UnitPlus: Assisting Developer Testing in Eclipse

Yoonki Song[1] Suresh Thummalapenta[2] Tao Xie[3]
Department of Computer Science
North Carolina State University, Raleigh, USA
{[1]ysong2, [2]sthumma}@ncsu.edu, [3]xie@csc.ncsu.edu

ABSTRACT

In the software development life cycle, unit testing is an important phase that helps in early detection of bugs. A unit test case consists of two parts: a test input, which is often a sequence of method calls, and a test oracle, which is often in the form of assertions. The effectiveness of a unit test case depends on its test input as well as its test oracle because the test oracle helps in exposing bugs during the execution of the test input. The task of writing effective test oracles is not trivial as this task requires domain or application knowledge and also needs knowledge of the intricate details of the class under test. In addition, when developers write new unit test cases, much test code (including code in test inputs or oracles) such as method argument values is the same as some previously written test code. To assist developers in writing test code in unit test cases more efficiently, we have developed an Eclipse plugin for JUnit test cases, called UnitPlus, that runs in the background and recommends test-code pieces for developers to choose (and revise when needed) to put in test oracles or test inputs. The recommendation is based on static analysis of the class under test and already written unit test cases. We have conducted a feasibility study for our UnitPlus plugin with four Java libraries to demonstrate its potential utility.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging—*Testing tools*;

General Terms: Languages, Reliability, Experimentation.

Keywords: Developer testing, Test code reuse

1. INTRODUCTION

In the software development life cycle, unit testing is an important phase that helps early detection of bugs and ensures the overall quality of the developed software. In general, a unit test case consists of two parts: a test input and a test oracle. A test input often includes methods that affect fields of the class under test and a test oracle often verifies the affected fields through the class' methods whose return type is non-void. Test inputs can either be written manually or generated automatically. Although existing automated approaches are effective in automatically generating test inputs, they often suffer from the problem of insufficient test oracles [2], especially in the absence of specification.

Rompaey et al. [4] propose a metric-based heuristical approach for ranking test cases to identify poorly designed test cases. Their approach suggests to refactor those test cases that violate unit test criteria. Their approach helps to minimize the maintenance cost for test code but may not be effective in increasing the effectiveness of the existing test cases. Orstra developed by Xie [6] tries to increase the effectiveness of the existing test cases by augmenting an automatically generated unit-test suite with regression oracle checking. But as their approach is non-interactive, it is not possible for developers to incorporate their domain knowledge while generating test oracles.

In many situations, developers still manually write test code for test inputs, because the developers' domain or application knowledge can be incorporated there. Even for test oracles, developers still need to manually write test code in assertions, e.g., invocations of methods whose return types are non-void. On the other hand, writing effective test oracles manually is often not a trivial task as developers need to refer to the intricate details of the class under test. Often manually written test code in test inputs or oracles is the same as or similar to some previously written test code. It is tedious for developers to repeatedly type in these pieces of the same or similar test code.

To assist developers in writing test code more efficiently, we have developed an Eclipse plugin, called UnitPlus, that runs in the background and recommends test-code pieces for developers to choose (and revise when needed) to put in test oracles or test inputs. In particular, UnitPlus accepts a class and related existing test suites (including the test suite that developers are working on) as inputs. UnitPlus identifies all public methods of the given class and classifies them into two categories: state-modifying methods and observer methods. A method is classified as a state-modifying method, if the method affects (i.e., writes) the value of at least one field (or its transitively reachable field) of the given class. A method is classified as an observer, if the method's return type is non-void. Sometimes, a method can be both a state-modifying and observer method, if the method affects some field of the given class and its return type is not `void`.

UnitPlus parses the existing test suites and collects method sequences used for producing non-primitive method arguments and values used for primitive method arguments. The

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

```
01:public class Person {
02:  private String fName;
03:  private int fAge;
04:  private Address faddr;
05:  public Person() {
06:    fName = ""; fAge = 0; }
07:  public Person(String name, int age) {
08:    fName = name; fAge = age; }
09:  public String getName() { return fName; }
10:  public void setName(String name) { fName = name; }
11:  public int getAge() { return fAge; }
12:  public void setAge(int age) { fAge = age; }
13:  public void getAddress() { return faddr; }
14:  public Address setAddress(Address addr) {
15:    faddr = addr; }
16:}
17:public class Management {
18:  private Person[] fPeople;
19:  private int fCount;
20:  public Management() {
21:    fPeople = new Person[10]; fCount = 0; }
22:  public void add(Person p) { fPeople[fCount++] = p; }
23:  public int howmany() { return fCount; }
24:  public boolean isEmpty() {
25:    return fCount == 0 ? true : false; }
26:  public boolean isFull() {
27:    return fCount == 10 ? true : false; }
28:  public boolean exists(Person p) {
29:    for (int i = 0; i < fCount; i++)
30:      if (fPeople[i].getName().equals(p.getName()) &&
31:        fPeople[i].getAge() == p.getAge())
32:        return true;
33:    return false; }
34:}
```

Figure 1: Example classes `Person` and `Management`.

parsed information, referred as *TestCodeDB*, is loaded by UnitPlus. Whenever developers add or change test cases in the test suite, UnitPlus updates the *TestCodeDB* on the fly to reflect the changed information. The *TestCodeDB* is used to recommend test code for producing method arguments in test inputs or oracles. The recommended values for method arguments can be either method sequences or primitive values based on the type of the argument.

UnitPlus runs in the background when developers write new test cases or modify existing test cases. After writing test code (e.g., a state-modifying method) in the test input of the test case, the developers can request UnitPlus to recommend *relevant* observer methods as *relevant* test code to choose (and revise when needed) to put in test oracles. UnitPlus identifies an observer method to be *relevant* for a state-modifying method if the intersection between the affected-field set of a state-modifying method and the accessed-field set[1] of the observer method is not empty. The rationale is that the side effects (i.e., affected fields) of the state-modifying method on the receiver object state need to be observed and asserted (by the relevant observer methods) to make sure these side effects are expected.

Sometimes, a state-modifying method in the test input or a recommended observer method in the test oracle may need method argument values. UnitPlus recommends test code of *relevant* method sequences or values (from *TestCodeDB*) for providing needed method arguments in the test input or oracle. UnitPlus identifies a method sequence or value to be *relevant* for a method argument if the object produced by the method sequence or the value is of the same type as the method argument. The rationale is that the same or similar test code to be written by developers may have already been written by the developers in the past.

[1]The accessed-field set includes read fields (as well as written fields if the observer method is a state-modifying method).

```
01: public class ManagementTest extends TestCase {
02:   public void testAdd() throws Exception {
03:     Management mgmt = new Management();
04:     mgmt.add(new Person("Jane Doe", 20));
05:     assertEquals(1, mgmt.howmany());
06:   }
07: }
```

Figure 2: Sample JUnit test suite for `Management`

```
01:public class ManagementTest extends TestCase {
02:  public void testAdd() throws Exception {
03:  Management mgmt = new Management();
04:  mgmt.add(new Person("Jane Doe", 20));
05:  assertEquals(1, mgmt.howmany());
06:  assertEquals(false, mgmt.isEmpty());
07:  assertEquals(false, mgmt.isFull());
08:  Person person1 = new Person();
09:  person1.setName("Jane Doe");
10:  person1.setAge(20);
11:  Address addr = new Address();
12:  addr.setCity("A");
13:  addr.setZipcode("12345");
14:  person1.setAddr(addr);
15:  assertEquals(true, mgmt.exists(person1));
16:}
```

Figure 3: Sample JUnit test suite augmented with recommended test code in the test oracle

2. EXAMPLE

We next explain our UnitPlus approach through an example shown in Figure 1 and illustrate how UnitPlus can help developers in writing unit test cases efficiently. The sample code Figure 1 shows two classes `Person` and `Management` along with their fields and methods. The `Person` class contains two fields `fName` and `fAge` and a few state-modifying and observer methods. For example, methods `setName` and `setAge` are state-modifying methods as they affect values of fields `fName` and `fAge`, respectively, and methods `getName` and `getAge` are observer methods as their return types are not `void`.

Figure 2 shows a sample test suite, either written manually or generated automatically, for the `Management` class in the form of JUnit. In general, each JUnit test case consists of two kinds of statements: non-assertion and assertion statements. The non-assertion statements form the test input and the assertion statements form the test oracle. For example, in the `testAdd` test case shown in Figure 2, Lines 3 and 4 contain non-assertion statements followed by Line 5, which contains an assertion statement for verifying values of some affected field (`fCount`) through verifying the return of the observer method (`howmany`).

In the given test case, verifying the number of persons (reflected by the `fCount` field) after adding a `Person` object may not be sufficient to check the entire functionality provided by the `add` method. The test case can be made more effective by adding new test oracles. To recommend test code in augmenting the test oracle, UnitPlus initially classifies methods of the given classes into state-modifying and observer methods. UnitPlus also identifies the affected fields and the accessed fields for each state-modifying and observer methods, respectively. Tables 1 and 2 show the set of state-modifying and observer methods along with their affected and accessed

Method Name	Affected Fields
`Management.add(Person)`	`fCount, fPeople`
`Management.add(String, int)`	`fCount, fPeople`
`Management.CONSTRUCTOR()`	`fCount, fPeople`
`Person.CONSTRUCTOR()`	`fAge, fName`
`Person.CONSTRUCTOR(String, int)`	`fAge, fName`
`Person.setAge(int)`	`fAge`
`Person.setName(String)`	`fName`

Table 1: Set of state-modifying methods of the `Management` and `Person` classes.

Method Name	Accessed Fields
`Management.exists(Person)`	`fCount, fPeople`
`Management.howmany()`	`fCount`
`Management.isEmpty()`	`fCount`
`Management.isFull()`	`fCount`
`Person.getAge()`	`fAge`
`Person.getName()`	`fName`

Table 2: Set of observer methods of `Management` and `Person` classes.

fields, respectively. Column "Method Name" gives the signature of the method and Columns "Affected Fields" and "Accessed Fields" give the set of affected fields and accessed fields, respectively, by the corresponding method.

UnitPlus uses the information shown in Tables 1 and 2 to compute a relation between the observer and state-modifying methods. The relation is computed by calculating the intersection between the accessed-field set of the observer method and the affected-field set of the state-modifying method. The computed relation describes which observer methods are associated with a given state-modifying method. For example, UnitPlus identifies that the `Management.add` method, which affects fields `fPeople` and `fCount`, is associated with the observer methods `howMany`, `isEmpty`, `isFull`, and `exists`. Whenever UnitPlus identifies the `Management.add` method in a test case, UnitPlus recommends the associated observer methods as test code in augmenting the test oracle. Figure 3 shows the sample test suite with the augmented test oracle including the recommended test code. The augmented test oracle can verify more behaviors of the the `Management.add` method and thereby can provide better fault-detection capability.

Sometimes, the recommended observer methods may need method arguments including non-primitive arguments. To reduce developers' effort, UnitPlus learns from existing test cases and recommends method sequences or primitive values for producing the required argument type. For example, consider the test oracle shown in Line 15 of Figure 3. The observer method `exists` requires an argument of the `Person` class type. UnitPlus learns the method sequence that produces an object of `Person` class from existing test cases and recommends the method sequence to the developers. In the current example, the recommended method sequence for producing the object of the `Person` class is shown between Lines 8 and 14. The suggested method sequence includes method calls for creating an object of the `Address` class as the method `setAddr` of the `Person` class has a non-primitive argument of the type `Address`.

3. APPROACH

Our UnitPlus approach consists of three major components: the side-effect analyzer, observer-method recommender

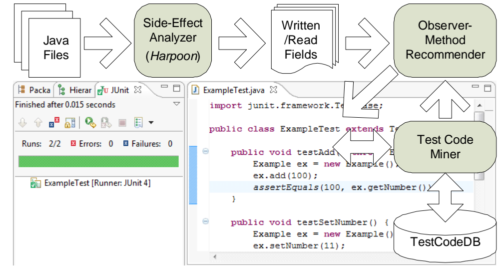

Figure 4: Overview of our approach

(OMR), and test code miner (TCM). Figure 4 shows an overview of the major components of our approach.

The side-effect analyzer accepts the given class as input and classifies the public methods of the given class into two categories: state-modifying and observer methods. A method is classified as state-modifying if the method affects at least one field (or its transitively reachable field) of the given class. A method is classified as an observer method if the return type of the method is non-void. Along with the classification of the methods, the side-effect analyzer also identifies the affected and accessed fields for each state-modifying and observer methods, respectively. In our current implementation, we used Harpoon [3] as a side-effect analyzer.

The OMR component assists developers by recommending test code in augmenting the test oracle in the form of assert statements. This component accepts the sets of state-modifying and observer methods as input and computes relationships among state-modifying and observer methods. An observer method is identified as *relevant* to a state-modifying method if the intersection between the accessed-field set of the observer method and the affected-field set of the state-modifying method is not empty. For example, consider that an observer method, say OM_1, accesses fields F_1 and F_2 of the class $ExClass$, and the state-modifying method, say SMM_1, affects fields F_2 and F_3. The OMR component calculates the intersection between sets $\{F_1, F_2\}$ of OM_1 and $\{F_2, F_3\}$ of SMM_1. As the intersection results in set $\{F_2\}$, which is not empty, the OMR component identifies that OM_1 is relevant to SMM_1. The component uses the computed relationships while recommending test code in augmenting the test oracle. For example, if developers add a state-modifying method to the test case, the OMR component identifies the relevant observer methods and recommends them to the developers as test code in augmenting the test oracle.

The TCM component helps developers in writing test code by suggesting method sequences or argument values for the recommended observer methods. Initially, TCM gathers method sequences or primitive values used by the existing test cases along with the locations where these method sequences or primitive values are written. The gathered information is stored in memory, and is referred as `TestCodeDB`. When OMR requests TCM for the values of a method argument, TCM checks for the available method sequences or primitive values, and recommends them as a list ranked by the distance from the working location to their respective locations, with a higher preference to a nearer one. The rationale behind this nearness heuristic is that the developers often tend to reuse the nearest available method sequences or primitive values among all available method sequences

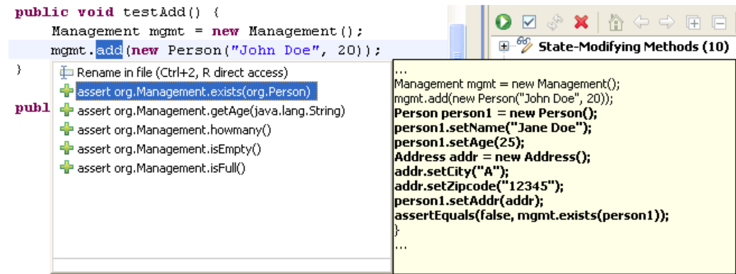

Figure 5: Screen snapshot of the UnitPlus Eclipse plugin showing the list of recommended observer methods

or primitive values. We plan to incorporate other ranking heuristics in future work.

In our current implementation, the OMR and TCM components use Eclipse JDT [1] for parsing Java source files. The whole UnitPlus plugin was built upon an existing Eclipse plugin, called moreUnit [5], which assists developers while they are writing unit test cases. moreUnit provides several basic functionalities such as switching between the test case and the class under test, test case creation, and refactoring. UnitPlus can be invoked from the Eclipse by selecting the desired state-modifying method and by pressing `Ctrl+1`. A snapshot of the UnitPlus plugin is shown in Figure 5. The snapshot shows a set of recommended observer methods for the state-modifying method `add`. The developers can choose any of the recommended observer methods and UnitPlus automatically adds the selected observer method to the test case.

4. FEASIBILITY STUDY

We next describe our feasibility study conducted with four different subjects. Our study results show that the existing test suites of these libraries can be augmented with our recommended test code in additional test oracles to make these test suites more effective.

4.1 Subjects

The four subjects used in our study are open source libraries with existing test suites. The `JSort`[2] library provides sorting algorithms. The `JBell`[3] is a Java library that enables developers to perform operations such as collection filtering and/or sorting. The `JAutomata`[4] is a library used for creating, manipulating, and displaying finite-state automata within the Java platform. `StringTree`[5] is a library for text transformation and processing package.

Table 3 shows the characteristics of all four subjects that are used in our study. In particular, the columns show the subject name, number of classes, number of test suites, and the total number of test cases in all test suites. Column "SMM" gives the number of state-modifying methods in each subject library. For each SMM, we present the total number, and the number of SMMs invoked and not invoked by the test code. Column "OM" gives the number of observer methods in each subject library.

4.2 Study Results

Figure 6 shows the study results of UnitPlus with all four subject libraries. The figure consists of a chart for each subject library used in our study. The x axis represents state modifying methods of the subject (labelled with ID numbers) and the y axis represents the number of observer methods associated with each state-modifying method. For each state-modifying method, we show both the number of recommended observer methods (black bar) and the number of observer methods that are actually invoked (white bar) in the test suites of each subject library. We observed that some libraries have a few state-modifying methods for which there are neither recommended observer methods nor invoked observer methods in the existing test suite. We ignored such state-modifying methods from the charts shown in Figure 6.

For subject libraries JAuto, JBel, and StringTree, our results show that the test oracles of the existing test suites can be further augmented, because in the existing test cases not all relevant observer methods are used to verify behavior for many state-modifying methods. For example, for the first state-modifying method (denoted by 1 in the x axis) of the JAuto library, the number of recommended observer methods is 17. However, none of these observer methods are verified after the state-modifying method in the existing test suite of the JAuto library. Our results indicate that for the JSort library, the developers invoke all recommended observer methods for each state-modifying method.

We also found some interesting results on the test suites of each library. In our study, we found that the existing test suites are not covering all state-modifying methods. These results are shown in the column "Not Invoked" of Table 3. We are currently inspecting these cases in details for explanations.

5. DISCUSSION

UnitPlus classifies methods of the class under test into state-modifying methods and observer methods. Based on our criteria used for classification, a method can be both a state-modifying and observer one. When this type of observer methods is used as test oracles, the recommended relevant observer methods for a state-modifying method can modify the state of the underlying object. Therefore, UnitPlus expects developers to decide whether to use a recommended observer method as there can be side-effects when the recommended observer method is also a state-modifying method.

UnitPlus currently needs developers to manually write expected return values for the recommended observer methods. In future work, we plan to automatically capture the actual return values of the recommended observer methods and then the developers need only confirm the captured return values. We expect that automatic capturing of the

[2]`http://sourceforge.net/projects/jsort`
[3]`http://sourceforge.net/projects/jbel`
[4]`http://sourceforge.net/projects/jautomata`
[5]`http://sourceforge.net/projects/stringtree`

Subject	Classes	Test Suites	Test Cases	SMM			OM
				Total	Invoked	Not Invoked	
`JSort`	11	7	24	6	6	0	3
`JBel`	55	26	80	46	41	5	30
`JAutomata`	84	17	48	93	36	57	96
`StringTree`	58	30	169	169	104	65	131

Table 3: Characteristics of the subject libraries used in the study

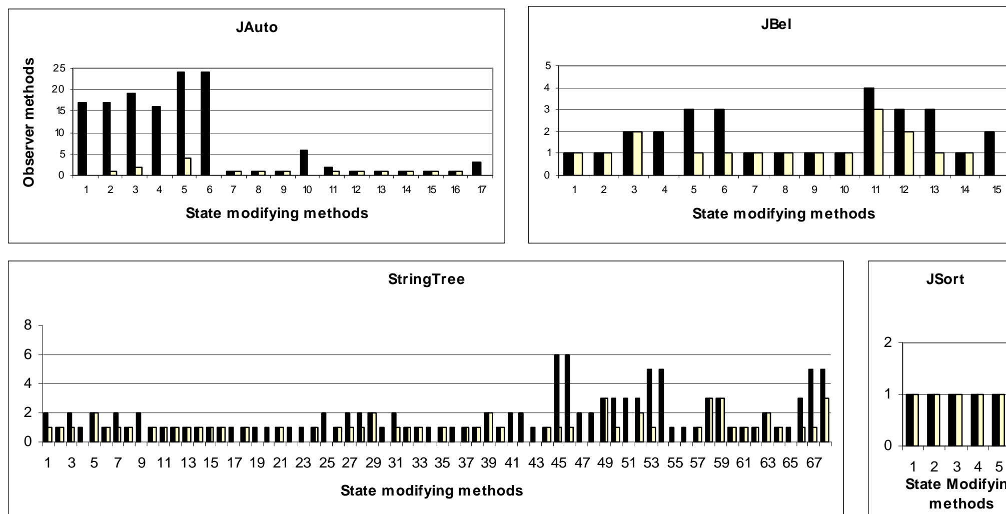

Figure 6: Study results

`Legend: Black:Recommended observer methods, White:Existing observer methods`

actual values can help to further reduce the efforts of developers while writing unit test cases.

So far we have conducted a feasibility study to show that UnitPlus can recommend additional test oracles that can be added to make the existing test cases more effective. Although the benefits of test oracle augmentation was demonstrated by an experiment conducted previously in assessing the Orstra approach [6], in future work, we plan to conduct a more comprehensive evaluation to validate whether the augmented test oracles indeed improve the fault-detection capability of those test cases.

In general, there can be more than one method sequence available in the existing test cases for instantiating a required non-primitive type. The current implementation sorts the available method sequences based on the distance from the working location to locations of method sequences to be recommended. In future work, we plan to investigate and include several other ranking heuristics for prioritizing those identified method sequences. One such ranking criterion is to sort the identified method sequences based on the frequency of each identified method sequence.

6. CONCLUSION

Manual test case creation in creating test inputs and oracles is a tedious process. We have developed an Eclipse plugin, called UnitPlus, that can assist developers in writing unit test cases more efficiently. UnitPlus runs in the background and recommends relevant test code in test oracles whenever the developers enter a test input in the test case. The recommended test code in test oracles can increase the effectiveness of the test case and thereby can help in finding more bugs. UnitPlus also tries to reduce developers' effort while they are writing test cases by automatically recommending method sequences or values that can instantiate a given method argument type. We conducted a feasibility study on UnitPlus with four different subjects and showed that existing test suites can be augmented with our recommended test code in additional test oracles to make these test suites more effective.

7. REFERENCES

[1] M. Aeschlimann, D. Baumer, and J. Lanneluc. Java tool smithing extending the Eclipse Java development tools. In *Proc. EclipseCon, Tutorial*, 2005.

[2] M. Amorim, C. Pacheco, T. Xie, D. Marinov, and M. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *Proc. 21st International Conference on Automated Software Engineering*, pages 59–68, 2006.

[3] S. Ananian. FLEX compiler infrastructure for Java, 2003. `http://cycleserv2.csail.mit.edu/Harpoon`.

[4] B. Rompaey, B. Bois, and S. Demeyer. Characterizing the relative significance of a test smell. In *Proc. 22nd International Conference on Software Maintenance*, pages 391–400, 2006.

[5] V. Wahler, C. Walton, P. Ombredanne, and C. Jones. moreUnit, 2007. `http://moreunit.sourceforge.net`.

[6] T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *Proc. 20th European Conference on Object-Oriented Programming*, pages 380–403, 2006.