named integration testing techniques are decomposition-based integration testing techniques. By this, it is meant that the system is functionally decomposed into a tree-like structure. Starting from the main program, we break down each subsystem using the list of units it calls. These units become the child nodes of the starting node and they are further broken down into their own constituent units. The units at the leaves of the tree are those that have no calls to other procedures or functions.

A partial functional decomposition of the simple automated teller machine system can be found below. We start with the main system and check the possible function calls from the system. These are calls to the Terminal I/O, Manage Sessions and Conduct Transaction functions. The decomposition goes on with each node till a leaf node is reached.

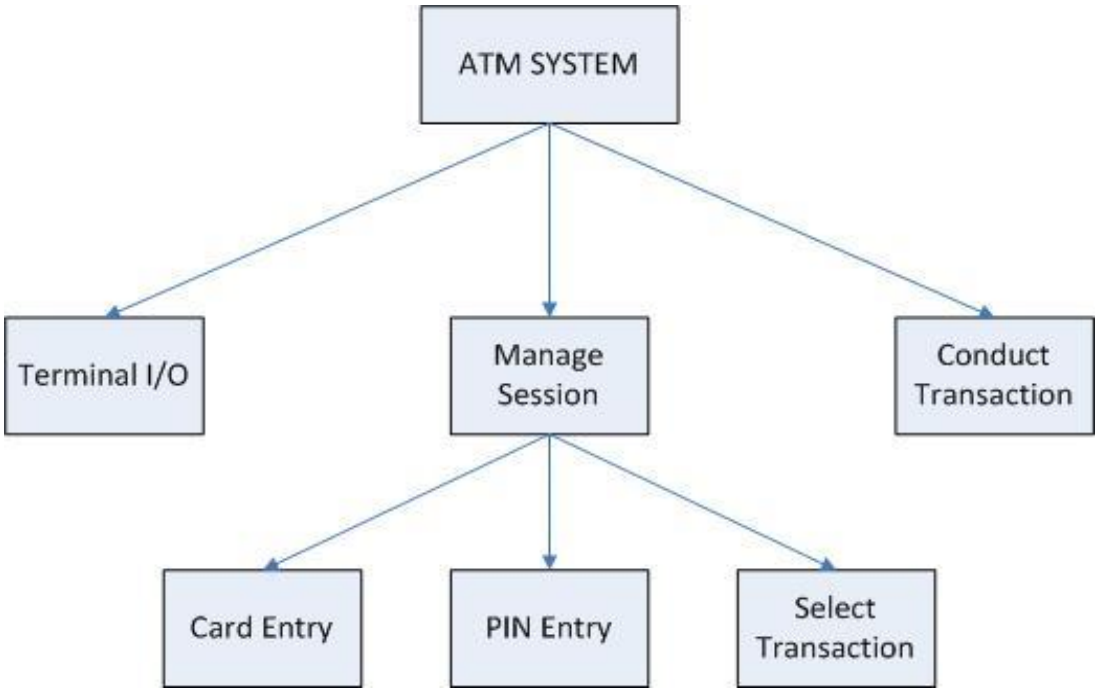


Figure 2-2: A partial functional decomposition of the SATM

2.3.1.1 "Big-bang" integration testing

The big-bang approach to integration testing is a very simple approach. It is regarded as a simple approach as there is no need to write throw-away code (stubs and drivers) as we shall see in the cases of the other three approaches. In addition little planning and resources are needed for the big-bang approach. As a result it is very common amongst many novice programmers. It involves combining all the tested units and performing tests on the combined system as a whole. The usual problem with this approach is the level of complexity involved in locating the source of errors when unexpected events occur.

2.3.1.2 Bottom-up integration testing

In the bottom-up approach, testing starts at the bottom of the tree (the leaves). In the case of the simple automated teller machine (SATM) system testing would start with the three leaf units i.e. the Card Entry, PIN Entry and the Select Transaction units. Testing then proceeds upwards towards the root of the tree till the entire system is covered.

At each stage in bottom-up integration, the units at the higher levels are replaced by *drivers. Drivers* are throw-away pieces of code that are used to simulate procedure calls to child units [9]. They are also responsible for passing test case parameters to the child units under test, collecting the returned values and carrying out verification on the results to determine whether the test was successful or not. For example if running an integration test for the SATM system, the manage session unit would be replaced by a *driver* that calls its three child units.

The good thing about the bottom-up approach is that it usually requires less throw-away code (*drivers*) than in the top-down approach (which requires *stubs*) as we shall see shortly. This is a as a result of the structural property of trees.

A disadvantage of the bottom-up strategy is the lack of a system prototype until towards the end of the testing process (which is contrary to the case of the top-down approach as we shall see shortly).

2.3.1.3 Top-down integration testing

Top-down integration testing starts from the top and then proceeds to its child units. At each testing phase, the units at the lower levels are replaced by *stubs*. *Stubs* are like drivers in the sense that they are both temporarily replacing real world code. They are however quite different in intended functionality. A stub mimics the process of receiving an object as input and giving the required output based on predefined properties just like the real code would do.

Stubs are used to feed test cases to their parent modules during testing. In order to run the tests with multiple test cases, either one of two approaches could be used. The first approach is to write multiple versions of the *stub* giving them different test case values. Alternatively, test data could be placed in external files so that the *stub* can read the data from the files on different test executions.

Two general guidelines to consider when planning for a sequence of executing the modules using top down approach are:

- Critical sections of the system should be added to the sequence as early as possible.
- Modules containing I/O should be added as early as possible.

The motivation for the second is mainly so that test case representation can be simplified as early as possible [1].

Stubs can also be very useful hardware replacements in a testing scenario. They could easily be used to emulate communication of the system with an external device. This is particularly useful when the real hardware is unavailable for testing purposes. For example in the SATM system, the PIN entry unit could be replaced by an emulator (stub) that sends PIN information to the manage sessions unit.

A good advantage of this approach is the concept of an early system prototype. The drawback of this approach is the extra coding involved in writing stubs. Besides writing an emulator could be a very complex task [10].

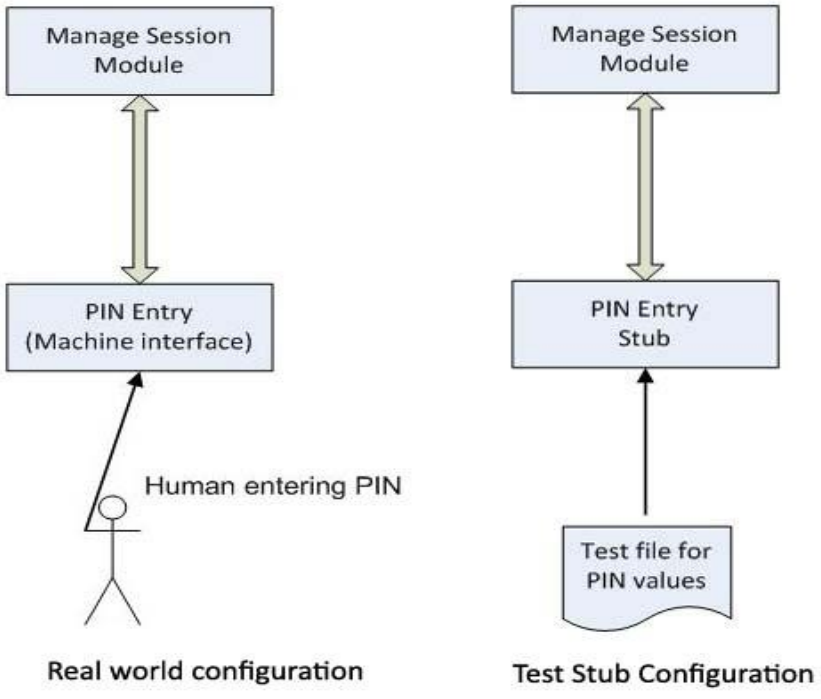


Figure 2-3: a modelled view of the difference between a stub configuration and a real world configuration

2.3.1.4 Sandwich integration testing

The sandwich approach combines both top-down and bottom-up integration. The system is divided into two sections. Units in the higher sections are tested using the top-down integration approach while units in the lower sections are tested using bottom-up integration. The combination of the use of stubs and drivers is done to optimise the amount of throw-away code written.

The sandwich approach uses less throw-away code than the top-down approach but still contains one of the advantages of using the top-down approach – namely the availability of a system prototype at an early stage in the testing phase.

There exist different variations to the sandwich approach. The underlying difference between them remains the way the system is divided into sections.

2.3.2 Test automation for integration testing

UML diagrams are very useful when performing integration tests because test cases could be automatically derived from statechart diagrams and use-case diagrams. Siemens Corporate Research [11] has addressed this exact issue of automating test case generation and execution with UML. This helps to achieve a design based testing environment. What is then required from the developers is a detailed definition of the system's dynamic behaviour using UML Statechart diagrams and a good description of the test requirements. It is from these that test cases are generated and executed.

Recent research is looking into other frameworks for automatic generation of integration tests. Substra [12] is a good example of such a system. Substra's framework is based on the fact that integration tests are concerned with the way the components of a software system interface or interact with one another. Well designed components usually interact with other components via their interfaces. Substra then generates integration tests based on the sequence of method calls obtained from an initial run of the subsystem under test.

2.4 SYSTEM TESTING

System testing begins after completion of integration testing. System testing is done to prove that the system implementation does not meet the system requirements specification. Test planning for system testing is usually one of the first to be processed as all that is needed is the system requirements specifications and this is usually available very early in the project development lifecycle. System test planning and system testing are normally performed by a test team if there is one.

System test planning phase is very dependent on the high-level design specification in the development process. As a result any errors made in translating the requirements specification and the design specification would be very drastic as it would propagate downwards to the lower levels of test and development.

2.4.1 Forms of testing under system testing

System testing is normally run on a system test machine and usually in a specially configured environment that simulates the end user environment as realistically as possible. When designing test cases for system testing several categories of testing should be considered [1]. The use of some or all these testing categories varies from project to project because not all of them are always applicable.

Facility Testing

This aims to determine that all functionality listed in the user requirement specification has been implemented. Facility testing could be performed without a computer as a checklist of implemented functionality would suffice.

Volume Testing

In volume testing, the program under test is subjected to huge volumes of data to see how it copes. For example a graphics application would be fed a ridiculously large bitmap file to edit or a compiler

would be fed with source files containing outrageous amounts of code. The aim of volume testing is to show that the system cannot handle the amount of data it has specified in its objectives [1].

Stress Testing

Stress testing puts the program under heavy load or stress. According to *Glenford Myers* [1],

> "…a heavy stress is a peak volume of data or activity, encountered ***over a short period of time."***

Stress testing is very easily confused with volume testing. However, the key difference is the element of ***time*** involved in stress testing. Borrowing from a good example from *Glenford Myers* ***"The Art of Software Testing, Second Edition",*** while a volume test would test if a typist could cope with typing up a large report, a stress tests would test if the typist could type at a rate of 55 words per minute.

Stress testing is usually applicable to programs with varying loads. A good candidate for stress testing is an online application that interacts with a database. A typical stress test would try to determine the maximum number of concurrent users the application could handle and also explore the systems reaction when the number of concurrent users exceeds the maximum.

Usability Testing

Usability testing is concerned with determining how easily a human would interface with the system or utilise the system. When designing usability test cases it is important to study the end users the program is targeted towards. Important factors to consider would be the age of the audience, the educational background and possibly the inclusion of accessibility features for the disabled should be considered. It is also important to ensure that the program outputs meaningful, non-offensive messages.

Security Testing

Security tests attempts to break the program's security. Test cases would normally be designed to attempt to access resources that should not be accessible at all or without the required privileges. An example would be an attempt to corrupt the data in a database system. Internet based application are a good candidate for security testing due to the continuous growth in the number of e-commerce applications.

Performance Testing

Performance testing is related to stress testing. The performance of a system is measured as how fast the system works under certain workloads. While performance testing determines if a system works as fast as a stipulated rate under certain conditions, stress testing determines the performance of a system under extreme workload conditions.

Configuration Testing

Many software systems are now built to operate under multiple operating systems and different hardware configurations. Web applications are a special case as there are many different web browsers and a given web browser would operate differently on different operating systems. Configuration testing therefore determines how a system performs under different software and hardware configurations.

Compatibility Testing

Many programs are built to replace old systems. As a result, these programs usually have compatibility requirement with the old system. A good example would be the replacement of an application that utilizes a database management system as it back end data store. It is clear that the new application must be compatible with the old database. Hence compatibility test aims to show that the system compatibility objectives were not met.

Installability Testing

This is particularly necessary in systems with automated installation processes. If any of the automated installations malfunctions then the whole installation process could go wrong. This ends

up producing a malfunctioning system since all features were not installed. This fault might not be clear to the end user as he only sees the end result of it all (possible system instability or missing features).

Reliability Testing

Software reliability testing is very important for systems that must maintain a given uptime. A good example of this would be a database system or a web hosting system, many of which have a targeted uptime of 99.97 % over the system life time. Other ways by which system reliability could be measured would be the mean time between failures (MTBF) [1].

Recovery Testing

Recovery testing is executed to show that the recovery functions of a system do not work correctly. Systems such as database management systems and operating systems need the ability to recover from different forms of failures (hardware failures, human data entry errors, programming errors). These failures could be simulated or emulated when performing tests.

One of the objectives of recovery testing would be to minimize the mean time to recovery (MTTR) so that the system is back up and running as soon as possible after a failure.

Documentation Testing

The accuracy of the user documentation is also important in the system testing process. This is usually done by using the documentation for designing test cases. For example if a stress test case is identified, the actual test case would be written by using the documentation as a guideline.

Testers normally model the system under test as a finite state machine. This helps them to have a clearer view of the system behaviour which would be the major focus in design system test cases.

From the finite state machine above, it is a lot easier to identify system threads. A typical system test thread on the SATM system would be the insertion of an invalid card. This is clearly a system level action as it is a possible user action.

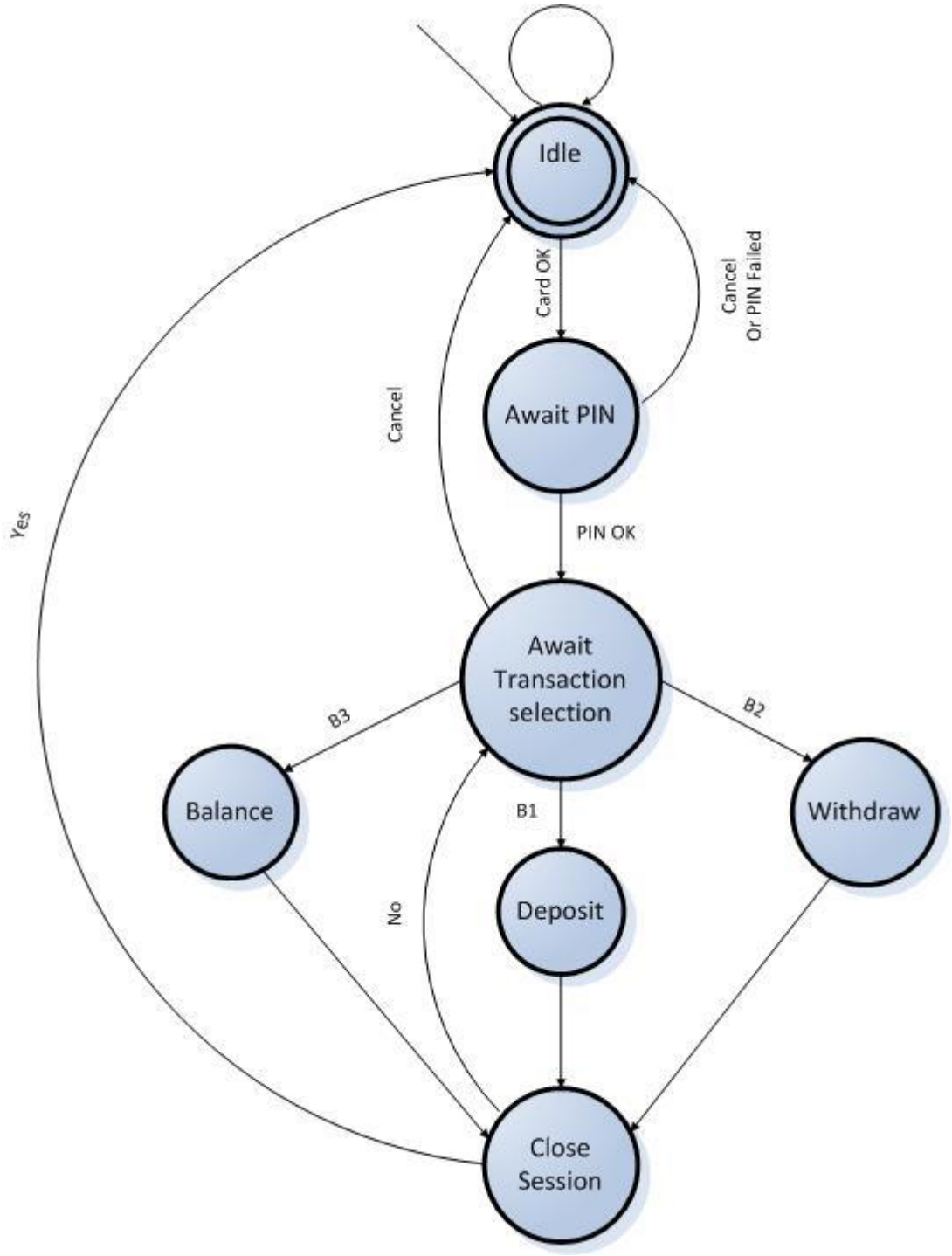


Figure 2-4: SATM finite state machine

2.4.2 System testing vs. integration testing

Many find it difficult to differentiate between system testing and integration testing. One of the more common ways of differentiating between the two is to ask the questions what is the system supposed to do? And how is the system supposed to do what it does? Another way of viewing this is that system testing attempts to show that the system implementation does not match its high level design specification and that integration testing attempts to expose flaws between the system implementation and the system's detailed design specifications.

Another difference is that system testing takes a functional view of testing rather than a structural view which is taken by integration testing.

2.5 ACCEPTANCE TESTING

Acceptance testing is concerned with showing that the end product does not meet the user requirement i.e. that the user is not satisfied with the system. Since acceptance testing is based solely on user requirements specs, it is usually the first to receive full planning.

The acceptance test plan is ideally written by the end-user. This is not possible in all projects due to the extremely large user-base for some application. In these cases a group of testers would be brought forward to represent the end-users and write the test plan.

Acceptance testing is very user centred and is normally performed by the users in an environment that is just like the deployment environment. It could be carried out by either benchmark testing or pilot testing.

The major objective of benchmark testing is to measure the system's performance in the end-user environment. This is usually simulated with the required configuration settings.

Pilot testing on the other hand is concerned with installing a system on a user site (or a user simulated environment) for testing against continuous and regular use. Two forms of pilot testing are Alpha Testing and Beta Testing.

Alpha testing is an internal acceptance testing carried out by the test team. This is usually done in preparation for beta testing. In beta testing, the system is released to a limited number of people to carry out further tests. Because the system is now in the hands of the public there is no formal methodology for testing. The most common method of testing is continuous use of the system to find out its weakness. These weaknesses are sent back to the developers as bug reports and these bugs are fixed in the next build of the system.

Sometimes acceptance testing also involves compatibility testing. This occurs when a system is being developed to replace an old one. Compatibility testing is used to determine if the new system can co-exist with the old system without loss of data

2.6 REGRESSION TESTING

Unlike the previous levels of testing discussed, regression testing spans through the testing phase. Regression testing is carried out whenever the system is modified either by adding new components during testing or by fixing errors. Its aim is to determine if modification to the system has introduced new errors in the system. Regression testing is a very important aspect of the system maintenance and Li et al. reported that the quality of a system is directly related to good regression testing [8].

2.6.1 Regression testing strategies

Regression testing could occur as a result of two general types of modification – adaptive maintenance where the system specification is modified and the other is corrective maintenance where the specification is not modified [8,13].

Corrective regression supports the reuse of test cases (since specification has not changed). However, adaptive regression requires the generation of new test cases to suit the new specification. In the case of corrective maintenance, an important question still arises – should all the previous tests cases be run or should only some of them be run?

Its is clear that running all the previous test cases (also known as the *retest-all* strategy) would be a time consuming and wasteful process especially when the change to the system is minor [8]. The more practical thing to do is to use the *selective* strategy (where some of the existing test cases are chosen to be run). This reduces the cost of regression testing but we are faced with another slight problem i.e. how to identify the test cases to run.

Chen et al. compared selective regression to the common selective compilation technique using the *make* utility [14] although determining the dependencies between the test cases run and the source code would be extremely complicated. Onoma et al. [15] identifies the typical steps involved in regression testing as:

- Modification of affected software artefacts after software has been modified- software artefacts include requirements documentation and user documentation.
- Appropriate test cases are selected to be used for regression testing on the modified program. This is an important process as the test cases must cover all aspects of modification in the program.
- Execution of the test cases selected from the previous process and examination of the test results to identify failure or faults in the modified system.
- Identification and mitigation of faults in the system.
- Updating the test case history to identify test cases that have become invalid for the modified program.

2.6.2 Selective regression testing techniques across different levels of testing

Regression testing needs to be performed at all levels of maintenance. This is evident in a study performed by Hetzel which showed that the probability of introducing an error during program modification is between 50 and 80% [16,17]

All three different levels of test require a slightly different regression testing approach. Li et al. cited that McCarthy [18] showed how to automate regression testing at the unit level using the *make* utility. In this method the make file for the test program contains information about the dependencies of the test cases and the unit under test. Input is sent to the test program from a data file and output is written to a file. The first time the test program passes, a copy of the test results is made for later reference. When future changes are made, test cases are run on the modified units and their results are saved. These results are then compared to the results from the original test runs. From this the changes can be seen and the test results are updated for later reference.

Regression testing at the integration level often makes use of a firewall concept [19]. A firewall is a construct used to separate modules that have change from those that have not changed during regression testing. Change could be direct change or indirect. Direct in the sense that the unit itself has changed and indirect in the sense that the unit is a direct descendant or ancestor of the changed unit or any unit composed of the changed unit or any unit that is contained in the changed unit.

Regression testing at the system level is similar to regression testing at the unit level. The only difference being that it is performed on a larger scale at the system level.

In general, in order to be able to perform selective regression testing at any level, one must keep extensive records of test cases and a good record of which part of the system each case covers. That way, a test case can be selected for regression test if a part of the system associated with that test case is modified.

2.6.3 Automated regression test generation

The usual problems with the selective and the retest-all regression testing strategy are that one is very expensive (retest-all) and none of the strategies guarantee revelation of software faults [16].

Korel and Al-Yami [16] presented a new system that aims at generating regression test cases, such that each case uncovers at least one error. The idea is to find test cases such that the modified program and the original program give different outputs for the same input. For this to be an inexpensive solution, the test data for the test cases are also automatically generated using existing automated test data generating methods. Experiments on their system showed that this approach increases the chances of finding software errors when compared to existing approaches.

3. SUMMARY

The cost of maintaining software is much more than the cost of producing the software [8]. In order to reduce this maintenance cost, more time and resources is being invested in extensively testing the software before deployment.

We have seen different levels of testing. Unit testing is done at the lowest level of software components to ensure the implementation fit the functional specification. Integration testing is done to ensure that the tested units interface correctly. System testing is done to determine if the functionality of the system as a whole is as was specified in the user requirements specification. Acceptance testing verifies that the end user(s) is (or is not) satisfied with the system. We have also seen regression testing which occurs throughout the software maintenance phase. Regression testing is performed to ensure that the modifications applied to a system have not adversely changed the system behaviour (i.e. that the system still conforms to the requirements specification after modification.

Finally, It is important to test at different levels because sources of errors in the software can then be easily tracked and fixed. It also provides a working subsystem which can be compiled into libraries for later use in other projects or for sale as fully tested compiled library.

4. Bibliography

[1] Glenford J. Myers. *The Art of Software Testing,* Second Edition 2004, *John Wiley and Sons, Inc*. Chapters 5 & 6.

[2] Rodney Parkin. Software Unit Testing, IV & V Australia, *The independent software testing specialists, 1997*

[3] Rick D. Craig and Stefan P. Jaskiel. Systematic Software Testing, Artech House Publishers, 2004. Chapter 4.

[4] James H. Andrews, Susmita Haldar, Yong Lei and Felix Chun Hang Li. Tool Support for Randomized Unit Testing. *Proceedings of the First International Workshop on Random Testing, July 2006 (RT'06).* pp. 36-45.

[5] http://jub.sourceforge.net. Last accessed 10/01/07 17:56

[6] David Stotts, Mark Lindsey and Angus Antley. An Informal Formal Method for Systematic JUnit Test Case Generation, *Technical Report TR02-012 April 2002.* pp 2-12.

[7] Guttag J. V and Horning J. J. The Algebraic Specification of Abstract Data Types. *Acta Informatica 10 (1978).* pp. 27-52

[8] Yuejian Li and Nancy J. Wahl. An Overview of Regression Testing. *Software Engineering Notes Vol 24, no. 1, January 1999.* pp 69-73.

[9] Paul C. Jorgensen. Software Testing: A Craftsman's Approach, Second Edition. *CRC Press, June 2002.* Chapters 1-4 & 12.

[10] Ron Patton. Software Testing, Second Edition. *Sams Publishing, July 2005.* Chapters 2 & 3.

[11] Jean Hartmann, Claudio Imoberdoff and Michael Meisinger. UML-Based Integration Testing, *International Symposium on Software Testing and Analysis, ACM Press (2000).* pp. 60-70

[12] Yuan, H. and Xie, T. 2006. Substra: a framework for automatic generation of integration tests. In *Proceedings of the 2006 international Workshop on Automation of Software Test* (Shanghai, China, May 23 - 23, 2006). AST '06. ACM Press, New York, NY, pp. 64-70.

[13] Leung, H.K.N and L. White. A Study of Integration Testing and Software Regression at the Integration Level. *Proc. Conf. Software Maintenance,* San Diego, Nov. 1990, pp. 290-301.

[14] Chen, Y. F, Rosenblum D. S.O and K. P Vo. TestTube: A System for Selective Regression Testing. *Proc. 16th International Conference Software Engineering,* Sorrento, Italy, May 1994, pp. 211-220.

[15] Onoma, A. K., Tsai, W., Poonawala, M., and Suganuma, H. 1998. Regression testing in an industrial environment. *Commun. ACM* 41, 5 (May. 1998), pp. 81-86.

[16] Korel, B. and Al-Yami, A. M. 1998. Automated regression test generation. In *Proceedings of the 1998 ACM SIGSOFT international Symposium on Software Testing and Analysis* (Clearwater Beach, Florida, United States, March 02 - 04, 1998). W. Tracz, Ed. ISSTA '98. ACM Press, New York, NY, pp. 143-152.

[17] Hetzel W., The Complete Guide to Software Testing, QED Information Sciences, Wellesley, Mass.,1984

[18] McCArthy A. Unit and Regression Testing. *Dr. Dobb's Journal,* February 1997, pp. 18-20, 82 & 84.

[19] White L. and H.K.N Leung. Regression Testability. *IEEE Micro,* April 1992, pp. 81-84.

[20] http://sourceforge.net/projects/xunit/ Last accessed 10/01/07 18:01

5. APPENDIX A

Below is a decomposition tree for the Simple Automated Teller Machine System.

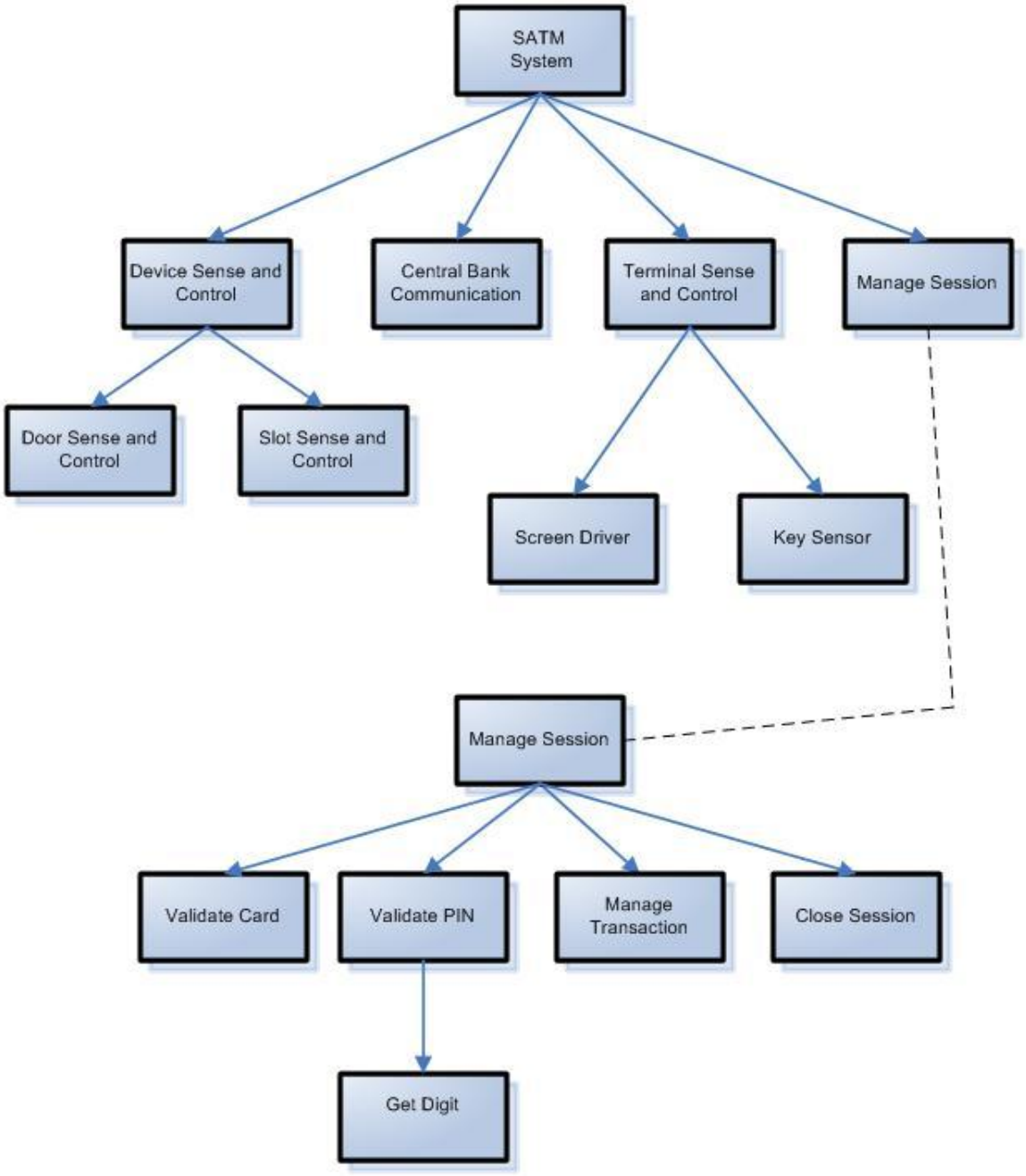


Figure 5-1: A decomposition tree for the SATM system

Below is a more complete outline structured functional decomposition of the SATM System. The numbering scheme represents the levels of the components. This is taken from Paul Jorgensen's "*Software Testing: A Craftsman's Approach, Second Edition*".

1 SATM System
- 1.1 Device Sense and Control
  - 1.1.1 Door Sense and Control
    - 1.1.1.1 Get Door Status
    - 1.1.1.2 Control Door
    - 1.1.1.3 Dispense Cash
  - 1.1.2 Slot Sense and Control
    - 1.1.2.1 Watch Card Slot
    - 1.1.2.2 Get Deposit Slot Status
    - 1.1.2.3 Control Card Roller
    - 1.1.2.4 Control Envelope Roller
    - 1.1.2.5 Read Card Strip
- 1.2 Central Bank Communication
  - 1.2.1 Get Pin for PAN
  - 1.2.2 Get Account Status
  - 1.2.3 Post Daily Transactions
- 1.3 Terminal Sense and Control
  - 1.3.1 Screen Driver
  - 1.3.2 Key Sensor
- 1.4 Manage Session
  - 1.4.1 Validate Card
  - 1.4.2 Validate PIN
    - 1.4.2.1 GetPIN
  - 1.4.3 Close Session
    - 1.4.3.1 New Transaction Request
    - 1.4.3.2 Print Receipt
    - 1.4.3.3 Post Transaction Local
  - 1.4.4 Manage Transaction
    - 1.4.4.1 Get Transaction Type
    - 1.4.4.2 Get Account type
    - 1.4.4.3 Report Balance
    - 1.4.4.4 Process Deposit
    - 1.4.4.5 Process Withdrawal