Towards Generating Human-Oriented Summaries of Unit Test Cases

Manabu Kamimura
Software Innovation Laboratory
Fujitsu Laboratories Ltd
Kawasaki, Japan
kamimura.manabu@jp.fujitsu.com

Gail C. Murphy
Department of Computer Science
University of British Columbia
Vancouver, Canada
murphy@cs.ubc.ca

***Abstract*—The emergence of usable unit testing frameworks (e.g., JUnit for Java code) and unit test generators (e.g., CodePro for Java code) make it easier to create more comprehensive unit testing suites for applications. Unfortunately, test code, especially generated test code, can be difficult to comprehend. In this paper, we propose generating human-oriented summaries of test cases. We suggest an initial approach based on a static analysis of the source code of the test cases. Our goal is to help improve a human's ability to quickly comprehend unit test cases so that appropriate decisions can be made about where to place effort when dealing with large unit test suites.**

***Index Terms*—Test case comprehension, summarization**

I. Introduction

The emergence of usable unit testing frameworks, such as JUnit[1] for Java code, makes it easier to create comprehensive unit test suites for applications. For example, the JFreechart application[2] (v. 1.0.14) includes 2217 test methods, a ratio of approximately 3.5 test methods per class in the application.

More unit tests are typically seen as good; for instance, more unit tests should catch regressions caused by code changes earlier in development. But are more unit tests good in all dimensions? Imagine that you are a developer who joins a software development project with lots of unit test cases. When a code change causes many test cases to fail, which test case should you investigate first? What if you need to update the test cases as you change the code? How do you learn what all of the cases test? What if your test case suite includes generated tests from the growing number of test generation tools (e.g., CodePro[3] and JTest[4])?

If you are lucky, as much or more care was taken in writing or annotating the test cases as the code to which the tests apply: the names of the test cases are meaningful, the code within the test case is clean and straightforward and all variables are well-named. A simple scan of the test case code allows a developer to determine what the test case is doing. Unfortunately, such test case code is not that common. Figure 1 shows two examples of unit test cases from JFreechart. From the names of these test cases, a developer can determine that they are about zoom, but it is difficult to tell, without reading the contents of each test case in depth, how the test cases compare and whether the test cases consider more than zoom functionality.

```
public void test2502355_zoom() {
      DefaultXYDataset dataset = new DefaultXYDataset();
      JFreeChart chart = ChartFactory.createXYLineChart(
"TestChart", "X","Y", dataset, PlotOrientation.VERTICAL,
false, false, false);
      ChartPanel panel = new ChartPanel(chart);
      chart.addChangeListener(this);
      this.chartChangeEvents.clear();
      panel.zoom(new Rectangle2D.Double(1.0, 2.0, 3.0,
4.0));
      assertEquals(1, this.chartChangeEvents.size());
}

public void test2502355_zoomInBoth() {
     DefaultXYDataset dataset = new DefaultXYDataset();
     JFreeChart chart = ChartFactory.createXYLineChart(
"TestChart", "X","Y", dataset, PlotOrientation.VERTICAL,
false, false, false);
     ChartPanel panel = new ChartPanel(chart);
     chart.addChangeListener(this);
     this.chartChangeEvents.clear();
     panel.zoomInBoth(1.0, 2.0);
     assertEquals(1, this.chartChangeEvents.size());
}
```

Fig. 1. Some Unit Test Cases from JFreechart

Summary for test case: `test2502355_zoom`:

Calls zoom on a `panel` object using a `Rectangle2D.Double` object.
Checks the size of the `chartChangeEvents` object is equal to 1.

Summary for test case: test2502355_zoomInBoth:

Calls zoomInBoth (1.0, 2.0) on a `panel`
Checks the size of the `chartChangeEvents` object is equal to 1.

Fig. 2. Summaries for the Test Methods in Fig. 1

We hypothesize that a developer can benefit from a consumable and understandable textual summary of a test case. In this paper, we provide an initial step towards generating such summaries. Our focus is on identifying interesting facts

[1] http://www.junit.org
[2] http://www.jfree.org/jfreechart/
[3] https://developers.google.com/java-dev-tools/codepro/doc/
[4] http://www.parasoft.com/jsp/products/jtest.jsp

about the test cases, largely centering on how similar test cases are different from each other (Section III). Figure 2 shows the summaries we can currently generate for the two similar test cases shown in Figure 1. From the summaries, a developer can see that the end result of each test case is the same; the size of the object referred to by `chartChangeEvents` is 1. However, each method, as indicated by the name, operates primarily on testing a different zoom method: `zoom` on a panel object in the first test case versus `zoomInBoth` on a panel object in the second test case. In this particular case, one could infer part of this information from the method names, however, this information is not always so apparent; the names of the test methods do not always correspond to the most important methods in the test case. Our approach is able to pick out this pertinent information from an analysis of the method bodies of the test cases (Section IV).

II. Our Previous Work

Existing work on helping developers comprehend test cases has focused on the generation of graphical representations (e.g., [1]) (Section VI). To our knowledge, this work is the first to consider the generation of textual summaries for test cases. In earlier work at UBC, we have shown that textual summaries of crosscutting source code and bugs can help in source code change [2] and bug duplication tasks [3]. The work described in this paper introduces a new abstractive summarization approach summarize a previously unconsidered software artifact, test cases.

III. Approach

Our approach to generating a test case summary is based on static analysis of the test case's source code. Our current focus is on generating summaries for Java unit test cases written to the JUnit test framework. Given this focus, the input to our approach is a JUnit test suite for which our approach generates a textual summary for each test method in the test suite.

We consider each test method (i.e., test case) in the test suite in turn. We focus on method invocations, including arguments to the method invocations, present in the test case. We extract each method invocation in the test method and identify which are verification statements; that is, statements which check whether an expected value has occurred. We categorize all other method invocations as operations.

We then determine how unique a particular method invocation is relative to other test cases. The uniqueness of a method invocation helps identify the focus of a given test case. For example, Figure 1 shows two test cases. Both test cases are similar, yet one includes a `zoom` method invocation and the other a `zoomInBoth` method invocation. This difference in method invocation captures the essential difference between the test cases. To better explain our approach, we provide more detail on the three most notable steps: classifying method invocations, comparing the difference between the test cases, and identifying key facts about the test case. We then briefly describe how we generate the textual summary.

A. Classifying Method Invocations

A unit test case typically has four phases: setup, exercise, verify and teardown [4]. The verify phase is critical because it is the phase that determines whether there is a problem with the system under test. We refer to each statement in the verify phase as a verify statement. For JUnit test cases, we recognize verify statements as those using `assert*` calls, such as `assertEquals`. The differences between the setup, exercise and teardown phases are not as obvious as the verify phase. The classification of method invocations into the other three phases may depend on which part is interesting in the test case. Many patterns have been proposed to classify the phases from various viewpoints [4]. In our approach, we separate method invocations into verify and operation (i.e., all non-verify) statements.

B. Determining Unique Method Invocations

To identify which method invocations in a test case are relatively unique, we store the number of occurrences of each method invocation across the test suite. It is stored as a hash table that contains the method invocation as the key and the number of occurrences as the value. The key for a method invocation includes the method name, the class name of the object on which the method is invoked and the arguments to the invocation. We also include information about whether the method invocation is inside "try/catch" or "if or loop" statements; this information indicates whether the statement executes under some particular condition checked in the method.

We also need to represent information about the arguments to each method invocation. Our first choice is to use the variable name or constant value present in the source code. We use a heuristic to determine if the name is sufficient to comprehend. For example, if the variable name is short, such as two characters like "a1", we think this name is too short to comprehend. We follow the data flow in the method to see if the value comes from a variable named more meaningfully. If we cannot find a longer name, we use the argument type. When we follow the data flow and reached to the class name, we also determine whether the class is meaningful or not by using user-defined information that classifies meaningful class name. The user-defined information as not meaningful is set as "`ArrayList`" or "`Hashtable`" in this paper so we will determine this class name as not meaningful and use the variable name instead.

Separate hash tables are used for method invocations in verify and operation statements.

C. Determining Key Facts

To form the textual summary for a test case, we successively consider the least occurring verify and operation method invocations. The least occurring invocations are the most unique for each test case. If there are two method invocations with the same number of occurrences, we pick the invocation that appears later in the source code for the test case, similar to [4]. We repeat the process of picking invocations until the generated summary reaches a predefined length,

which is set to one tenth of the character number of the target test case for this paper.

D. Generating the Summary

Given the key method invocations, we can generate a textual summary using a pre-defined template. All operation invocations are described first using a "calls <methodname> on <objectname>" format. We then output all verification invocations using templates for various verification operations, such as "checks the <methodname> of <object> is equal to <value>".

IV. Preliminary Evaluation

Can the key information in a test case be identified to enable the generation of a useful summary? To investigate this question, we considered the application of our approach to the JFreechart application, which has largely reasonable Javadoc summary comments with reasonably well-named methods. This application thus provides a case to study in which we can compare against the developer-provided documentation given by the test method name and the method's summary Javadoc.

We first considered whether the information identified as key using our approach is the same as the information provided in the developer-provided text consisting of the test method name and the method's summary Javadoc. As a target, we picked the 14 test methods in the `ChartPanelTest` class. We applied our approach to generate key facts for each test case. We then analyzed the developer-provided text, extracting as key developer-provided facts the method name. Figure 1 provides an example. The key facts generated by our approach for this example are "Call `zoom` and check the size of `chartChangeEvents` is equal to 1". The developer-generated provided key facts in the comments for this test method are "a call to the zoom() method" and "generates just one ChartChangeEvent". There is a description "just one" in the developer-provided comment and we consider the generated summary thus captures the right facts because our generated facts express that `chartChangeEvent` is 1. In this case, the key facts generated by our approach match the key facts in the developer-provided documentation.

We found that 13 of the 14 generated summaries captured the relevant features in the developer-provided documentation. The inconsistency in one summary was that the developer-provided documentation used the word "Constructor", whereas the generated summary used the class name `ChartPanel`.

We also compared the length (in terms of number of characters) of the generated textual summaries and the length of the original test method to see how much shorter the summary is than the original method. For the 14 test cases considered, the summaries are 18-23% of the original length; some specific examples are shown in Table 1.

Next, we considered whether our technique can help a developer comprehend generated test cases. We generated test cases for `XYSeriesCollection` from the JFreechart application using the CodePro[5] tool. The test class generated using CodePro has 2371 lines with 86 methods. It is larger than the developer-created test class, which has 431 lines with 16 methods.

TABLE I. Compression Rate of Test Case Summaries

Test case name	Length and Compression Rate of Summaries		
	Number of characters in the test case	*Number of characters of the summaries*	*The length of summaries per method (%)*
zoom	607	137	22.57
zoomIn Both	591	121	20.47
zoomOut Both	594	123	20.71
zoomIn Domain	742	135	18.19
zoomIn Range	737	133	18.04

Figure 3 and Figure 4 shows the key facts our approach extracts from the generated test case code. In Figure 3, we can see from the summaries where the test cases for `XYSeriesCollection` differ. The first and third tests are highly similar but differ in the argument when constructing an instance of `XYSeriesCollection`. The key facts can help a developer understand the variety of test cases generated. Figure 4 alerts the developer to a potential problem. In Figure 4, the key facts about the test cases for `addSeries` are the same. On deeper inspection, it turns out that the test generation tool failed to run and only the initialization process of each case was generated. Scanning the facts extracted by our approach can point the developer to look at these cases to see if there is a problem.

testAddSeries_1	XYSeriesCollection (XYSeries(Day())) on XYSeriesCollection(), addSeries (series) on XYSeriesCollection()
testAddSeries_2	XYSeriesCollection (XYSeries(Day())) on XYSeriesCollection(), addSeries (series) on XYSeriesCollection()

Fig. 3. Looking into Test Cases with Summaries (Raw Facts)

testXYSeriesCollection_1	XYSeriesCollection, Check Equals for (0.5, result.getIntervalPositionFactor(), 1.0) , Check Equals for (false, result.isAutoWidth())
testXYSeriesCollection_2	XYSeries (Day()) , XYSeriesCollection (series) , Check Equals for (1, result.getSeriesCount()) , Check Equals for (false, result.isAutoWidth())
testXYSeriesCollection_3	XYSeriesCollection (series) , Check Equals for (0.5, result.getIntervalPositionFactor(), 1.0) , Check Equals for (false, result.isAutoWidth())

Fig. 4. Completely same Test Cases with Summaries (Raw Facts)

[5] https://developers.google.com/java-dev-tools/codepro/doc/

V. Discussions

We have provided initial steps towards generating a human-oriented summary of a test case based on an analysis of the source code of the test case. We believe that test case summaries show promise to help a developer understand test cases more efficiently and more effectively. In some cases, there may be other means of presenting the extracted facts than text; for instance, the facts can be presented by highlighting the relevant code in a test case in a development environment. Such highlighting has been shown to be effective in other cases [8]. Highlighting may be particular effective for understanding generated test cases, where with our approach, very different parts of a test case---the method name and arguments called only in those test cases---could be drawn to the attention of a developer. We have developed a preliminary approach of this highlighting for the Eclipse development environment.

Much more work is needed to characterize the kinds of test cases and to understand how this affects the human-oriented descriptions generated.

VI. Related Work

Many testing techniques have been developed to help in specialized cases. For example, Rothermel and colleagues have considered relating test cases to faults to determine which test case mostly reveals the faults [5]. These specialized approaches can be very effective at responding to regression test situations. In this paper, we are more interested in the general case of test case comprehension. Similar to studies that show how good comments can help programmers quickly understand what a method does [6] [7], we believe that test case descriptions can help support a developer to understand test cases more efficiently and more effectively. Others have also considered how to help in test case comprehension. For example, Cornelissen and colleagues have considered how to depict the behavior of a test case through sequence diagrams created from trace information [1]. This method may help to see how objects interact in a test case but does not necessarily help a developer deal with large test suites. We are not aware of any other work that focuses on generating textual descriptions of test cases.

VII. Summary

Unit test cases are not write-once, read-once. After initial development, just like code, unit test cases must be evolved. Unfortunately, unit test case code is not always easy to comprehend. In this paper, we have introduced the idea of generating summaries of test cases to ease comprehension. We have provided initial steps towards generating such a summary based on an analysis of the source code of the test case. We investigated our approach by applying it to the open source JFreeChart application, which has several hundred classes, showing we can capture the facts expressed by the developer as important and that we can pinpoint issues with test cases generated for the application.

Much more work is needed to make truly usable human-oriented summaries. For instance, what do developers need in a generated summary to ease their work? How do developers use test case generation tools? We will use our method to find what the key summary information is for the developer. When a code change causes many test cases to fail, how does the developer find the test case to investigate first? Is static analysis information sufficient to generate useful descriptions or is execution information needed? Are different summaries needed to support developers in different tasks? How can generated summaries augment the use of specialized testing support, such as test case prioritization? Work in this area will further the emerging area of software artifact summarization.

References

[1] Bas Cornelissen, Arie van Deursen, Leon Moonen and Andy Zaidman: Visualizing Testsuites to Aid in Software Understanding , In Proc. of CSMR 2007, pp. 213-222, 2007.

[2] Sarah Rastkar, Gail C. Murphy, and Alexander W. J. Bradley Generating Natural Language Summaries for Crosscutting Source Code Concerns, In Proc. of ICSM 2011 pp.103 - 112 2011.

[3] Sarah Rastkar, Gail C. Murphy, Gabriel Murray. Summarizing Software Artifacts: A Case Study of Bug Reports. In Proc. of ICSE2010, pp 505-514, 2010.

[4] Gerard Meszaros: xUnit Test Patterns: Refactoring Test Code Addison-Wesley Signature Series (Fowler), 2007.

[5] Gregg Rothermel, Roland H. Untch, Mary Jean Harrold: Prioritizing Test Cases For Regression Testing. In IEEE TSE 27(10), pp. 929-948, 2001.

[6] Giriprasad Sridhara, Lori Pollock and K. Vijay-Shanker: Automatically Detecting and Describing High Level Actions within Methods, In Proc. of ICSE 2011, pp. 101-110, 2011.

[7] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock and K. Vijay-Shanker: Towards Automatically Generating Summary Comments for Java Methods, In Proc. of ASE 2010, pp. 43-52, 2010

[8] Gerard K. Rambally, The influence of color on Program Readability and Comprehensibility In Proc. Technical Symposium on Computer Science Education (SIGCSE), pp. 173–181. ACM Press.1986