Patterns of Test Oracles - A Testing Tool Perspective

YUNG-PIN CHENG, Dept. of CSIE, National Central University, TAIWAN
TSUNG-LIN YANG, Dept. of CSIE, National Central University, TAIWAN

Continuous integration (CI) has become a state-of-art software engineering practice in modern software industry. The key practice in CI is automatic test regression. A successful test automation can provide immediate feedback to developers and then encourages frequent code change to meet the spirit of agile methods. Unfortunately, the cost of building test automation varies significantly from different application domains. It is often inevitable to adopt testing tools of different levels to ensure the effectiveness of test regression. While adopting these testing tools, it is required to add test oracles by developers to determine whether a test is passed or failed. Nevertheless, there are different kinds of test oracles to choose, understand, and apply. Sometimes, a testing tool may constrain the feasibility and selection of test oracles. It is important for developers to understand the pros and cons of test oracles in different testing tools. In this paper, patterns of test oracles are described, particularly from a testing-tool perspective in practice. These patterns are collected to clarify possible confusion and misunderstanding in building test oracles.

Categories and Subject Descriptors: H.5.2 [**Software and Its Engineering**]: Software testing and debugging—*test oracles*

General Terms: Software Testing

Additional Key Words and Phrases: Software Testing, Test Oracle, Test Automation, Regression Testing, Assertion

ACM Reference Format:

Cheng, Y.-P. and Yang, T.-L. 2017. Patterns of Test Oracles - A Testing Tool Perspective. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 22 (October 2015), 12 pages.

1. INTRODUCTION

Software testing [Bertolino 2007; Lewis 2016; Myers et al. 2011; Naik and Tripathy 2011] is the major quality assurance approach in software industry. In a traditional testing, efforts to improve quality have centered around the end of the product development cycle; that is, the detection and correction of software defects. This traditional quality assurance method, however, faces dramatical challenges in the modern, fast-paced, and iterative Agile development. While applying effective testing methods to find bugs in a complete system is still critical, building tests incrementally from bottom-up is gaining momentum nowadays.

In a recent trend, introducing continuous integration (CI) into software development cycle has become a state-of-the-art software engineering practice in modern software industry. The key practice in CI is automatic test regression[Engström and Runeson 2010]. A successful test automation can provide immediate feedback to programmers. With automatic test regression, programmers are encouraged to boldly modify the code to meet the spirit of Agile methods. In a fast-paced Agile software development cycle, critical project health indexes can only be provided by test automation. Test automation is considered as the most important practice to prevent backward progress.

This work is supported by the Widget Corporation Grant #312-001.
Author's address: Yung-Pin Cheng, No. 300, Zhongda Rd., Zhongli District, Taoyuan City 32001, Taiwan; email: ypcheng@csie.ncu.edu.tw; Tsung-Lin Yang No. 300, Zhongda Rd., Zhongli District, Taoyuan City 32001, Taiwan; email: ylin0603@gmail.com

A test is an execution path that goes through some code of interest, which is created by an ordering sequence of input data and user inputs. A set of good tests can manifest critical bugs in the run. In practice, these tests are still derived by humans. The number of tests can range from hundreds to thousands. When code is modified, it is obligated to rerun all the tests, which is called regression testing. Unfortunately, rerunning all the tests can be tedious and time-consuming if performed manually. So, seeking out methods to automate manual testing has been a practical challenge in nowadays CI.

The key components of a successful test automation are as follows:

(1) to precisely replay, regress, or execute the tests, aka *test execution engine*, and

(2) to determine the correctness of a test run mechanically, aka *test oracle*.

To be able to precisely or faithfully replay a test is in general a difficult problem, particularly when graphical user interfaces (GUIs), multi-threading, networking, etc. are involved. Take web applications for example. A straightforward sequence of mouse click with absolute timing can fail from time to time because the objects of an html page are loaded nondeterministically. So, most testing tools with a test execution engine (or a test driver) typically limit their replay capability on specific platforms or domains.

Once a test run can be replayed automatically, the last thing is to add test oracles to determine the correctness of the run; that is, whether a test is passed or failed. The entity that judges the correctness of a test run is called a test oracle. Humans are the most powerful test oracles. However, the whole idea of test automation is to replace the humans by machines. There are several types of test oracles in software testing(see [Hoffman 1998; Oliveira et al. 2015]). In these papers, test oracles were mostly explained by more of an academic fashion. In practice, particularly in a CI, the implementation and design choices of test oracles often depend on and constrained by the adopted testing tools or testing frameworks. So, the major difference of this paper from [Hoffman 1998; Oliveira et al. 2015] is we describe the test oracles from a practical and popular testing tool perspective and in the form of pattern languages. These patterns suggest some concrete choices while building test automation in a CI.

There are three patterns described in this paper. The first pattern is specification-oriented test oracle. This pattern actually serves as a reference pattern for explanatory purpose, which briefly explains the testing principle, complexity, and feasibility behind the test oracles. Since omniscient test oracles are often infeasible and impractical to build, the first pattern explains why software engineers choose the directions in second pattern and third pattern to achieve test automation. The second pattern describes the popular test-oriented unit oracle, which is the major approach to achieve white-box test automation nowadays. Despite the popularity, the cost and drawbacks of adopting unit testing in practice is described. The third pattern is graphical-oriented test oracle. Testing a black-boxed complete system is indispensable in industry. Unfortunately, a complete system complicates the testing in many ways. The third patterns will describe the popular tools and approaches for building test automation and how the test oracles are implemented.

In Fig. 1, we plot a figure to illustrate the difficulties, cost, and time for test automation when the granularity of code under test (CUT) grows from unit code to system code. The granularity of code can grow from code blocks, methods/functions, classes, modules, components, subsystem, to system. Examples of code of low granularity are white-boxed methods and classes. Examples of code of high granularity are black-boxed programs, processes, or a distributed system composed of several processes. In principle, the difficulties, cost, and time to execute a test grow significantly as the code granularity increases, so are test oracles. For example, a test in xUnit[Meszaros 2006] can take less than a millisecond to complete, whereas a GUI test may take a few minutes to complete.

Why is there such a growth in cost, time, and efforts when code granularity increases? Here, we give an explanation by examples from bottom-up, starting from unit tests. An xUnit test is in general written to test a single class. When the class has interactions to other classes, test stubs and mocks[Osherove 2013] to fake the behaviors of other classes are required for the test. A test stub is an entity that mimics the correct behaviors of a particular test. A mock is an entity that helps asserting the tests when the correctness cannot be determined by the return values of class under test. In principle, xUnit test frameworks do not prevent you from testing a group

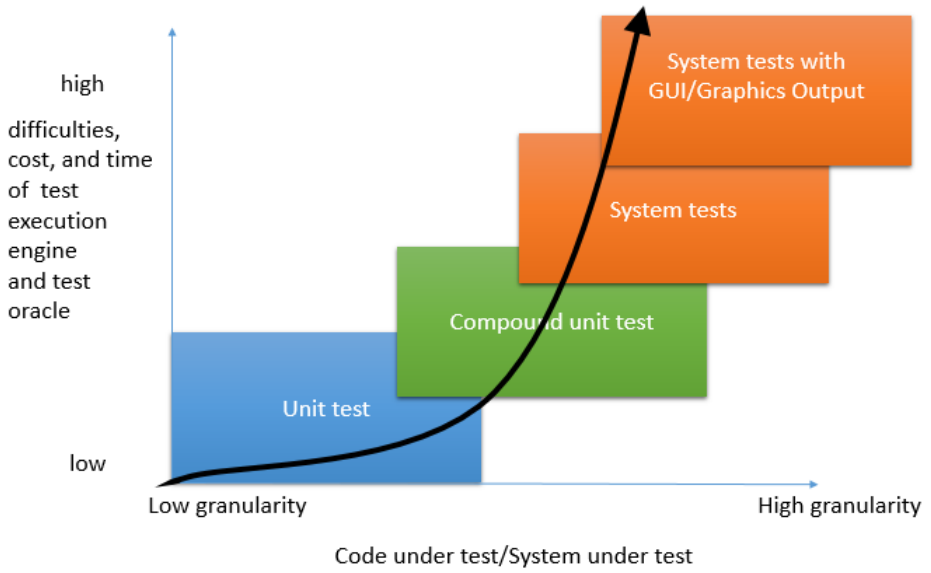


Fig. 1. The technical difficulties of test execution engine and test oracles as the granularity of code under test grows.

of objects working together. For example, let the class under test be X which interacts with class Y (faked by a test stub in unit tests) and Z (faked by a mock in unit tests). To test X,Y, and Z working together, a naive thinking would be to create the needed objects from Y and Z for the test. Such an attempt will actually level up the testing complexity from unit testing to integration testing. Unfortunately, it is not easy and straightforward as it may seem in practice. For example, Y could be some class in a library. Creating an object from Y could mean creating a series of related objects from the library first. To make it worse, the creation of an object from Z could mean to start a server program. Simple leveling up from unit test to integration test could result in dramatical increase of testing cost and efforts.

Another significant increase of testing cost and efforts is when a subject under test becomes black-boxed or coded with GUIs. For example, when GUIs are involved, tests can only be triggered via GUI events. Unlike the unit tests, where the correctness of tests can mostly be asserted by the return values of methods under test, white-box programming techniques for building test oracles may no longer be feasible. Furthermore, when a system under test is black-boxed, a new testing activity called deployment testing is introduced. Deployment is the activity to execute the system under test in a real environment correctly. Deployment testing is indispensable for test automation. However, automatic deployment can sometimes be expensive, this is why the DevOps technology [Loukides 2012; Httermann 2012] were proposed in recent years to tackle the problems of CI.

In reality, tests of different granularities are not always replaceable; that is, a system test cannot be simply replaced by a set of unit tests. This is a well-known fact in many disciplines. So, in practice, testing tools of different granularities are often combined to address the quality issue together.

2. SPECIFICATION-ORIENTED TEST ORACLE

2.1 Context

In software testing theory, the ultimate correctness of a software system is whether the behaviors of the system conforms to a set of specifications. So, it is not surprising to design and implement test oracles based on specifications. Test oracles that can validate whether a system under test conforms to specifications are called

“true oracle.”[Hoffman 1998](or equivalently, a pseudo oracle in [Oliveira et al. 2015; Davis and Weyuker 1981]). A true oracle is an artifact that faithfully reproduces all relevant results for a system under test (SUT) using independent platform, algorithms, processes, compilers, code, etc. That is, the same values can be fed to a SUT and a true oracle and you can compare the results to determine the correctness of SUT. Unfortunately, a true oracle seldom exists for a given SUT because if it does, the true oracle can simply replace the SUT for the job.

Since building a true oracle is impossible for most cases, sometimes, partial oracle [Davis and Weyuker 1981] may exist for some problems; that is, code under test may have a straightforward method to check if its output goes wrong. Test oracles that perform such a check is called a partial oracle.

2.2 Problem

It is impossible for you to exhaustively explore the relations between inputs and outputs to assert the correctness of an SUT. Instead, when the outputs of an SUT go wrong, you have an algorithm to check.

2.3 Solutions

Before building a specification-oriented test oracles, it is crucial to understand the specifications of code under test and then decide whether it is tested as in a white-box or a black-box manner. The correctness of code under test is often composed by a set of specifications. Each specification should be addressed by test oracles separately. Since a true test oracle for a problem is often impossible to build, you should instead explore erroneous system behaviors that do not conform to the specifications and see if these erroneous behaviors can be recognized by an algorithm.

2.4 Examples

It is comparatively easy to build white-box specification test oracles than black-box specification test oracle. In the following sample code, we use a sorting function to explain how a specification oracle can be constructed.

```
void sort(int input_array[], int output_array[], int n) {
    ....   // code of implementation
    ....   // code of implementation
    ....   // code of implementation

    // here begins the test oracle
    for (i=0;i<n; i++) {
       assert(output_array[i] <= output_array[i+1]) ;
    }
}
```

In the sample code above, a partial oracle is added at the end of the function. Theoretically speaking, if a true test oracle exists for the function, it can actually compare the `input_array` and `output_array` to assert the results. However, it is apparently impractical to derive such a test oracle. So, instead the test oracle at the end of sorting function is to assert when it goes wrong. By checking two consecutive `output_array` elements, we can determine if the results fail to conform the specifications. Note that, we do not check if the results succeed in conforming the specifications. In other words, it is a partial test oracle. Note that here we use the term “fail to conform the specifications” instead of “fail the test” because the two terms have different meanings. Specification test oracles are often added at the end of code under test, not “at the end of a test”. They are supposed to guard the correctness for all inputs.

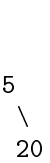
One difficult problem of applying specification-oriented test oracle in practice is you do not know when the partial oracles are precisely enough to conform to specs. For example, a straightforward partial test oracle for an AVL tree is as follows:

```
... // code and operations to an AVL tree
assert(abs(AVL.left.getHeight()- AVL.right.getHeight()) <= 1 ); //
```

This oracle looks reasonable and sound at first. Unfortunately, an AVL tree is not just a balanced tree. It is also a binary search tree. So, you should implement more oracles to check if an AVL tree is a binary search tree. Following is an example oracle of making such an attempt.

```
void check_left_smaller_than_right(AVLnode n) {
  if (n is a terminal node) return ;
  assert(n.left.key < n.key);
  assert(n.right.key > n.key) ;
  check_left_smaller_than_right(n.left);
  check_left_smaller_than_right(n.right);
  return ;
}
```

At first glance, this partial oracle looks reasonable. However, the following binary tree can pass the oracle; that is, the oracle fails to manifest a bug as follows:



You can go on to further refine the test oracles but it is indeed too difficult for programmers in practice. Debugging the test oracles so that there are no false positives and false negatives (aka. type 1 and type 2 errors) can be a challenge.

Sometimes, a partial oracle can be extended to become an *"invariant"* if you find that some properties should hold at each state or each instance of objects. For example, an example of an invariant from [de Champlain 1997] is in the following:

```
public void dec() {
  Assertion.require( count > 0 );
  count--;
  Assertion.ensure( invariant() );
}
protected boolean invariant() {
  return lower <= count && count <= upper;
}
```

In this example, the variable `count` plays the role of counter. When it is decreased or increased, it should not exceed its lower bound and upper bound. This property should hold for any states of an object, so it is called an invariant.

In many other cases, their code under test may not have straightforward or intuitive specification oracles to build. Take a basic data structure, hash table, as an example. Defining a set of partial test oracles to check the correctness of a hash table implementation is neither straightforward nor intuitive.

2.5 Know Uses

The idea of adding a specification-oriented test oracle is equivalent to post-conditions in design by contract programming [Meyer 1992], which is a software correctness methodology uses preconditions and postconditions to assert the correctness for a piece of code. The idea of pre-condition, post-condition, and loop invariants techniques originated from Tony Hoare [Hoare 1969] in his 1969 Communications of the ACM paper. The ideas of specification-oriented test oracle are strongly emphasized in the programming language Eiffel[Meyer and Software

]. In many popular programming languages, design by control supports typically appear as tools or add-on for the languages.

2.6 Consequences

A specification-oriented test oracle actually guards all inputs. So, in principle, you can apply test generation techniques to stress the SUT, without writing tests one by one manually. It is a dream in software quality assurance. Unfortunately, the advantages of applying specification-oriented test oracle may look promising as it may seem. However, as explained in the above, most programmers in practice have difficulties in defining correct specification test oracles for their code.

2.7 Related Patterns

The patterns of contract can be found in [Des ; de Champlain 1997] which describes the idiom that lets you apply assertions to guarantee pre-conditions and post-conditions of methods and invariants on the state of objects.

3. TEST-ORIENTED UNIT ORACLE

3.1 Context

As described in introduction section, when the granularity of code under test grows from a unit (white box), a module, to a system (black box), the cost of test automation increases significantly in general. The increased cost includes the efforts of adopting the right testing tools, creating test scripts, and maintaining the test scripts. Because of these reasons, building test automation has been considered an expensive technique in the past. So, instead of building black-boxed and system-wise test automation, let's introduce standardized unit tests and build test automation in a white-box manner.

3.2 Problem

You have completed your code under test in the units of methods (or functions in non-object-oriented programming languages like C) and classes. Your code under test is subject to change as the project grows and evolves. You need to maintain and monitor the code correctness every time code is changed.

3.3 Solutions

Although the cost of adopting a testing framework is considered much lower than other testing tools, choosing a right xUnit test framework for your programming language is still required. An xUnit test framework[Beck 1989; Meszaros 2010; Meszaros 2006] provides essential base class libraries and assertion commands for you to build test cases.

Unit code such as a class is not a complete program, so, an unit test framework is responsible for the creation of a test driver to run the code under test. For example, code, such as a library function, without a `main()` is not actively executable in operating systems. xUnit test frameworks hide the test drivers behind the object-oriented structure so that programmers can focus on writing tests without worrying about the details of how tests are executed.

To write tests for your class under test, you need to write test case classes that inherit base classes provided by the test framework. A test case class often contains several test case methods. These test methods will be invoked automatically by the hidden test driver. Each test method should be implemented by the author of code under test. It is not common to have other people write the unit tests for the code under test.

A test method typically contains two parts. The first part is to create objects from the class under test and then invoke the methods of these objects. The second part is to assert the results, which is the test oracle. Following is a list of assertion commands provided by the test frameworks.

```
Contains                DoesNotContain
Empty                   Equal
False                   InRange
IsNotType               IsType
NotEmpty                NotEqual
NotInRange              NotNull
NotSame                 Null
Same
Throws                  Throws True
```

Unlike specification test oracles which guard all inputs, the test oracle in a test method typically works only for the test method. This is why the test oracle is called test-oriented. A test is often composed of a set of selected input values to be fed to the code under test. These input values are often carefully selected. They can be representative values for some partitions in partition testing, boundary values in boundary testing, or triggering specific branches in structural testing. Given such input values, programmers can manually compute the expected outputs of code under test if it is implemented correctly.

3.4 Examples

The following is a sample test code in C++:

```
class Datetest : testcase {
....
  public void DateShouldBeEqualEvenThoughTimesAreDifferent() {
    DateTime firstTime = DateTime.Now.Date;
    DateTime later = firstTime.AddMinutes(90);
    Assert.NotEqual(firstTime, later);
    Assert.Equal(firstTime, later, new DateComparer());
  }
...
}
```

In this sample code, a test case class inherits the `testcase` from test framework. Each of its methods implements a test. In this example, a `DateTime` object is created and invoked by selected input values. The last two assert commands work as test oracles to determine the correctness automatically.

3.5 Know Uses

The rise of the xUnit test framework [Beck 1989; Meszaros 2006] introduced such a low-cost test automation technique. Most importantly, it is applicable to many platforms and programming languages, and irrelevant to application domains. It is arguably safe to say that the test automation in nowadays CI is mainly achieved by xUnit tests. In some application domains whose test automation cost increases rapidly as code granularity grows, it is possible that regression tests are all xUnit tests.

3.6 Consequences

Compared to specification-oriented test oracle, the simplicity and practicality make unit test framework become state-of-the-art test automation for modern software development. The largest cost of adopting xUnit test framework is the paradigm shift of programming culture. In the past, quality assurance by testing is often the responsibility of testers. However, in the new era of modern software engineering, more testing responsibility and work have been shifted to programmers in writing unit tests. In [Hevery 2009], Hevery described the problems, efforts, and difficulties in introducing unit testing into Google programmers. Besides the culture shift, good coding habits in

object-oriented programming are required for adopting unit tests. Poorly designed object-oriented code was found to make test code hard to write.

In practice, test scripts (see next section) of many testing tools can be generated automatically by a capturing tool to record the testing steps from a tester. However, unlike those test scripts, unit tests are code written manually by programmers. So, it is subject to same problem of code maintenance in practice, such as spaghetti code that is hard to extend and change and the problem of program understanding when the original authors of code under test leave.

Another major problem of completely relying on unit tests is the fact that unit tests cannot replace system-wise tests. Unit tests are not necessarily an effective way to find bugs or detect regressions. Unit tests, by definition, examine each unit of your code separately. When you run your application for real, all units have to work together and the whole is more complex and subtle than the sum of its independently-tested parts. So, it can be far more effective to actually run the whole application together if you are trying to find bugs. Unit testing, undoubtedly, helps improving and monitoring the quality during regressions but it should not be adopted as the only technique for quality assurance.

3.7 Related Patterns

There are a lot of patterns of xUnit testing documented in [Meszaros 2006]. This book describes *test smell* and test double patterns, including *test stub, test spy, mock object, fake object*, etc. It is crucial to understand these patterns before writing xUnit tests.

4. GRAPHICAL-ORIENTED TEST ORACLE

4.1 Context

Many real world programs have graphical user interface (GUI) or graphical output. Code with GUIs must be built and executed together with a GUI subsystem in operating systems. So, it is often tested as a program; that is, via system-wise tests. It is impossible to test such programs by unit testing. The testing of these systems has a special name, called GUI testing. Manual GUI testing is still the most effective way to find bugs. During the tests, humans interacts with a system under test by a mouse and a keyboard. The correctness is often determined by the states, properties, or graphical output of GUI components as a whole.

GUI testing has some major differences than testing a console program. First, the control flows of a program are mainly controlled by mouse events. So, the test execution engine of GUI testing is considered much more difficult than non-GUI testing. Precisely replaying a GUI test is a difficult problem in practice. Take web applications for example. A straightforward sequence of mouse click with absolute timing can fail from time to time because the objects of an html page are loaded nondeterministically. Second, GUI is often platform dependent, which makes a GUI testing tool applicable for all platforms in general. Third, GUIs are the most frequently changed program components. Such a nature poses a great challenge to GUI testing tools. This often explains why selecting a right GUI testing tool for a project can be difficult in practice.

4.2 Problem

You want to automate the regression of GUI tests. Besides the problem of faithful replaying the mouse and keyboard events, test oracles must be added to the faithful replays to determine the correctness of test runs.

4.3 Solutions

Programmers build GUI applications by a GUI SDK. A GUI component has properties like component ID, caption text, name, color, hidden, enabled/disabled, etc. In a trustworthy GUI system (3D applications are not included), these properties should faithfully reflect its appearance (in image pixels). So, you have two ways to define a test oracle: by asserting GUI component properties or by asserting image pixels.

By GUI component properties. When you build your GUI applications by a GUI SDK, it is always possible for the GUI subsystem to trace and monitor the GUI component while your applications are running. For example, if you build an application under Visual Studio, it can prepare a UI map file which contains the logical names and physical descriptions of GUI objects created from your applications. Such kind of information allows testing tools such as *Coded UI* [MSDN] to build automated GUI tests. These automated GUI tests, typically in the form of a script, can trigger mouse and keyboard events automatically during replay. The test scripts typically can be generated by a capturing tool which records the test behaviors performed by a human tester. However, there are no test oracles in these scripts. You need to add assertions in the scripts to make the tests really useful.

By screen image pixels. Not all the platforms provide a back door for testing tools to monitor and retrieve GUI component properties. For example, if your program is not developed under Visual Studio, your program cannot be tested under *Coded UI* even though it is still a Windows application. The capability of exposing the internal data of GUI components is platform dependent or SDK dependent. Recall that humans can judge the program correctness by screen output (i.e., image pixels), so in principle, test oracles can be designed to mimic the work performed by human brain and eyes.

4.4 Examples

By GUI component properties. Since GUI testing is a very basic testing activity in practice, most popular platforms support ways to probe the GUI component properties during runtime. One popular system is the android Robotium UI test framework. Robotium is an open source library extending JUnit with plenty of useful methods for Android UI testing. It provides powerful and robust automatic black-box test cases for Android apps (native and hybrid) and web testing. Robotium supports its UI testing under JUnit (using Java as its script language) but indeed it is a system-wise testing tool not a unit testing tool. Fig. 2 is a Robotium test method written in JUnit. Before executing the test method, the android app (i.e., .apk) is actually started by the Robotium engine. In Fig. 2, *solo* is a simple android app which multiply two numbers from two text areas. The test code is to enter a “10” into a test area which is indexed as 0 and then enter a a “20” into a text area which is indexed as 1. The third statement is to click the button which has caption text, “multiply”.

The last statement is the test oracle, which asserts a string 200 displayed somewhere in the *solo*. The assertion is similar in xUnit, except GUI properties are used for assertion.

By screen image pixels. So, to emulate a human test oracle, one intuitive approach is to analyze screen output (image pixels). This approach is pixel-aware in this sense. One representative image-analysis test tool is Sikuli [Yeh et al. 2009]. Sikuli proposes a GUI scripting language called Sikuli (see Fig. 3), in which the sequence of operations

```
public void testDisplayBlackBox() {
    // Enter any integer/decimal value for first editfield, we are writing
    // 10
    solo.enterText(0, "10");

    // Enter any interger/decimal value for first editfield, we are writing
    // 20
    solo.enterText(1, "20");

    // Click on Multiply button
    solo.clickOnButton("Multiply");

    // Verify that resultant of 10 x 20
    assertTrue(solo.searchText("200"));
}
```

Fig. 2. The Google test framework for Android apps.

of a test, called visual workflow, is recorded. The advantage of Sikuli script is its visual and easy-to-understand characteristics. The `assertExist` is the test oracle in a Sikuli script, where a desired image should appear at the point of test.

Another work that is closely related to Sikuli is T-Plan Robot[T-Plan]. Unlike Sikuli, T-Plan Robot is run at a separate machine from SUT. Running testing tools on a separate machine can guarantee no performance interference on the execution of SUT. However, the problem is how to intercept mouse/keyboard events from SUTs. The answer of T-Plan Robot is to run a VNC (Virtual Network Computing) server on the SUT machine. VNC is a common tool for users to remotely control another computer. It transmits the keyboard and mouse events from one computer to another and graphical screen updates back in the other direction over a network. So, T-Plan robot can be platform independent wherever a VNC server can be installed.

In Fig. 4, another image-based testing tool KORAT[Cheng et al. 2015] with keyboard capture/replay is illustrated. As shown in the figure, KORAT is run at a separate machine from SUT but the SUT's video output is connected to a video image capture card installed in the KORAT machine. A network port of SUT is connected to a switch which allows network traffic to send information to KORAT, which is used for precise assertion in a replay run. An ARM-cortex M4 development board is implemented as a USB emulator to intercept and send keyboard/mouse events to SUT's USB ports. There are several kinds of assertions in KORAT:

(1) assert the ASCII output

(2) assert the image output via image analysis and recognition (similar to Sikuli and T-Plan Robot)

(3) assert the OCR (optical character recognition) output

Recognizing images into human readable words is a powerful capability of humans. So, supporting such kind of assertions allows powerful test oracles to be built.

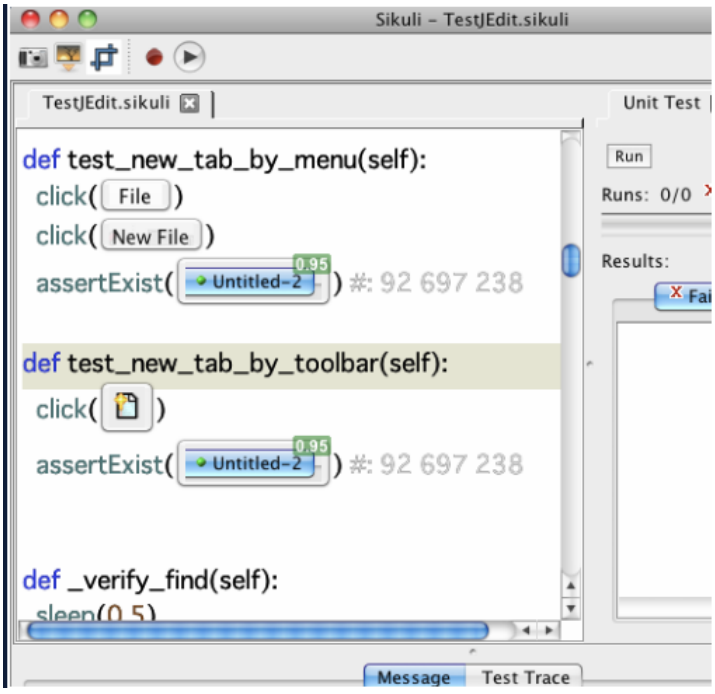


Fig. 3. The Sikuli screen-shot.

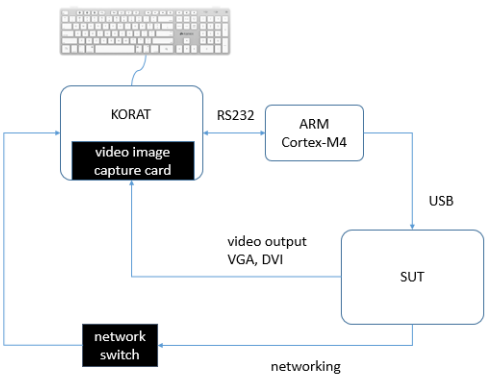


Fig. 4. KORAT Capture/Replay architecture with keyboard.

4.5 Known Uses

GUI testing tools have been available and widely adopt by software industry for some time. Besides the examples described in the previous section, there are plenty of GUI testing tools available for GUI automation[Wikipedia]. It is critical to choose a right GUI testing tools when a development team decides to automate the tests.

4.6 Consequences

If a GUI subsystem support accessing the GUI component properties, precision is guaranteed whereas test oracles based on image analysis are always subject to the problem of false positives and false negatives. In addition, GUI components are the frequently changed entities in a software system. Their screen coordinates, appearance, color, etc. can be changed from time to time. Test replays based on images can be fragile in practice. However, these problems can be easily overcome by providing GUI component internal properties to testing tools at runtime.

On the other hands, image-based testing tools are potentially more flexible and user friendly. Systems like KORAT can work in different platforms, such as BIOS, DOS, Windows, Linux, and game boxes such as Xbox and PS4. Recent astonishing progress of deep learning in image processing and recognition makes this approach much more promising. In addition, this approach mimics human test behaviors, so adding test oracles can be done without any programming capabilities.

4.7 Related Patterns

It is very hard to implement GUI testing without shooting yourself in the foot. Writing the tests in the adopted testing tool seems relatively easy, maintaining them would become harder and harder over time to some point where it was impractical to maintain because GUI tends to change often and frequently. There are comparatively less patterns of GUI testing documented in academic. If there are patterns or experiences shared to public, they are often bound to a specific testing tool and seldom described in a formal manner. As a result, the experiences and patterns for a testing tool may not apply to another testing tool. Although it is beyond the scope of this paper, it is necessary to collect and extract common patterns to avoid the troubles of adopting GUI testing tools.

5. CONCLUSIONS

In this paper, three patterns of test oracles in different code granularity are described. While adopting test automation, these patterns describe the state-of-the-art test oracles that are mostly likely to use. We also analyze and comment the pros and cons in applying the test oracles respectively. So far, there is no testing method alone to cover the quality assurance problem in test automation. By clarifying the test oracle patterns and understanding

their technical challenges, cost, and efforts, this paper can help selecting the right combinations of testing methods to build a successful test automation.

REFERENCES

Kent Beck. 1989. Simple Smalltalk Testing:With Patterns. https://web.archive.org/web/20150315073817/http://www.xprogramming.com/testfram.htm. (1989).

Antonia Bertolino. 2007. Software Testing Research: Achievements, Challenges, Dreams. In *2007 Future of Software Engineering (FOSE '07)*. IEEE Computer Society, Washington, DC, USA, 85–103. DOI:http://dx.doi.org/10.1109/FOSE.2007.25

Yung-Pin Cheng, Jen Wei Kuo, Ben Cheng, and Chia-Hung Kuo. 2015. A Non-intrusive, Platform-Independent Capture/Replay Test Automation System. In *17th IEEE International Conference on High Performance Computing and Communications, HPCC 2015, 7th IEEE International Symposium on Cyberspace Safety and Security, CSS 2015, and 12th IEEE International Conference on Embedded Software and Systems, ICESS 2015, New York, NY, USA, August 24-26, 2015*. IEEE, 1122–1127. DOI:http://dx.doi.org/10.1109/HPCC-CSS-ICESS.2015.138

M. D. Davis and E. J. Weyuker. 1981. Pseudo-oracles for non-testable programs.. In *In Proceedings of the ACM conference (ACM 1981)*. New York, USA, 254–257.

Michel de Champlain. 1997. The Contract Pattern. In *Pattern Language of Programs (PLoP'97)*. Montocello, IL, USA.

Emelie Engström and Per Runeson. 2010. *A Qualitative Survey of Regression Testing Practices*. Springer Berlin Heidelberg, Berlin, Heidelberg, 3–16. DOI:http://dx.doi.org/10.1007/978-3-642-13792-1_3

Misko Hevery. 2009. How To Write Hard To Test Code & What To Look For When Reviewing Other Peoples Hard To Test Code âĂŤ Tutorial. In *OOPSLA2009*. Orlando, USA.

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. DOI:http://dx.doi.org/10.1145/363235.363259

Douglas Hoffman. 1998. Analysis of a Taxonomy for Test Oracles. *Quality Week* (1998).

Michael Httermann. 2012. *DevOps for Developers* (1st ed.). Apress, Berkely, CA, USA.

W.E. Lewis. 2016. *Software Testing and Continuous Quality Improvement, Third Edition*. CRC Press. https://books.google.com.tw/books?id=fgaBDd0TfT8C

Mike Loukides. 2012. *What is DevOps* (1st ed.). ORelly Media.

Gerard Meszaros. 2006. *XUnit Test Patterns: Refactoring Test Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Gerard Meszaros. 2010. XUnit test patterns and smells: improving the ROI of test code. In *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, 299–300. DOI:http://dx.doi.org/10.1145/1869542.1869622

Bertrand Meyer. 1992. Applying "Design by Contract". *IEEE Computer* 25, 10 (1992), 40–51. DOI:http://dx.doi.org/10.1109/2.161279

Bertrand Meyer and Eiffel Software. Eiffel Programming Languages. Available as https://www.eiffel.org/. (????).

Microsoft MSDN. Use UI Automation To Test Your Code. https://msdn.microsoft.com/en-us/library/dd286726.aspx. (????).

G.J. Myers, C. Sandler, and T. Badgett. 2011. *The Art of Software Testing*. Wiley. https://books.google.com.tw/books?id=GjyEFPkMCwcC

K. Naik and P. Tripathy. 2011. *Software Testing and Quality Assurance: Theory and Practice*. Wiley. https://books.google.com.tw/books?id=neWaoJKSkvgC

Rafael A. P. Oliveira, Upulee Kanewala, and Paulo A. Nardi. 2015. Automated Test Oracles: State of the Art, Taxonomies, and Trends. *Advances in Computers* 95 (2015), 113–199. DOI:http://dx.doi.org/10.1016/B978-0-12-800160-8.00003-6

R. Osherove. 2013. *The Art of Unit Testing: With Examples in C#*. Manning Publications Company. https://books.google.com.tw/books?id=2GRRmgEACAAJ

T-Plan. T-Plan Robot. Available as http://www.t-plan.com/robot/. (????).

Wikipedia. Comparison of GUI testing tools. Available as https://en.wikipedia.org/wiki/Comparison_of_GUI_testing_tools. (????).

Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. 2009. Sikuli: using GUI screenshots for search and automation. In *Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology, Victoria, BC, Canada, October 4-7, 2009*. 183–192. DOI:http://dx.doi.org/10.1145/1622176.1622213

Received May 2015; revised September 2015; accepted February 2015