AUTOMATIC GENERATION OF TUTORIAL FROM UNIT TESTS
単体テストを用いたチュートリアルの自動生成手法

by

Hiroaki Mikami
三上裕明

A Senior Thesis
卒業論文

Submitted to
the Department of Information Science
the Faculty of Science, the University of Tokyo
on February 2, 2014
in Partial Fulfillment of the Requirements
for the Degree of Bachelor of Science

Thesis Supervisor: Takeo Igarashi　五十嵐健夫
Professor of Information Science

ABSTRACT

Understanding and learning the usage of the APIs are important to programmers, but they are time consuming tasks. A tutorial which is a document composed of some sample codes, their explanations and a list of sample codes is one way to understand the usage of the APIs. Tutorials are currently written by library developers; however, writing a tutorial is a tedious work. Therefore, there are cases that how to use the latest APIs is not described in a tutorial because developers does not update the tutorial. To deal with this issue, this thesis proposes a method that automatically generates a tutorial from unit tests. This method first generates executable sample codes from unit tests. It then uses program visualization technique to explain the sample codes. Future more, it analyses a dependencies between tests to make a list of sample codes. The results of a user study showed that tutorials generated by this method are more effective in helping programmers learn APIs than the existing auto-generated document.

論文要旨

API の理解と使い方の学習は、プログラマにとって重要であるが、時間のかかる作業である。API を理解するための方法の一つとして、チュートリアルがある。チュートリアルは、複数のサンプルコード及びその説明と、それらのリストからなるドキュメントである．現在、チュートリアルはライブラリの開発者が記述しているが、チュートリアルを書くことは手間がかかる。そのため、チュートリアルが更新されず、最新の API の使い方が、チュートリアルで説明されていない場合がある．この問題に対処するため、本論文では、単体テストからチュートリアルを自動的に生成する手法を提案する。この手法では最初に、単体テストから実行可能なサンプルを生成する。次に, サンプルの説明を生成するために、プログラム可視化の技法を用いる．また、サンプルコードのリストを作るために、単体テスト間の依存関係を解析する。ユーザースタディにおいて、この手法を用いて生成されたチュートリアルを用いることで、既存の自動生成されたドキュメントの場合よりも API を効率的に理解できるという結果が得られた。

Acknowledgements

I thank to Professor Takeo Igarashi for his comments, Daisuke Sakamoto for his advice and suggestions on drafts. And I thank to Jun Kato and Hidehiko Masuhara for their opinions about our approach.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Recently, application programming interfaces (APIs) of the libraries that third parties provides(for example, open source libraries) are widely used in the popular software in order to save cost and improve quality of the source codes [10], [12]. In particular, it is extremely difficult to develop modern software without these APIs. However, because of the APIs complexity, it is difficult and time consuming for developers to understand the usage of the APIs and to use the APIs [27], [30], [28].

Currently, most libraries are documented to help for understanding the APIs. Although the documentations can be valuable to understand the APIs, previous researches have shown that it is very difficult to write the documentations and to maintain them. As a result, many documentations are out of date and erroneous [8], [9].

Many studies have shown that the code examples are key resources for understanding and using the APIs [30], [27], [28]. So some automatic code example generation methods have been proposed [36], [15], [34], [20], [5]. These example generation methods use the generated samples in an API documentation such as Javadoc or a code recommendation system. For example, *eXoaDocs*[20] aims at adding code examples into documents created by Javadoc and MAPO [34] is the tool that recommends code snippets by mining the client code repositories. But API documentation and code recommendation system have two problems:

- it is difficult to determine which elements (e.g. functions, methods and classes) are optional [30], and
- they are not suitable to understand and learn the complex APIs usages that are found in frameworks. For example, many frameworks assume that developers use the APIs in specific order [8].

To compensate for the problems of API documentation, some libraries provide the documentations called tutorial or getting started document. The tutorial is composed of the list of sample codes and their explanations. And the order of the list is not mechanical order such as alphabetical order but is created so that developers can understand the APIs if they read it in order. By using tutorials, developers can know how to use the basic APIs of the library [8].

However, the tutorials must be written and maintained by the contributors and writing the tutorials is more difficult than writing the API documentations [8]. As a result, the tutorials more often become out of date than the API documentations, for example, the code samples in the tutorial may not be able to be compiled in the latest version of APIs and the APIs in the tutorial may be currently deprecated. Although there are techniques to help for creating samples used in the tutorial [16][1], contributors still must write all of the code example used in the tutorial even if they use this technique. To deal with this issue, this thesis proposes a method that automatically generates a tutorial. This method is related to unit testing, program visualization technique and dependencies between tests.

Unit testing have become popular recently because of the spread of the agile development methods [3]. The test codes can be sources of the code examples and have some advantages [15]. First, the test codes are self-contained and executable unlike other code example sources such as web page and client code repository. Second, test codes are reliable and latest APIs are used in them because API contributors write and maintain test codes. Program visualization is technique that shows program execution to programmers. Many tools that visualize program have been proposed for the purpose of debugging and education [17], [7], [29], [22]. Program visualization is helpful for learning the relation between the source code that is a static representation of the program and the program execution that is a dynamic process of the program [17] [6]. Dependencies between test is a concept that is related to debugging tests and maintaining tests. Because of their existence, small bug of software causes failure of many test cases like a domino effect and debugging of software becomes difficult. So it should be removed from the test codes, but cannot be removed completely [21] [11]. There are several methods that use dependencies between test to localize defects quickly [21], [13].

The proposed method consists of four modules. Test code analysis module analyses the test code and classifies the test programs in order to use these classification results in the other modules. Code example extraction module extracts the executable code examples from unit tests, and clustering them. Explanation generation module generates the html pages that contain code examples and their explanations by using program visualization technique. The tutorial generation module generates order relation between examples by using dependencies between tests and dependencies in the program and makes tutorials by using this order relation.

In order to evaluate usefulness of the proposed method, we ran the user study. The results of the user study indicates that developers can perform more programming tasks if they use the tutorial generated by the tutorial than if they use Javadoc that contains code examples extracted from unit tests. The main contributions of this thesis are

- a method that automatically generates lists of the tutorials by using dependencies between tests

- an algorithm that extracts executable code examples from unit tests, and

- implementing the prototype of the method.

Chapter 2

Related Works

2.1 Barrier to learn APIs

Many studies [27], [30], [28] have been conducted to identify the barrier in learning APIs. Robillard et al. analyzed documentations. They showed that code examples are key resources to learn APIs and classified code examples into snippets, tutorials and applications [27] [28]. Scaffidi [30] investigated the inadequacy of documentations and showed the weakness of tutorials and API documentations.

Dagenais et al. [8] [9] analyzed documentations in open source projects and the way to maintain them. They indicated that tutorial maintenance is more difficult than API documentation maintenance and tutorials are more appropriate to learn how to use framework than API documents.

2.2 Method to help learn APIs

Several approaches to generate code examples automatically have been proposed. Kim et al. [20] proposed *eXoaDocs* that extracts code examples from client code repositories. That method separated into four modules: summarization, representation, diversification and ranking. Xie et al. proposed a framework that gets a method name, class name or package name as inputs and extracts method invocation sequence from open source repository [34]. Nasehi et al. shewed that unit tests can be good resources of code examples, and proposed an approach that extracts code examples from unit tests and integrates them into API documentation to improve the learnability of APIs [25]. Ghafari summarized the problems concerning code example extraction from unit tests and proposed an overview of the algorithm that extracts code examples from unit tests for code recommendation systems [14] [15]. The algorithm consists of two steps: identify units under test and extract usage example from test codes. Zhu et al. proposed the algorithm that extract code examples from test codes [35] [36]. That algorithm uses *test scenario* and four code pasterns. *Test scenario* is a self-contained subset of test code and explains an independent API usage.

Several code completion systems that use code examples have been proposed. Mandelin et al. defined *jungloid* as a code example that explains how to generate single output type from single input type, and proposed an algorithm that generates *jungloids* from API signatures and sample codes [24]. GraPacc proposed by Nguen et al. [26] is a tool that extracts context-sensitive information from the code under editing finds the best match code pattern and uses the pattern to complete code. Lv et al. [23] proposed a new algorithm that extracts method invocation sequences to instantiate objects from API library codes and selects the sequences that are best matching to destination and source types. Blueprint

[4] is an interface that searches the sample codes in web and presents the search results.

2.3 Method to maintain documents

Ginosar et al. [16] proposed an interface that helps write multi-stage code examples by using revision control algorithm. The author can propagate a change of a certain stage to other stages by using that interface. Cumiki [1] is a web service that helps programmers teach and learn a program by annotating the existing code. CommentWeaver [19] is an extension of Javadoc to reduce the amount of comments that need to be written by developers. Python doctest module [2] finds texts that look like Python program and its expected results from the docstring that is a string literal documenting the program. It can be used to check that the contents of the docstring is consistent with the program and to write the document including executable examples.

2.4 Program visualization

Several tools that visualize run-time behavior of program have been proposed for educational or debugging purposes. jGRASP [7] is a lightweight Java integrated development environment (IDE) that visualizes data structures. It was developed for educational purposes. Vebugger proposed by Rozenberg et al. [29] is a debugger that can visualize the state of selected objects by using templates written in HTML and CSS. Online Python Tutor [17] is a web-based program visualization tools for educational purposes. It is easy for computer novices to use Online Python Tutor because they need not to install or configure software. Lotoza et al. proposed a new approach that visualizes call graph interactively to help programmers navigate along control flow.

2.5 Dependencies between tests

Van et al. [32] proposed test refactoring approach that reduces test code duplication and dependencies between tests to improve readability and maintainability of the unit tests. Gaelli et al. [11] showed that 84% to 95% of the test cases have dependencies between tests by conducting four case studies.

JExample [21] is a JUnit extension that is developed to help programmers debug test code by using dependencies between tests. Programmers can use dependencies between tests and reuse a test case by adding *@Depends* annotations into test cases. Gaelli et al. [13] defined dependencies between tests by using *coverage set* that is a set of methods that are invoked by the test case, and proposed an algorithm that extracts this dependencies from test codes written in xUnit frameworks. Haensenberger et al [18]. applied the dependencies based on *coverage set* to JExample, and proposed an approach that automatically converts test cases written in JUnit into test case written in JExample that contains *@Depends* annotations. The approach can extracts the dependencies between instances under test.

Chapter 3

Automatic Tutorial Generation from Unit Tests

In this chapter, we describe the method that automatically generate a tutorial. Figure 3.1 shows the overview of the proposed method. First, Test Code Analysis Module receives the library source code and its test codes as inputs, and classifies the test programs into three groups by analyzing the run-time information of the unit tests. Next, Code Example Extraction Module receives these groups as inputs. It outputs the code examples of the library by generating code examples and clustering them. Explanation Generation Module, then, receives these examples and outputs the html pages that show the code examples and visualize the run-time behavior of the example. Finally, Tutorial Generation Module receives these html pages and the code groups of the test codes. And it outputs the tutorial of the library by extracting the dependencies between tests and using them.

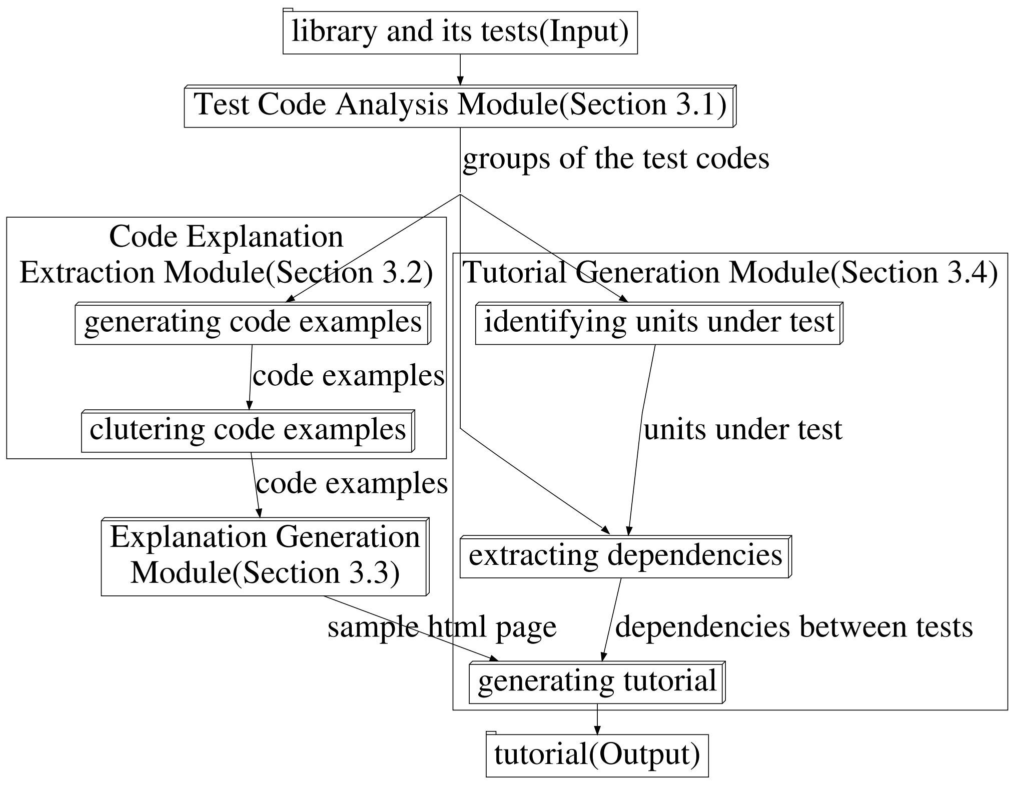

library and its tests(Input)
Test Code Analysis Module(Section 3.1)
groups of the test codes
Code Explanation Extraction Module(Section 3.2)
generating code examples
code examples
clutering code examples
code examples
Explanation Generation Module(Section 3.3)
sample html page
Tutorial Generation Module(Section 3.4)
identifying units under test
units under test
extracting dependencies
dependencies between tests
generating tutorial
tutorial(Output)

Figure 3.1: An overview of the proposed automatic tutorial generation method

The prototype of this method developed in this thesis is for libraries that are written in Java and are tested by using JUnit4[1]. Its backend was written in scala[2] and its frontend was written in JavaScript. However, they can be extended to the libraries that are written in a general object oriented language that uses class and are tested by using a general testing framework.

3.1 Test Code Analysis Module

In this module, the run-time information of unit tests are analyzed and we classify the programs in the test codes into the following groups:

- programs that works as a part of the main method,
- programs that works as the member of the main class, and
- substitute programs such as mock objects and stubs.

We call the first list *Mains*, call the second list *Members*, and call the third list *Substitutes* below.

The test codes that use current testing framework are sometimes distributed to reduce the amount of the test codes and to save test maintenance cost, and contain substitute programs that implement only functions used in tests. As a example of *Mains*, *Members* and *Substitutes*, consider the program in Listing 3.1. This is a simple test case of *java.io.PriorityQueue* class and shows typical test code written by using JUnit4. To execute *testAdd* test case, we must invoke *setup* and *testAdd* method in order. These methods should be main method in the generated example, therefore they are in *Mains*. Although *checkQueue* method is declared in the test code, it is in not *Mains* but *Members* because it is a general assertion method about queues. Finally, *MockComparator* class declared in *MockComparator.java* is in *Substitutes* because it is a mock class.

To obtain three lists, this thesis focuses on a currently running object called *this* in Java or C++, and defines *Mains* , *Members* and *Substitutes* as follows.

Mains is a list of members(e.g. method, field and constructor) that satisfies at least one of the two conditions:

- It is a method that is invoked first in the test case.
- Its *this* and *this* of the first method invocation are equal and there is no caller method.

Members is a list of members that satisfies at least one of the two conditions:

- Its *this* and *this* the element in *Mains* is equal and there is a caller method.
- It is static member and it is declared by the class that declares one of *Mains*.

Finally, *Substitutes* is a list of members that are declared in test codes and are not in either *Mains* or *Members*.

To analyze a test case using this definitions, we obtain the information of method invocation and field access during the test case execution. Next, we use a stack containing this information to distinguish *Mains* and *Members*. This stack is used as the call stack, and a method has no caller method if it is invoked when the stack is empty.

The detail of the proposed algorithm is follows.

[1]http://junit.org/
[2]http://www.scala-lang.org/

MockComparator.java

```java
package test;
import java.util.Comparator;

public class MockComparator extends Comparator<String> {
    public int compare(String e1, String e2) {
        ...
    }
}
```

PriorityQueueTest.java

```java
package test;

import static org.junit.Assert.*;
import org.junit.Test;
import org.junit.Before;
import java.util.PriorityQueue;

public class PriorityQueueTest {
    protected PriorityQueue<String> queue;

    public static void checkQueue(PriorityQueue<String> queue, int size,
          String front) {
        assertEquals(size, queue.size());
        assertEquals(front, queue.element());
    }

    @Before
    public void setup() {
        queue = new PriorityQueue<String>(11, new MockComparator());
    }

    @Test
    public void testAdd() {
        this.queue.add("Foo");
        this.queue.add("Bar");
        checkQueue(queue, 2, "Bar");
    }

    @Test
    public void testRemove() {
        ...
    }
}
```

Listing 3.1: A typical example of test case written by using JUnit4.

Step. 1 Obtain the classes that is declared in the test codes by using eclipse JDT[3].

Step. 2 Prepare the empty stack.

Step. 3 By using javassist[4], insert the following program before the field access and the beginning of the method.

(a) Obtain a member information and *this* of the member.

(b) Check *this* of the member obtained in (a), and add the member into the appropriate list according to the definitions.

[3]https://eclipse.org/jdt/

[4]http://www.csg.ci.i.u-tokyo.ac.jp/ chiba/javassist/

(c) Push the method invocation information to the stack if the member is a method.

Step. 4 Insert the program that pops the stack in the ending the method in the same way as Step. 3.

Step. 5 Execute each unit test by using JUnit4 API, and obtain *Mains*, *Members* and *Substitutes*.

In this thesis, javassist is used to simplify the implementation. However, this algorithm can be implemented by combining the language parser and the compiler or the interpreter even if the language does not have the library that changes the behavior of the program dynamically.

Consider *testAdd* test case in Listing 3.1 as a sample input of this algorithm. In this example, the algorithms analyses the program as follows:

1. got *MockComparator* and *PriorityQueueTest* as the classes declared in the test code by parsing two java files.
2. added *setup* method into *Mains* because *setup* was invoked firstly.
3. pushed *setup* to the stack.
4. added the constructor of *MockComparator* into *Substitutes* because *this* of the constructor was different from the *this* obtained in 2 and the constructor was declared in *MockComparator* class.
5. ignored the constructor of *PriorityQueue<String>* because it was not declared in classes obtained by 1.
6. added *queue* field into *Members* because its *this* and *this* obtained in 2 were equal and the stack was contained *setup*.
7. analyzed the test case until the end of execution, similarly.

The final output of the algorithm was

- the elements of *Mains* were *setup* and *testPush*,
- the elements of *Members* were *queue* and *checkStack*, and
- the element of *Substitutes* was the constructor of *MockComparator*.

3.2 Code Example Extraction Module

In this module, executable code examples firstly are generated by using the three groups described in Section 3.1. Next, we cluster code examples and select a representative code examples for each cluster.

3.2.1 Generating code samples from unit tests

This section describes the algorithm that generates code samples from unit tests. Some methods or ideas that extract code examples from unit tests have been proposed [36], [15]. Unlike them, the proposed algorithm generates executable code examples.

To generate executable code examples, we declare a class that contains the main method generated from *Mains* and other members generated from *Members* to generate executable code examples. The detail of our code examples generation algorithm is as follows:

Step. 1 Find the method that has *org.junit.Test* annotation from *Mains* and obtain the class that declares this method.

Step. 2 Declare a class with the same name as the class obtained in Step. 1. This is because some test cases use the information of the declaring class, for example, the class name might be used as a string.

Step. 3 Declare the fields and methods that are in *Members* as the members of the class declared in Step. 2.

Step. 4 Add the method that its name is main into the class declared in Step. 2. This method works as the main method.

Step. 5 Generate the body of the main method by lining up the body of the methods that are in *Mains*. Put the method body in braces({, }) to deal with the situation that there are duplicate names of local variable.

Step. 6 Remove `super.` from generated program, for example `super.setup();` is replaced by `setup();`.

Step. 7 Obtain the package declaration from the file declaring the class in Step. 1 and add this declaration into the sample.

Step. 8 Obtain the import declarations from the file declaring *Mains* or *Members* and add this declaration into the sample.

Step. 9 Obtain the files that declares *Substitutes* and add those into the sample.

Consider *testAdd* test case in Listing 3.1 as a sample input, again. First, because *PriorityQueueTest.testAdd* method has *org.junit.Test* annotation and is in *Mains* , the name of a main class is `PriorityQueueTest`. Because *Members* are *queue* field and *checkStack* method, they are declared as a member of *PriorityQueueTest*. Next, the main method is generated by putting the bodies of *setup* and *testAdd* methods in braces and lining up them. Because *Mains* and *Members* are declared in *PriorityQueueTest.java*, the package declaration and the import declarations are obtained from *PriorityQueueTest.java* and is added into the example. Finally, *MockComparator.java* is added into sample because the element of *Substitutes* is declared in *MockComparator.java*.

As a result, the proposed algorithm generates Listing 3.2 from *testAdd* test case in Listing 3.1.

3.2.2 Clustering similar code samples

There are often some similar test cases in test codes. The generated tutorial becomes difficult to understand if we use all of the samples from these similar test cases. So it is necessary to assemble the similar examples.

The algorithm that assembles similar examples is based on the algorithm algorithm [36]. In that algorithm, the samples are clustered by using the similarity between samples, and select the representative sample for each cluster.

The similarity relies on the method invocation sequence of the sample. In this thesis, the method invocation sequence is basically defined as a sequence that contains methods that are invoked by *Mains* or *Members* and are not declared in the test codes. In the case of the sample in Listing 3.2, the method invocation sequence are

- the constructor of *PriorityQueue<String>*, MockComparator.java

```
package test;
import java.util.Comparator;

public class MockComparator extends Comparator<String> {
    public int compare(String e1, String e2) {
        ...
    }
}
```

PriorityQueueTest.java

```
package test;
import static org.junit.Assert.*;
import org.junit.Test;
import org.junit.Before;
import java.util.PriorityQueue;

public class PriorityQueueTest {
    protected PriorityQueue<String> queue;
    public static void checkQueue(PriorityQueue<String> queue, int size,
          String front) {
        assertEquals(size, queue.size());
        assertEquals(front, queue.element());
    }

    public void main() {
        {
            queue =
                new PriorityQueue<String>(11, new MockComparator());
        }
        {
            this.queue.add("Foo");
            this.queue.add("Bar");
            checkQueue(queue, 2, "Bar");
        }
    }
}
```

Listing 3.2: The executable code example that is extracted from *testAdd* test case in Listing3.1. The main class is *PriorityQueueTest*.

- *PriorityQueue<String>.add*,
- *PriorityQueue<String>.add*,
- *PriorityQueue<String>.size*, and
- *PriorityQueue<String>.element*.

But there is a problem with the definition. For example, although the similarity of *testCase1* and *testCase2* in Listing 3.3 is 0.75 by the definition, the API usages explained by them are fully equal. To deal this problem, we remove consecutive duplicated method invocation subsequences so that the similarity of *testCase1* and *testCase2* in Listing 3.3 is 1. In detail, we remove all $s[i, j]$ defined by the formula 3.1.

```
public void testCase1() {
    Stack<int> stack = new Stack<int>();
    for (int i = i; i < 10; i++) {
        stack.push(i);
    }
    assertEquals(9, stack.lastElement());
}
public void testCase2() {
    Stack<int> stack = new Stack<int>();
    for (int i = i; i < 6; i++) {
        stack.push(i);
    }
    assertEquals(5, stack.lastElement());
}
```

Listing 3.3: An example that the similarity is not 1 but the programs explain completely same API usage.

$$s[i, j] = s[j + 1, 2j - i + 1] \tag{3.1}$$

The symbol $s[i, j]$ denotes the method invocation subsequence that begins at i and ends at j.

Listing 3.4 is a program that use swing of Java. The method invocation sequence by the initial definition is as follows:

1. the constructor of *JPanel*,
2. the constructor of *JLabel*,
3. *JPanel.add*,
4. the constructor of *JLabel*,
5. *JPanel.add*,
6. the constructor of *JButton*,
7. and *JPanel.add*.

Because the formula is satisfied if i is 2 and j is 3, we remove the constructor of *JLabel* and *JPanel.add*. As a result, the method invocation sequence in Listing 3.4 is the constructor of *JPanel*, the constructor of *JLabel*, *JPanel.add*, the constructor of *JButton* and *JPanel.add*.

```
@Test
public void test1() {
    JPanel panel = new JPanel();
    panel.add(new JLabel("foo"));
    panel.add(new JLabel("bar"));
    panel.add(new JButton("button"));
}
```

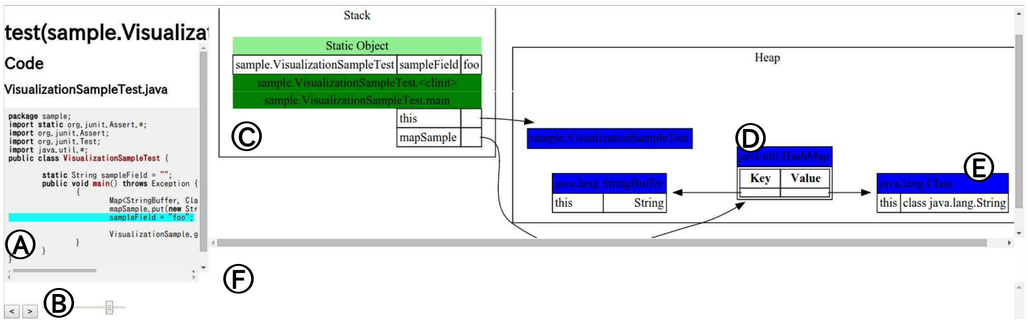
Listing 3.4: A sample program that tests swing classes

3.3 Explanation Generation Module

In this module, we generate explanations of code examples by using program visualization technique. The program visualization method in this thesis is based on the method used in Online Python Tutor [17]. That method is divided into backend and frontend. Backend executes the code example and obtains the execution trace of the example. And frontend visualizes the execution trace in web browser.

In the original method, the data such as an instance of class is converted into JSON format[5]. We encode basically the data in JSON format by using the information of the fields of the class. However, the example used by this method is written in Java unlike a program of Online Python Tutor and is the more complex program than the program for educational purpose that is main purpose of Online Python Tutor.

To address these differences, the encoding method differ from the method described above in two cases. First, we encode the instance in JSON format by using *toString* method instead of the field information if the instance is not declared in the target library. For example `new java.lang.StringBuffer("test")` is encoded as `test` if *StringBuffer* class is not declared in the target library. This is in order to remove the unimportant information to learn APIs and to prevent the image of the program execution from being bloated. Next, we encode the instance in JSON format as a data structure rather than as a class if the instance belongs to *java.util.Map*, *java.util.List*, *java.util.Set* or *Array*. For example, the instance of *java.util.Map* is encoded as the set of the pair of key and value. This is because the data structure of Java, such as map, list and set is a complex class in reality.



```
package sample;
import static org.junit.Assert.*;
import org.junit.Assert;
import org.junit.Test;
import java.util.*;
public class VisualizationSampleTest {

        static String sampleField = "";
        public void main() throws Exception {
                {
                        Map<StringBuffer, Cla
                        mapSample.put(new Str
                        sampleField = "foo";

                        VisualizationSample.g
                }
        }
}
```

Figure 3.2: A html page generated by this method. A) the sample code, B) the buttons and the slider that step forward and backward, C) the area that visualizes stack, D) the instance of *java.lang.Class*, E) the instance of progn-java.util.HashMap and F) the area that shows console outputs of the program.

In frontend, we generate the html page from the execution trace like Figure 3.2. Because of the encoding method mentioned above, the instance of *java.lang.Class* in this screenshot is visualized as string(see Figure 3.2-D) and the instance of *java.lang.HashMap* is visualized as a pair of key and value(see Figure 3.2-E). The only difference from the original visualization method is how to visualize the information of the static fields. Although static fields are rendered in the heap area in the original method, static fields are rendered in the stack area in the proposed method(see Figure 3.2-C). This is in order to reduce

[5]http://json.org/

the number of objects placed in the heap area.

In implementation, we use Play Framework[6] to generate JSON texts, Viz.js[7] to decide the layout of data and use highlight.js[8] for the purpose of syntax highlight.

3.4 Tutorial Generation Module

In this module, we extract dependencies between tests, first. Next, we generate a list of sample codes by using dependencies between tests and dependencies between classes. Finally, we generate a list of sample codes for each API in the similar way to help readers learn the usage of the specific API.

3.4.1 Identifying units under test

We identify the methods under test to use in extracting dependencies between tests. The algorithm used in this thesis is combined the algorithm using dynamic slicing proposed [15] and the algorithm using naming convention [36]. We describe the method that identifies the units under test of the certain test case.

First, we collect the methods that are invoked by *Mains* or *Members* and are not declared in the test codes and treat them as the candidates of units under test.

```
public void testWithoutAssertion() throws Exception {
    ...
    java.io.FileReader reader = new java.io.FileReader(file);
    reader.read();
    ...
}
```

Listing 3.5: An example of the test case that has no assertion statement

Next, the algorithm using dynamic slicing is tried to apply. Although the algorithm assumes that the test case has some assertion statements, there are test cases that have no assertion statements. For example, *testWithoutAssertion* in Listing 3.5 is a typical test case that has no assertion statements and it tests whether *FileReader.read* method can be executed without throwing any exceptions.

If test case has no assertion statement, the algorithm using naming convention is tried to apply. However, the name of test case sometimes does not contain the information of the units under test. For example, the name test case has no information of units under test if the name is associated with the id of the bug tracking systems. To deal with such cases, We change the definition of the name similarity between the test case name and the name of candidate method as shown in formula 3.2.

$$NameSimilarity = \frac{2 \times |C1 \cap C2|}{|C1| + |C2| - 2 \times |CommonWords|} \tag{3.2}$$

Here the symbol $C1$ denotes the set of words that is extracted by splitting camel-cased name of the method, $C2$ denotes the words set of the test case name

[6]https://www.playframework.com/
[7]https://github.com/mdaines/viz.js/
[8]https://highlightjs.org/

and $CommonWords$ denotes the set of the common words in the names of the candidate methods and the test case name.

Using $CommonWords$ prevents the influence of the string deciding the program structure (e.g. package name) and decrease the name similarity when the test case name has no information of units under tests. Consider the case that the test case name is *someLongPackageName.TestBugs.test001* and the name of one of the candidate method is *someLongPackageName.Class* as a example. Although the test case name does not indicate that the units under test of this test case is *someLongPackageName*, the name similarity of the original definition in [36] is about 0.67 that seems to be high on account of the influence of the package name(*someLongPackageName*). If all candidate methods belong to *someLongPackageName* package or its subpackages, the name similarity of the formula 3.2 becomes 0.

If the similarity of all of the candidates is less than a parameter, that is namely $threshold$, the results of the algorithm using naming convention is not used. In this thesis, $threshold$ is set to 0.

Finally, the method invoked lastly is identified as the units under test if the units under test cannot be identified by both of the two algorithm. Because the test case that has no assertion statements often tests whether the program can be executed to the last(see Listing 3.5 as a example), the method invoked lastly is probably the most important method in the test case.

3.4.2 Extracting dependencies between tests

In this section, we described the definition of dependencies between tests used in this approach and the algorithm to extract the dependencies. This algorithm is based on the dependencies based on *coverage set* [13].

While the dependencies based on *coverage set* is proposed to debug unit tests efficiently, the proposed dependencies between tests is defined to determine the test cases that are needed in tutorial and to generate the order relation used for making a list of the tutorial.

To achieve two purposes, we defined two types of dependencies between tests: (1) *normal dependencies* and (2) *internal dependencies*. A test case X depends *normally* on a test case Y if all of the units under the test case X are invoked from *Mains* or *Members* of the test case Y. And a test case X depends *internally* on a test case Y if all of units under the test case X are invoked during the test case Y execution and the test case X does not depends *normally* on the test case Y. The fact that X depends *normally* on Y means that the API usages explained by Y help readers understand the API usages of the test case X. And the fact that X depends *internally* on Y means that understanding API usages of Y is needed to understand the internal processing of X.

Compiler.java

```
public class Compiler {
    ...
    public Compiler() { }
    public Compiler(Settings settings) { ... }
    public String convertTabToSpace(String str) {
        return str.replace("\t", "␣␣␣␣");
    }
    public AST parse(String text) {
        String program = convertTabToSpace(text);
        return ... ;
    }
    public byte[] compile(AST ast) {
        return ... ;
    }
}
```

test codes of Language

```
@Test
public void testConvertTabToSpace() {
    assertEquals("␣␣␣␣", new Compiler().convertTabToSpace("\t"));
}

@Test
public void testParse() {
    Compiler lang = new Compiler();
    AST ast = lang.parse(...);
    assertFalse(ast != null);
    ...
}

@Test
public void testCompile() {
    Compiler lang = new Compiler(new Settings());
    AST ast = lang.parse(...);
    byte[] resutls = lang.compile(ast);
    ...
}
```

Listing 3.6: An implementation of a programming language and its test cases.

As a example of two dependencies, consider Listing 3.6 that are the compiler of a programming language and its test case. First, the units under *testConvertTabToSpace* is *Compiler.convertTabToSpace*, the units under *testParse* is *Compiler.parse* and the units under *testCompile* is *Compiler.compile* by the algorithm described in section 3.4.1. Next, since *testCompile* invokes *Compiler.parse* from *testCompile* that is in *Mains*, *testCompile* depends *normally* on *testParse*. Finally, since *testParse* invokes *Compiler.convertTabToSpace* from *Compiler.parse* that is not in either of *Mains* and *Members*, *testParse* depends *internally* on *testConvertTabToSpace*.

The algorithm to generate the graph of the dependencies between tests is as followings.

Step. 1 To assemble test cases into equivalent test cases, we generate the graph G' that

- its node set is a set of the test cases
- and its edge set contains the edge from the test case X to the test case Y if the test case Y *depends* on the test case X.

Step. 2 For each test case X, We finds the test case that are contained in the cycle containing the test case X and treat these test cases and the test case X as the equivalent test case of X.

Step. 3 We generate the graph G that

- its node set is a set of the equivalent test cases obtained in Step. 2
- and its edge set contains the edge from X to Y if the graph G' contains the edge from one of the test case in Y to one of the test case in X.

Step. 4 Finally, the transitive reduction of the graph G is computed. And we obtained the graph of the dependencies between tests.

We apply this algorithm to both of the *normal dependencies* and *internal dependencies*, and obtain G_{normal} and $G_{internal}$.

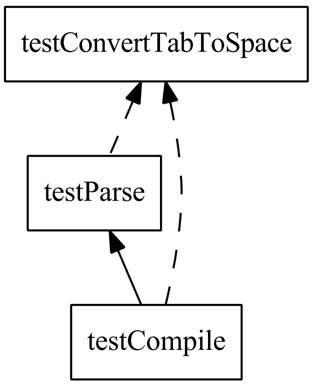


Figure 3.3: The dependencies between tests in Listing 3.6. The dotted line represents the edge of $G_{internal}$ and the normal lines represent the edges of G_{normal}.

Figure 3.3 shows G_{normal} and $G_{internal}$ of the Listing 3.6. The dotted line is the edge of $G_{internal}$, that is *internal dependencies*, and the normal lines are the edges of G_{normal}, that is *normal dependencies*.

3.4.3 Generating a tutorial

In this section, we describe the method that generates a tutorial from the results so far. First, this method obtains the test cases that do not depends on any test cases, and treats these as the examples that show practical usages of the library. Next, it generates a tutorial that helps readers learn API usages that is explained by these test cases to show API overview. Finally, it generates the tutorials for each methods to help learn the usages of the methods that are under test. And we use the algorithm that generates lists from sample code sizes, dependencies between tests and dependencies between classes under test to generate a list of tutorial.

The detail of the method to generate tutorial is as follows:

Step. 1 We generates dependencies between classes that are declared in the library. The class X depends on the class Y if the class X cannot be compiled without the class Y.

Step. 2 We obtain the equivalent test cases that are not depended *normally* on any equivalent test cases and the equivalent test cases that are not depended *internally* on any equivalent test cases. We name the intersection of the two equivalent test cases *seeds*. *Seeds* are thought to show the practical usages of the library.

Step. 3 We find the test cases that is depended on by *seeds* and add these test cases into *seeds*. To decide the test cases used in the tutorial explaining the API overview, we continue this operation until *seeds* do not change. The test cases that use the deprecated methods are excluded from *seeds* because such test cases do not explain latest API usages.

Step. 4 We generate a tutorial from *seeds* obtained in Step. 3 by using the following algorithm:

a We cluster the *seeds* by using the class under test that is a class declaring units under test to assemble the usages of the same class.

b For each cluster, we obtain the subgraph of G_{normal} induced by the cluster, and compute a topological sort [33] by using the number of methods that are invoked by *Mains* or *Members*.

c We generate an order relation among the clusters by using the dependencies between classes obtained in Step. 1 and the dependencies between tests. In this relation order, the cluster X is more than the cluster Y if and only if the clusters satisfy one of the two conditions: (1) there is a path in dependencies between tests from the element of the cluster Y to the element of the cluster X and (2) there is a path in the dependencies between classes from the class under the cluster Y to the class under the cluster X.

d We compute a topological sort by using the number of classes that are used by the test cases in the cluster to generate a list of the clusters.

e By converting the test case into the HTML page that shows the sample and visualizes it, we generate a tutorial that explain the API overview. The title of each HTML page is a name of the test case that is example source.

Step. 5 We do Step. 3 and Step. 4 for each method that is under test. Initial *seeds* are the test cases that their units under test is that method.

Consider the dependencies in Figure 3.4 as an input of the method. Figure 3.4 represents the dependencies between tests. The dotted lines mean the edge of *internal dependencies* and the normal lines mean the edge of *normal dependencies*. Further more, we assume that there are no edges in the dependencies between classes generated in Step. 1 of the algorithm.

First, because *testCompile* is not depended on by any test cases, the initial *seed* is *testCompile*. Next, *seeds* of the tutorial that shows the API overview are *testCompile*, *testWriteFile*, *testParse* and *testReadFile* because G_{normal} has paths that are from *testCompile* to these test cases.

We cluster *seeds* and obtain two clusters: (1) *testReadFile*, *testWriteFile* and (2) *testParse*, *testCompile* because the class under test of first cluster is *IO* and the class under test of second cluster is *Compiler*. Next, we compute a topological sort for each cluster, and obtains two lists: *testReadFile*, *testWriteFile* and *testParse*,

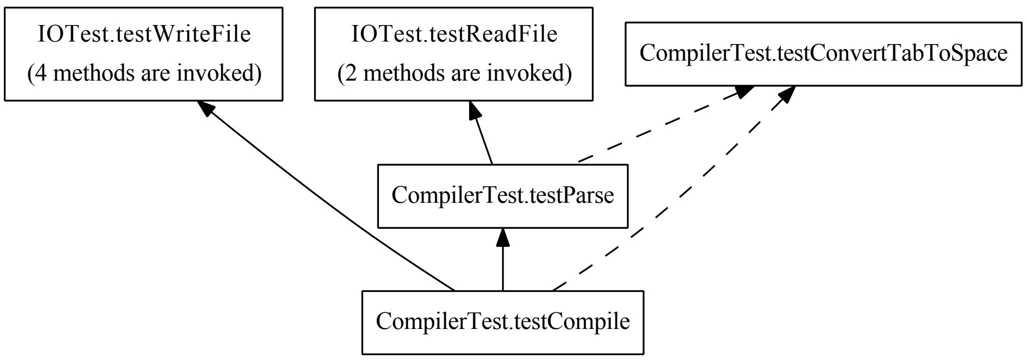


Figure 3.4: An example of the dependencies between tests.

testCompile. Third, we generates an order relation defining that the cluster of *IO* is more than the cluster of *Compiler*. Finally, we obtain Figure 3.5 as the list of the tutorial that shows the API overview.

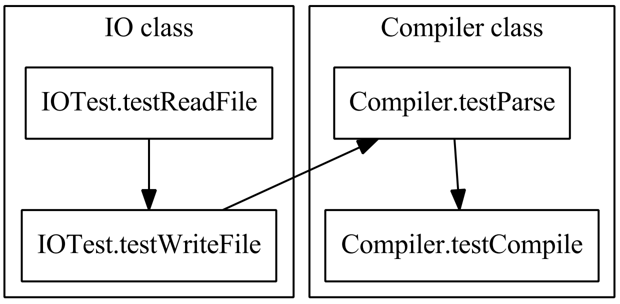
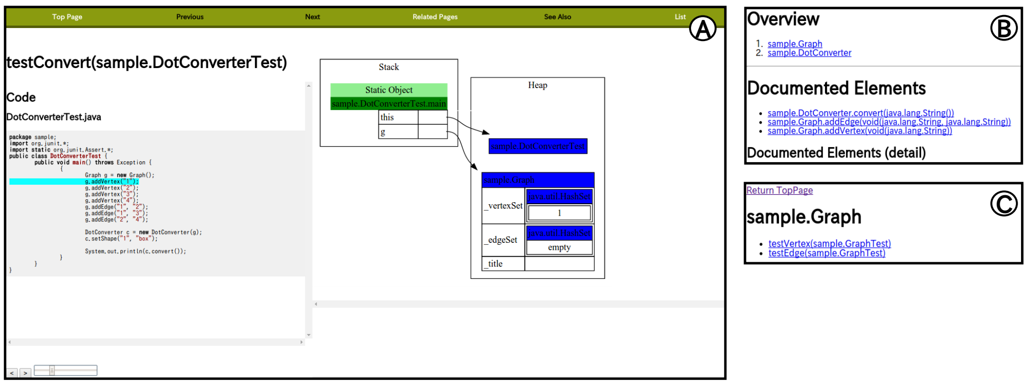


Figure 3.5: The tutorial that shows overview of Figure 3.4 API.

Figure 3.6 is an example of the whole method. It shows the tutorial that is generated from Appendix A by this method.



testConvert(sample.DotConverterTest)

Code

DotConverterTest.java

```
package sample;
import org.junit.*;
import static org.junit.Assert.*;
public class DotConverterTest {
        public void main() throws Exception {
                {
                        Graph g = new Graph();
                        g.addVertex("1");
                        g.addVertex("2");
                        g.addVertex("3");
                        g.addVertex("4");
                        g.addEdge("1", "2");
                        g.addEdge("1", "3");
                        g.addEdge("2", "4");

                        DotConverter c = new DotConverter(g);
                        c.setShape("1", "box");

                        System.out.println(c.convert());
                }
        }
}
```

Overview

1. sample.Graph
2. sample.DotConverter

Documented Elements

- sample.DotConverter.convert(java.lang.String())
- sample.Graph.addEdge(void(java.lang.String, java.lang.String))
- sample.Graph.addVertex(void(java.lang.String))

Documented Elements (detail)

Return TopPage

sample.Graph

- testVertex(sample.GraphTest)
- testEdge(sample.GraphTest)

Figure 3.6: The tutorial that is generated from Appendix A. A) The html page that shows example, B) The top page of the generated tutorial and C) The list of *Graph* class tutorial.

Chapter 4

User Study

We conducted a user study to evaluate the proposed approach. The user study had the following goals.

1. Investigating how useful the tutorial generated by the proposed approach is to lean APIs.

2. Collecting feedback about the improvements of the approach.

In the user study, we documented Commons CLI[1] by using the proposed approach. Commons CLI is the library for parsing command line arguments and for printing help message of the command line options. The reasons of selecting Commons CLI are as follows. First, since APIs should be used in specific ways to use this library, tutorials for the library are in high demand. Second, the APIs of the library is not too complex to conduct a user study in a short time. Third, the test codes of the library are rich and contain various kinds test cases.

We used Javadoc that is added code examples that were extracted by *UsETeX*[36] for comparison. The reasons for choosing the *UsETeX* are two: (1) *UsETeX* uses the same information as the proposed method because it is a tool that extracts code examples from test codes and (2) the proposed approach uses its sample clustering algorithm and its algorithm identifying the units under test.

4.1 Methodology

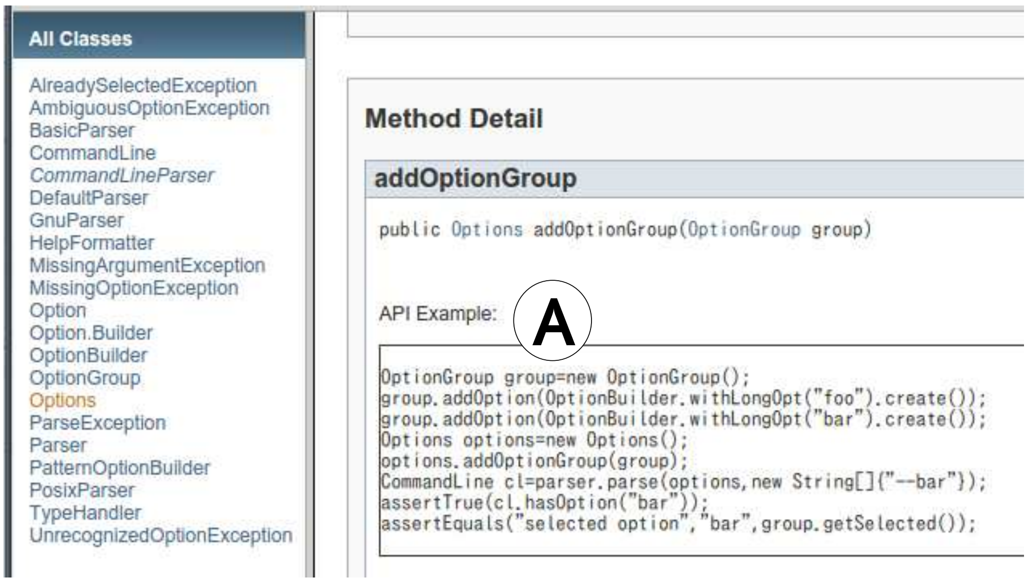
Participant	Age	Gender	Program Ability	Java Ability	Commons CLI experience
P1	22	M	7 years	2 years	nothing
P2	22	M	3 years	1 years	nothing
P3	23	M	12 years	1 years	nothing

Table 4.1: Participants of the user study.

We recruited to a 60 minute user study 3 participants described in Table 4.1. Participants answered the pre questionnaire about age, gender and programming experience. The results of the pre questionnaire is described in Table 4.1.

After the pre questionnaire, participants were asked to implement simple echo command for 20 minutes by using Javadoc documentation. Figure 4.1 shows the top page of this documentation and it contains code examples. While the participant implemented echo command, an observer took notes about the experiment.

[1]http://commons.apache.org/proper/commons-cli/


All Classes

AlreadySelectedException
AmbiguousOptionException
BasicParser
CommandLine
CommandLineParser
DefaultParser
GnuParser
HelpFormatter
MissingArgumentException
MissingOptionException
Option
Option.Builder
OptionBuilder
OptionGroup
Options
ParseException
Parser
PatternOptionBuilder
PosixParser
TypeHandler
UnrecognizedOptionException

Method Detail

addOptionGroup

```
public Options addOptionGroup(OptionGroup group)
```

API Example: A

```
OptionGroup group=new OptionGroup();
group.addOption(OptionBuilder.withLongOpt("foo").create());
group.addOption(OptionBuilder.withLongOpt("bar").create());
Options options=new Options();
options.addOptionGroup(group);
CommandLine cl=parser.parse(options,new String[]{"--bar"});
assertTrue(cl.hasOption("bar"));
assertEquals("selected option","bar",group.getSelected());
```

Figure 4.1: The Javadoc document used in the user study. A) the code example extracted by *UsETeX*

The echo command implementation task was separated into 5 programming subtasks:

A. configuring an instance of *Options* class. In Commons CLI, *Options* class represents the command line options of the application under development. To parse the command line arguments properly, the participants were asked to create an instance of *Options* by using constructor and to set the information of each option(e.g. `--help`, `-n`) to the instance.

B. parsing the command line arguments. *CommandLineParser* is an parser interface that provides the method parsing the command line arguments by using the instance of *Options* and *DefaultParser* is a concrete subclass of *CommandLineParser*. The participants were asked to create an instance of *DefaultParser* by using the constructor and to parse the command line arguments by using *CommandLineParser.parse* method.

C. checking the specific option is set. *CommandLine* is a return type of *CommandLineParser.parse* method and it provides *hasOption* method that checks whether the specific option is set. The participants were asked to invoke *hasOption* method at the point related to the option. For example, it is required to write `hasOption("help")` for checking that `help` option is set at the point that determines whether to print help message.

D. obtaining the arguments that do not belong to the options. *CommandLine* also provide *getArgList* method that returns the arguments that do not belong to the options. The participants were asked to invoke *getArgList* method after *CommandLineParser.parse* method invocation.

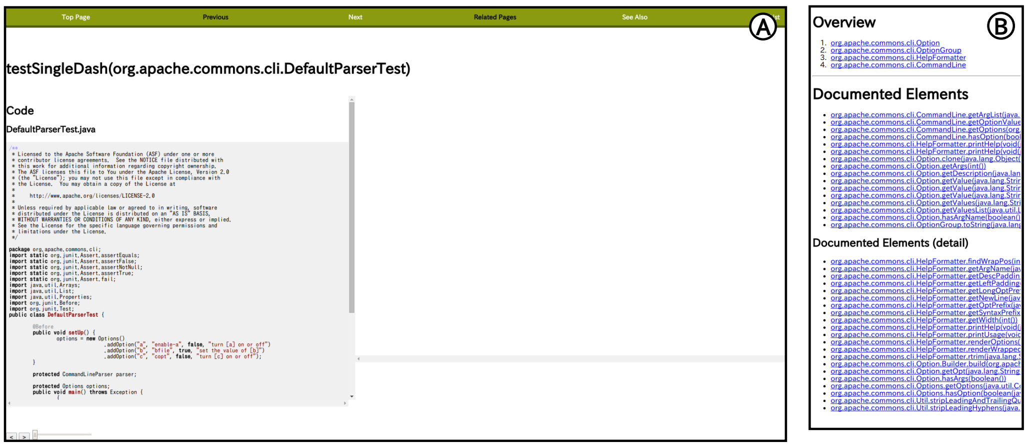
E. printing the help message. *HelpFormatter* provides *printHelp* method that outputs the help message of the command line options that is represented by the instance of *Options*. The participants were asked to create an instance of *HelpFormatter* by using the constructor and to print the help message by using *printHelp* method and the instance of *Options*.

The programs unrelated to above tasks were prepared beforehand because we wanted to maximize the time that participants did the programming tasks related to Commons CLI. For example, the program printing the list of string, that is necessary for echo command but is not related to Commons CLI, was prepared beforehand.

```
...
CommandLine cl;
if (cl.hasOption("h")) {
    // print help message
    ...
}
...
```

Listing 4.1: An example of the program written by a participant. He completed Task C but did not complete Task B.

After the task was finished, we count the number of the completed tasks. A participant completed a task if the APIs mentioned above were used properly. Consider Listing 4.1 as an example of the completed task and not completed task. A participant writing this program could understand that *CommandLine.hasOption* is used for determining whether h option is set, but could not understand how to obtain the instance of *CommandLine*. Task B is not completed because *CommandLineParser.parse* method is not used, but task C is completed because *CommandLine.hasOption* is properly used.



testSingleDash(org.apache.commons.cli.DefaultParserTest)

Code

DefaultParserTest.java

```
/**
 * Licensed to the Apache Software Foundation (ASF) under one or more
 * contributor license agreements.  See the NOTICE file distributed with
 * this work for additional information regarding copyright ownership.
 * The ASF licenses this file to You under the Apache License, Version 2.0
 * (the "License"); you may not use this file except in compliance with
 * the License.  You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.apache.commons.cli;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.fail;
import java.util.Arrays;
import java.util.List;
import java.util.Properties;
import org.junit.Before;
import org.junit.Test;
public class DefaultParserTest {

        @Before
        public void setUp() {
                options = new Options()
                                .addOption("a", "enable-a", false, "turn [a] on or off")
                                .addOption("b", "bfile", true, "set the value of [b]")
                                .addOption("c", "copt", false, "turn [c] on or off");
        }

        protected CommandLineParser parser;

        protected Options options;
        public void main() throws Exception {
                {
```

Overview

1. org.apache.commons.cli.Option
2. org.apache.commons.cli.OptionGroup
3. org.apache.commons.cli.HelpFormatter
4. org.apache.commons.cli.CommandLine

Documented Elements

- org.apache.commons.cli.CommandLine.getArgList(java.
- org.apache.commons.cli.CommandLine.getOptionValue
- org.apache.commons.cli.CommandLine.getOptions(org.
- org.apache.commons.cli.CommandLine.hasOption(bool
- org.apache.commons.cli.HelpFormatter.printHelp(void(
- org.apache.commons.cli.HelpFormatter.printHelp(void(
- org.apache.commons.cli.Option.clone(java.lang.Object(
- org.apache.commons.cli.Option.getArgs(int())
- org.apache.commons.cli.Option.getDescription(java.lan
- org.apache.commons.cli.Option.getValue(java.lang.Strir
- org.apache.commons.cli.Option.getValue(java.lang.Strir
- org.apache.commons.cli.Option.getValue(java.lang.Strir
- org.apache.commons.cli.Option.getValues(java.lang.Stri
- org.apache.commons.cli.Option.getValuesList(java.util.L
- org.apache.commons.cli.Option.hasArgName(boolean()
- org.apache.commons.cli.OptionGroup.toString(java.lang

Documented Elements (detail)

- org.apache.commons.cli.HelpFormatter.findWrapPos(in
- org.apache.commons.cli.HelpFormatter.getArgName(jav
- org.apache.commons.cli.HelpFormatter.getDescPaddin
- org.apache.commons.cli.HelpFormatter.getLeftPadding
- org.apache.commons.cli.HelpFormatter.getLongOptPre
- org.apache.commons.cli.HelpFormatter.getNewLine(jav
- org.apache.commons.cli.HelpFormatter.getOptPrefix(jav
- org.apache.commons.cli.HelpFormatter.getSyntaxPrefix
- org.apache.commons.cli.HelpFormatter.getWidth(int())
- org.apache.commons.cli.HelpFormatter.printHelp(void(
- org.apache.commons.cli.HelpFormatter.printUsage(voic
- org.apache.commons.cli.HelpFormatter.renderOptions(
- org.apache.commons.cli.HelpFormatter.renderWrappec
- org.apache.commons.cli.HelpFormatter.rtrim(java.lang.
- org.apache.commons.cli.Option.Builder.build(org.apach
- org.apache.commons.cli.Option.getOpt(java.lang.String
- org.apache.commons.cli.Option.hasArgs(boolean())
- org.apache.commons.cli.Options.getOptions(java.util.Co
- org.apache.commons.cli.Options.hasOption(boolean(jav
- org.apache.commons.cli.Util.stripLeadingAndTrailingQu
- org.apache.commons.cli.Util.stripLeadingHyphens(java.

Figure 4.2: The tutorial used in the user study. A) the example page of the tutorial. and B) the top page of the tutorial.

Next, we explained to participants how to use the tutorial by using the library about data structure as a sample. Similarly to the echo command implementation, participants were asked to implement simple cat command for 20 minutes by using the tutorial generated by the proposed approach. Figure 4.2 shows the top page and the example page of this document. The programming subtasks that were necessary for implementing cat command is same as the subtasks of echo command.

The programming task order and the relation of the tasks and the documents are different for each participant. P1 implemented echo command by using the tutorial first, P2 implemented cat command by using the Javadoc first and P3 implemented echo command by using the Javadoc first.

After two tasks were finished, participants answered post questionnaire about the followings:

Q1 Did you feel that the tutorial was more useful than Javadoc?

Q2 Did you feel that program visualization in the tutorial is useful for understanding APIs?

Q3 Did you feel that the order of the example was natural?

And we got some comments about the improvements of the tutorial.

During the user study, participants used PC-GL17414GU running Ubuntu 14.04 with Core i5-3317U and 4GB RAM, LCD-MF225XBR as a sub display for displaying a document, MA-117HBK as a mouse and TK-FCM005BK as a keyboard. And they use emacs 24.3[2] as a text editor and Google Chrome 39.0[3] as a web browser.

4.2 Results

The result of the user study is summarized in Table 4.2 and the result of the post questionnaire is summarized in Table 4.3.

4.2.1 How useful the tutorial generated by the proposed approach is to lean APIs.

The result of the user study shows that regardless of the differences of conditions, all of the participants could do more programming subtasks by using the tutorial than by using the Javadoc. P2 and P3 answered that the tutorial generated by the proposed method is more useful than Javadoc: P2 rated 6/7 and P3 rated 5/7 about the tutorial usability(see Table 4.3).

P1 said that he felt that using Javadoc was easier than using the tutorial because the code example contained by Javadoc was short and it was easy to begin to write code. In fact, P1 wrote the codes related to 4 subtasks during the experiment with Javadoc. On the other hand, he wrote the codes related to 3 subtasks during the experiment with the tutorial.

P1 and P3 felt the order of the code example was natural: P1 rated 7/7 and P3 rated 5/7 about the order of the example(see Table 4.3). P1 said that there was no problem about the order, however, the code examples had too many information about API usages to understand them easily. However, all of the participants answered that program visualization in the tutorial was not useful to understand APIs. P1 said that seeing the visualized program state was time consuming. P2 told us that it was a problem that the area of code examples was narrow due to the program visualization area.

4.2.2 Feedback about the improvement of the approach

P1 and P2 told us that it is a problem that the code examples extracted by the proposed algorithm are too long and too complex.

P2 suggested two improvements about program visualization:

[2]http://www.gnu.org/software/emacs/
[3]https://www.google.co.jp/chrome/

- to show the differences of the visualized program state between the previous step and the current step,
- and to sync the code example and visualized state. For example, if user click second line of code example, the tutorial visualizes the program state corresponding to second line.

The lack of the code explanation in natural language was pointed out as a problem. For example, P1 said that he needed more comments to understand the code examples. And P2 told us that it is desirable that he can guess easily the content of code example from its title. P3 said that if the tutorial shows the input and output of the sample, it becomes more easy to understand code example. P2 and P3 said that because inferring the method signature from code example is tedious, it is important that the tutorial provides the information of the signatures for each methods in code examples.

Participant	**A**	**B**	**C**	**D**	**E**
P1	tutorial				tutorial
P2	tutorial		both		
P3	tutorial	both	Javadoc	tutorial	tutorial

Table 4.2: Summary of the user study results. The cells contain tutorial if the participant completed a task successfully by using the proposed tutorial only, contain Javadoc if the participant completed a task by using the Javadoc only and contain both if the participant completed a task by using both of the documentations.

Participant	**Q1 Usability**	**Q2 Program visualization**	**Q3 Example order**
P1	3	1	7
P2	6	3	4
P3	5	3	5

Table 4.3: The results of the post questionnaire. Each question is 7-point scale. Rating of 1 represents strong disagreement and Rating of 7 represents strong agreement.

Chapter 5

Discussion and Future Works

5.1 Discussion

In the user study, all participants completed more task by using the proposed tutorial than by using Javadoc and two-thirds participants felt that the proposed tutorial is more helpful than Javadoc. On the other hand, all participants disagree that program visualization in the tutorial was useful to learn APIs, and pointed out that there are several improvements of the tutorial.

The results of the user study indicated that a tutorial generated by the proposed approach was more effective in helping users learn the complex API usages than the API documentation. Because writing tutorials is difficult and time consuming tasks, this approach may be useful to maintain tutorials.

However, this algorithm has some limitations and problems. This approach generates the executable code examples to apply program visualization technique. But the examples is big and complex to make code examples executable certainly, therefore the readability of examples is low. Further more, there are some cases that the proposed algorithm generates not-executable examples. For example, the generated example becomes not-excusable if the test class overrides the method of its base class(see Listing 5.1).

```
public class SubTestClass {
    @Override
    public void assertSomeCondition(...) {
        ...
    }

    @Test
    public void testCase1() {
        ...
        assertSomeCondition(...);
        ...
    }
}
```

Listing 5.1: A test case that generates an not-executable code example.

In this method, the explanations of code examples written in natural language are poor and improper. The only resources for natural language explanation in this method is the names of the test cases that are used as titles of the code examples. Although the name of the test case contains much information about the content of the test case, it does not always contain appropriate information to use for documentations.

5.2 Future Works

We believe the following future works are important to improve the usability of generated tutorials.

Extracting high quality code examples

Readability of the code examples is important and the improvement of readability of examples can increase the usability of generated tutorials. Restyling code examples by using the static program analyzing technique may improve the readability of the code examples. Similarly the code example may be separated into simple code examples by using this technique.

Generating automatically code explanations

While the results of the user study indicates that the explanations written in natural language is more suitable to tutorials than program visualization, this method cannot generate appropriate natural language explanations. It may be possible to generate natural language explanations by using pattern matching technique such as [31].

Extending to semi-automatic methods

Although the automatic generation of tutorials save maintenance cost, it cannot always generate a suitable tutorial. Therefore, the method can be more practical by enabling developers to use annotations like Javadoc. The annotations need to be helpful for not generating the tutorials but for understanding the test cases to save maintenance costs of tutorials.

Chapter 6

Conclusion

This thesis proposed the algorithm that automatically generates tutorials from unit tests by using program visualization technique and dependencies between tests. We developed the algorithm to extract the executable code examples by using the run-time behavior of the test cases. And we defined the dependencies between tests for the purpose of generating tutorial by using the information of the units under test. The results of an user study showed that this method generates a natural order of the test cases and the tutorial generated by this method helps programmers learn unfamiliar APIs and write the program using these APIs. However, an user study found that there are several improvement in this method to generate more efficient tutorials.

Appendix A

Sample Inputs

A.1 Library Source Code

Graph.java

```java
package sample;
public class Graph {
    public class Pair {
        public Pair(String v1, String v2) {
            this.v1 = v1;
            this.v2 = v2;
        }
        public String v1;
        public String v2;
    }
    private java.util.Set<String> _vertexSet;
    private java.util.Set<Pair> _edgeSet;
    private String _title;
    public Graph(String title) {
        _title = title;
        _vertexSet = new java.util.HashSet<String>();
        _edgeSet = new java.util.HashSet<Pair>();
    }
    public Graph() {
        _title = "";
        _vertexSet = new java.util.HashSet<String>();
        _edgeSet = new java.util.HashSet<Pair>();
    }

    public String getTitle() {
        return _title;
    }
    public void addVertex(String vertex) {
        _vertexSet.add(vertex);
    }
    public java.util.Set<String> getVertexSet() {
        return _vertexSet;
    }
    public void addEdge(String v1, String v2) {
        _edgeSet.add(new Pair(v1, v2));
    }
    public java.util.Set<Pair> getEdgeSet() {
        return _edgeSet;
    }
    public java.util.Set<String> neighbors(String vertex) {
        java.util.Set<String> neighbors = new java.util.HashSet<String>();
        for (Pair p: _edgeSet) {
            if (p.v1.equals(vertex)) {
                neighbors.add(p.v2);
            } else if (p.v2.equals(vertex)) {
                neighbors.add(p.v1);
            }
        }
        return neighbors;
    }
}
```

DotConcverter.java

```java
package sample;

public class DotConverter {
    private Graph g;
    private java.util.Map<String, String> shapes;
    public DotConverter(Graph g) {
        this.g = g;
        shapes = new java.util.HashMap<String, String>();
    }
    public void setShape(String v, String shape) {
        shapes.put(v, shape);
    }
    public String convert() {
        String r = "graph␣" + g.getTitle() + "␣{\n";
        for(String v: g.getVertexSet()) {
            if (shapes.containsKey(v)) {
                r += (v + "[shape=\"" + shapes.get(v) + "\"]\n");
            } else {
                r += (v + "\n");
            }
        }
        for (Graph.Pair p: g.getEdgeSet()) {
            r += p.v1 + "␣--␣" + p.v2 + "\n";
        }
        r += "}";
        return r;
    }
}
```

A.2 Test Codes

GraphTest.java

```java
package sample;

import org.junit.*;
import static org.junit.Assert.*;

public class GraphTest {
    @Test
    public void testVertex() {
        Graph g = new Graph();
        g.addVertex("foo");
    }
    @Test
    public void testEdge() {
        Graph g = new Graph("title");
        g.addVertex("foo");
        g.addVertex("bar");
        g.addEdge("foo", "bar");
    }
}
```

DotConcverterTest.java

```java
package sample;

import org.junit.*;
import static org.junit.Assert.*;

public class DotConverterTest {
    @Test
    public void testConvert() {
        Graph g = new Graph();
        g.addVertex("1");
        g.addVertex("2");
        g.addVertex("3");
        g.addVertex("4");
        g.addEdge("1", "2");
        g.addEdge("1", "3");
        g.addEdge("2", "4");

        DotConverter c = new DotConverter(g);
        c.setShape("1", "box");

        System.out.println(c.convert());
    }
}
```

References

[1] cumiki. http://cumiki.com/. Accessed: 2015-01-18.

[2] doctest. https://docs.python.org/2.7/library/doctest.html. Accessed: 2015-01-18.

[3] Kent Beck. *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[4] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 513–522. ACM, 2010.

[5] Raymond PL Buse and Westley Weimer. Synthesizing api usage examples. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 782–792. IEEE, 2012.

[6] JH Cross, T Dean Hendrix, Larry A Barowski, et al. Combining dynamic program viewing and testing in early computing courses. In *Computer Software and Applications Conference (COMPSAC), 2011 IEEE 35th Annual*, pages 184–192. IEEE, 2011.

[7] James H Cross II, T Dean Hendrix, David A Umphress, Larry A Barowski, Jhilmil Jain, and Lacey N Montgomery. Robust generation of dynamic data structure visualizations with multiple interaction approaches. *ACM Transactions on Computing Education (TOCE)*, 9(2):13, 2009.

[8] Barthélémy Dagenais and Martin P Robillard. Creating and evolving developer documentation: understanding the decisions of open source contributors. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 127–136. ACM, 2010.

[9] Barthélémy Dagenais and Martin P Robillard. Recovering traceability links between an api and its learning resources. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 47–57. IEEE, 2012.

[10] Prem Devanbu, Sakke Karstu, Walcélio Melo, and William Thomas. Analytical and empirical evaluation of software reuse metrics. In *Proceedings of the 18th international conference on Software engineering*, pages 189–199. IEEE Computer Society, 1996.

[11] Markus Gaelli, Oscar Nierstrasz, and Stéphane Ducasse. One-method commands: Linking methods and their tests. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004.

[12] John E Gaffney Jr and Thomas A Durek. Software reuse—key to enhanced productivity: some quantitative models. *Information and Software Technology*, 31(5):258–267, 1989.

[13] Markus Galli, Michele Lanza, Oscar Nierstrasz, and Roel Wuyts. Ordering broken unit tests for focused debugging. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 114–123. IEEE, 2004.

[14] Mohammad Ghafari. Extracting code examples from unit test cases. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 667–667. IEEE, 2014.

[15] Mohammad Ghafari, Carlo Ghezzi, Andrea Mocci, and Giordano Tamburrelli. Mining unit tests for code recommendation. In *ICPC*, pages 142–145, 2014.

[16] Shiry Ginosar, De Pombo, Luis Fernando, Maneesh Agrawala, and Bjorn Hartmann. Authoring multi-stage code examples with editable code histories. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, pages 485–494. ACM, 2013.

[17] Philip J Guo. Online python tutor: Embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 579–584. ACM, 2013.

[18] Lea Haensenberger, Adrian Kuhn, and Oscar Nierstrasz. Using dynamic analysis for api migration. *Program Comprehension through Dynamic Analysis*, page 32, 2008.

[19] Michihiro Horie and Shigeru Chiba. Tool support for crosscutting concerns of api documentation. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, pages 97–108. ACM, 2010.

[20] Jinhan Kim, Sanghoon Lee, Seung-won Hwang, and Sunghun Kim. Adding examples into java documents. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, pages 540–544. IEEE, 2009.

[21] Adrian Kuhn, Bart Van Rompaey, Lea Haensenberger, Oscar Nierstrasz, Serge Demeyer, Markus Gaelli, and Koenraad Van Leemput. Jexample: Exploiting dependencies between tests to improve defect localization. In *Agile Processes in Software Engineering and Extreme Programming*, pages 73–82. Springer, 2008.

[22] Henry Lieberman and Christopher Fry. Zstep 95: A reversible, animated source code stepper. *Software Visualization: Programming as a Multimedia Experience*, pages 277–292, 1997.

[23] Chen Lv, Wei Jiang, Yue Liu, and Songlin Hu. Apisynth: a new graph-based api recommender system. In *ICSE Companion*, pages 596–597, 2014.

[24] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the api jungle. *ACM SIGPLAN Notices*, 40(6):48–61, 2005.

[25] Seyed Mehdi Nasehi and Frank Maurer. Unit tests as api usage examples. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[26] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 69–79. IEEE Press, 2012.

[27] Martin P Robillard. What makes apis hard to learn? answers from developers. *Software, IEEE*, 26(6):27–34, 2009.

[28] Martin P Robillard and Robert Deline. A field study of api learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.

[29] Daniel Rozenberg and Ivan Beschastnikh. Templated visualization of object state with vebugger. In *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*, pages 107–111. IEEE, 2014.

[30] Christopher Scaffidi. Why are apis difficult to learn and use? *Crossroads*, 12(4):4–4, 2006.

[31] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 101–110. IEEE, 2011.

[32] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. *Refactoring test code.* CWI, 2001.

[33] Eric W Weisstein. Topological sort. *From Mathworld–A Wolfram web resource*, 2000.

[34] Tao Xie and Jian Pei. Mapo: Mining api usages from open source repositories. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 54–57. ACM, 2006.

[35] Zixiao Zhu, Yanzhen Zou, Yong Jin, and Bing Xie. Generating api-usage example for project developers. In *Proceedings of the 5th Asia-Pacific Symposium on Internetware*, page 34. ACM, 2013.

[36] Zixiao Zhu, Yanzhen Zou, Bing Xie, Yong Jin, Zeqi Lin, and Lu Zhang. Mining api usage examples from test code. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 301–310. IEEE, 2014.