Software Testing

Test Design and the Project Life Cycle

Prepared for Dr. Stephen A. Bernhardt
at the University of Delaware
on December 18, 2002.

By Derek Callaway
English 410, Section 11
Technical Writing

Abstract

The software development process consists of an indeterminate number of fundamental steps that together comprise the project life cycle. All of these steps carry out software testing in one form or another. Some organizations have an entire team delegated exclusively to software testing. (Royer 16-17,20) As a result, a substantial amount of a software development project's budget is allocated solely toward testing. This establishes the need to utilize formal techniques in order to trim cost. (Amman and Black, *Coverage* 20) Such techniques are the subject of an ample amount of scholarly investigation and are generally classified into two complementary integration approaches (top-down and bottom-up) and fall into one of a pair of distinct methods (black-box and white-box). In this report, the distinguishing characteristics and merits of each are presented, as well as their relative disadvantages and ways to mitigate their limitations.

Table of Contents

Introduction

Inevitably, human imperfection will be a factor in any computational endeavor. As the old saying goes, a machine is only as perfect as its inventor. A typical piece of software that has utility will include many people in its life cycle. (Royer 3) In order to rectify the mistakes of any individual involved with a particular software project, a certain amount of testing must be applied. There are many things to take into consideration when determining the appropriate way to go about said testing. (Demillo et al. 27-28; Amman and Black, *Abstracting* 5)

Erroneous Computing

Computers users are people that basically instruct a machine to perform switch toggles. These switch toggle sequences tend to become very large rather quickly. With such a lengthy list of ones and zeroes, the electronic device is destined to go astray in some manner of indirection from a human mistake. Computational mishaps can be described by the terminology that follows.

- error – That which is caused by a human; this could be anyone related to project including but not limited to programmers, designers, and end-users. (Royer 3)
- fault – An error that causes software to behave unexpectedly. (Royer 5)
- failure – Faults that prevent the software from executing. (Royer 5)

Encountering erroneous computing in software is essentially inescapable for all those involved. Errors in computer science are the result of a person that is ignorant. This ignorance is what a software development process strives to regulate through testing.

Test Design Methods

Test design falls into two clearly outlined methods called black-box and white-box testing. They have many other self-evident names. Black-box test design is sometimes called behavioral, functional, opaque-box, or closed-box. White-box testing is also called structural, glass-box, or clear-box. (Rivest) Deciding which test design method to utilize usually depends upon the current step in the project life cycle.

Test methods can be done by a human or a computer. Manual testing is performed by a human. Automated testing is carried out by a computer. Albeit

faster and quite effective in many contexts, automation is not always fruitful. Goldfine reports on a research initiative by the National Institute of Standards and Technology (NIST) in which two programmers spent over 36 hours each creating automated tests using Sun Microsystems' Assertion Definition Language (ADL). The automata were used to test conformance of four system functions to Portable Open Systems Interconnect (POSIX) standards. The final test results were in a large part inconclusive. (1, 2, 5, 29) One should be aware of the implications of manual and automated testing and how they relate to the two primary test design methods.

Black-Box

Black-box is a test design method which verifies that the functionality of a program is proper. It can be defined as determining which inputs produce faulty outputs. To do this, input sets, or test cases must be generated from software specifications. (Demillo et al. 20, comp.software) Luckily, there are existing algorithms that are extremely useful for automating test case generation. In particular, there are algorithms for computation of permutations and combinations. (Goodaire and Parmenter, 213-218) Specifications can be acquired from an authoritative manual, design documents, development files, or even word-of-mouth for lack of something more official. The documentation of other component software such as operating systems, shared interfaces, and specialized execution environments may also come into play. A human software tester could arduously carry out the analysis of specifications by looking over them carefully or a computer could "grok" the specs automatically.

As should be expected, there are a few very tough problems existent in black-box testing design. If test automation is desired, a plan of attack must be formulated to tackle these problems. First of all, one must realize that initial and successive input may contribute to putting the program into an extremely large number of possible states. In some cases, the number of potential states may be uncountable. Program output is derived from these internal execution states and is the predicate of a faulty program. (Allman and Black, *Coverage* 5) Furthermore, combinatorial algorithms likely to be used for generating test cases can be NP-complete. This means that they are calculated in nondeterministic polynomial time. In other words, there currently exists no efficient solution to the computation that the algorithm addresses. (Goodaire and Parmenter 254) As a result, it is often necessary to systematically reduce the spaces that are representative of conceivable input and output; anything that operates along the lines of this conjecture is called a reduction method. The

following subsections present some effective ways to deal with these problems.

Irrelevant Variable Removal

Irrelevant variable removal depends on the notion that if some property holds for a specification, variables that don't relate to this property are not crucial to testing. This idea can be extended into partially relevant variable removal by eliminating variables from a problem that relate to one or few properties. (Allman and Black, *Coverage* 5) As a rather simple example, envision a program with a new version in which the functionality of a particular input option from a previous release was removed but the optional input was retained for backward compatibility. This input option can be removed from the test set.

Monitored Variable Abstraction

Allman and Black nicely summarized this technique as, "if only certain values or ranges of a monitored variable influence the values of other variables, the monitored variable may be replaced with an abstract variable." (*Abstracting* 5) Take for instance an application that expects as input an integer value from a specific range. Input possibilities can be abstracted into three quantities that are less than the range, in the range, and greater than the range. So, if the range were from -10 to 10, the lesser variable could take the value -15, the in range variable might be set to 5, and the greater variable could be defined as 15.

Bounded Timers

The passage of time inside a computer may not coincide with time in the real-world. A process may be shown incorrect if operations are not performed in a suitable length of time. Timers with lower and upper bounds on estimated time intervals between events can be used to catch a fault. (Allman and Black, *Abstracting* 6) If a software specification correctly says that a program will perform an action every ten seconds then it must, else it has been implemented incorrectly.

Semi-Automated Abstraction

Semi-automated abstraction, another approach to space reduction is based upon transitions between program states. Initially, a complete list of transitions between abstract states is compiled. If the existence of the transition between two abstract states is disproved, that transition is removed from the list of cases to test. . (Allman and Black, *Abstracting* 6) Take for example a calculator program that has five states: a beginning state, an input state, a calculation state, an output state, and an end state. The beginning state displays some program information then gives control to the main program loop is made up of the input, calculation, and output states. The input state receives the input and computes it by moving on into the calculation state. The result is displayed in the output state and the program returns to the input state. The end state is reached when a special command is given as input. According to this specification, this program may not proceed directly from the beginning state to the end state; there are middle states to be modeled. It also shows that the program cannot changeover from the input state to the output state. The beginning to end and input to output transitions could be deleted from the list of all transitions for the calculator program's testing. Falsifying transitions is a fairly complicated procedure. Nonetheless, semi-automated abstraction can turn out to be very useful, especially when used in conjunction with other techniques.

Temporal Strength Reduction

In order to test a computer program with many states, previous execution states must be taken into account as they relate to the current state. Instead of saving previous states, traits describing the current state should be saved before moving onto the next in the interest of storage space preservation. . (Allman and Black, *Abstracting* 6) For example, if specifications require that at each successive state a integer variable needs to be greater than a certain number, then compute the truth value of this specified predicate instead of storing the number and analyzing it later. This technique saves storage space and facilitates early fault detection. Another good thing about this technique is that it can detect faults

as they happen as opposed to when the tested program is finished executing.

Ubiquitous Abstraction

> As with many aspects of software testing, a conservative blend of techniques is most effective. Using a variety of reduction methods in all sections of the testing process has been named ubiquitous abstraction. If the current abstractions fail to prove the existence of a fault they can be particularized further and tested again. (Allman and Black, *Abstracting* 6) Black-box testing in this manner helps minimize costs. Program faults will be discovered earlier which is a good thing because the later they are noticed the harder they are to fix. Not detecting an error in the current step of the software development process causes it to propagate into future steps.

Do not underestimate the profound effect that these approaches can have on an input space. Of course, the output space may be partitioned as well but partitioning the input space is especially useful because it reduces the number of cases to be tested. Consequently, the number of times that the program must be executed during a test phase is also reduced. Input space partitioning is a reduction method that divides program inputs into equivalence classes. (Demillo et al. 20) This division can be accomplished by the aforementioned techniques. The equivalence of the members of each partition can be approximated in order to preserve computational resources.

Good judgment must be used when deciding upon how to implement a black-box test so that the usefulness of the method is maximized. Typical software usage can be regarded as black-box testing and makes up a large part of software validation. Remember that black-box testing is limited to dynamic analysis.

White-Box

White-box test design affirms the structural integrity of a given software project. It is characterized by the inspection of program instructions or source code. (Demillo et al. 21, comp.software.testing) Naturally, white-box test methods are part of the coding and unit testing stage of the program's life cycle. (Falk, Kaner, and Nguyen 41) Code walkthroughs, a

form of white-box testing have been shown to be just as good as dynamic testing by a non-author. (Falk, Kaner, and Nguyen 46)

Having access to the program's source gives a tester a good guess as to which inputs will cause a transfer of program control to take place. Henceforth, it is easier to determine which execution paths need to be tested. (Falk, Kaner, and Nguyen 46; Royer 104) A path is regarded as a unique set of transitional states that take a program from start to finish. The precise path taken is determined by program logic. (Falk, Kaner, and Nguyen 43-44) Idealistically, a complete white-box test would cover all possible paths, but this is likely to be impractical given resource and time constraints for a sizable project. (Royer 104) Sharply defined strategies must be used for measuring the effectiveness of path testing.

Path Testing

Describing the coverage of possible execution paths in white-box testing is done in a number of different ways. The fundamental measures for path testing are statement coverage, decision coverage, and condition coverage. (Kit 91; Falk, Kaner, and Nguyen 43) Not too unsurprisingly, an actual software situation may employ a mixture of these three techniques. All coverage types stem from statement coverage since statements are the primordial units of source code. Following are brief definitions of each including a few hybrids.

- statement coverage – All statements are executed at least once. (Kit 91) It may sometimes be referred to as line coverage. (Falk, Kaner, and Nguyen 43)

- decision coverage – Statement coverage and evaluation of each decision's consequents. (Kit 91; Falk, Kaner, and Nguyen 43) This is also called branch coverage. (Kit 89)

- condition coverage – Statement coverage and evaluation of each way that a branch outcome may be decided. There may be many ways that a condition can return the same value. (Falk, Kaner, and Nguyen 44)

- decision/condition coverage – A combination of statement, decision, and condition coverage. (Kit 91)

- multiple/condition coverage – The epitome of complete path coverage; statement coverage combined with all possible consequences of conditions. Note that a decision may have multiple constituent conditions. (Kit 91)

Deciding upon a path testing technique to employ is based largely on the intricacy of the program under scrutiny. (Royer 104) Obviously, it would be silly to utilize multiple/condition coverage on an application with several thousand statements in its source code and a dozen conditions for each of many decisions.
Good judgment must be used when deciding which path testing technique should be applied; these kinds of choices are frequently the subjects of review meetings.

Path testing is a powerful tool for program verification. Unfortunately, there are problems that path testing is unable to solve. Such weaknesses of the white-box test design method will be highlighted in the next subsection.

Issues in Structural Testing

In some PC programs that were released in the early eighties, a bug existed which path testing would have neglected to uncover. Hitting the space bar during the startup of the programs would force a cold reboot disk operations were performed while hardware interrupts (more specifically, the one responsible for the space bar) were enabled. The interrupt was an unforeseen action so no interrupt handling code was written. (Falk, Kaner, and Nguyen 211) The preceding was an example of a race condition. The programmer assumed that an event wouldn't occur during the time interval for program initialization. Timing-related errors are one of the many issues that arise when putting structural testing under consideration.

Race conditions

As shown above, a race condition bug comes about when a programmer makes a presumption about the order of events that take place during execution. As a demonstration, take two events, event one and event two; event one almost always happens before event two because of the logical organization of program instructions. Race conditions take place when the coder presumes that event one will definitely take place before event two. (Falk, Kaner, and Nguyen 421) Hopefully, the author will apprehend that the occurrence of event two before event one can happen under uncommon circumstances.

Deprecation and Unsupported Code

If a project phases out a feature (so-called deprecation) or retains code that isn't intended to be supported in current or future releases, then a costly "high maintenance test case" becomes evident. This matter can be avoided by maintaining compatibility. (Kit 114) Notice that this issue can increase cost even for a non-commercial development initiative because it squanders time.

Peaceful Co-Existence

White-box testing cannot determine if a given application will conflict with other software on a computer. Program developers cannot know what software may be installed on a user's computer. (Falk, Kaner, and Nguyen 421) Two or more programs may overwrite each other's data on disk. There can be many possibilities for resource contention. Concurrently executing processes may inadvertently compete for the right to access the same processor.

Hardware Inconsistencies

Idiosyncrasies of underlying hardware can cause problems in software. (Falk, Kaner, and Nguyen 421) If the software is portable, it is highly probable that the programmers and/or testers will not have immediate access to all hardware

> configurations that the software will operate under. Hardware inconsistencies are usually exposed during validation.
>
> Notice that these issues can usually be revealed by black-box testing techniques. For example, evidence of race conditions can be stumbled upon during the volume and load/stress testing forms of black-box design. (Kit 101) Additionally, it is difficult to simulate actual end-user usage with white-box testing because most software users are ignorant of source code. Irregardless, white-box test design is a very potent method of testing software. A tester that utilizes path testing has a chance at being extremely successful at locating bugs.

In practice, it is desirable to use both methods so that testing is not limited by the shortcomings implicit in one or the other. If you know something about the inside of a program you can test it better from the outside and vice versa. The terms gray-box and translucent-box specify the intermingling of white-box and black-box test design methods. Take care not to confuse the two with static and dynamic analysis. Static analysis studies structure. Dynamic analysis surveys functional properties. (Falk, Kaner, and Nguyen 46) White-box testing is a form of static analysis but may at the same time perform dynamic analysis. Black-box testing is strictly dynamic.

The Project Life Cycle

The discrete steps inherent in the software development process are termed the project life cycle. The conventional definition of project life cycle is generalized more or less and not all programs follow it exactly. The project life cycle must be understood in order to recognize software testing as a whole. It can be used as a guide to determine the appropriate type of software testing needed. According to Royer the steps in the project life cycle are:

- ✓ Requirements Definition
- ✓ Top-level Design
- ✓ Detailed Design
- ✓ Coding and Unit Testing
- ✓ Component Integration and Testing
- ✓ Configuration Item (CI) Level Testing (16-17)

Note that the last half of the cycle is a direct test of the software's completeness. Although it may not involve physically testing a piece of software, each step is essential to the software testing process. The steps are detailed below.

Requirements Definition

The requirements definition step of the project life cycle is simply a statement of the problem. Once the groundwork for the problem has been laid out, the project team may consider a possible solution. (Royer 17) After an agreeable solution is devised, top-level design can begin.

Top-level Design

During the top-level design step, necessary interfaces external and internal to the program are defined. These interfaces take on a number of different forms; for instance, interfaces between individual software components or interfaces between the software and a user. This step may be accompanied by the composition of documents which specify the required interfaces and other architectural necessities. A document of this type is sometimes called a Software Design Document (SDD) or Top-Level Design Document (TLDD). These documents may include schematics such as data flow diagrams. (Royer 18)

Data Flow Diagrams

A data flow diagram allows a designer to visualize data transfer or message-passing in a computer system through symbolic representation. An example of a data flow representation is UML, the Unified Modeling Language (Unified). Data flow diagrams are to programmers as blueprints are to drafting artists.

It is important to note that the software still needs to be tested even though it has manifested itself only in the form of design documents. After a preliminary design is completed, it must be tested against the true intentions of the project. A preliminary design review (PDR) involving technical and marketing staff may be held with a customer concerning the top-level design. (Royer 18) This design review can be thought of as a form of white-box static analysis because of access to structural definitions (design documents) and the absence of program execution.

Detailed Design

Also known as low-level design, detailed design more formally specifies the interfaces established by the top-level design step. These elaborated specifications are documented in an SDD or Low-Level Design Document (LLDD). The LLDD provides a technical run-down of system units. The relevant information for each individual unit is also kept in places such as a Software Development File (SDF), Unit Development Folder (UDF) or a programmer's notebook. SDF's list information a software engineer needs to be aware of in order to develop a given portion of a project. Detailed design should be tested by an internal review with participating personnel. A customer could be invited to a separate Critical Design Review (CDR). (Royer 18-19) Some sort of review is necessary for proper verification of this step.

Coding and Unit Testing

Coding takes place when a programmer converts SDF information into a computer language. This entails lost of white-box testing in the forms of "code walkthrough" and "peer review" after all compile-time errors have been resolved. Code walkthrough is the personal proofreading of code by the author and peer review is proofreading by others. This proofreading will reduce potential run-time errors before unit testing. Unit testing executes the unit's code with respect to specific test data and test cases. After unit testing is completed the particular program unit's code is kept under some sort of configuration control so that the developers of other components are mindful of the status of the unit in question. (Royer 19)

Component Integration and Testing

Component Integration and Testing checks the correctness of interfaces between separate units or components. If they fail to verify, the unit probably needs to be modified and unit tested again. (Royer 20) Observe the variety of ways in which a component may be integrated:

- Top-down
  - Breadth-first
  - Depth-first

- Bottom-up
- Mixed
  - Sandwich (Muccini)

Top-down

Components can be categorized as high-level, middle-level, low-level, etc. based on their location in a sequence of procedure calls. Top-down integration verifies high-level procedures, first. (Muccini; Demillo et al. 22-23) This is done by including dummy procedures called stubs which emulate the functionality of the lower-level procedures. (Royer 142) Stubs are less complex than the real procedures so they are easier to prove correct. The top-down approach does not mutually exclude testing and integration like bottom-up and may improve programmer morale by allowing him or her to preview the product before it evolves into a fully-functional form. (Demillo et al. 19-22) Two widely used styles of integrating program pieces are breadth-first and depth-first. (Muccini)

Breadth-first

Imagine if a hierarchical program or subprogram were to be represented as a tree with the main function taking the form of the root and other vertices representing various modules. Breadth-first top-down integration tests modules that are a given distance from the root in combination with one another. Vertices of equal distance are said to be on the same level of a tree. It is rather elegant because each individual end-to-end path need not be considered.

Depth-first

Depth-first top-down integration can be represented as a directed graph. Stubs are replaced with modules that are successively deeper in the call sequence for each parent module. It is apparent that modules may be integrated and tested more than once. This accounts for the fact that modules may be called at different levels in a call stack. In fact, many programs do not have a hierarchical top-down design methodology. In summary, each module is tested by

continually replacing stubs with actual program units. (Muccini)

Bottom-up

The bottom-up approach to integration testing is best when the meat of a program is in components at a low-level of control. (Demillo et. Al. 22-23) Drivers treat the current unit as an individual. Drivers are special units that verify other units by offering them test data. They can be thought of as test harnesses. (Muccini) A disadvantage of bottom-up integration is that the final program cannot be foreknown as with the top-down approach. (Royer 143)

Mixed

Utilizing a mixture of top-down and bottom-up integration testing is practical because it allows the development process to enjoy the advantages of both.

Sandwich

Sandwich is a particular type of mixed testing which pairs together the results of comprehensive top-down and bottom-up integration. This type of testing suffers the disadvantage of neglecting middle-level procedures (Mosley)

The top-down and bottom-up integration approaches are regarded as incremental testing strategies. (Demillo et. al 19) There is both manual and automated component integration. As might be expected, someone or something can integrate program or system components by the top-down or bottom-up approaches using black or white box test design methods with static or dynamic analysis.

CI Level Testing

This final stage of testing tests the project as a whole. It concludes with the successful demonstration of a working piece of software, but actually extends itself indefinitely in the form of validation.

Keep in mind that project life is defined in terms of a cycle. Hence, it has the property that it may revert back to a previous step. This happens when a step fails. A step fails if there is an unsuccessful verification. Verification is the process of confirming that the current step in the project life cycle complies with previous step or steps. A more specific type of verification called validation tests a "final product" (the term is being used loosely) that has been released to the public to be sure that it fits specifications. (Royer 17)

Conclusion

All matters in software testing are offspring of the fundamental theorem of computing. Also known as Alan Turing's halting theorem or halting problem, it shows why one cannot write a program to test if an arbitrary program of arbitrary input will eventually halt; even if this were possible, the program could show to be self-contradictory. (Abelson, Sussman, and Sussman 387) The halting problem demonstrates why software testing is a problem. It is impossible to show that an arbitrary program is correct for arbitrary inputs. Conversely, by making good design and testing choices, a program can be shown to be incorrect. Be sure to choose your software testing tools and techniques wisely.

Bibliography

Abelson, Harold. Structure and interpretation of computer programs / Harold Abelson and Gerald Jay Sussman, with Julie Sussman; foreword by Alan J. Perlis. Cambridge, Mass.: MIT Press; New York: McGraw-Hill, 1996.

Ammann, Paul E. Abstracting formal specifications to generate software tests via model checking [microform] Paul E. Ammann, Paul E. Black. Gaithersburg, MD: U.S. Dept. of Commerce, Technology Administration, National Institute of Standards and Technology, 1999.

Ammann, Paul E. A specification-based coverage metric to evaluate test sets [microform] Paul E. Ammann, Paul E. Black. Gaithersburg, MD: U.S. Dept. of Commerce, Technology Administration, National Institute of Standards and Technology, 1999.

Demillo, Richard A. et al Software Testing and Evaluation. Menlo Park, Calif: Benjamin/Cummings Pub. Co., 1987.

Goldfine, Alan H. Experience report: comparing an automated conformance test development approach with a traditional development approach [microform] Alan Goldfine, Gary Fisher, Lynne Rosenthal. Gaithersburg, MD: U.S. Dept. of Commerce, Technology Administration, National Institute of Standards and Technology, 1998.

Goodaire, Edgar G. and Parmenter, Michael M. Discrete Mathematics with Graph Theory. 2nd ed. Upper Saddle River, NJ: Prentice Hall Pub. Co., 2002.

Kaner, Cem et al. Testing computer software / Cem Kaner, Jack Falk, Hung Quoc Nguyen. New York: Van Nostrand Reinhold, 1993.

Kit, Edward. Software testing in the real world: improving the process / Edward Kit; edited by Susannah Finzi. Wokingham, England: Reading, Mass.: Addison-Wesley Pub. Co., 1995.

Muccini, Henry. "Software Testing." *Henry Muccini*. <http://www.henrymuccini.com/Testing.htm> (Dec. 15 2002)

Rivest, Raymond. "comp.software.testing Frequently Asked Questions (FAQ)" 1 Dec. 2002. Online posting. Newsgroup: comp.software.testing. USENET. 13th Dec. 2002.

Royer, Thomas C. Software testing management: life on the critical path / Thomas C. Royer. Englewood Cliffs, N.J.: Prentice Hall, 1993.

"Unified Modeling Language Resource Center." *Rational Software*. 2002. <http://www.rational.com/uml> (Dec. 10, 2002)