Journal of Software Engineering and Applications, 2014, 7, 94-103
Published Online February 2014 (http://www.scirp.org/journal/jsea)
http://dx.doi.org/10.4236/jsea.2014.72011

Obsidian: Pattern-Based Unit Test Implementations

James Bowring, Hunter Hegler

Computer Science Department, College of Charleston, Charleston, SC, USA.
Email: bowringj@cofc.edu

Received January 1st, 2014; revised January 30th, 2014; accepted February 7th, 2014

ABSTRACT

There exist many automated unit test-generator tools for Java with the primary task of generating test cases, comprised of inputs and a corresponding oracle, each of which is explicitly paired with a specific supporting test implementation. The authors posit that this explicit pairing, or conflating, of test implementation with test case is unnecessary and counter-productive. The authors address this problem by separating the conflated concerns into two distinct tasks: 1) instantiating test implementations and 2) instantiating test cases. This paper focuses on automating the first task in support of the second with the goal of freeing the test engineer to concentrate on test case instantiation. The authors present a new open-source test-preparation tool Obsidian that produces robust, comprehensive, and maintainable unit test implementations. Obsidian, built on the JUnit framework, uses a set of context patterns and associated algorithms combined with information from the Java Reflection API to generate these unit test implementations from Java byte code. These context patterns guide Obsidian to prepare test implementations that guarantee compilation, support exception handling, enable multiple test cases when required, and provide a suitable location for assertions about the test case outcome(s). Obsidian supports regression testing and test-driven development through its novel audits of the testing process.

KEYWORDS

Unit-Testing; Automated Testing; Testing Tool

1. Introduction

The software-process models extreme programming [1] and test-driven development [2] have inspired software test engineers to automate unit-testing tasks. There now exist many automated test-generator tools and most have the primary task of generating test cases comprised of a set of inputs and a corresponding oracle. The second task of these tools is to pair each of these generated test cases with a test implementation. We discuss the most popular such tools in Section 6. We posit that this pairing, or conflation, of implementation with test case is unnecessary and counter-productive. Furthermore, the generated test implementations may be hard to extend to other test cases due to their specialization, as suggested by Robinson [3]. We address this problem by separating the two conflated concerns into two tasks: 1) instantiating test implementations and 2) instantiating test cases. This paper focuses on automating the first task in support of the second through the generation of robust and complete test implementations in Java, thus freeing the test engineer to concentrate on test case instantiation.

We present a new open-source test-preparation tool Obsidian, built on the JUnit framework, which produces comprehensive test implementations that are ready to support any test case. Obsidian uses the Java Reflection API [4] in concert with a set of context patterns, abstracted from basic method signatures, to generate test implementations from Java byte code. These implementations support multiple test cases consisting of input and outcome data. These data can be generated by hand or by automation and then inserted into an Obsidian-prescribed location within the test implementation.

Obsidian's strategy for generating a method's test implementation uses Java Reflection to classify the method based on two Boolean conditions of its signature: 1) the presence of parameters, and 2) the presence of thrown exceptions. Each classification maps to a context pattern. A context pattern specifies how to prepare the test implementation to guarantee compilation, support exception handling, enable multiple test cases when required, and provide a suitable location for assertions [5] about the test case outcome(s).

Obsidian also provides a prescribed ordering of method test invocations within the test structure for the class. This order is based on the classification of each method's signature, using the bottom-up incremental approach prescribed by Myers [6] so that each method's test can depend on previously tested methods to change or access class or object attributes.

For every concrete class referenced in the class hierarchy under test, Obsidian also provides a hierarchy of specialized object-equality methods. These methods exist in a hierarchy at the class, package, and global levels. The test engineer can use these methods to easily make and maintain assertions about class or object attributes. Initially, the lower-level implementations simply call the next higher implementation in the hierarchy.

As the code base under test evolves, Obsidian can track and then report to the user that methods have test implementations and any class attributes that have been introduced since the last execution of Obsidian.

The main contributions of this work are:

- A set of context patterns to guide the generation of test implementations.
- Obsidian as a new open-source test-preparation tool for Java that uses context patterns to automatically generate unit test implementations.
- The creation of a novel abstraction for assertions as an equality method hierarchy; the implementation of accountability audits to manage the testing of attributes and methods over time; the generation of unit test implementations from byte code only.

Section 2 presents the theoretical underpinnings for Obsidian's approach, followed by a discussion of assertion handling in Section 3. Then, in Section 4 we present our strategies for building method test implementations using Java Reflection. In Section 5 we present Obsidian's workflow, followed by background material and related work in Section 6. Finally, we present our conclusions and future work in Section 7.

2. Context Patterns

Obsidian's strategy for generating a method's test implementation begins with classifying the method to determine which of four context patterns will prescribe its test implementation. A context pattern provides structural requirements for the test implementation so it can process one or more test cases, each of which consists of inputs, outcomes, and assertions about the outcomes. A context pattern also specifies how to guarantee exception handling and compilation.

To determine the context pattern for a method, Obsidian considers two boolean properties of a method's signature: 1) whether the method requires parameters, and 2) whether the method explicitly throws exceptions. These two properties dictate the structural requirements of a method's test implementation. When a method requires parameters, Obsidian's test implementations must facilitate the definition and execution of one or more test cases. Likewise, in the presence of exceptions, Obsidian's test implementations must support test cases where a thrown exception is expected as well as test cases where an exception is not expected. We define these as expected and unexpected exceptions, respectively.

Table 1 presents a summary of the four context patterns as conjunctions of the corresponding boolean properties "has parameters" and "throws exceptions". Each context pattern provides a standardized template for generating method test implementations from a set of building blocks. The resulting code structure facilitates modification and maintenance by a test engineer. Furthermore, these standardized implementations provide a defined insertion point for multiple test cases consisting of input and outcome data, which can be provided by hand or by a test-case generating tool. The basis of this standardization is a sequence of building blocks of code that are included or excluded based on the specific context pattern selected.

The following subsection presents the building blocks for context patterns followed by a subsection for each of the four Obsidian context patterns, including illustrative code examples.

2.1. Building Blocks for Context Patterns

Table 2 lists the sequence of basic building blocks for

Table 1. Obsidian's four context patterns determined by method signature.

Has Parameters	Throws Exceptions	Context Pattern	Pattern Nick Name
F	F	Call	CALL
F	T	Try-Catch	TRY
T	F	Test Case Iterator	CASE
T	T	Exception Test Case Iterator	TRY-CASE

Table 2. **Obsidian's sequence of building blocks for context patterns.**

Context Pattern	Basic Setup	Expected Exception Blocks	Array of Normal Test Cases	Start Test Case Iteration	Start Try-Catch Construct	Method Call	Close Try-Catch Construct	Assertions	End Test Case Iteration
CALL	▯					▯		▯	
TRY	▯				▯	▯	▯	▯	
CASE	▯		▯	▯		▯		▯	▯
TRY-CASE	▯	▯	▯	▯	▯	▯	▯	▯	▯

generating a method test implementation and shows when the block is used based on the context patterns. The common building blocks among all context patterns are Basic Setup, Method Call, and Assertions. The Basic Setup block includes code for Obsidian housekeeping and room for the test engineer to refine the setup, including preparing the object's state. The Method Call block includes code to support and call the method under test. These method calls follow the requisite signature relative to whether they are static, have a return type, or have parameters. The Assertions block includes assertions when possible, as described in Section 3. However, this block may only provide the location for the test engineer to place assertions.

The remaining building blocks are presented with the explanations of the four context patterns below.

2.2. Call Pattern (CALL)

The CALL context pattern handles methods that do not require parameters and that do not throw exceptions. It uses the building blocks Basic Setup, Method Call, and Assertions. An example of this context pattern is presented in Figure 1. This example, as well as the following examples illustrating the other context patterns, is from the Java 1.6 code base's Java.lang.object. Note that this context pattern does not require any test case data as there are no parameters.

This example is of an Obsidian-generated test implementation of the method public toString(). Obsidian uses JUnit's attribute "@Test" at the beginning of each test implementation (line 1). Lines 3 - 5 contain Obsidian's setup code, including a call to setTested of object methodMap, which provides method accountability, as described in Section 4.5. Line 8 is a placeholder for the test engineer to enter the expected result string. Line 11 produces the actual result from invocation, and line 14 provides the areEqual assertion, one of Obsidian's object equality methods, as presented in Section 4.4.

2.3. Try-Catch Pattern (TRY)

The TRY pattern handles methods that do not require parameters but that do throw exceptions. Obsidian surrounds the method call with a try-catch as specified in Table 2, which forces the test to fail if an exception is thrown,

```
1  @Test
2  public void testToString(){
3    System.out.println(‘‘Testing Object.toString
         ()’’);
4    methodMap.setTested(‘‘toString’’);
5    fail(‘‘This is a Generated Unit Test and
         Should be Modified’’);
6
7    //Expected Result
8    String expResult = ‘‘’’;
9
10   //Result
11   String result = instance.toString();
12
13   //Assertions
14   areEqual(expResult, result);
15 }
```

Figure 1. **Obsidian-generated test code for context pattern CALL.**

while providing the test engineer a helpful report. Note that any thrown exception is unexpected since there are no parameters.

An example of this context pattern is presented in Figure 2. This context pattern does not require any test case data as there are no parameters. This example is of an Obsidian-generated test of the method public Object clone() throws CloneNotSupportedException. In Line 14, result (initialized in line 11) gets a call to method clone(). Line 17 forces the test to fail if there is the unexpected exception CloneNotSupportedException.

2.4. Test Case Iterator Pattern (CASE)

The CASE context pattern handles methods that require parameters but that do not explicitly throw exceptions. With this context pattern and the TRY-CASE context pattern that follows, Obsidian provides support to the test engineer for multiple test cases within the same test implementation. This is in contrast to the approach of the other tools discussed in Section 6. The problem confronting users of these other tools is that in order to test a method using more than one test case, the user must duplicate the test implementation for each test case or build custom code to iterate through a set of test cases. JUnit solves this problem with "parameterized tests" that require, for any method test, a special method with the attribute "@parameters" that enables the JUnit runner to

```
@Test
public void testClone(){
  System.out.println(``Testing Object.clone()''
       );
  methodMap.setTested(``clone'');
  fail(``This is a Generated Unit Test and
       Should be Modified'');

  //Expected Result
  Object expResult = new Object();

  //Result
  Object result = new Object();

  try{
    result = instance.clone();
  } catch(CloneNotSupportedException ex){
    //Unexpected Exception
    fail(``Unexpected '' + ex.toString() );
  }

  //Assertions
  areEqual(expResult ,result);
}
```

Figure 2. Obsidian-generated test code for context pattern TRY.

iterate through a number of test cases. However, this solution requires specialized, hand-coded, test-class constructors and special test-class fields for each parameter. Obsidian solves this problem by supplying the required code to store, execute, and evaluate a multi-dimensional object array of test case inputs and outcomes as an Obsidian Test Case Iterator.

An example of this pattern is presented in **Figure 3** as an Obsidian-generated implementation of the method public void testEquals() that provides the multi-dimensional Object array placeholder for test cases at lines 7 - 11, and an iterator for this array with a placeholder for assertions at lines 14 - 18.

2.5. Exception Test Case Iterator Pattern

The TRY-CASE context pattern handles methods that both require parameters and throw exceptions. As described at the beginning of this Section, Obsidian's method test implementations are designed to handle expected as well as unexpected exceptions. Unexpected exceptions, such as those handled in the TRY context pattern (above), are thrown when not intentionally provoked by an input parameter. In the presence of parameters, Obsidian allows the test engineer to provoke a method into throwing exceptions to verify proper exception handling inside the method. If an exception is thrown by any test case not designed to provoke an exception, Obsidian considers this to be an unexpected ex- ception and therefore a test failure.

Thus, there are two types of test cases for this TRY-CASE context pattern: 1) those designed to provoke de-

```
 1 @Test
 2 public void testEquals(){
 3   System.out.println(``Testing Object.equals()'
        ');
 4   methodMap.setTested(``equals'');
 5   fail(``This is a Generated Unit Test and
        Should be Modified'');
 6
 7   //Test Cases
 8   Object[][] testCases = {
 9     {new Object(), new Object()}
10     // {new Object(), new Object()}
11     };
12
13   //For Each Test Case
14   for(int tc = 0; tc < testCases.length; tc++){
15     boolean actualResult = instance.equals(
          testCases[tc][0]);
16
17     //Assertions
18   }
19 }
```

Figure 3. Obsidian-generated test code for context pattern CASE.

clared exceptions, and 2) those that are not so designed. To handle these test cases, Obsidian defines an Exception Test Case Iterator as a collection of specialized Test Case Iterators (defined in the CASE context pattern above). For test cases designed to provoke declared exceptions, a Test Case Iterator is specified for each type of declared exception.

These Test Case Iterators are contained within Expected Exception Blocks (see **Table 2**). If the test case is designed to provoke a specific declared exception and the exception is then thrown, the test case passes. If a different exception, or no exception, is thrown, then the test case fails. Based on this clear distinction, Obsidian can generate Expected Exception Blocks with completed assertions and require only that the test engineer supply the test case inputs. For test cases that are not designed to provoke exceptions, Obsidian generates a final Test Case Iterator for normal test cases including a try-catch for unexpected exceptions. In this case, the test engineer must supply the assertions at the Obsidian-specified location.

An example of this pattern is presented in **Figure 4** as an Obsidian-generated test of the method public void wait (long) throws Interrupted Exception. Lines 7 - 10 initialize the test cases for the expected exceptions of type Interrupted Exception. Lines 12 - 23 automatically handle these test cases, as described in the previous section. Lines 24 - 41 are in the form of a normal Test Case Iterator with the added lines 31 - 40 to handle normal test cases and to handle unexpected exceptions. Per previous examples, line 36 is a placeholder for assertions.

3. Support for Assertions

The Assertions Block of an Obsidian method test imple-

```
@Test
public void testWait()
  System.out.println(``Testing Object.wait()'')
      ;
  methodMap.setTested(``wait'');
  fail(``This is a Generated Unit Test and
      Should be Modified'');

  //Expected InterruptedException Test Cases
  long[][] InterruptedExceptionTestCases = {
     {0L, 0L}
      // {0L, 0L} };

  //For Each InterruptedException Test Case
  for(int tc = 0; tc <
      InterruptedExceptionTestCases.length; tc
      ++){
    try{
      instance.wait(
          InterruptedExceptionTestCases[tc][0])
          ;

      //If Expected Exception wasn't thrown
      fail(``Expected Exception not thrown.'' +
          `` Test case #'' + (tc + 1)
        + ``:'' + InterruptedExceptionTestCases
            [tc][0]);
    } catch(Exception ex){
      areEqual(ex, new InterruptedException();
    }
  }
  //Normal Test Cases
  long[][] testCases = {
    {0L, 0L}
    // {0L, 0L} };

  //For Each Normal Test Case
  for(int tc = 0; tc < testCases.length; tc++){
    try{
      instance.wait(testCases[tc][0]);

      //If needed, access outcome via getter or
          Reflection

      //Assertions

    } catch(Exception ex){
      fail(``Unexpected '' + ex.toString() );
    }
  }
}
```

Figure 4. Obsidian-generated test code for context pattern TRY-CASE.

mentation contains the least amount of automatically generated code of any building block and thus requires the most effort on the part of the test engineer. Obsidian attempts to generate simple assertions [5] for Constructors and other methods that are determined to be Getters and Setters. It is the innate simplicity of these methods that allows Obsidian to generate effective assertions within their test implementations.

Testing the Constructors of a class is very important since almost every succeeding test will assume that the Constructors have no faults. The goal of a Constructor is to correctly instantiate the class. This means that each non-static field is either set to its default value or it is in some way transformed using the parameters of the Constructor. To handle Constructors, Obsidian generates an assertion for each non-static field that verifies that its expected value, held in a local Expected Value Field, is equal to the actual value discovered through Reflection. The expected value will either be the field's default value or some transformation of one or more of the parameters. Obsidian generates a test-class field called a Default Value Field for each non-static field in the class. This allows Obsidian to automatically set a field's expected value to its Default Value Field and to write an assertion comparing the two. Therefore, once the test engineer has specified all of the default values, the assertions for fields not transformed using the parameters are complete. Note that Default Value Fields are initially set using the construction strings from Algorithm 1, as described in Section 4.3, and often will not need editing. For fields that are transformed using the parameters, the test engineer must define the Expected Value Field for that field in terms of the test case input parameter.

Testing Getters and Setters is much simpler. For Getters, defined as non-static methods with no parameters that return a value, Obsidian generates an assertion that the field being accessed is equal to its default value or is defined in terms of the parameters of one or more of the constructors. If the field is not transformed using the parameters of the Constructors, the test engineer only needs to set an Expected Value Field equal to the Default Value Field of the field being accessed. If the field being accessed is transformed using the parameters of one of the Constructors, the test engineer must then write the Expected Value Field in terms of the input parameters to said Constructor. Obsidian presently cannot tell which fields are affected by Constructors and therefore the inputs previously mentioned as well as the call to the Constructor would have to be written by hand. Future work, in Section 7, includes plans to add this functionality using static analysis.

For Setters, defined as non-static methods that have one parameter and do not return a value, Obsidian generates an assertion that the value of the field being modified is equal to in the input parameter to the Constructor. Obsidian supplies an Actual Result Field that the test engineer should set to the value of the field being modified by either calling a previously tested Getter or through Reflection.

For the Assertions Block of all other public methods, Obsidian provides the test engineer only guidance.

4. Strategies for Generation

We seek to automate as much of unit test generation as

```
comment: Java generic type $T$;
Data: Dictionary of Primitive constructionStrings $D$
Input: Class< $T$ > type, Set of Visited Constructors
       $VC$
Result: constructionString
1  $constructionString$ ←“ ” comment: empty string;
2  if type is an abstract class or an interface then
3      $type \leftarrow getConcreteSubstitution(type)$;
4  if type is a primitive then
5      $constructionString$ ←lookup constructionString for
       type in $D$;
6  else if type is Array then
7      $constructionString$ ←“new type[0]...(number of
       dimensions)”;
8  else if type is an abstract class or an interface then
9      comment: no concrete substitution found above;
10     $constructionString$ ←“null”;
11 else if type has no constructors then
12     $constructionString$ ←“null”;
13 else
14     comment: the general case;
15     $NVC \longleftarrow NotVisitedConstructors(type, VC)$;
16     if $NVC = \emptyset$;
17     then
18         $constructionString$ ←“null”;
19     else
20         $constructor \leftarrow NVC$’s simplest constructor;
21         $VC \longleftarrow VC \cup constructor$;
22         $parameters$ ⟵constructor’s parameters;
23         $constructionString$ ←“new type(”;
24         for $p \in parameters$ do
25             $constructionString \leftarrow constructionString +$
               $buildConstructionStringFromType($
               $typeOf$(p), $VC$) + “,”;
26         end
27         comment: strip last comma first;
28         $constructionString \leftarrow constructionString$+“)”;
29     end
30 end
31 comment: prevents incomplete construction strings;
32 if “$null''$ ∈ constructionString then
33     $constructionString$ ← “null”;
34 return constructionString;
```

Algorithm 1. Build Construction String From Type.

possible and to organize any remaining tasks in an efficient manner. To do this, we evaluated each stage of the unit test generation process to find opportunities for refinement and automation. We developed and use several guiding strategies that inform our approach, each of which is made possible by the Java Reflection API. These strategies are:

- method test invocation ordering,
- default object creation,
- object equality by indirection,
- method accountability,
- field accountability

We first present our use of Reflection and then explain these guiding strategies in the following sections.

4.1. Java Reflection API

Java provides an opportunity to developers with its Reflection API. Reflection allows a program to dynamically look inside itself or another program and get information about internal properties and state of the program. Reflection occurs during execution and is therefore dependent on the Java Virtual Machine (JVM). Given a class object, reflection can return information about all of the class’ methods, constructors, and fields. For example, Reflection can return a method’s signature including its visibility, its parameter types, its return type, and what kind of exceptions it might throw.

When Obsidian traverses a class hierarchy to generate test implementations, it uses Reflection to collect and organize information about methods, fields, types, and exceptions. With Reflection, Obsidian can generate unit tests from class interfaces only, allowing a test engineer to easily write unit tests for a component that supplies only compiled class files. This feature therefore supports test-driven development [2]. When they are available, Reflection also gives access to private methods and fields. In future versions, Obsidian will perform static analysis to generate additional refinements, based on these private methods and fields, to the test implementation code (see Section 7). Each of the five strategies described below depend on the information collected by Reflection.

In addition to supporting code analysis, Reflection can be used directly inside test implementations to retrieve the values of an object’s fields during the execution of the program. This direct access to the values of fields without having to call accessor methods helps to reduce associations among method tests and therefore strengthens the method tests.

4.2. Method Test Invocation Ordering

As presented in the Introduction, Obsidian uses a specific execution ordering of method signature types to drive its production of test implementations: 1) Constructors, 2) Getters, 3) Setters, 4) public methods, and 5) private methods (future work).

This ordering uses the bottom-up incremental approach prescribed by Myers [6] so that each method’s test can depend on previously tested methods to change or access class or object attributes. Obsidian uses the information it collected during Reflection to classify all the methods of a class as members of this order and to then generate the method test implementations in the specified order. As described in the typical Obsidian workflow presented in Section 5, the test engineer is encouraged to build confidence in the class by incrementally writing test cases using the method tests in this ordering.

4.3. Default Object Creation

The generation of unit tests requires the instantiation of objects for a variety of purposes, as for example, creating an instance of the class under test or creating instances of a method's parameters. Obsidian uses the information it collects through Reflection to build meaningful construction strings for instantiating objects.

This approach to building construction strings is summarized in the recursive **Algorithm 1**, called build Construction String From Type. A construction string is the string that when written into the test code will, after compilation and execution, produce a new instance of the needed object.

The algorithm uses a dictionary of default construction strings for primitive types (Data) and takes as input the object type and a working set of visited Constructors, VC. At lines 2 - 3 the algorithm tests whether type has an implementation, and if not, tries to find a suitable substitution using the method get Concrete Substitution, which is not presented herein.

At lines 4 - 5, the algorithm determines if type is a primitive and, if so, finds it in the dictionary and returns a default string such as "0" for type int. At lines 6 - 7, if type is an Array, the construction string becomes "new type[0]..." with the appropriate dimension. At lines 8 - 10, the algorithm double-checks to determine if a concrete substitution was found at line 3. If not, the algorithm sets the construction string to "null". At lines 11 - 12 the algorithm checks for constructors using Reflection and, if none exist, sets the construction string to "null".

Finally, at line 14, the algorithm proceeds to the general case. At line 15 it retrieves a set of Constructors, Not Visited Constructors (NVC), for type excluding any already visited as tracked by VC. If there are no remaining un-visited Constructors for type, it sets the construction string to "null" at line 18. Otherwise, at line 20, the algorithm chooses the Constructor from this set, NVC, with the fewest parameters and with parameter-count ties settled by the order of implementation. Then, working recursively, the algorithm walks the parameter list of the selected Constructor to compose construction strings for each parameter at lines 24 - 26. The construction string is finalized at line 28. At lines 32 - 33, the algorithm rejects the construction string if it contains any "null" elements by setting it to "null". The construction string is returned at line 34.

4.4. Object Equality by Indirection

At the heart of unit testing is the use of assertions to compare two objects where one object is the outcome of a method call and the other is the expected value. JUnit has more than 25 flavors of assertion that the test engineer must choose from and then configure for every use in any test code. These assertions rely on the equals() method of a class, which defaults to the equals() method of the Object class in the absence of a class-specific definition. The Object class comparison is equivalent to reference equality as expressed by the "==" operator. Obsidian explicitly simplifies and automates the use of assertions through indirection.

Obsidian provides two binary and two unary assertion methods that are used in lieu of any of the 25 possible JUnit assertions. The binary assertion methods are Obsidian's equality methods that are are named areEqual and areNotEqual. The unary assertion methods are Obsidian's null test methods that are named isNull and isNotNull. All four methods behave as JUnit assertions in that their return type is void and they use the JUnit Assert.fail() method to throw exceptions.

Obsidian creates a new class at the global level of the test-class hierarchy named Global Equality Methods and includes in it an areEqual equality method for each of the types found by Reflection in the class hierarchy. Every one of these areEqual methods uses the JUnit Assert.fail() method by default and thus the task for the test engineer is to proactively replace the Assert.fail() call with a definition of equality for the type.

Obsidian provides pass-through implementations of the equality methods in a special class at each package called Package Equality Methods that each call the method in Global Equality Methods with the same signature. Similarly, within each test class another pass-through implementation calls the appropriate method in Package Equality Methods. This chaining provides a way for the test engineer to inject customized definitions at any level of the class hierarchy. Only those objects instantiated within the appropriate scope have associated equality methods at any given level of the hierarchy. Thus, the only required task here for the test engineer is to insert, in each Global Equality Method, a definition of equality for the specific type. A call within a method's unit test implementation to areEqual (obj1, obj2) where obj1 and obj2 are instances of some Example Type will use indirection and method overloading to invoke the correct equality method at the global level, unless an intervening package level or class level definition has been supplied.

In the case of primitive types, Obsidian provides a complete implementation in Global Equality Methods of the equality methods that do not fail by default, as object equality for primitives is well-defined.

4.5. Method Accountability

Obsidian provides support for continuous test evolution and regression testing through the use of Method Accountability. Method Accountability ensures that all current methods in a class have been considered for unit test generation. The JUnit framework is only concerned with failed assertions, so it only reports whether or not each test passes. If no test exists for a method in a class, there is no warning and the test engineer may erroneously believe that the class has been thoroughly tested.

Obsidian solves this problem by modifying the JUnit-specified test method setUpClass to instantiate a special class called Method Map. Method Map provides operations for managing a class' methods based on the information collected during Reflection. Each method is labeled as hasUnitTest, hasNoUnitTest, or ignored. When the test suite is invoked, Obsidian uses reflection to detect all the current methods in the class and initially labels each as hasNoUnitTest. The engineer can choose to ignore methods by invoking the Method Map's setIgnored() method with the method's signature as the parameter. Currently, Obsidian considers all private methods to be ignored by default. As the test code is executed, a call to the Method Map object sets the method under test as hasUnitTest. Once all unit tests in a test class have executed, a method accountability audit checks for any methods still labeled hasNoUnitTest. This audit is implemented as a special JUnit method annotated with @Test so that if any methods without unit tests exist, Obsidian forces a failure for the test class and produces a report to the test engineer with details of the missing test implementations.

4.6. Field Accountability

Obsidian also provides class field accountability similarly to how it does for methods. If, since the last invocation of Obsidian, a new field has been introduced, then Obsidian will force an error and produce a report alerting the test engineer that a new field has been added to the code and that Obsidian should be executed again.

5. Workflow

We have defined a workflow for completing an Obsidian test class, to help test engineers understand Obsidian. The workflow, outlined below, consists of a sequence of tasks, each of which exists as a more precise directive written at the appropriate place in each test implementation with local detail to guide the test engineer. When appropriate, the test engineer will need to introduce specific meaningful test case data into the test implementations at the location specified by Obsidian. The test engineer builds confidence in the class by gradually accomplishing this sequence of tasks and repeatedly running the appropriate tests. We recommend using the Netbeans or Eclipse IDE as they both support the JUnit framework and Obsidian produces a compatible test package containing the test class hierarchy.

The Obsidian Workflow is defined as follows:

1) Equality Methods
2) Ignoring Methods
3) Ensure Default instance is instantiated in setup
4) Default Value Fields
5) For each method:
   a) Expected Exception Test Cases (if needed)
   b) Normal Test Cases (if needed)
   c) Write Assertions as needed for: Constructors, Getters, Setters, and public Methods

First, the test engineer defines any Equality Methods needed in the test class. It is recommended that this be done on the Global or Package level. Obsidian provides a comprehensive list of the needed Equality Methods at the end of each test class.

Second, Method Accountability ensures that all methods have been considered for testing. As tests are executed, methods are set to "tested". At the end of the test class if "untested" methods are found, Method Accountability fails, and the user is notified. If a method is meant to be untested, it can be ignored by calling the setIgnored method. To do this, in the JUnit @BeforeClass method, a call is inserted to method Map.setIgnored() with the method name as the argument. By default, all private methods are set to ignore because Obsidian does not currently generate tests for private methods.

Third, the test engineer must check that the default instance of the class under test is initialized in the JUnit @Setup method. This is the instance of the class that Obsidian uses to invoke all non-static methods. Obsidian tries to create this instance by looking for a default constructor. If no default constructor is found, the instance is not created. If the test engineer finds that no default constructor was discovered, then they must identify a reasonable substitute to be used throughout the test class.

Fourth, the test engineer gives meaningful values to the Default Value Fields. For each non-static field in the class, Obsidian has created a test-class level Default Value Field. A comment preceding each Default Value Field supplies the respective field name. The test engineer then sets the Default Value Field appropriately. These values may be used while testing constructors, getters, or any other method where they are of use. Because **Algorithm 1** supplies default values for primitive types and uses the simplest Constructors found for a class, often the Default Value Fields are correctly initialized during test class generation.

Finally, for each method, the test engineer must define the test cases. Each test case is comprised of the test case input and outcome data plus assertions. Obsidian provides specialized arrays to accept the input and outcome data and also specifies the location of the related assertions. The test engineer, when instructed, provides these data for the Expected Exception Blocks and then for the Normal Test Cases. Assertions will be customized to the constraints of the different classifications of methods being tested, as described below.

For Constructors, Obsidian generates an assertion verifying that, for each non-static field in the class, an Obsidian-generated Expected Result Field is equal to that field's value after invocation of the Constructor. As explained above, Obsidian sets these Expected Result Fields to the field's Default Value Field. The test engineer must modify these Expected Result Fields to reflect their relationship to the inputs. For example, if a field is transformed using one of the inputs, the test engineer would set this field's Expected Result Field equal to that transformation of the input.

For Getters, the test engineer sets the generated Expected Result Field to the accessed field's Default Value Field. If the field accessed is transformed using the parameters of one or more of the Constructors, the test engineer may choose to substitute one of these Constructors.

For Setters, the test engineer accesses the modified field after the invocation of the setter either by Reflection or by calling a previously-tested Getter. An Actual Result Field is generated by Obsidian for holding this value, as well as an assertion comparing the inputs to the Actual Result Field.

For all remaining public method tests Obsidian provides assertion guidance after the method invocation. This assists the test engineer in writing appropriate assertions about the state of the instance or static fields of the class using the supplied input data and the gathered outcomes of method calls.

6. Background

Research into several automation techniques, such as random testing, has led to the development of a number of automated testing tools. Of these tools, some notable academic members are EvoSuite [7], Randoop [8], Pex [9], Jartege [10], CUTE [11], Jcrasher [12] and the commercial members include Agitator [13], and Jtest [14]. The focus of this research and these tools has been concentrated on the generation of test cases with little attention given to engineering the test implementations that support these test cases.

We understand that many of these tools are purely academic and intended only to show the application of their guiding principles. However, the test implementations produced by many of these tools demonstrate the need for robust, managed, comprehensive test implementations like those generated by Obsidian. For example, when Agitator generates tests for a method that throws exceptions, a new method test implementation is generated for each type of exception. Each of these tests has a single test case hard-coded into the implementation. This is equivalent to a single expected exception test case without the ability of easily extending to additional test cases.

An example of the conflation of test implementation with test case that inspires our approach is illustrated by Randoop. Randoop uses feedback-directed random testing to find test cases that reveal faults. The nature of this technique requires that an amount of time be specified for Randoop to run, otherwise it could run forever. If no fault-revealing test cases are found before Randoop reaches its time limit then it generates empty test implementations for the methods under test. Thus, if the test engineer wanted to specify one or more test cases for these methods, they would have to build the implementations from scratch.

These and similar issues are part of the landscape of test automation and state-of-the-art automation tools. Humans still play a large role in the completion and verification of test suites as described by Pachecho and Ernst [15]. Obsidian tries to make test suites, including generated test cases, more maintainable through standardized test implementation code structure.

7. Conclusions and Future Work

We presented Obsidian as a new testing tool for Java programs that automatically generates comprehensive unit test implementations built on the JUnit 4 framework and guided by a set of four context patterns. These context patterns specify how to prepare the method test implementation so as to guarantee compilation, support exception handling, enable multiple test cases when required, and to specify where assertions are needed about the test case outcome(s). We also presented a set of five strategies for generating these unit test implementations: test execution ordering, default object creation, object equality by indirection, method accountability, and field accountability.

Obsidian makes a novel contribution by providing a structured way to separate the generation of test implementations from the generation of test cases. We have identified additional areas for research and improvement that will make Obsidian more robust and easier to integrate into a test engineer's workflow.

First, Obsidian's generated test implementations are based solely on information gathered about method signatures through dynamic analysis. We will research to add static analysis techniques to Obsidian's processes. For example, a powerful addition would have Obsidian statically determine associations between a method's parameters and the fields of a class to inform the details of the test implementation and automation of assertion generation. Thus, in Getter test implementations where the field being accessed by the Getter is directly transformed using the parameters of one or more of the constructors, we seek to have Obsidian build a test implementation that calls the associated constructor with provided test input data.

Second, we will develop front-end tools that support the test engineer in maintaining the testing code base by providing seamless assistance in inserting test case data and specifying the associated assertions.

Third, we intend to research how to make Obsidian compliant with existing test case generators. One approach is an API which allows these generators to add their test case data and assertions to Obsidian implementations programmatically. Another is to define a standardized XML format that test case generators could produce and Obsidian could read. Furthermore, we intend to customize how assertion failures are handled to support reporting of the pass/fail status of all test cases regardless of any intervening assertion failures. This change will make Obsidian compliant with the requirements of fault analysis and visualization tools such as Tarantula [16].

Fourth, due to the nature of Obsidian's pattern-based generation technique many Obsidian unit test implementations are very similar to one another. This standardization is good for maintainability but reveals that another layer of abstraction may be possible. We plan to research how to design and produce a set of generic and universal test implementations.

Finally, we are continuing to develop an open source community with a web presence for Obsidian with a portal at www.ObsidianTest.org. Readers interested in obtaining the code can clone or fork it from https://github.com/johnnyLadders/obsidian.

Acknowledgements

Funding was provided by grant EAR-0930223 from the National Science Foundation. We thank the College of Charleston Department of Computer Science for providing support for CIRDLES.org.

REFERENCES

[1] K. Beck, "Extreme Programming Explained: Embrace Change," Addison-Wesley, Reading, 2000.

[2] K. Beck, "Test-Driven Development: By Example," Addison-Wesley, Boston, 2003.

[3] B. Robinson, M. D. Ernst, J. H. Perkins, V. Augustine and N. Li, "Scaling up Automated Test Generation: Automatically Generating Maintainable Regression Unit Tests for Programs," *Proceedings of the* 2011 26*th IEEE/ACM International Conference on Automated Software Engineering*, Washington, 2011, pp. 23-32. http://dx.doi.org/10.1109/ASE.2011.6100059

[4] G. McCluskey, "Using Java Reflection," 1998. http://java.sun.com/developer/ALT/Reflection/

[5] D. S. Rosenblum, "Towards a Method of Programming with Assertions," In: *Proceedings of the* 14*th International Conference on Software Engineering*, ACM, New York, 1992, pp. 92-104. http://doi.acm.org/10.1145/143062.143098

[6] G. J. Myers, "The Art of Software Testing," Wiley, New York, 1979.

[7] G. Fraser and A. Arcuri, "Evosuite: Automatic Test Suite-generation for Object-Oriented Software," *ACM Symposium on the Foundations of Software Engineering* (*FSE*), 2011, pp. 416-419.

[8] C. Pacheco and M. Ernst, "Randoop: Feedback-Directed Random Testing for Java," *Conference on Object Oriented Programming Systems Languages and Applications*: *Companion to the* 22*nd ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications Companion*, Vol. 21, 2007, pp. 815-816.

[9] N. Tillmann and J. De Halleux, "Pex-White Box Test Generation for Net," *Tests and Proofs*, Vol. 4966, 2008, pp. 134-153.

[10] C. Oriat, "Jartege: A Tool for Random Generation of Unit Tests for Java Classes," *Quality of Software Architectures and Software Quality*, Vol. 3712, 2005, pp. 242-256. http://dx.doi.org/10.1007/11558569_18

[11] K. Sen and G. Agha, "Cute and Jcute: Concolic Unit Testing and Explicit Path Model-Checking Tools," *Computer Aided Verification*, Vol. 4144, 2006, pp. 419-423. http://dx.doi.org/10.1007/11817963_38

[12] C. Csallner and Y. Smaragdakis, "Jcrasher: An Automatic Robustness Tester for Java," *Software*: *Practice and Experience*, Vol. 34, No. 11, 2004, pp. 1025-1050. http://dx.doi.org/10.1002/spe.602

[13] M. Boshernitsan and R. Doong, A. Savoia, "From Daikon to Agitator: Lessons and Challenges in Building a Commercial Tool for Developer Testing," In: *Proceedings of ISSTA* 2006, ACM, New York, 2006, pp. 169-180.

[14] Parasoft, Jtest, "Java Static Analysis, Code Review, Unit Testing, Runtime Error Detection." http://www.parasoft.com/jsp/products/jtest.jsp

[15] C. Pacheco and M. Ernst, "Eclat: AutomatIC Generation and Classification of Test Inputs," ECOOP Object Oriented Programming, 2005, pp. 734-734.

[16] J. Jones and M. J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique," *Proceedings of the* 20*th IEEE/ACM International Conference on Automated Software Engineering*, 2005, pp. 273-282.