Software Test Selection Patterns and Elusive Bugs

William E. Howden
CSE, UCSD
howden@cs.ucsd.edu

Abstract

Traditional white and black box testing methods are effective in revealing many kinds of defects, but the more elusive bugs slip past them. Model-based testing incorporates additional application concepts in the selection of tests, which may provide more refined bug detection, but does not go far enough. Test selection patterns identify defect-oriented contexts in a program. They also identify suggested tests for risks associated with a specified context. A context and its risks is a kind of conceptual trap designed to corner a bug. The suggested tests will find the bug if it has been caught in the trap.

Keywords Testing, patterns, defects, elusive, models, design

1. Introduction

1.1. Background

Since there is no automatic procedure for selecting tests that is guaranteed to reveal all of the defects in a program, a variety of imperfect testing strategies are used. It is common to use an empirical approach, to select tests that have historically been found to be useful in finding bugs.

The goal of the work described in this paper is to develop a systematic general approach to testing that is empirically based, and which can be used to discover both common and elusive bugs. Elusive bugs are the kinds that depend on some combination of conditions, or which may require tests that incorporate some knowledge of the program or application area under test.

1.2. Historical approaches

The two major historical approaches to testing are black box and white (or glass) box testing. White box testing, sometimes called structured testing, requires the use of tests that cause each branch or other component of a program to be tested. Black box, or functional testing, uses tests that are based on a program's specifications. Black box testing usually incorporates a partition or decomposition of a program's input domain into subsets. The idea is that the program is "the same" over each partition subset, and it is sufficient to choose one test from each.

The problem with white box testing is that elusive bugs often do not show up until some combination of actions is performed by a program. This means that covering a program's branches is not enough. It is necessary, in an execution, to execute a particular combination of branches. In other difficult cases, it is not so much the necessity of executing a particular combination of branches, but that of executing a branch with particular kinds of data.

Attempts to extend white box testing so that it will find hard-to-discover combinational defects include more advanced coverage measures such as def-use [10]. In general, this and other attempts to force the testing of significant combinations of statements or branches has not been found to be significantly more effective than ordinary branch testing. One problem seems to be that such methods are ignorant of the concepts used in designing and creating a program.

Attempts to make the coverage testing of program components more selective include weak mutation testing [3]. This may be useful for simple faults, but it seems that it is only by chance that it will be effective for defects which involve higher level programming or design and requirements level concepts.

Black box testing usually includes the testing of boundary cases in an input or output domain. This is more defect oriented than simple coverage. However, it and related methods do not appear to go far enough. They do not include information about the kinds of things that are being done by a program. And, by definition, they do not get inside it and base tests on how the program works.

1.3. Patterns

It seems that in order to find elusive bugs, we have to include more information about a program in the testing process. Test selection patterns hold that promise.

The patterns movement began with design patterns [2]. Prior to this, design methods were either very generic, such as Structured Design, or they were associated with specific application areas, such as the use of layers for operating system design. Design patterns are generic but are personalized to a particular design context. For example, the Creator Pattern gives rules for identifying the class X whose instances should have the responsibility for creating any necessary instances of a class Y.

Design patterns have been very successful. They identify best practices, and do it in an abstract way so that patterns are applicable to a wide variety of different possible concrete situations.

The phrase "test patterns" has been used in different ways, including the following four:

i) design for testability. The use of interfaces for constructor parameter specifications so that either actual production run objects or test objects can be passed is a commonly used test pattern.

ii) test process. The inclusion of integration testing in a testing process, to be performed after unit testing, is listed as a test pattern by some authors.

iii) testware constructs. In the Object-Mother pattern a factory object is used to create structures of instances in required states for a planned test.

iv) test selection. Standard testing methods, such as Category-Partition, in which the tester divides input and output domains into functional equivalence subclasses and chooses interior and boundary points for each class, are described as test patterns in Binder [1].

In this paper, we use the phrase to refer to test selection methods. The test selection patterns described here include standard methods, but seen from a new point of view, in addition to newer methods that allow bug cornering application concepts to be used in the testing process. One of the goals of the new methods is to incorporate the spirit of design patterns: to identify test patterns that personalize the testing process to a particular context as opposed to completely generic methods such as white and black box testing.

Some kinds of more modern testing methods, such as model-based testing, do take conceptual program information into account. In model-based testing [e.g. 9], it is assumed that a state model for a program is available. It is used to guide the selection of tests that are related to the abstract conceptual aspects of the program that are elucidated in the model. This kind of testing is part of the pattern based test selection strategy suggested here, where it corresponds to a class of test selection patterns.

Perhaps the most extensive use of the phrase "test patterns" occurs in Binder's monumental work on testing [1]. He uses the phrase in a general sense to include: test process activities, testware, and test selection methods. His approach to test selection methods is model-based in the sense that tests are selected that cover a corresponding model. Binder's approach is more general than state model testing, in that many different kinds of models are used, such as class models and decision tables. The model-based part of the test patterns approach that is described in this paper is similar to this, but there are some important distinctions.

The work described here is part of the PASTA (Pattern Activated Software Testing and Analysis) project. The idea in this project is to organize and construct testing and analysis patterns. This paper is restricted to testing. To distinguish the test selection approach described here from other approaches, it will be referred to as the PASTA approach.

2. Test selection patterns

2.1. Pattern templates and contexts

For the purposes of this paper, a pattern template is used which has 5 parts:

Name
Context
Risks
Tests
Examples

A context describes a situation in which a defect might occur, and is based on an occurrence of one of the three kinds of artifacts described below. The risks section of a template describes potential defects that are associated with the use of that kind of context artifact. The tests section suggests tests that will find the occurrence of such defects.

PASTA patterns fall into three major groups, according to three different kinds of context artifacts:

Program or physical artifacts
Design mechanisms or "themes"
Models

Program artifacts are basic entities such as variables and expressions. They occur in programs, but also in designs and specifications. Test patterns associated with these kinds of entities are generic in the sense that they can be used with any kind of program or relevant development product.

Design mechanisms or themes are conceptual devices that are used in developing a program. They are sometimes oriented towards particular program

application areas, or standard parts of programs. This orientation can be used to group them. Sample groups include: data processing, user interface, graphics, and distributed processing. Design mechanisms are the principal technique for constructing test patterns that are based on application concepts.

Design mechanisms appeared in limited form in Brian Marick's well known testing book [7]. The basic strategy proposed in his book is to look for "clues" in a program or its specifications. Clues include what are referred to here as program artifacts, and also include what Marick calls "cliche′s". Cliche′s are simple examples of design mechanisms. Marick lists two kinds of cliche′s: searching and sorting. The idea of a cliche′ is expanded in PASTA to include different kinds of design mechanisms from different areas.

The inclusion of models in PASTA test selection patterns is similar to the use of models in model-based testing. There is however, a subtle but important difference. In the approach described here, models (and also design mechanisms and program artifacts) are part of the context in a test pattern. In a PASTA model-based test selection pattern, the suggested tests relate to the context model and its risks, and so may suggest tests that have model aspects in them, but the model-based fault model came first. In Binder's approach to model-based testing, he starts by defining model-based tests independently of context. These are, basically, tests that cover a model. His context based fault models are not model-based, only the suggested testing methods. In some cases, his context/fault model describes classes of faults quite independently of the suggested model-based testing strategy, and there is no direct connection.

The PASTA approach is more in the spirit of the testing methods described in James Whittaker's *How to Break Software*, which starts with defect types and then identifies tests to expose those defects [13].

2.2. Patterns and risks

Two general kinds of risks appear in PASTA test selection patterns:

i) *fault-based risks*. These occur when empirical evidence indicates that there is a risk of certain kinds of faults occurring in the use of an associated kind of context artifact.

ii) *failure-based risks*. These occur when empirical evidence indicates that there is a risk of program failure when an associated kind of artifact is used with certain kinds of defect-prone data.

Many defects involve some quirk or complication that is a significant factor in the occurrence of the defect, particularly for elusive bugs. In this case we may be able to identify a secondary risk. A *primary risk*, and its suggested tests, refers the simple, uncomplicated form of a risk. A *secondary risk*, and its tests, involve the complication. When a secondary risk occurs the pattern template will contain complication, secondary risk and secondary risk test sections. In some cases the primary risk test section may be omitted. Some of the examples below contain both primary and secondary risks.

2.3. Patterns and tests

In the case of failure-based risks, tests are often a part of the definition of the risk. For example, a risk in the use of the search design mechanism is that that it will fail to work if the searched item is not present. The suggested test follows immediately from the risk description.

Fault-based risks are less suggestive with respect to tests. The tester must find tests that will reveal the occurrence of the fault if it is present. The suggested tests should be more than "devise tests that will reveal the fault". It should give suggestions on how to build such a test, as in the pattern examples given below.

2.4. Patterns and examples

Test patterns must include one or more examples. This is because patterns are both informal and abstract. The examples "inform" the pattern. It is examples that give them their substance and which make them understandable. For the purposes of this paper the following template for a defect description will be used:

Name
Application Brief description of the program/system
Defect Brief description of the defect that occurred
Failure Invalid behavior that occurs
Fault The program bug that causes this
Error The human error leading to the fault
Severity Critical, etc
Source Phase When the bug was introduced e.g. design, requirements, implementation, enhancement
Detection Phase When it was detected

Note that we do not always have all of the above information.

The following sections describe some sample patterns for each kind of context artifact.

3. Program artifacts and patterns

3.1. Program artifacts

Two kinds of program artifacts will be discussed in this subsection: variables and expressions. We will consider one test selection pattern for each. These artifacts may occur as parts of design mechanisms and models, as well as programs, so the tests that are suggested here are also relevant during the consideration of these higher level pattern contexts. The difference is that when they occur in a mechanism/theme or model context it is possible to define tests that take additional, more application dependent information into account

Artifact patterns subsume traditional program coverage testing, since they require the testing of the program elements used to define coverage measures. Test patterns are more general than simple coverage, since they can identify kinds of important testing risks not readily incorporated in automated coverage measurement, such as in the following sample pattern and its associated complication.

3.2. Variable test patterns

Pattern Maximum Boundary Values
Context Artifact: A scalar variable x
Risks A program fails when a variable takes on its largest possible value
Tests Construct a test in which the variable takes on its maximum value, for each occurrence of the variable

Complication The maximum value of x at any time is bounded by the current value of another variable y
Risks The program may fail when x is at its absolute maximum (i.e. equals y when y is maximum)
Tests Construct tests where x takes on its absolute maximum

The defect in the following example will be revealed if it is tested using the above Maximum Boundary Values pattern.

Example Dating system bad delete, initial state
Application Simple dating system where members can be added and deleted
Defect If the user attempts to delete a non-existent member in a session, before an existing member has been deleted, an out of bounds array index is produced. This occurs because there is a loop index x that iterates from zero to the position where an item in a vector is stored, or to an upper bound y when no item is found. Initially, y is set to the size of the vector that is used to store the dating data, which would make it out of bounds by one. If a user is deleted, then y is reduced, so that x will now always be in bounds.
Failure Program crash, error message
Fault Bad logic
Error Fuzzy reasoning
Severity Critical
Source Phase Detailed design
Detection Phase Post release

3.3. Expression test patterns

Pattern All Terms Relevant
Context Artifact: Boolean expression
Risks The program fails to make certain distinctions because the effects of one of the terms is masked and does not affect the expression's outcome.
Tests For each basic term t, construct a test in which the rest of the expression is fixed in such a way that as we vary the value of t the value of the whole expression varies. If possible, relate this term to the expected alternative program behavior and use tests in which that behavior should vary.

An example of the use of this pattern is included below in the discussion of design mechanisms.

4. Design mechanisms and patterns

4.1. Mechanisms and program application areas

These kinds of patterns are important for boring down on a hiding bug. They include general mechanisms such as Marick's search and sort cliche′s. Mechanisms may be associated with classes of programs. An example from each of two areas is included here: User Interfaces and Data Processing.

4.2 User interfaces

Pattern User interfaces: Validated Data Entries
Context Design Mechanism: The user enters a set of integers or other data items in one or more slots, whose range validity is checked
Risks Failure to completely check each entry item
Tests Enter invalid data for each slot that should cause a change in the program behavior

Complication Interdependent validity specifications. The validity of one item depends on the current entries of the others
Risks Not all interdependencies are tested
Tests Try invalid data in each position, both for the position check and the relationship validity checks

Example Nokia Cell phone clock set
Application A certain (older) model Nokia cell phone contains a menu item that allows the user to set the current time clock using number and position changing keys. The time can be in either 24 or 12 hour representation. There are four digits. The last two digits have non-interdependent validity checks. The first entry can be 0,1 or 2. The phone will stop you from entering an invalid number in this position. The second digit can be a 0,1,....9, but depending on the first digit some entries are not allowed. For example, if the first digit is a 2, you cannot enter an 8.
Defect The phone fails to check the first number being entered against the value of the second. So if the time is 18:00 and you try to enter a 2 in the first position, it will let you. (This is rejected by a later time setting mechanism, but the entry should be rejected as it is entered, as are other illegal entries).
Failure Illegal time entered
Fault Bad logic
Error Fuzzy reasoning
Severity Low
Source Phase Detailed design
Detection Phase Post release

The Validated Data Entry pattern, with its complication, identifies a set of necessary tests that were apparently omitted from the cell phone tests. We can consider what other kinds of testing might have worked. Since there is an implicit expression for specifying validity, we could use expression testing.

We have an input space with 4 variables, say h1,h2,m1,m2, for the first and second hour digits and the first and second minutes digit. Each digit has a range specification:

0<=h1<=2
0<=h2<=9
0<=m1<=5
0<=m2<=9

One kind of necessary test will require trying to enter invalid values. But there are also some validity relationships:

h1 = 0,1 and h2 = 0-9
or
h1 = 2 and h2 = 0,1,2,3

We could apply the All Terms Relevant expression testing pattern here, which is essentially equivalent to the use of this pattern for this example.

4.3. Data Processing

Pattern Implicit Account Break
Context Design Mechanism: It is common to have a file of records that has been sorted by account number that has to be sequentially processed. It is necessary, when reading through the sequence of records, to recognize when the account changes (i.e. recognize an account break), in order to do special processing at that time.
Risks Failing to recognize the account break
Tests Designed to give different output if an account break is missed

Complication Alternative kinds of records occur, requiring different kinds of processing
Risks For some kinds of records the implicit account break processing is omitted and not detected
Tests Test the account break detection feature for each kind of record

Example General ledger accounting system
Application Accounting system that periodically has to update accounts from a transaction file. The records can be financial or non-financial, resulting in different kinds of processing.
Defect The account break is not checked for when a non-financial record is encountered. So if you have an account that has financials, that ends with a non-financial, and that is followed by an account with financials, then the entries for the two accounts will get merged together.
Failure Incorrect account totals
Fault Missing code
Error Fuzzy reasoning
Severity High
Source Phase Detailed design
Detection Phase Post release

5. Models and Patterns

5.1. Models

Two kinds of models will be discussed here: functional and state models. We can consider both simple and compound versions of these models. Compound models contain two or more interacting simple models. In the case of state models, a compound model consists of a system of communicating components, each of which could be modeled as a state machine.

5.2. Functional models

Functional models are black box models for which there is an input and an output specification from which tests can be generated. In the introduction, this was introduced as an historically important kind of model that is commonly used for testing programs or program

components. Functional testing is very general and can be extended to include all kinds of functions, such as a low level function that indexes computations in a program loop, or a higher level functional slice of a program that computes a result for a particular kind of data. Much of the testing that occurs in other models, such as state models, can be described as instances of functional testing. For example, a transition in a state model is a function whose input is a state and an event, and whose output is a new state.

We can also use functional testing for abstract functions that give an overview of some system action. In the above data processing account break example, we could formulate the processing that takes place for each record as a function that takes the record and the current program state as input, and produces a new state plus a flag that indicates if an account break has occurred.

We will look at two examples here. The first is a simple concrete function.

Pattern Invalid Input Data
Context Functional Model: user can cause a function to be invoked that uses input data to return a result
Risks Function fails to check for invalid input data
Tests Test for invalid data for each kind of input. Look at ranges/domains and choose invalid data outside of each boundary

Example Illegal customer data (Jessica Chiang)
Application A banking application can be used to determine certain "profitability" measures of selected customers. The factors that are used in this function are: net interest revenue, other revenue, direct expense, indirect expense, and risk provisions. Each of these is based on a combination of the data plus certain equations.
Defect The program does not detect illegal data, and in particular, if there are negative numbers where they should not appear. It simply produces wrong results.
Failure Incorrect output
Fault Missing code
Error Oversight
Severity Moderate
Source Phase Design
Detection Phase System testing

The above example illustrates the application of the patterns approach to straightforward functional black box testing. There are no complications, and no special application specific concepts are needed for discovering this simple, non-elusive bug. In the next example, we consider a pattern related to the abstract account break function mentioned above. Application of the following pattern to the testing of this function will reveal the defect. In this case the bug is elusive, and the extra information in the complication section of the pattern is critical.

Pattern Multiple Subfunction Domain Subclasses
Context Functional Model: A function occurs whose input domain can be partitioned. Each partition element results in the application of a different subfunction.
Risks Program fails to work for some subfunctions
Tests Test each subfunction/partition element
Complication There is a common task that must be performed for each of the subfunctions, along with their unique tasks.
Risks Programmer assumes that common task is done in a common place and does not have to be done for each subclass
Tests Construct tests for each subdomain that will reveal if the common subtask is missing

5.3. State models

As discussed earlier, model-based testing often refers to the practice of generating tests that will "cover" all of the transitions in a state model. The emphasis in the patterns approach is somewhat different. As in model-based testing, the tester identifies the occurrence of a model but at this point we look to patterns to indicate possible tests associated with model-based risks, rather than simply trying to cover the model with tests.

Two sample patterns are given here. The first is a single model pattern for a simple bug. The second involves a compound state model in which two components communicate over a channel, and includes an elusive bug complication.

Pattern State Transition Input Validity
Context State model: Transition events are associated with the receipt of input, resulting in a transition to a new state
Risks Input received on a transition is invalid
Tests Construct invalid data tests for each transition, which are such that if the input data associated with the transition is invalid this is observable as unexpected program behavior

Example Cell phone message receipt (Angela Molnar)
Application A cell phone can receive short messages from a web site. When the message is received, it is stored and there is a transition to a message received state.

Defect If a message longer than 128 characters is received, the cell phone is unable to save and store it. It simply freezes.
Failure System freeze
Fault Missing failure detection/correction capability
Error Unknown
Severity Critical
Source Phase Design, requirements
Detection Phase Post Deployment

This is another example where we could have used a functional model. In this case the function corresponds to a state transition. In fact, we can view state models as being a tool for recognizing functions for functional testing.

In the next pattern, the suggested testing method involves the familiar domain partitioning technique used with functional models, but this time in conjunction with a communications channel.

Pattern Communications Channel Domain Partition
Context Different kinds of data are sent from one component to another, which can be modeled by a domain partition. For each partition element, the destination component will exhibit a unique different kind of behavior.
Risks The sender may not correctly send data for some partition element
Tests A test for each partition element

> *Complications* There is an initial domain partition that, for the purposes of communication, is mapped on to a different domain decomposition.
> *Risks* There may not be a simple 1-1 mapping and implementation for the derived partition may not be correct or complete.
> *Tests* Identify all derived partition elements and construct tests for each of them.

Example Outdoor light management system (Ryan Shyffer)
Application A system was built for turning lights off and on at a city's parks. It did this using paging hardware. A series of characters would be sent out, each of which specified what to do for the four 15 minute periods in an hour. For example, 1100 would indicate a half hour on and a half hour off. However, only 10 characters could be sent, so that some possibilities were eliminated. Basically, these were the ones where there was an isolated 15minute period in the hour in which the lights were on/off but in the adjacent periods they were off/on. The eliminated combinations were 1010, 0101, 1011, 1101, 0100, and 0010.

Defect The problem was that the system did not correctly behave for the derived partition element 1001.
Failure Incorrect behavior
Fault Missing/bad code
Error Oversight
Severity Critical
Source Phase Design/programming
Detection Phase Beta testing

6. Summary and Conclusions

Simple coverage measures and black box testing are effective for simple bugs, but more elusive defects may be too well hidden. It is necessary to "corner" them in a conceptual box inside of which they can be more easily discovered. Test selection patterns offer a promising way of doing this, while at the same time providing a uniform approach that includes both traditional methods such as coverage and black box testing, as well as more contemporary testing strategies such as model-based testing.

Design mechanisms and secondary risks can be used to introduce the kinds of application dependent concepts or refinements that may be necessary to trap an elusive bug. The identification of design mechanisms such as, for example, "Implicit Account Breaks", makes it possible to have personalized testing patterns in the spirit of a design pattern. The identification of secondary risks guides the tester in the development of more refined tests. Finally, test patterns emphasize the inclusion of examples, which inform a pattern. Examples suggest additional possible refinements in the application of the pattern.

A list of test selection patterns may seem to be the same thing as a list of defect taxonomies like those found in, for example, [1] and [5], but the emphasis and motivation are different. Bug taxonomies are lists of types of defects whereas test patterns are descriptions of testing rules. In some cases a defect category suggests a test but in others it does not. The difference is clearer when the defect categories focus on failures, as in [4] and [12]. The idea with failure lists is for the tester to look at possible kinds of failures in order to be jogged into seeing possible risks associated with a product. With PASTA test patterns you look for context artifacts, and then use patterns to identify risks associated with the use of such artifacts, leading to the use of associated tests or analysis methods.

A list of test selection patterns may also resemble an inspection checklist. A risk may be manageable by inspection if its manifestation can be recognized by reading the code or other development artifact. The analysis side of PASTA includes inspections for

situations like this. But risk management using inspections is not always possible for several reasons. In the case of failure-based risks, an inspection may be effective only in those cases where the code is simple enough to be mentally executed. In the case of fault-based risks, it may not be easy to see if the fault is present, and a test may be easier to perform than an attempted analysis. For example, interfaces are a common form of risk. If the interface is documented, consistency inspection may be sufficient. If it is not, carefully chosen interface tests may be the only feasible approach.

Two potential problems with the patterns approach are: i) difficulty in accessing and using the right test selection patterns and ii) having an incomplete set of patterns. If there are too many patterns, and it is necessary to manually read through them all to see which are applicable, the approach may not be acceptable. At present, the PASTA collection of patterns is organized in a simple hierarchical directory. As it grows, it may be necessary to use something more sophisticated.

In order for the PASTA approach to work, it is necessary to develop a comprehensive set of design mechanism and model patterns. However, even if we only have an incomplete set, the approach is still feasible for two reasons. First, we can incorporate traditional methods as test patterns, so we still have the testing power of non-pattern oriented methods while gaining additional testing discernment with those patterns that have been developed. Second, the exercise in examining existing patterns for relevance, even if none are found, may suggest new design mechanisms or model complications that could be used to improve the testing of the program under evaluation.

Test patterns were a new and popular topic several years ago. Brian Marick started a test patterns web site [8] and several patterns workshops were organized but the interest abated. The web site has not been developed since 2001 and the workshops were discontinued. The recent publication of a Java testing patterns book [11] may motivate new interest in the area. As for the PASTA project, current work includes: augmenting the patterns collection, identifying additional design mechanisms and exploring a test and analysis patterns-oriented development process.

7. References

[1] Robert V. Binder, *Testing Object Oriented Systems*, Addison-Wesley, 1999.

[2] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns*, Addison-Wesley, Reading, MA, 1994.

[3] William E. Howden, Weak Mutation Testing and Completeness of Test Sets*, IEEE Transactions on Software Engineering*, Volume 8, Number 4, July 1982.

[4] Ajay Jha & Cem Kaner, " Bugs in the brave new unwired world." *Pacific Northwest Software Quality Conference,* Portland, OR, October 2003.

[5] Cem Kaner, Jack Falk, and Hung Quoc Nguyen, *Testing Computer Software*, Thomson Computer Press, 1993.

[6] Manfred Lange, Its Testing Time! Patterns for Testing Software, Gemplus GmbH, 2001.

[7] Brian Marick, *The Craft of Software Testing*, Prentice Hall, 1995.

[8] Brian Marick, http://www.testing.com/test-patterns.

[9] Harry Robinson, Finite State Model-based Testing on a Shoe-String, Microsoft, 1999.

[10] Sandra Rapps and Elaine Weyuker, Selecting software test data using data flow information, *IEEE Transactions on Software Engineering*, vol. 11, no. 4, April 1985.

[11] Jon Thomas, Mathew Young, Kyle Brown, Andrew Glover, *Java Testing Patterns*, Wiley, 2004

[12] G. V. Vijayaraghavan, *A Taxonomy of E-commerce risks and failures,* M.Sc. Thesis, Florida Tech, 2003.

[13] James A. Whittaker, *How to Break Software*, Addison Wesley, 2001.