Software Testing Techniques Essay, Research Paper

The George Washington UniversitySchool of Engineering & Applied Science Software Testing Techniques By: Haneen Al-AbdulrazzaqCourse : CS270 Software Engineering & Development. Outline:I. Introduction.II. Types of Testing Strategies in Conventional Programming Languages.A) White Box Testing. 1) Statement Coverage. 2) Edge Coverage. 3) Condition Coverage. 4) Path Coverage. B) Black Box Testing. 1) Black Box testing Techniques.a) Boundary Value Analysis.b) Equivalence Class Partitioning.C) Gray Box Testing.D) Unit Testing.E) Integration Testing.1) Non-incremental Testing.2) Incremental Integration.a) Top-down Approach.b) Bottom-up Approach.c) Sandwich Approach.F) Validation Testing.1) Alpha Testing.2) Beta Testing.III. Testing Object-Oriented Software.A) The Effect of Object-Oriented Concepts.1) The Impact of Encapsulation on Testing.2) The Impact of Information Hiding on Testing.IV. Conclusion. Introduction Testing is one area of software engineering in which the gap between research knowledge and actual practice is very large. Many times testing is confused with debugging or software quality assurance. To clearly show the difference between the three concepts mentioned above, a formal definition of each concept will be stated: Testing: Testing is the process of examining something with the intention of finding errors. While testing may reveal a symptom of an error, it may not uncover the exact cause of the error. Debugging: Debugging is the process of locating the exact cause of an error and removing that cause. Software Quality Assurance: Software QA assures the effectiveness of a software quality program within a software engineering organization. Testing takes up as much as 40% of the software engineering effort. This is why the importance of testing a software product is very highly weighted. Testing cannot show the absence of defects it can only show that defects are present. In order to cut down on effort and time spent on testing, the developer(s) must integrate the testing process with the development process. In the last decade of this century Object-Oriented programming has become one of the mainstream implementation technologies. When making the transition to a new technology, we expect that some of what we currently know about software testing still holds. One point that must be stressed is that the testing of an Object-Oriented system is different. It is different because of the nature of Object-Oriented programming. Many characteristics of Object-Oriented programming affect the old testing strategies used for conventional languages but with little alterations those strategy will still be as effective when testing an Object-Oriented software product. In this paper we will look at the different testing methods and concepts used in conventional programming languages. Moreover, we will look at some Object-Oriented testing methodologies. Types of Testing Strategies in Conventional Programming Languages: 1. White Box Testing:White-box testing is the testing of the underlying implementation of a piece of software (e.g. source code) without regard to the specification (external description) for that piece of software. The nature of typical errors makes white-box testing very important. The goal of white-box testing is to identify such items as (unintentional) infinite loops, paths through the code which should be allowed but which cannot be executed, and dead (unreachable code). One essential concept in white-box testing is coverage. The tester must achieve full coverage of the software code. In order to understand clearly the concept of coverage, we will discuss the different types and meaning of coverage.Coverage: coverage is a measure of the number and type of statements executed, as well as how they are executed. The are four different types of coverage:1. Statement coverage: In this type, the tester aims to test all executable statements at least once. This type is the weakest type of coverage for many reasons. For one, executing a statement once and observing that it behaves properly does not mean that it is correct. Another reason is that the definition of a statement is very inconsistent. The convention here is to use the BNF definition of a statement.2. Edge coverage: Here the tester aims to execute all edges of a control flow graph that make each condition generate both true and false values. Since the focus here is on the flow of the control in the program, the tester might regard a sequence of edges starting at N(1) and ending at N(k) as one edge from N(1) to N(k): N(1) N(2) N(3) N(k-1) N(k) N(1) N(k) Using the edge coverage, all conditions the control the flow of the program will be executed. Test cases that make each condition generate both true and false values at different times are required. 3. Condition coverage: In this type of coverage, all edges of a control flow graph are traversed as well as all possible values of the constituents of compound conditions are executed at least once. In other words, effective test cases for this type of coverage must include the different values taken by each constituent of the compound Boolean expression in a condition statement. 4. Path coverage: This type of coverage looks at all paths that lead from the initial node to the final node of a control flow graph. Basically, the tester will examine all different paths of the graph and come up with test cases that cover each individual path. An example will illustrate this type of coverage in a clear manner:Example of path coverage:0: procedure Sort:1: Do while records remain2: read record;3: if record filed\_1 = 0 then4: Process record;5: store in buffer;6: increment counter;7: elseif record field\_2 = 0 then8: reset counter;9: else10: process and store record;11: end else-if;12: end if;13: end do-while;14: end sort; 2. Black Box Testing:Black-box testing is the testing of a piece of software without regard to its underlying implementation. Specifically, it dictates that test cases for a piece of software are to be generated based solely on an examination of the specification (external description) for that piece of software. The goal of black-box testing is to demonstrate that the software being tested does not adhere to external specifications. There are quite a number of black-box testing techniques. Two of the most productive black box techniques are “Boundary Value Analysis” and “Equivalence Class Partitioning”.Black Box Testing Techniques:1. Boundary Value Analysis: This technique requires that test cases be generated which are on, and immediately around, the boundaries of the input and output for a given piece of software. The tester must focus on the boundaries. For example, the tester will look at the range of the input a to b, and then provide test cases with a, b, a-1, b+1 (if it is an integer range, otherwise, slightly less than a and slightly more than b). Moreover, the tester can have a set of input values and then test with the minimum of the set, the maximum of the set, the min of the set – 1, and the max of the set + 1. Furthermore, for output values, the tester might try to push the boundaries of the output values. As for internal data structures with limits, the tester can test those structures at their limits (e.g. an array with maximum size n).2. Equivalence Class Partitioning: This technique requires that for each input, an equivalence class is developed, representing the set of valid and invalid input conditions. An equivalence class is a collection of items which can all be regarded as identical at a given level of abstraction (e.g. a set of data items which will all evoke the same general behavior from a given software module).There are other techniques that could be used depending on the nature of the software product. In situations where there are many different combinations of inputs possible, a black-box technique called “cause-effect graphing” might be used. This technique helps software engineers identify those specific combinations of inputs that will be the most error-prone. 3. Gray Box Testing: Gray-box testing is testing based on an examination of both the specification for a piece of software, and the underlying implementation (e.g. source code) for that piece of software. Any good testing effort is a carefully planned combination of black-box, white-box, and gray-box testing techniques. Low-level testing (testing small amounts of software like a single function) usually involves a significant amount of white-box testing. Higher-level testing (testing larger amounts of software like system testing) is almost exclusively black-box testing. Unit Testing: In conventional programming languages the module is considered a unit. When performing unit testing the focus is obviously on the module. This kind of testing can be classified as white-box testing and is usually done by the developer. One major area to test here is the interface of the module. For example, the tester must check to see that the parameters of the module are compatible with the arguments; number; type; units; compatible attributes; and last but not least the order of the parameters. Moreover, the tester must perform similar parameter checks on internal calls to other routines. The tester will also want to check the I/O interface. This means that the tester will be looking at such things as file attributes, formats, buffer sizes, EOF handling, and I/O error handling. Another important factor in unit testing is the testing of data structures. Here, the tester will be checking typing problems, initialization and default values, incorrect variable names, and the underflow/overflow or other exceptions. Additional checks performed include basis path testing, computational errors, and error handling. Integration Testing: In conventional languages the approach is that once a unit (usually a subprogram) is tested in isolation, it is then integrated into the larger system. This can be done in two ways: A non-incremental testing (a.k.a. Big Bang Testing) and an incremental integration. 1. Non-incremental Testing (”Big Bang Testing”): This approach means that the tester tests each unit in isolation, simultaneously integrate all units, and then attempt to test the resulting whole. This approach is not advisable, unless the system being tested is a very small non-critical system.2. Incremental Integration: This is the alternative approach to non-incremental testing and is the more useful approach of the two. The idea is that the tester tests each unit in isolation. Then the tester integrates each unit, one at a time into the system, testing the overall result as he/she goes. Types of approaches to incremental testing include:+ Top-Down+ Bottom-Up+ Sandwich (combination of the former two approaches) The Top-Down Approach:This approach means that the tester starts from the main module and tests the rest of the modules in a top-down manner. There are two methods to perform such testing, depth-first is one and breadth-first is the other (a tester can also combine both). One of the difficulties to overcome when using the top-down approach is that not all modules of the system have been completed. In order to solve this problem the idea of using stubs was introduced. When a tester is testing a module and during its execution, an external procedure call was made, however, that procedure is not yet developed, the tester will need to build a stub that simulates the procedure’s behavior. A stub is, in other words, a procedure with the same I/O parameters as the missing procedure but with simplified behavior.

E.g. Top-Down Integration – Depth First The Bottom-Up Approach: This approach is the opposite of the former approach. In this approach the tester will be working with the low-level modules and working his/her way up. The low-level modules are combined into clusters that perform a subfunction. A driver is defined for interaction with the cluster. Then the cluster is tested and finally, the drivers are removed and the clusters are combined. A driver simulates the work of the higher-level modules, as opposed to the lower-level modules being simulated by stubs. An illustration will help clarify the concept: E.g. Bottom-Up Integration Validation Testing: Validation testing is the process of checking that what has been specified is what the user actually wanted, whereas verification testing is the testing of items, including software, for conformance and consistency with an associated specification. For validation testing the tester should be able to answer the question: “Are we doing the right job?” For verification testing the tester should be able to answer the question: “Are we doing the job right?” Validation testing is the testing of the full software. The tester must confirm that the software functions in a manner that can be reasonably expected by the customer. The tester must always make the comparison of the full product to the requirements specification. At the end of the validation testing the result would either be acceptance, or deviation will be uncovered. In the case of acceptance it means that the product meets the customer’s requirements and is ready to be delivered. However, if deviation was found, then the company must start negotiation with the customer as to what to do at this point in time. Once the software product has undergone system testing, two approaches for validation can be taken:1. Alpha Testing: Here the software is put to actual use in the company producing the application. The reason for this is to test the product in a realistic environment.2. Beta Testing: The software product is delivered to a selected group of customers for evaluation purposes. By getting the users feedback, the developing company can then perform any changes necessary before releasing the official version of the software in the market. Testing Object-Oriented Software: The discussion of software testing above was related to conventional languages rather than object-oriented ones. The question that raises itself is: “Is the testing of object-oriented software different, and if so, how is it different?” This question will be answered by looking at the nature of object-oriented languages and how the testing methodologies mentioned above apply to it. First of all, when testing object-oriented software the tester must take into account the unique characteristics of this type of programming. Furthermore, the tester must attempt to start with the testing as early as possible which means that the testing should start during the object-oriented analysis and design phases. Next, the tester must provide test cases that handle the unique characteristics of object-oriented software. Finally, the strategies used for unit and integration testing of conventional languages must be changed. The Effect of Object-Oriented Concepts: Some of the object-oriented concepts are very powerful especially in constructing the structure of object-oriented software. Two concepts that have a major impact on testing strategies are information hiding and encapsulation. Information hiding requires that we suppress (or hide) some information regarding an item. The general idea is that we show only that information which is necessary to accomplish our immediate goals. If we were to show more information we increase the chances of errors, either at the present time, or when the software is later modified. There are degrees of information hiding, e.g., C++’s public, private and protected members. Encapsulation, on the other hand, describes the packaging (or binding together) of a collection of items. Common low level examples of encapsulation include records and arrays. Procedures, functions, and subroutines are another way of encapsulating information. Object-oriented approaches require higher levels of encapsulation, e.g., classes. Among other things classes can encapsulate operations, other objects, and exceptions. Depending on the programming language and the decisions of the software engineer, items, which are encapsulated in a class, will have varying degrees of visibility (information hiding). The Impact of Encapsulation on Testing: Object-oriented approaches use different encapsulation strategies than do the more conventional approaches. First of all, the basic testable unit will no longer be the subprogram. Second of all, the strategies for integration testing will have to be modified. The subprogram is the basic building block from which applications are created. In a classic waterfall approach to software development, subprogram units are usually well defined by the end of the design phase. Even before there was any code written, a good subprogram unit had a well-defined interface, and performed a single specific function. Once an individual subprogram unit was thoroughly tested, it was rarely, if ever tested as a unit again. If a subprogram unit was reused (either in the same application or in another application), however, it’s appropriateness had to be re-determined in each context. In an object-oriented environment, we are dealing with larger program units (classes), so the concept of a subprogram is not quite the same as it was in the traditional approaches. Specifically, in OO programming, the specification of the subprogram (its interface) is separated from its implementation (its body). We refer to the specification as an “operation”, and to the implementation as a “method”. To further complicate matters, one operation can be supported by several methods. In order to keep things simple for the sake of comparison, let us assume that an operation and one of its methods are the equivalent of a subprogram in a more traditional environment. In this case, a class can encapsulate many subprograms. In OO systems, the subprogram can be thought of as being bound (encapsulated) within a larger entity (a class). Moreover, these subprograms will work in connection with the other items encapsulated within the same object. Consequently, in an object-oriented environment, attempting to test a subprogram in isolation is pointless. In conclusion, the smallest testable unit is no longer the subprogram but the classes and instances of classes in which it is encapsulated. Another object-oriented concept that plays a large role in the testing of a subprogram is inheritance. For example, suppose that a subprogram has been thoroughly tested within the context of a given class. Next, suppose that a subclass was created based on that class and it inherited the tested subprogram from the superclass. Even though the subprogram has been tested within the context of the superclass, one cannot guarantee it will work properly within the context of the subclass, unless it is re-tested within the context of the subclass. This principle is called anti-extensionality. Anti-extensionality states that, black-box test cases that were created for the subprogram within the superclass, are probably not entirely appropriate for the same subprogram within context of a subclass derived from the superclass. Integration testing in an object-oriented approach is not equivalent to that done in a non-object-oriented approach. In the latter, our smallest testable unit is the subprogram, and during integration testing (depending on the strategy used) we would be integrating one (or a few) subprograms at a time. Integrating subprograms into a class, one at a time, testing the whole as we go may not be an option. There are usually direct or indirect interactions among the components that make up the class. For example, one operation may require that the object is in a specific state – a state that can only be set by another encapsulated operation, or combination of encapsulated operations. The Impact of Information Hiding on Testing: Object-oriented programmers prefer the black box nature of objects. Specifically, a user of an object is denied access to the underlying implementation. This creates problems during testing. Since we cannot directly inspect the underlying implementation we must seek some other strategy. For example, let us consider a simple list object. In its interface there are a number of operations, e.g., “add”, “delete”, and “length”. Suppose an item was added to the list using the “add” operation. How can we know that the specific item was actually added? A general approach is to first establish an acceptable level of reliance on those operations that do not change the state of the object, but rather return information on the object’s state. For example, if we could trust the “length” operation in our list object, we would expect to see the length increase by one whenever an object is added. To say the very least, test designers would have to plan class-testing strategies very carefully in order to ensure proper coverage. Conclusion: Software can be tested at various stages of the development and with various degrees of strictness. Like any development activity, testing consumes effort and effort costs money. Developers should plan for between 30% and 70% of a project effort to be consumed on verification and validation activities, including software testing. Efficiency and quality are best served by testing the software as early in the life cycle as possible, with full regression testing whenever changes are made. The later a bug is found the higher the cost of fixing it, so it is far more beneficial to identify bugs as early as possible. Designing tests will help to identify bugs, even before the tests are executed, so designing tests as early as possible in a software development process is a useful way of reducing the cost of identifying and correcting bugs. One should remember that the software should be tested against what it is specified to do, not against what it actually observed to do. The effectiveness of testing effort can be maximized by selection of an appropriate testing strategy, good management of the testing process, and appropriate use of testing tools to support the testing process. The net result will be an increase in quality and a decrease in costs, both of which can only be beneficial to a software developers business. Last but not least, in the issue of object-oriented software testing, much of what is known about testing technology does apply. However, object-orientation brings with it its own specialized set of concerns. Therefore, while the old testing strategies still apply, they must be altered in order to work in an object-oriented environment. Bibliography [Balfour, 1988]. B. Balfour, “On ‘Unit Testing’ and other Uses of the Term ‘Unit’,” MCC ‘88 Military Computing Conference, Military Computing Institute, 1988, pp. 127 – 130. [Beizer, 1990]. B. Beizer, Software Testing Techniques, second edition, Van Nostrand Reinhold, New York, New York, 1990. [Binder, 1996]. Robert V. Binder, “The FREE Approach to Testing Object-Oriented Software: An Overview,” RBSC Corporation, 1996. [Binder, 1995]. Robert V. Binder, “Object-Oriented Testing: Myth and Reality,” Object Magazine, 1995. [Fiedler, 1989]. S.P. Fiedler, “Object-Oriented Unit Testing,” HP Journal, Vol. 36, No. 4, April 1989. [Howden, 1987]. W. Howden, Software Engineering and Technology: Functional Program testing, McGraw-Hill, New York, New York, 1987. [IPL, 1996]. Information Processing Ltd., “An Introduction to Software Testing,” IPL, Eveleigh House, Bath, UK, 1996. [Meiler Page-Jones, 1995]. Meiler Page-Jones, “What Every Programmer Should Know About Object-Oriented Design,” Dorset House Publishing, New York, New York, 1995. [Myers, 1979]. G.J. Myers, The Art of Software testing, John Wiley and Sons, New York, New York, 1979. [Perry and Kaiser, 1990]. D.E. Perry and G.E. Kaiser, “Adequate Testing and Object-Oriented Programming,” Journal of Object-Oriented Programming, Vol. 2, No. 5, January/February 1990, pp. 13-19.