Master Thesis

Java Interaction Testing

2008

Leiden University
Leiden Institute of Advanced
Computer Science [LIACS]

Li Xuan No. S0631361

Acknowledgments

I greatly appreciate the effort of my supervisor Frank S. de Boer， who gave me quite a deal of guidance and helped for my mater thesis. Particularly, he put effort in directing me in the preparation of my thesis and when revising the manuscript with endless patience and insightful suggestions. I would also like to express my thanks to my mother Wang Xi-xuan & my husband Xiao-Chun Xing, who always support and encourage me with her endless care.

Besides that, I thank all other teachers who ever taught and instructed me during the past two years. I could finally graduate and get my Computer Science Master degree in their careful guidance from Leiden Institute of Advanced Computer Science.

Table Contents

Java Interaction Testing

July 25th 2008

Xuan Li
LIACS, Universiteit Leiden,
Niels Bohrweg 1
2333 CA Leiden, The Netherlands
+31 614589860
xli@liacs.nl

Prof. Dr. F.S. de Boer
LIACS, Universiteit Leiden,
Niels Bohrweg 1
2333 CA Leiden, The Netherlands
+31 715277069
frb@cwi.nl

ABSTRACT

Compared to traditional testing methods like Junit and Jmock, "testing based on sequence diagram" can be more efficiently applied to test the interaction traces between system under test (SUT) and the environment running the SUT. Mock objects come out in order to set up the compatible test environment. Besides that, through the tailor-made mock objects in this project - LiMock, the interaction of different objects within System under test (SUT) becomes possible to be detected to some extent. Although this kind of specific testing method is not considered mature, it would be popular for its easily edited xml input and handy testcase definition. And most important is that the practical interaction can be tested applied anytime during the software development. Case Census-Voter is the demonstated model. It verifies the advantanges of LiMock comparing Jmock and Junit. As a result, it is convienced that sequence based "Java Interaction Test" could present its potential specifically in the area of development of software interaction testing.

Keywords

Interaction Testing, Census-Voter, Junit, JMock, Assert, Use Case, QA, SUT, AUC, Test Environment, Parser, Scenarios, Sequence Diagram, LiMock, Interaction Testing, XML, Internal Interaction, CUT

1.INTRODUCTION

In current software industry, it always happens that a tester does not only need to assert the ouput result of system, but also has to test the interactions between the system and testing environment. Interaction testing comes forth with the purpose of testing this kind of interaction. Interaction testing happens in a specific environment. The interaction environment consists of testing code, system under test (SUT) and its dependent source auxiliary component (AUC). SUT is the target of the whole test program. The interface-level outputs and interactions between the SUT and its dependent environment is the focus of the testing.

AUC also plays an important role despite what its name implies. Unlike state based testing, it is critical for the SUT to interact with another party so that the interactions can take place and be tested. Ideally, AUCs are available before the development of SUT starts. However, it is usually not the case in the reality. Mock objects come in handy to replace the objects with which the SUT collaborates. With the help of Mock objects, testing can happen as soon as any part of implementation of SUT finishes. Testing code usually includes importing or generating input data, applying testing logic and verifying testing result. In the test environment, only the SUT is provided, all the testing code and probably AUC ought to be generated from test case offered by testers. The effient way of starting the test is to utilize simple XML defining designed UML sequence diagrams. And it is consider as an easy way for interaction testing. Testing interaction between two different java objects - "Java Interaction Testing" is right one of the concrete models of "Interaction Testing".

2.DESCRIPTION

2.1The Problem

In industrial practice, it sometimes occurs that the client companies would like to make sure of whether the software interfaces work well and fit their own system before purchasing the software or application form the software companies. Therefore, the testing becomes one of the most important affairs. Test experts should assure of the software quality (QA). And interaction testing turns into one part of QA.

We can set up an example from Census-Voter case. Census represents the SUT and Voter is the AUC. The testing code cannot compile and execute without the full implementation of both Census and Voter. (See Figure 1)

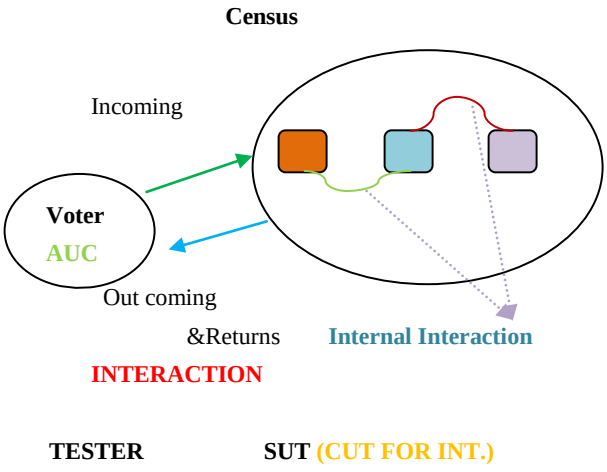

Figure 1

As a result of testing, the interaction between the SUT & test environment can be detected through the Incoming & Out coming returns. Let's take a close look, we suppose that Census is the person who collects the Voters' opinions, and Voters are the people (probably not only one person) who always vote with many opinions. All the opinions should be known and recorded correctly by Census. Then the Tester comes out for assuring the jobs of Census. While Voter votes something, e.g. a string in programming, Census should receive it and record it, then a report of all recorded votes is available. However, if there are many voters vote and/or do other behavior many times, how the Census handle? Can it send the each record correctly to Tester? We cannot suppose that if Voter votes, Census receive and record correctly, because the assumption would cause information missing or mistaken. There are many risks during the procedure. A series of problems may appear:

- **Do all the Voters have the bahvaior"vote"? Is there any exception that only part of Voters vote, the others doing nothing or the others wasn't aware of the voting?**

- **Does Census indeed calls all the Voters? Does Census forget to calls some of the Voters?**
- **Are all votes from voters correctly received by Census? The record of Census is complete & correct or not?**

Thus, the interaction testing becomes unavoidable and significant for verifying above problems. And the interaction testing urgently needs an optimized solution to realize.

2.2Possible Solution

First, a test environment can be set up with testing code and object Voter & Census. Although the Voter is also provided by testers, it is better to isolate Voter (AUC) from the testing code for good testing habit. By the reason that mock objects (or mocks for short) are perfectly suitable or testing a portion of code logic in isolation from the rest of the code and mocks replace the object with which the methods under test collaborate, thus offering a layer of isolation. [10] Consequently mock objects are introduced at this time. It provides a suitable mechanism to create mirror stubs [8] of SUT and its expected behavior. Then a mock object, mock voter, is instantiated from a modified class mirroring the original Voter class. And the new mocked object MockVoter interacts with Census instead. In another word, Census interacts with MockVoter not with original Voter [See Figure 2]. However, what kind of Frameworks offering mock objects are effective, Jmock or Junit mock object framework or any other ones? There is still a concatenate problem coming out. It can be studied further in the coming chapters. Second, the fact the interaction between Voter [MockVoter] and Census happens frequently has to be considered. The sequence diagram can right map the frequent cases with several set like – Voter[1].vote, Census report votes of Voter[1] to tester; Voter[2].vote, Census report votes of Voter[2] to tester;… Voter[n].vote, Census report votes of voter[n] to tester. Once the interaction sequence diagram is set, the suitable Test-driven development (TDD) is not difficult to implement**.** Last but not least, the Test-driven development (TDD) is expected not to be Ad hoc testing. ("Ad hoc means "for this purpose". It generally signifies a solution that has been custom designed for a specific problem or task, is not-generalizable, and cannot be adapted to other purpose." [Wiki definition]) A general template would be better for future use of interaction testing.

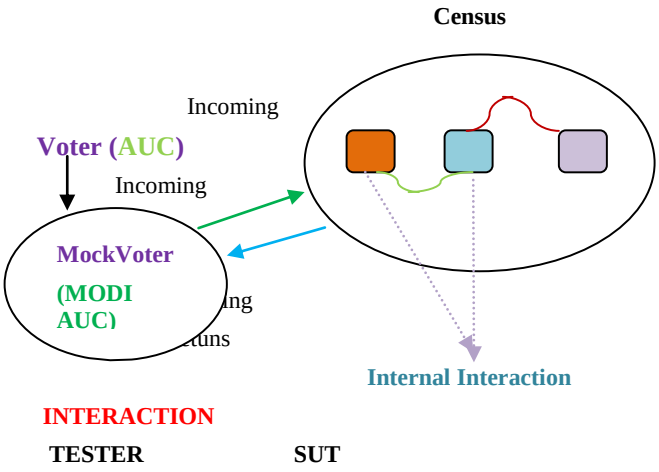

Figure 2

3.SEQUENCE DIAGRAM DESIGN

3.1Test Specification

In the reference document [3] "Generating Mock Classes from Observational Traces for Testing Java Programs". And it proposed to generate mock classes from an abstract syntax in List 1, which described the basic test specification – The signature of the environment is specified by the class Voter which contains a Boolean field fvote and its corresponding get-method vote. [1]&[2] In

another word, each Voter does the behavior “votes” with its sole identity. (Line 3-6). Census should provide Java’s HashSet to store the record of Voters’ voting. (Line 8-9). The testing mocks the record of voters in a loop (Line 20), and then analyzes the copied record (Line 24) and return the fvote method (Line 25).

List 1: Specification of the voting example

```
1    import java.util.HashSet
2
3    provided Voter {
4    vote() : bool;
5    fVote : bool;
6    } 7
8    required Census {
9    census(HashSet) : bool;
10    }
11
12    c : Census;
13    called : HashSet = new HashSet();
14    voters : HashSet = input.read;
15    conj : bool = true;
16
17    new !Census()! { 18 c:=?return()
19    };
20    c!census(voters.clone()) { 21 while (called.size() <
voters.size()) do { 22 (this : Voter)?vote() where
(called.contains(this) = false) {
23    called.add(this);
24    conj:=conj & this.fVote;
25    !return(this.fVote)!;
26    } }
 27   x:=?
```

Yet the abstract syntax is not so straightforward to understand and master although it is also tailor-made for case Census-Voter. For that, I developed another way which utilizes the simple XML not abstract syntax to define the testing scenarios. The scenarios can generate mock objects of Voter and obtain the expected return method in a situable test mechanism. In practical, the XML statement involves the behavior of the tester and the expected behavior Census reacted like map with the set– Voter[1].vote, Census Census.report [Voter(1)]; Voter[2].vote, Census.report [Voter(2)];… Voter[n].vote, Census.report [Voter(n)].

3.2Test Sequence Diagram

However, why we specially define the behaviors of Voter & Census in XML statement? And what are they used for? That’s because though the bidirectional behaviors, the interaction between Census & Tester happens. In Wikipedia, Interaction is defined as “**Interaction** is a kind of action that occurs as two or more objects have an effect upon one another. The idea of a two-way effect is essential in the concept of interaction, as opposed to a one-way causal effect… **Interactive computation** involves communication with the external world during the computation. This is in contrast to the traditional understanding of the computation which assumes a simple interface between a computing agent and its environment, consisting in asking a question (input) and generating an answer (output)…” In Census-Voter case, if Voter votes something when required and Census return another method, successful interactions are said to occur between Censuses with its dependent testing environment. As for the testing process: first, a new instance (object) of the component class Census is created by calling its constructor method and waiting for the return value which is assigned. Second, the tester calls the method census of component object Census, which passes a copy of the HashSet of voters to tester. Last, Census is asked to return the report

trace of all the voters as expected. To report the trace, the tester has to find out the votes of the voter object. After all the calls are received by the census, the control goes back to the testers. And individual census report can be accessed later by invoking overloaded report method of census object. Then a sequence diagram is built as follows:

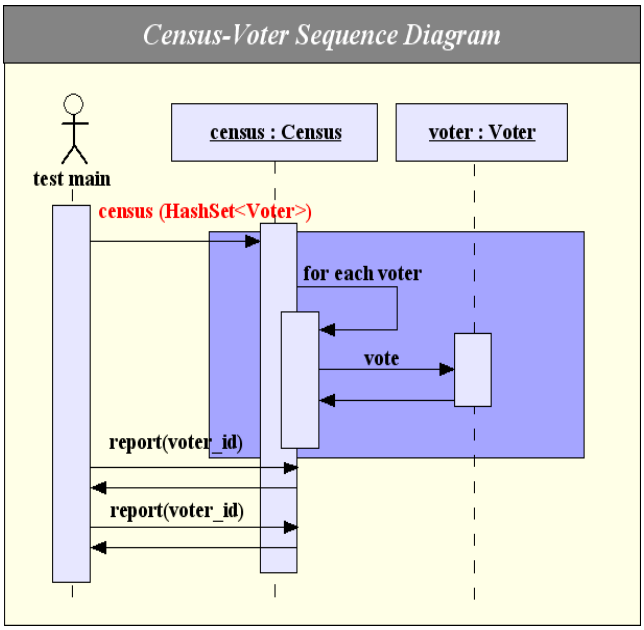

Figure 3

When the sequence diagram and test design are ready, the class diagram can be set up next in Figure 4 according to instruction of the reference paper:

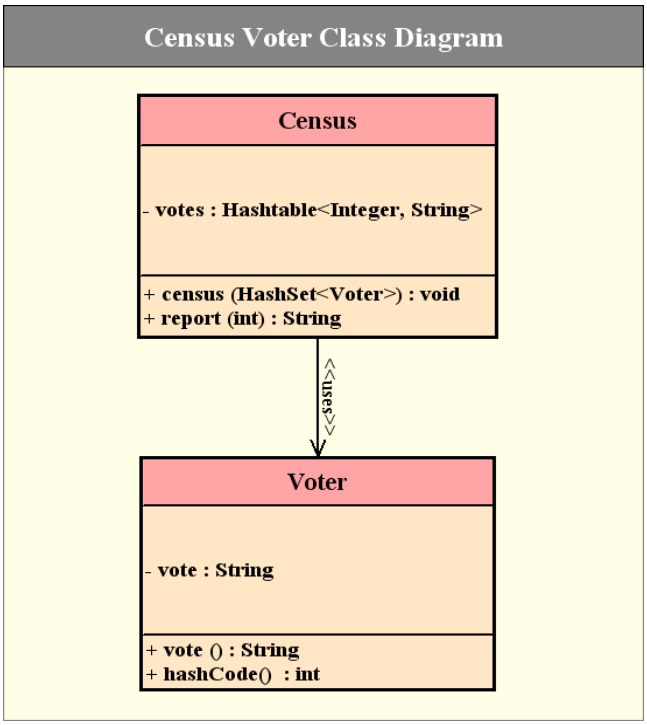

Figure 4

4.LiMock

4.1Code Generation in LiMock

LiMock provides code generation from an XML file. As a testing framework, it helps testers to concentrate on defining test scenarios and monitoring results with minimized programming skill requirements and coding efforts.

LiMock's work flow, as illustrated in Figure 5, starts from the XML file, TestingCase.xml. Information of declaration, instantiation and initialization of CUT and Mock objects are defined in the structured tags in the TestingCase.xml. The testing scenario is also defined, in the "expections" tag found in appendix 1, to specify the order of interactions, the interaction invoking objects and their parameters as well.

Of course, the information of the testing code that is going to be generated, such as class name and file path, are present.

The TestingCaseReader class reads the data in the structued xml tags and creates a list of key-value pairs, key: mock.num and value: 2 for example, for further processing by TestingComponentResolver (resolver). The xml reader takes advantage of Java Streaming API for XML (Stax), introduced in Java 6.0.

It is the resolver's responsibility to create test component objects from the key-value-pair properties. LiMock modulizes the essential components for the code generation procedure, e.g. ClassInfo (for CUT and Mock classes), Invocation (for interaction) and ParameterInfo(for passing to the interactions). The objects of these module classes are encapsulated with the TestCase, highlighted in pink in Figure 5, as the top-most container. Now all the information that are previously defined in the TestingCase.xml is contained in the testCase object of TestCase class.

Before we move on to the actual code generation, there might be questions about why there are two steps, reading and resolving, from xml data to module objects. The xml reader is extracting data while the component resolver is utilizing the extracted data. It is a good implementation practice to keep the classes performing different tasks separated. Moreover, xml reader shouldn't be aware of the resolving logic and how the data are used. The separation of two classes also helps to reduce the changes that are needed in the implementation if the xml structure has changed or the modulization of LiMock is altered.

After all the information is settled, the code generation is quite straight forward. The MockClassGenerator and TestClassGenerator are responsible for generating the Mock classes (AUC) and Test class (testing code) respectively. They both write the information of package, importing library, class header, fields and methods using the Java syntax into the specified output files. Once the test class is ready, the test can be started by running it as a JUnit class.

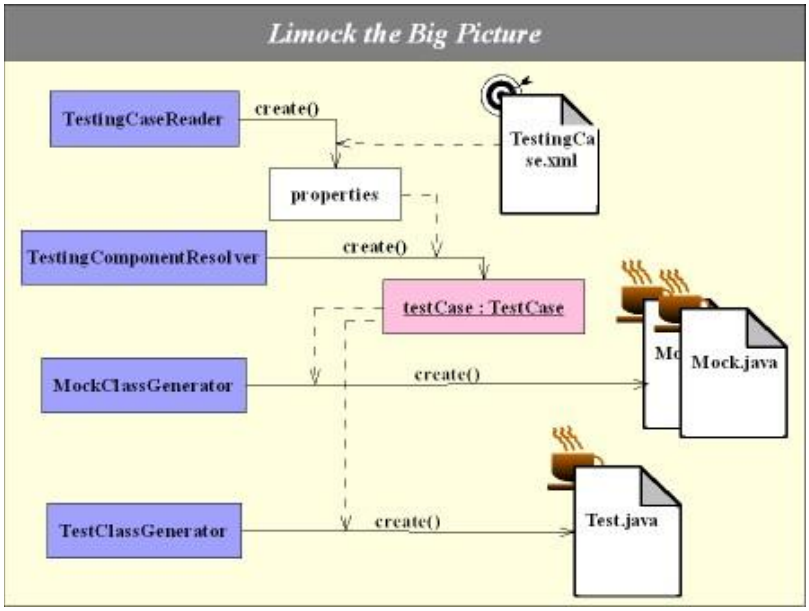

Figure 5

4.2 Step into the XML

If the code generation doesn't catch too much of your attention and interests, a close look of the XML will prove that LiMock is your friend of testing. Going through the list 2 one step at a time, it won't take long for you to realize how testing, and just testing no programming, becomes easy.

List 2: TestCensus.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<test>
        <name>Census</name>
        <src_path>C:\Users\XuanLi\Programming\workspace\IBT\src</src_path>
        <class_path>edu.leiden.msc.limock.test.TestCensus</class_path>
        <cuts><cut>
        <src_path>C:\Users\XuanLi\Programming\workspace\IBT\src</src_path>
        <class_path>edu.leiden.msc.limock.test.assets.census.cut.Census</class_path>
        <methods><method>
                <name>census</name>
                <parameters>
                        <parameter>
                <type>java.util.HashSet</type>
                <value>voters</value>
                        </parameter>
                </parameters>
                </method>
                <method>
                <name>report</name>
                <parameters>
                <parameter>
                <type>java.lang.Integer</type>
        <value>voter1.hashCode()</value>
                </parameter>
                </parameters>
                </method>
                <method>
                <name>report</name>
                <parameters>
                        <parameter>
                <type>java.lang.Integer</type>
        <value>voter2.hashCode()</value>
                </parameter>
                </parameters>
                </method>
                …
        </methods>
        </cut></cuts>
        <mocks><mock>
        <src_path>C:\Users\XuanLi\Programming\workspace\IBT\src</src_path>
        <class_path>edu.leiden.msc.limock.test.assets.census.Voter</class_path>
        <instances><instance>
        <collection_class>java.util.LinkedHashSet</collection_class>
        <value>voters</value>
        <parameters/>
        </instance>
        <instance>
                <collection_class/>
                <value>voter1</value>
                <parameters>
                        <parameter>
                <type>java.lang.String</type>
                <value>"a"</value>
                        </parameter>
                </parameters>
</instance>
<instance>
        <collection_class></collection_class>
        <value>voter2</value>
        <parameters>
                <parameter>
        <type>java.lang.String</type>
        <value>"b"</value>
                </parameter>
        </parameters>
</instance>
…
</instances>
</mock>
        <initializations>
                <initialization>
        <instance>voters</instance>
        <name>add</name>
        <parameters>
                <parameter>
        <type>java.lang.Object</type>
        <value>voter1</value>
                </parameter>
        </parameters>
                </initialization>
                <initialization>
        <instance>voters</instance>
        <name>add</name>
        <parameters>
                <parameter>
        <type>java.lang.Object</type>
        <value>voter2</value>
                </parameter>
        </parameters>
                </initialization>
                …
        </initializations>
        <expectations>
                <expectation>
        <instance>voter1</instance>
        <name>vote</name>
        <parameters/>
                </expectation>
                <expectation>
        <instance>voter2</instance>
        <name>vote</name>
        <parameters/>
                </expectation>
```

```
            …
            </expectations>
        </mocks>
</test>
```

First of all, the standard xml header is applied. The test case is inside a "test" root. The name, source path - file path (src_path) and class path - Java package name (class_path) will be used to define the testing code, output of code generation. The definition of CUTs comes after that. One test can have as many CUTs as wanted. Each of them is specified with their source path, class path and the interactions/methods that need to be tested. Methods are recognized by their signatures that include name (name), parameter list (parameters) and exception list (not fully implemented yet). All the parameters have a type and the value of its type. Mock objects use the same tags as source path and class path. Multiple instances can be instantiated for one mock class in order to fullfil the testing requirment. Besides the value and a parameter list, an instance may also have a collection type (collection_class) which is a concrete classes in the Java Collection API. Some initializations can be applied after the declaration of mock objects if necessary.

Right after all the mock objects are done, the test sequence comes on the scene. Since all that defined here are supposed to happen in the test, the tags are precisely enough to be named as "expectation". With exactly the same structure as initialization, each expectation has an instance (instance) to call a service (by name) with a parameter list (parameters). The expectations follow the order as they are defined in the XML file. And that's it. Save the file and pass it to TestClassGenerator, the testing code should be right there after LiMock has done the job for you. (Ellipses in the list 2 indicate omitting tag definitions of the same tag group, e.g. omit method in methods)

List 3 LiMock: MockVoter Class

```
1.  @Override
2.  public java. lang. String vote() {
3.          Method thisMethod = null;
4.          Invocation invocation = null;
5.          try {
6.                  thisMethod = this. getClass(). getMethod ("vote", (Class[]) null);
7.                  invocation = Trace. getInstance(). getInvocationMap(). get( this. hashCode());
8.                  invocation. setReturnValue (new ReturnValue (thisMethod. getGenericReturnType()));
9.          } catch (Exception e) {
10.                 e. getMessage();
11.                 System.exit(1);
12.         }
13.         Trace. getInstance(). notify( invocation);
14.         try {
15.                 String result = ""+super. vote(); invocation. getReturnValue(). setValue(result); return result;
16.         } catch (Exception e) {
17.                 e. printStackTrace();
18.         }
19.         return "";
20. }
```

4.3Running Result

At this time, a complete implementation is carried out from a rigorous testing design. Then running result attracts attention with the questions like “How LiMock works? Does it give a success result as expected?”

Running from Eclipse 3.2 with jdk1.6 (jdk1.6 provides the XML processing API as mentioned). After TestCensus.xml in Appendix 1 was defined, MockVoter.java was generated with number of records of Voter’s voting and a compatible testing case (TestCensus.java). The testing case deploys a kind of reporting mechanism, found in line 13 and 15 of List 3, which notifies the trace object when vote method

gets called at run time and also records the return value of the method, if any, or the exception thrown. Besides that, JUnit is used to provide the assertion API and the default main method. The Figure 6 gives an overview of this:

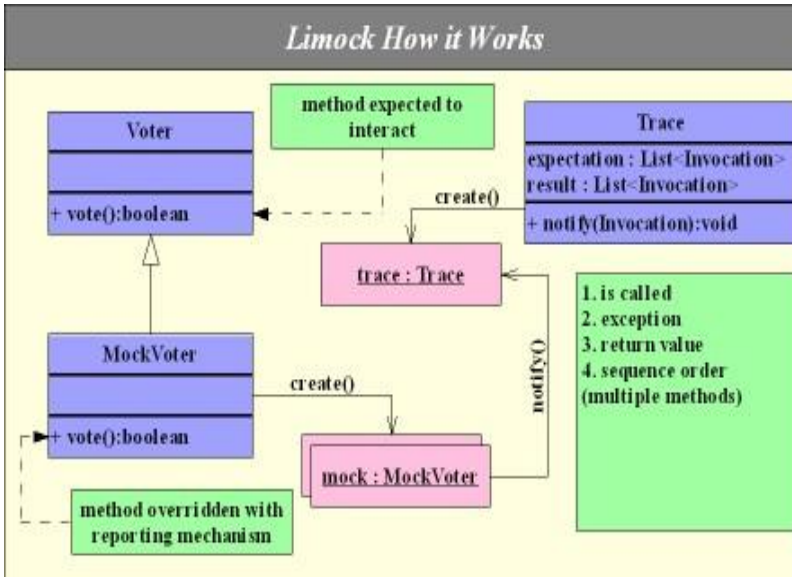

Figure 6

In practice, I set up 11 voters and would like census records 11 reports with return values. The result from eclipse JUnit execution console:

```
ivoking: census.census(voters)
ivoking:
census.report(voter1.hashCode()) returns
a
ivoking:
census.report(voter2.hashCode()) returns
b
ivoking:
census.report(voter3.hashCode()) returns
c
ivoking:
census.report(voter4.hashCode()) returns
d
ivoking:
census.report(voter5.hashCode()) returns
e
ivoking:
census.report(voter6.hashCode()) returns
f
ivoking:
census.report(voter7.hashCode()) returns
g
ivoking:
census.report(voter8.hashCode()) returns
h
ivoking:
census.report(voter9.hashCode()) returns
i
ivoking:
census.report(voter10.hashCode())
returns j
ivoking:
census.report(voter11.hashCode())
returns k

expected@1MockVoter[voter1].vote returns
a
expected@2MockVoter[voter2].vote returns
b
expected@3MockVoter[voter3].vote returns
c
expected@4MockVoter[voter4].vote returns
d
expected@5MockVoter[voter5].vote returns
e
expected@6MockVoter[voter6].vote returns
f
expected@7MockVoter[voter7].vote returns
g
expected@8MockVoter[voter8].vote returns
h
expected@9MockVoter[voter9].vote returns
i
expected@10MockVoter[voter10].vote
returns j
expected@11MockVoter[voter11].vote
returns k

Expected sequence size: 11
Result sequence size: 11
```

"Invoking: census.census(voters)" represents that Voter is called by Census, and "ivoking: Census.report(voter1.hashCode()) returns a…" means that the census reports the value "a" from the unique object voter1. Then look at lower description "expected@1MockVoter [voter1].vote returns a" to see the expectation of tester. Compare the two return values "a", the same results are obtained. And Census always correctly

reports all the voting of Voters. It verifies that Census & Voter successfully interacts with each other.

5.EXPERMENT

The above Census-Voter interaction testing is only testing interaction between SUT and its dependent environment. Further, there is something more that we can test via using LiMock. For instance, through the analysis the contribution of interaction, the in-depth interaction in container (CUT) is possible to be detected more or less. This time Cenus becomes CUT instead of SUT. See again Figure 1. Internal Interaction.

5.1 Interaction Analysis

An interaction takes place when it is invoked at run time. In programming, one interaction often needs to invoke one or more other interactions in order to complete its tasks. The interaction could happen from the interface level but also happens inside the systems with not unique object.

In-depth interactions are usually not directly visible during testing because of its nature. Whereas, it is still possible to find out a way to investigate their mystery with the information interface interaction reveals under the black box condition. Using the help of class inheritance and method override, for instance, the modified MockVoter Object is synchronous with Voter Object for inheritance. The mock objects are the ones instantiated during testing. Upon invocation, the testing mechanism is not only able to mark the calling trace with hash code and sequence of intended or expected method(s) but also record the return value(s), if any, or exception thrown. And the sub-classing layering provides isolation between the super class and mock class. Any changes that are made to the super class are automatically inherited in the mock class, for instance, if original Voter class is changed, its subclass MockVoter is synchronously changed. Consequently code redundancy is minimized and testing code maintainability is greatly improved as well.

5.2Internal Interaction Problem

A simple example can illustrate this. In CUT, we don't know how many objects with how many methods inside. The CUT already works with auxiliary code A.java. The CUT also defines 3 different initialization methods but not known by a tester. It is a completely black box even no given interface methods. If the tester has to check the methods or sequence inside the CUT as expected or not, what he can do? Is it possible? The answer is "Yes" from interaction analysis.

5.3Internal Interaction Solution & Result

LiMock can also be applied to this case. The whole SUT can be treated as one object A. An XML file like TestSimple.xml in Appendix 2, which mocks the A.java and also defined the expected methods, can be the tester's input. Then the same framework of LiMock shows its effect and generated new MockA.java & TestSimple.java for testing some of internal methods inside CUT (In Depth Tesing). And the running result:

```
ivoking:cut.setUp(a)
edu.leiden.msc.limock.test.assets.simple
.A init1() called
edu.leiden.msc.limock.test.assets.simple
.A init2() called
edu.leiden.msc.limock.test.assets.simple
.A init3() called
expected@1MockA[a].init1 returns
expected@1MockA[a].init2 returns
expected@1MockA[a].init3 returns
Expected sequence size: 3
Result sequence size: 3
```

From that, the tester can assure of that init1(), init2(), init3() methods inside CUT are called in sequence as expected. Of course, this doesn't stand for everything inside CUT and it is still difficult to testing all the interaction inside CUT with details if the internal interaction does not influence outside interactions. Maybe in near future, the theory can be more mature, more interactions and their sequences and combinations can be tested.

6.INVESTIGATION

Moreover, I did investigation for the idea about "Testing Based Sequence Diagram" (initial idea of this project) for curiosity about if there are other related references. At last, I found Software Engineering Research Croup of Darmstadt University of Technology (Germany). They ever developed a tool called SeDiTec that also uses UML sequence diagrams for testing. The concept of using sequence diagrams as test specification is also applied to fulfill the demand of software development processes. As they have already mentioned, "because of the fact that the sequence diagrams were created in an early stage of software development and only needed to be complemented by test case data sets to specify tests in an intuitive ways, the test specification can be one of the first activities; therefore can test earlier and more efficiently". [4] However, the main differences between LiMock are that they used one-way sequence - there is no return method or value from SUT to tester. And actually one-way is not ideal for interaction detecting. And both SeDiTec and LiMock could be further developed for Java applications testing with user-friendly graphical user interfaces (GUI) which requires a lot of more work thought.

7.COMPARSION

7.1Traditional Testing Comparison

Traditional testing generally is done after the software project completed. The software bugs are under fix only after testers report. Therefore, many projects needs experienced tester to corporate with project design in order to avoid some predictable mistakes. Whereas, with the help of LiMock testing can be carried out as soon as one part of development is ready and don't need to wait all the development is finished, the testing will become more effective.

7.2Junit Comparison

Junit is a state-based testing framework implemented using Java programming language. It facilitates changes, simplifies integration and provides living document of the system [9]. Junit is only able to validate a return value from a method call or a change in a property of the object being tested. Even with Junit's own mock object in action, it cannot verify whether the interaction take place or not. Furthermore, it doesn't support automated testing. LiMock is designed to utilize the advantages of Junit. The main test class is actually derived from the Junit's TestCase class. LiMock works as an intelligent robot which helps to program and set up the testing input as ordered. It is obvious that LiMock is as quick to start as Junit and has more added values to verify the states and interactions in an automated test. [7]

7.3 Jmock Comparison

Jmock is another mock-object framework, in conjunction with Test-Driven Development. And the mock objects help design and test the relations between the objects entangling

the whole system. [5] In practice, I ever used Jmock library, and test its usage. It can verify that the expected interaction via method calls took place (Are all votes from voters correctly received by Census?), but it cannot test in-depth interaction discussed in chapter 5. Even if it can test part of external interactions, some knowledge of classes at source code level is required and tester also has to works as programmer for testing. This can be illustrated by the example code MockTest.java. To test the callTest method, the implementation details of the method has to be known. As highlighted in the list 4, it obvious becomes the white box testing and tester ought to know some programming. It also has to be pointed out that the duplication of the implementation code in the testing class will raise maintenance issues in the long run.

List 4: MockTest.java

```
package org.leiden.msc.test;

import org.jmock.Expectations;
import org.jmock.Mockery;
import org.jmock.Sequence;
import
org.jmock.lib.legacy.ClassImposteriser;

import junit.framework.TestCase;

public class MockTest extends TestCase {

      Mockery context = new Mockery(){{

setImposteriser(ClassImposteriser.INSTAN
CE);
    }};

      public void testCallTest() {
            final TestCall test =
context.mock(TestCall.class);
            final Sequence sequence =
context.sequence("s");
            context.checking(new
Expectations() {{
      one(test).test1();inSequence(seque
nce);
      one(test).test2();inSequence(seque
nce);
            }});
            test1();
            test2();
            context.assertIsSatisfied();
      }
}

class TestCall implements Test {
      public void callTest() {
            test1();
            test2();
      }
      public boolean test1() {
            System.out.println("test1
called");
            return false;
      }

      public void test2() {
            System.out.println("test2
called");
      }
}

interface Test {
      boolean test1();
      void test2();
}
```

8.ADVANTAGE

Comparing the above several testing method, LiMock shows its own advantages like:

- Interactions and in-depth interactions both can be monitored. The interaction between the Tester & Component can not only be detected, the internal methods of CUT also can be known.
- Isolate development code and testing code to minimize code redundancy and avoid copy-paste in testing. With Java inheritance, Mock object can directly automately gets the updated information from the super object.
- Automate testing code generation to maximize reusability and maintainability.
- No programming skills required. Only the xml sequence is filled, the whole testing process can be

realized. And the xml editing can be further facilitated with the help of GUI. When the testing case sequence button is pressed, the testing behavior happens and reaction would be automated generated.

9.CONCLUSIONS AND EXPECTATION

9.1Related Work

Above all, to analyze the case Census-Voter, I developed my own testing framework utilizing mock object – "LiMock" for testing interactions based on sequence diagram. Besides that, I also investigated the similar work with the same testing base - "Testing based Sequence Diagram". After that, through the comparison with traditional testing, Junit and Jmock, results are clearly in LiMock's favour regarding the whole procedure of generating testing code from testing-based sequence diagram. I believe it will have good prospect in near future in the area of software testing, especially for the automated testing.

9.2Future Work

As mentioned, GUI can be further improved (GUI for xml editing), as well as the definition of a collection of objects. Some compiling scripts, such as Apache ant, can be added for compiling and execution of large-scale project. The actual implementation does not yet support loop for sequence, for example there should be an easy way to define the number of the voters to be input and their property values and voters are accessed in loop instead of each object has its own call.

Further, the whole project expect XML input is assuming testing using Java langauge, the testing base sequence diagram may also fit other programming languege.

Last but not least, I have ever asked the advice of experienced QA Engineer & Coordinator of my working company. She said, "it maybe seldom happens in current industry, because generally the client always buy the complete software application , they rarely care about interaction; conversely the software companies often care about the interaction testing for the software development." For the practical software Quality Assurance (QA), a wonderful testing should not only involves checking the code like white box testing, but also black box testing from the anchor points of users. Thus, no one can predict the use of the specific testing method. And an idea or new technology always starts from this...

10.REFERENCES

[1] ‘Abrah’am, E., de Boer.F.S, Bonsanque. M.M., Gruner. A., Steffen,M: Observability , connectivity, and replay in a sequential calculus of classes. In Proceedings of FMCO2004, LNCS 3657, pp. 296–316. Springer (2005).

[2] ´Abrah´am, E., Bonsangue, M., de Boer, F.S., Steffen, M.:Object connectivity and full abstraction for a Concurrent calculus of classes. In: ICTAC’04, LNCS 3407 04).

[3] Author unknown (CWI PDF): Generating Mock Classes From Observational Traces for Testing Java (2007)

[4] Falk Fraikin, Thomas Leonhardt, Software Engineering Research Group, Darmstadt University of Technology: SeDiTeC- Testing Based on Sequence Diagrams (2002)

[5] jMock. www.jmock.org (2007).
URL http://www.jmock.org

[6] The Maude web page.
http://www.csl.sri.com/projects/maude (2005). O

[7] Vincent Massol with Ted Husted, JUnit in action, Manning Publications Co. (2004)

[8]Dorothy Graham, Erik Van Veenendaal, Isabel Evans, Rex Black: Foundations of Software Testing,, Cengage Learning EMEA (2008)

[9] Wikipedia: Unit Testing
http://en.wikipedia.org/wiki/Unit_testing

[10] Wikipedia: Mock Object
http://en.wikipedia.org/wiki/Mock_object

APPENDIX

APPENDIX 1

Census-Voter Case:

Census.java (PROVIDED CUT)

```
package edu.leiden.msc.limock.test.assets.census.cut;

import java.util.HashSet;
import java.util.Hashtable;

import edu.leiden.msc.limock.test.assets.census.Voter;

public class Census {
	private Hashtable<Integer, String> report = new Hashtable<Integer, String>();

	public void census(HashSet<Voter> voters) {
		for (Voter v : voters) {
			String result = v.vote();
			report.put(v.hashCode(), result);
		}
	}

	public String report(Integer key) {
		return report.get(key);
	}

	public void report() {
		for (Integer key : report.keySet()) {
			System.out.println("++" + key + ": " + report(key));
		}
	}

}
```

TestCensus.xml (INPUT[See List 2])

Voter.java (AUXILLIARY CODE)

```
package edu.leiden.msc.limock.test.assets.census;

public class Voter {

	private String name;
	private String vote;

	public Voter(String vote) {
		name = "voter@" + hashCode();
		this.vote = vote;
	}
	public String getName() {
		return name;
	}
	public String vote() {
		return vote;
	}
	public void setVote(String vote) {
		this.vote = vote;
	}
	@Override
	public boolean equals(Object obj) {
		if (this == obj) {
			return true;
		}
		if (obj != null && obj instanceof Voter) {
			Voter temp = (Voter)obj;
			return vote.equals(temp.vote());
		}
		return false;
	}
	@Override
	protected Object clone() throws CloneNotSupportedException {
		return new Voter(vote);
	}
	@Override
	public String toString() {
		return vote;
	}

}
```

MockVoter.java (GENERATED CODE)

```
package
edu.leiden.msc.limock.test.assets.census
;

import java.lang.reflect.Method;

import
edu.leiden.msc.limock.model.ReturnValue;
import
edu.leiden.msc.limock.model.Trace;
import
edu.leiden.msc.limock.model.Invocation;

public class MockVoter extends Voter {

	public MockVoter(java.lang.String
arg0) {
		super(arg0);
	}

	@Override
	public java.lang.String vote() {
		Method thisMethod = null;
		Invocation invocation =
null;
		try {
			thisMethod =
this.getClass().getMethod("vote",
(Class[])null);
			invocation =
Trace.getInstance().getInvocationMap().g
et(this.hashCode());

	invocation.setReturnValue(new
ReturnValue(thisMethod.getGenericReturnT
ype()));
		} catch (Exception e) {
			e.getMessage();
			System.exit(1);
		}

	Trace.getInstance().notify(invocat
ion);
		try {
			String result = "" +
super.vote();invocation.getReturnValue()
.setValue(result);return result;
		} catch (Exception e) {
			e.printStackTrace();
		}
		return "";
	}
}
```

TestCensus.java (TEST CODE)

```
package edu.leiden.msc.limock.test;

import java.util.Iterator;

import org.junit.Before;
import org.junit.Test;

import
edu.leiden.msc.limock.model.Invoker;
import
edu.leiden.msc.limock.model.Trace;
import
edu.leiden.msc.limock.model.Invocation;
import
edu.leiden.msc.limock.test.assets.census
.cut.Census;
import
edu.leiden.msc.limock.test.assets.census
.Voter;
import
edu.leiden.msc.limock.test.assets.census
.MockVoter;

public class TestCensus {

	private Trace trace;
	private Census census;
	private
java.util.LinkedHashSet<Voter> voters;
	private Voter voter1;
	private Voter voter2;
	private Voter voter3;
	private Voter voter4;
	private Voter voter5;
	private Voter voter6;
	private Voter voter7;
	private Voter voter8;
	private Voter voter9;
	private Voter voter10;
	private Voter voter11;

	@Before
	public void setUp() throws
Exception {
		trace = Trace.getInstance();
		census = new Census();
		voters = new
java.util.LinkedHashSet<Voter>();
		voter1 = new MockVoter("a");
		voter2 = new MockVoter("b");
		voter3 = new MockVoter("c");
		voter4 = new MockVoter("d");
```

```
            voter5 = new MockVoter("e");
            voter6 = new MockVoter("f");
            voter7 = new MockVoter("g");
            voter8 = new MockVoter("h");
            voter9 = new MockVoter("i");
            voter10 = new
MockVoter("j");
            voter11 = new
MockVoter("k");
            voters.add(voter1);
            voters.add(voter2);
            voters.add(voter3);
            voters.add(voter4);
            voters.add(voter5);
            voters.add(voter6);
            voters.add(voter7);
            voters.add(voter8);
            voters.add(voter9);
            voters.add(voter10);
            voters.add(voter11);
            try {
                  trace.expect(new
Invoker(voter1.getClass(), "voter1",
voter1.hashCode()), "vote",
(Object[])null);
                  trace.expect(new
Invoker(voter2.getClass(), "voter2",
voter2.hashCode()), "vote",
(Object[])null);
                  trace.expect(new
Invoker(voter3.getClass(), "voter3",
voter3.hashCode()), "vote",
(Object[])null);
                  trace.expect(new
Invoker(voter4.getClass(), "voter4",
voter4.hashCode()), "vote",
(Object[])null);
                  trace.expect(new
Invoker(voter5.getClass(), "voter5",
voter5.hashCode()), "vote",
(Object[])null);
                  trace.expect(new
Invoker(voter6.getClass(), "voter6",
voter6.hashCode()), "vote",
(Object[])null);
                  trace.expect(new
Invoker(voter7.getClass(), "voter7",
voter7.hashCode()), "vote",
(Object[])null);
                  trace.expect(new
Invoker(voter8.getClass(), "voter8",
voter8.hashCode()), "vote",
(Object[])null);
                  trace.expect(new
Invoker(voter9.getClass(), "voter9",
voter9.hashCode()), "vote",
(Object[])null);
                  trace.expect(new
Invoker(voter10.getClass(), "voter10",
voter10.hashCode()), "vote",
(Object[])null);
                  trace.expect(new
Invoker(voter11.getClass(), "voter11",
voter11.hashCode()), "vote",
(Object[])null);
            } catch
(NoSuchMethodException e) {
                  e.printStackTrace();
                  System.exit(1);
            }
      }

      @Test
      public void testCensus() {
            System.out.print("ivoking:
census.census(voters)");
            census.census(voters);
            System.out.print("ivoking:
census.report()");
            census.report();
            System.out.print("ivoking:
census.report(voter1.hashCode())");
            System.out.println(" returns
" + census.report(voter1.hashCode()));
            System.out.print("ivoking:
census.report(voter2.hashCode())");
            System.out.println(" returns
" + census.report(voter2.hashCode()));
            System.out.print("ivoking:
census.report(voter3.hashCode())");
            System.out.println(" returns
" + census.report(voter3.hashCode()));
            System.out.print("ivoking:
census.report(voter4.hashCode())");
            System.out.println(" returns
" + census.report(voter4.hashCode()));
            System.out.print("ivoking:
census.report(voter5.hashCode())");
            System.out.println(" returns
" + census.report(voter5.hashCode()));
            System.out.print("ivoking:
census.report(voter6.hashCode())");
            System.out.println(" returns
" + census.report(voter6.hashCode()));
            System.out.print("ivoking:
census.report(voter7.hashCode())");
            System.out.println(" returns
" + census.report(voter7.hashCode()));
            System.out.print("ivoking:
census.report(voter8.hashCode())");
            System.out.println(" returns
" + census.report(voter8.hashCode()));
            System.out.print("ivoking:
census.report(voter9.hashCode())");
```

```
            System.out.println(" returns
" + census.report(voter9.hashCode()));
            System.out.print("ivoking:
census.report(voter10.hashCode())");
            System.out.println(" returns
" + census.report(voter10.hashCode()));
            System.out.print("ivoking:
census.report(voter11.hashCode())");
            System.out.println(" returns
" + census.report(voter11.hashCode()));
            Iterator<Invocation>
expected =
trace.expectationSequence().iterator();
            int index = 1;
            while (expected.hasNext()) {
                  Invocation
expectedNext = expected.next();

      System.out.println("expected@" +
index++ + expectedNext.toString());
            }
            System.out.println("Expected
sequence size: " +
trace.expectationSequence().size());
            System.out.println("Result
sequence size: " +
trace.resultSequence().size());
      }
}
```

APPENDIX 2

Simple Case:

CUT.java (Container)

```
package
edu.leiden.msc.limock.test.assets.simple
.cut;

import
edu.leiden.msc.limock.test.assets.simple
.A;

public class CUT {
      public void setUp(A a) {
            a.init1();
            a.init3();
            a.init2();
      }
}
```

A.java (AUXUILIARY CODE)

```
package
edu.leiden.msc.limock.test.assets.simple;

public class A {
      public void init1() {

      System.out.println(A.class.getName
() + " init1() called");
      }

      public void init2() {

      System.out.println(A.class.getName
() + " init2() called");
      }

      public void init3() {

      System.out.println(A.class.getName
() + " init3() called");
      }
}
```

TestSimple.xml (INPUT FILE)

```
<?xml version="1.0" encoding="UTF-8"?>
<test>
```

```
	<name>Simple</name>
	<src_path>C:\LX_MscThesis\workspac
e\Msc_JavaInteractionTesting\src</src_pa
th>
	<class_path>edu.leiden.msc.limock.
test.TestSimple</class_path>
	<cuts>
		<cut>

	<src_path>C:\Users\XuanLi\Programm
ing\workspace\IBT\src</src_path>

	<class_path>edu.leiden.msc.limock.
test.assets.simple.cut.CUT</class_path>
			<methods>
				<method>

	<name>setUp</name>

	<parameters>

	<parameter>

	<type>edu.leiden.msc.limock.test.a
ssets.simple.A</type>

	<value>a</value>

	</parameter>

	</parameters>
				</method>
			</methods>
		</cut>

	</cuts>
	<mocks>
		<mock>

	<src_path>C:\LX_MscThesis\workspac
e\Msc_JavaInteractionTesting\src</src_pa
th>

	<class_path>edu.leiden.msc.limock.
test.assets.simple.A</class_path>
			<instances>
				<instance>

	<collection_class/>

	<value>a</value>

	<parameters/>
				</instance>
			</instances>
		</mock>
		<initializations/>
			<expectations>
				<expectation>

	<instance>a</instance>

	<name>init1</name>
					<parameters/>
				</expectation>
				<expectation>

	<instance>a</instance>

	<name>init3</name>
					<parameters/>
				</expectation>
				<expectation>

	<instance>a</instance>

	<name>init2</name>
					<parameters/>
				</expectation>
			</expectations>
	</mocks>
</test>
```

MockA.java (GENERATED MOCK A)

```
package
edu.leiden.msc.limock.test.assets.simple
;


import java.lang.reflect.Method;


import
edu.leiden.msc.limock.model.ReturnValue;
import
edu.leiden.msc.limock.model.Trace;
import
edu.leiden.msc.limock.model.Invocation;


public class MockA extends A {


	public MockA() {
		super();
	}


	@Override
	public void init1() {
		Method thisMethod = null;
```

```
            Invocation    invocation    =
null;
            try {
                  thisMethod           =
this.getClass().getMethod("init1",
(Class[])null);
                  invocation           =
Trace.getInstance().getInvocationMap().g
et(this.hashCode()+thisMethod.getName())
;

      invocation.setReturnValue(new
ReturnValue(thisMethod.getGenericReturnT
ype()));
            } catch (Exception e) {
                  e.getMessage();
                  System.exit(1);
            }

      Trace.getInstance().notify(invocat
ion);
            try {
                  super.init1();
            } catch (Exception e) {
                  e.printStackTrace();
            }
            return;
      }

      @Override
      public void init3() {
            Method thisMethod = null;
            Invocation    invocation    =
null;
            try {
                  thisMethod           =
this.getClass().getMethod("init3",
(Class[])null);
                  invocation           =
Trace.getInstance().getInvocationMap().g
et(this.hashCode()+thisMethod.getName())
;

      invocation.setReturnValue(new
ReturnValue(thisMethod.getGenericReturnT
ype()));
            } catch (Exception e) {
                  e.getMessage();
                  System.exit(1);
            }

      Trace.getInstance().notify(invocat
ion);
            try {
                  super.init3();
            } catch (Exception e) {
                  e.printStackTrace();
            }
            return;
      }

      @Override
      public void init2() {
            Method thisMethod = null;
            Invocation    invocation    =
null;
            try {
                  thisMethod           =
this.getClass().getMethod("init2",
(Class[])null);
                  invocation           =
Trace.getInstance().getInvocationMap().g
et(this.hashCode()+thisMethod.getName())
;

      invocation.setReturnValue(new
ReturnValue(thisMethod.getGenericReturnT
ype()));
            } catch (Exception e) {
                  e.getMessage();
                  System.exit(1);
            }

      Trace.getInstance().notify(invocat
ion);
            try {
                  super.init2();
            } catch (Exception e) {
                  e.printStackTrace();
            }
            return;
      }
```

```
}
```

TestSimple.java (GENERATED MOCK A)

```
package edu.leiden.msc.limock.test;

import java.util.Iterator;

import org.junit.Before;
import org.junit.Test;

import
edu.leiden.msc.limock.model.Invoker;
import edu.leiden.msc.limock.model.Trace;
import
edu.leiden.msc.limock.model.Invocation;
import
edu.leiden.msc.limock.test.assets.simple
.cut.CUT;
import
edu.leiden.msc.limock.test.assets.simple
.A;
import
edu.leiden.msc.limock.test.assets.simple
.MockA;

public class TestSimple {

      private Trace trace;
      private CUT cut;
      private A a;

      @Before
      public    void    setUp()    throws
Exception {
            trace = Trace.getInstance();
            cut = new CUT();
            a = new MockA();
            try {
                  trace.expect(new
Invoker(a.getClass(), "a", a.hashCode()),
"init1", (Object[])null);
                  trace.expect(new
Invoker(a.getClass(), "a", a.hashCode()),
"init3", (Object[])null);
                  trace.expect(new
Invoker(a.getClass(), "a", a.hashCode()),
"init2", (Object[])null);
            }                        catch
(NoSuchMethodException e) {
                  e.printStackTrace();
                  System.exit(1);
            }
      }

      @Test
      public void testSimple() {
            System.out.println("ivoking:
cut.setUp(a)");
            cut.setUp(a);
            Iterator<Invocation>
expected                                =
trace.expectationSequence().iterator();
            int index = 1;
            while (expected.hasNext()) {
                  Invocation
expectedNext = expected.next();

      System.out.println("expected@"   +
index++ + expectedNext.toString());
            }
            System.out.println("Expected
sequence         size:         "         +
trace.expectationSequence().size());
            System.out.println("Result
sequence         size:         "         +
trace.resultSequence().size());
      }
}
```

APPENDIX 3

LiMock:

CONTRL:

TestingCaseReader.java

```
package edu.leiden.msc.limock.control;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.InputStream;
import java.util.Properties;

import javax.xml.stream.XMLEventReader;
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamException;
import javax.xml.stream.events.XMLEvent;

/**
 * This class utilizes Java Streaming API for XML (Stax),
 * introduced in Java 6.0, for processing testing case XML file.
 * It goes through the testing case XML file and generates the
 * properties for further usage.
 *
 * @author Xuan Li
 *
 */
public class TestingCaseReader {

	private Properties props;
	private File testingCaseFile;

	public TestingCaseReader(String filePath) {
		props = new Properties();
		testingCaseFile = new File(filePath);
	}

	/**
	 * Processes the specified XML file of testing case and generates
	 * the properties which can be get with <code>getProps</code>.
	 */
	public void process() {
		try {
			// First create a new XMLInputFactory
			XMLInputFactory inputFactory = XMLInputFactory.newInstance();
			// Setup a new eventReader
			InputStream in = new FileInputStream(testingCaseFile);
			XMLEventReader eventReader = inputFactory.createXMLEventReader(in);
			// Read the XML document
			while (eventReader.hasNext()) {
				XMLEvent event = eventReader.nextEvent();
				if (event.isStartElement()) {
					if (event.asStartElement().getName().getLocalPart() == ("name")) {
						event = eventReader.nextEvent();
						String eventData = event.asCharacters().getData();

	System.out.println(eventData);

	props.setProperty("test.name", eventData);

	continue;
					}
					if (event.asStartElement().getName().getLocalPart() == ("src_path")) {
						event = eventReader.nextEvent();
						String eventData = event.asCharacters().getData();

	System.out.println(eventData);

	props.setProperty("test.src_path", eventData);

	continue;
					}
					if (event.asStartElement().getName().getLocalPart() == ("class_path")) {
						event = eventReader.nextEvent();
						String eventData = event.asCharacters().getData();
```

```
            System.out.println(eventData);

            props.setProperty("test.class_path", eventData);

            continue;
                                    }
                                    if
(event.asStartElement().getName().getLocalPart() == ("cuts")) {
                                                event
= nextStartEvent(eventReader);
                                                int
numOfCUTs = 0;
                                                while
(event.isStartElement()                         &&
event.asStartElement().getName().getLocalPart() == ("cut")) {

            event = processCUT(eventReader, numOfCUTs);

            numOfCUTs++;
                                                }

            System.out.println("cuts: " + event + " " +
numOfCUTs);

            props.setProperty("cut.num", "" + numOfCUTs);
                                    }
                                    if
(event.asStartElement().getName().getLocalPart()  ==  ("mocks"))
{
                                                event
= nextStartEvent(eventReader);
                                                int
numOfMocks = 0;
                                                while
(event.isStartElement()                         &&
event.asStartElement().getName().getLocalPart() == ("mock")) {

            event = processMock(eventReader, numOfMocks);

            numOfMocks++;
                                                }

            props.setProperty("mock.num", "" + numOfMocks);
                                                if
(event.asStartElement().getName().getLocalPart()            ==
("initializations")) {

            event = nextStartEvent(eventReader);

            int numOfinitializations = 0;

            while (event.isStartElement() &&
event.asStartElement().getName().getLocalPart()            ==
("initialization")) {
//
                        event = processExpectation(eventReader,
numOfinitializations, "");

                        event = processInitialization(eventReader,
numOfinitializations);

                        numOfinitializations++;

            }

            props.setProperty("initialization.num",        ""        +
numOfinitializations);
                                    }
                                                if
(event.asStartElement().getName().getLocalPart()            ==
("expectations")) {

            event = nextStartEvent(eventReader);

            int numOfExpectations = 0;

            while (event.isStartElement() &&
event.asStartElement().getName().getLocalPart()            ==
("expectation")) {
//
                        event = processExpectation(eventReader,
numOfExpectations, "");

                        event = processExpectation(eventReader,
numOfExpectations);

                        numOfExpectations++;

            }

            props.setProperty("expectation.num",        ""        +
numOfExpectations);
                                    }

            continue;
                                }

                            }
                        }
                        System.out.println(props);
                    } catch (FileNotFoundException e) {
                        e.printStackTrace();
```

```
            } catch (XMLStreamException e) {
                e.printStackTrace();
            }
        }

        private XMLEvent nextStartEvent(XMLEventReader eventReader) throws XMLStreamException {
            XMLEvent nextStartElement = null;
            while (eventReader.hasNext()) {
                nextStartElement = eventReader.nextEvent();
                if (nextStartElement.isStartElement()) {
                    break;
                }
            }
            return nextStartElement;
        }

        private XMLEvent processCUT(XMLEventReader eventReader, int index) throws XMLStreamException {
            XMLEvent event;
            do {
                event = nextStartEvent(eventReader);
                if (event.isStartElement()) {
                    if (event.asStartElement().getName().getLocalPart() == ("src_path")) {
                        event = eventReader.nextEvent();
                        String eventData = event.asCharacters().getData();

        System.out.println("CUT: " + eventData);

        props.setProperty("cut" + index + ".src_path", eventData);
                        continue;
                    } else if (event.asStartElement().getName().getLocalPart() == ("class_path")) {
                        event = eventReader.nextEvent();
                        String eventData = event.asCharacters().getData();

        System.out.println("CUT: " + eventData);

        props.setProperty("cut" + index + ".class_path", eventData);
                        continue;
                    } else if (event.asStartElement().getName().getLocalPart() == ("methods")) {
                        event = nextStartEvent(eventReader);
                        int numOfMethods = 0;
                        while (event.isStartElement() && event.asStartElement().getName().getLocalPart() == ("method")) {
                            event = processMethod(eventReader, numOfMethods, "cut" + index);

        numOfMethods++;
                        }

        System.out.println("processCUT: " + event);

        props.setProperty("cut" + index + ".method.num", "" + numOfMethods);
                        break;
                    }
                }
            } while (eventReader.hasNext());
            return event;
        }

        private XMLEvent processMock(XMLEventReader eventReader, int index) throws XMLStreamException {
            XMLEvent event;
            do {
                event = nextStartEvent(eventReader);
                if (event.isStartElement()) {
                    if (event.asStartElement().getName().getLocalPart() == ("src_path")) {
                        event = eventReader.nextEvent();
                        String eventData = event.asCharacters().getData();

        System.out.println("Mock: " + eventData);

        props.setProperty("mock" + index + ".src_path", eventData);
```

```
                    continue;
                } else if (event.asStartElement().getName().getLocalPart() == ("class_path")) {
                    event = eventReader.nextEvent();
                    String eventData = event.asCharacters().getData();

        System.out.println("Mock: " + eventData);

        props.setProperty("mock" + index + ".class_path", eventData);
                    continue;
                } else if (event.asStartElement().getName().getLocalPart() == ("instances")) {
                    event = nextStartEvent(eventReader);

        System.out.println("Mock:(instances) " + event);
                    int numOfInstances = 0;
                    while (event.isStartElement() && event.asStartElement().getName().getLocalPart() == ("instance")) {
                        event = processInstance(eventReader, numOfInstances, "mock" + index);

        numOfInstances++;
                    }

        System.out.println("instances: " + event);

        props.setProperty("mock" + index + ".instance.num", "" + numOfInstances);
                    break;
                }
            }
        } while (eventReader.hasNext());
        return event;
    }

    private XMLEvent processInstance(XMLEventReader eventReader, int index, String componentDesignation) throws XMLStreamException {
        XMLEvent event;
        do {
            event = nextStartEvent(eventReader);
            if (event.isStartElement()) {
                if (event.asStartElement().getName().getLocalPart() == ("collection_class")) {
                    event = eventReader.nextEvent();
                    if (event.isCharacters()) {
                        String eventData = event.asCharacters().getData();

        System.out.println("Instance: " + eventData);

        props.setProperty(componentDesignation + ".instance" + index + ".collection_class", eventData);
                    }
                    continue;
                } else if (event.asStartElement().getName().getLocalPart() == ("value")) {
                    event = eventReader.nextEvent();
                    String eventData = event.asCharacters().getData();

        System.out.println("Instance: " + eventData);

        props.setProperty(componentDesignation + ".instance" + index + ".value", eventData);
                    continue;
                } else if (event.asStartElement().getName().getLocalPart() == ("parameters")) {
                    event = nextStartEvent(eventReader);

        System.out.println("processInstance: " + event);
                    int numOfParameters = 0;
                    while (event.isStartElement() && event.asStartElement().getName().getLocalPart() == ("parameter")) {
                        event = processParameter(eventReader, numOfParameters,
                                componentDesignation + ".instance" + index);

        numOfParameters++;
                    }

        System.out.println("parameters: " + event);
```

```
            props.setProperty(componentDesignation + ".instance" + index + ".parameter.num", "" + numOfParameters);
                                    break;
                                }
                            }
                        } while (eventReader.hasNext());
                        System.out.println("processInstance(return): " + event);
                        return event;
                    }

                    private XMLEvent processMethod(XMLEventReader eventReader, int index, String componentDesignation) throws XMLStreamException {
                        XMLEvent event;
                        do {
                            event = nextStartEvent(eventReader);
                            System.out.println("processMethod: " + event);
                            if (event.isStartElement()) {
                                if (event.asStartElement().getName().getLocalPart() == ("name")) {
                                    event = eventReader.nextEvent();
                                    String eventData = event.asCharacters().getData();

                    System.out.println("Method: " + eventData);

                    props.setProperty(componentDesignation + ".method" + index + ".name", eventData);
                                    continue;
                                } else if (event.asStartElement().getName().getLocalPart() == ("parameters")) {
                                    event = nextStartEvent(eventReader);

                    System.out.println("processMethod: " + event);
                                    int numOfParameters = 0;
                                    while (event.isStartElement() && event.asStartElement().getName().getLocalPart() == ("parameter")) {
                                        event = processParameter(eventReader, numOfParameters,
                        componentDesignation + ".method" + index);
                    numOfParameters++;
                                    }

                    System.out.println("parameters: " + event);

                    props.setProperty(componentDesignation + ".method" + index + ".parameter.num", "" + numOfParameters);
                                    break;
                                }
                            }
                        } while (eventReader.hasNext());
                        System.out.println("processMethod(return): " + event);
                        return event;
                    }

                    private XMLEvent processParameter(XMLEventReader eventReader, int index, String methodDesignation) throws XMLStreamException {
                        XMLEvent event;
                        do {
                            event = nextStartEvent(eventReader);
                            if (event.isStartElement()) {
                                if (event.asStartElement().getName().getLocalPart() == ("type")) {
                                    event = eventReader.nextEvent();
                                    String eventData = event.asCharacters().getData();

                    System.out.println("Parameter: " + eventData);

                    props.setProperty(methodDesignation + ".parameter" + index + ".type", eventData);
                                    continue;
                                } else if (event.asStartElement().getName().getLocalPart() == ("value")) {
                                    event = eventReader.nextEvent();
                                    String eventData = event.asCharacters().getData();

                    System.out.println("Parameter: " + eventData);

                    props.setProperty(methodDesignation + ".parameter" + index + ".value", eventData);
                                    continue;
                                } else {
                                    break;
```

```
                    }
                }
            } while (eventReader.hasNext());
            return event;
        }

        private XMLEvent processInitialization(XMLEventReader eventReader, int index/*, String componentDesignation*/) throws XMLStreamException {
            XMLEvent event;
            do {
                event = nextStartEvent(eventReader);

        System.out.println("processInitialization: " + event);
                if (event.isStartElement()) {
                    if (event.asStartElement().getName().getLocalPart() == ("instance")) {
                        event = eventReader.nextEvent();
                        String eventData = event.asCharacters().getData();

        System.out.println("Initialization: " + eventData);
//
        props.setProperty(componentDesignation + ".initialization" + index + ".instance", eventData);

        props.setProperty("initialization" + index + ".instance", eventData);
                        continue;
                    } else if (event.asStartElement().getName().getLocalPart() == ("name")) {
                        event = eventReader.nextEvent();
                        String eventData = event.asCharacters().getData();

        System.out.println("Initialization: " + eventData);
//
        props.setProperty(componentDesignation + ".initialization" + index + ".name", eventData);

        props.setProperty("initialization" + index + ".name", eventData);
                        continue;
                    } else if (event.asStartElement().getName().getLocalPart() == ("parameters")) {
                        event = nextStartEvent(eventReader);

        System.out.println("processInitialization: " + event);
                        int numOfParameters = 0;
                        while (event.isStartElement() && event.asStartElement().getName().getLocalPart() == ("parameter")) {
//                          event = processParameter(eventReader, numOfParameters,
//                                  componentDesignation + ".initialization" + index);
                            event = processParameter(eventReader, numOfParameters, "initialization" + index);

            numOfParameters++;
                        }

        System.out.println("parameters: " + event);
//
        props.setProperty(componentDesignation + ".parameter.num", numOfParameters);

        props.setProperty("initialization" + index + ".parameter.num", "" + numOfParameters);
                        break;
                    }
                }
            } while (eventReader.hasNext());
            System.out.println("processMethod(before return): " + event);
            return event;
        }

        private XMLEvent processExpectation(XMLEventReader eventReader, int index/*, String componentDesignation*/) throws XMLStreamException {
            XMLEvent event;
            do {
                event = nextStartEvent(eventReader);

        System.out.println("processExpectation: " + event);
                if (event.isStartElement()) {
                    if (event.asStartElement().getName().getLocalPart() == ("instance")) {
                        event = eventReader.nextEvent();
```

```
                    String eventData = event.asCharacters().getData();

                    System.out.println("Expectation: " + eventData);
//                  props.setProperty(componentDesignation + ".expectation" + index + ".instance", eventData);

                    props.setProperty("expectation" + index + ".instance", eventData);

                    continue;
                } else if (event.asStartElement().getName().getLocalPart() == ("name")) {
                    event = eventReader.nextEvent();
                    String eventData = event.asCharacters().getData();

                    System.out.println("Expectation: " + eventData);
//                  props.setProperty(componentDesignation + ".expectation" + index + ".name", eventData);

                    props.setProperty("expectation" + index + ".name", eventData);

                    continue;
                } else if (event.asStartElement().getName().getLocalPart() == ("parameters")) {
                    event = nextStartEvent(eventReader);

                    System.out.println("processExpectation: " + event);
                    int numOfParameters = 0;
                    while (event.isStartElement() && event.asStartElement().getName().getLocalPart() == ("parameter")) {
//                      event = processParameter(eventReader, numOfParameters,
//                              componentDesignation + ".expectation" + index);
                        event = processParameter(eventReader, numOfParameters, "expectation" + index);

                        numOfParameters++;
                    }

                    System.out.println("parameters: " + event);
//                  props.setProperty(componentDesignation + ".parameter.num", numOfParameters);

                    props.setProperty("expecation" + index + ".parameter.num", "" + numOfParameters);
                    break;
                }
            }
        } while (eventReader.hasNext());
        System.out.println("processMethod(before return): " + event);
        return event;
    }

    public Properties getProps() {
        return props;
    }

    public void setProps(Properties props) {
        this.props = props;
    }

    public File getTestingCaseFile() {
        return testingCaseFile;
    }

    public void setTestingCaseFile(File testingCaseFile) {
        this.testingCaseFile = testingCaseFile;
    }

    public static void main(String[] args) {
        TestingCaseReader reader = new TestingCaseReader(args[0]);
        reader.process();
    }
}
```

TestingComponentResolver.java

```
package edu.leiden.msc.limock.control;

import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Hashtable;
```

```
import java.util.List;
import java.util.Properties;

import edu.leiden.msc.limock.model.InstanceInfo;
import edu.leiden.msc.limock.model.Invocation;
import edu.leiden.msc.limock.model.Invoker;
import edu.leiden.msc.limock.model.ClassInfo;
import edu.leiden.msc.limock.model.MethodInfo;
import edu.leiden.msc.limock.model.ParameterInfo;
import edu.leiden.msc.limock.model.TestCase;

/**
 * This class takes the output from <code>TestingCaseReader</code> class
 * and creates objects out of them for further usage.
 *
 * @author Xuan Li
 *
 */
public class TestingComponentResolver {

	private Properties props;
	private Hashtable<String, ClassInfo> instanceToClassInfoMap;
	private TestCase testCase;
	private List<Invocation> initializations;
	private List<Invocation> expectations;
	private List<ClassInfo> cuts; // TODO there can be a call sequence for CUTs also

	public TestingComponentResolver(Properties props) {
		this.props = props;
		instanceToClassInfoMap = new Hashtable<String, ClassInfo>();
		testCase = new TestCase(

	props.getProperty("test.name"),

	props.getProperty("test.src_path"),

	props.getProperty("test.class_path")
		);
		initializations = testCase.getInitializations();
		expectations = testCase.getExpectations();
		cuts = testCase.getCuts();
	}

	public void resolve() {
		try {
			resolveCUTs();
			resolveMocks();
		} catch (Exception e) {
			System.err.println("Fatal error:");
			e.printStackTrace();
			System.exit(1);
		}
	}

	private void resolveCUTs() throws ClassNotFoundException, NoSuchMethodException {
		int numOfCUTs = 0;
		String numAsString = props.getProperty("cut.num");
		try {
			numOfCUTs = Integer.parseInt(numAsString);
		} catch (NumberFormatException e) {
			e.printStackTrace();
		}
		if (numOfCUTs == 0) {
			System.err.println("Fatal error: No component under testing (CUT) defined!");
			System.exit(1);
		} else {
			for (int i = 0; i < numOfCUTs; i++) {
				resolveCUT(i);
			}
		}
	}

	private void resolveCUT(int index) throws ClassNotFoundException, NoSuchMethodException {
		ClassInfo cut = new ClassInfo();

	cut.setSrcPath(props.getProperty("cut"+index+".src_path"));

	cut.setFullyQualifiedClassName(props.getProperty("cut"+index+".class_path"));
//
	cut.setInstanceName(props.getProperty("mock"+index+".instance"));
		cuts.add(cut);
		resolveCUTMethods(cut, index);
```

```
	}

	private void resolveCUTMethods(ClassInfo cut, int index)
	throws ClassNotFoundException, NoSuchMethodException{
		int numOfMethods = 0;
		String numInString = props.getProperty("cut" + index + ".method.num");
		try {
			numOfMethods = Integer.parseInt(numInString);
		} catch (NumberFormatException e) {
			e.printStackTrace();
		}
		if (numOfMethods == 0) {
			System.err.println("Fatal error: no expecation defined!");
			System.exit(1);
		} else {
			for (int i = 0; i < numOfMethods; i++) {
				resolveCUTMethod(cut, index, i);
			}
		}
	}

	private void resolveCUTMethod(ClassInfo cut, int cutIndex, int methodIndex)
	throws ClassNotFoundException, NoSuchMethodException {
		Class<?> cutClass = Class.forName(cut.getFullyQualifiedClassName());
		String methodName = props.getProperty("cut" + cutIndex + ".method" + methodIndex + ".name");
		Method method = null;
		int numOfParas = Integer.parseInt(props.getProperty("cut" + cutIndex + ".method" + methodIndex + ".parameter.num"));
		List<ParameterInfo> parameters = null;
		if (numOfParas > 0) {
			parameters = new ArrayList<ParameterInfo>();
			for (int i = 0; i < numOfParas; i++) {
				resolveParameter("cut" + cutIndex + ".method" + methodIndex, i, parameters);
			}
			List<Class<?>> parameterTypes = new ArrayList<Class<?>>();
			for (ParameterInfo p : parameters) {
	parameterTypes.add(p.getClassType());
			}
			method = cutClass.getMethod(methodName, parameterTypes.toArray(new Class[parameterTypes.size()]));
		} else {
			method = cutClass.getMethod(methodName, (Class[])null);
		}
		MethodInfo methodInfo = new MethodInfo();
		methodInfo.setMethod(method);
		methodInfo.setParameters(parameters);
		cut.add(methodInfo);
	}

	private void resolveMocks() throws ClassNotFoundException, NoSuchMethodException {
		int numOfMocks = 0;
		String numAsString = props.getProperty("mock.num");
		try {
			numOfMocks = Integer.parseInt(numAsString);
		} catch (NumberFormatException e) {
			e.printStackTrace();
		}
		if (numOfMocks == 0) {
			System.err.println("Fatal error: No mock class defined!");
			System.exit(1);
		} else {
			for (int i = 0; i < numOfMocks; i++) {
				resolveMock(i);
			}
			resolvInitializations();
			resolvExpectations();
		}
	}

	private void resolveMock(int index) throws ClassNotFoundException, NoSuchMethodException {
		ClassInfo mock = new ClassInfo();
```

```
        mock.setSrcPath(props.getProperty("mock"+index+".src_path"));

        mock.setFullyQualifiedClassName(props.getProperty("mock"+index+".class_path"));
                Class<?> mockClass = Class.forName(mock.getFullyQualifiedClassName());

        mock.getConstructors().addAll(Arrays.asList(mockClass.getConstructors()));
                resolveInstances(mock, "mock"+index);
                testCase.getMocks().add(mock);
        }

        private void resolveInstances(ClassInfo classinfo, String designation) throws ClassNotFoundException, NoSuchMethodException {
                int numOfInstances = 0;
                String numAsString = props.getProperty(designation + ".instance.num");
                try {
                        numOfInstances = Integer.parseInt(numAsString);
                } catch (NumberFormatException e) {
                        e.printStackTrace();
                }
                if (numOfInstances == 0) {
                        System.err.println("Fatal error: No mock class defined!");
                        System.exit(1);
                } else {
                        for (int i = 0; i < numOfInstances; i++) {
                                resolveInstance(classinfo, i, designation);
                        }
                }
        }

        private void resolveInstance(ClassInfo classInfo, int index, String designation) throws ClassNotFoundException, NoSuchMethodException {
                String instanceValue = props.getProperty(designation + ".instance" + index + ".value");
                String instanceCollectionClass = props.getProperty(designation + ".instance" + index + ".collection_class");
                Class<?> instanceClassType = null;
                if (instanceCollectionClass != null) {
                        instanceClassType = Class.forName(instanceCollectionClass);
                }
//              System.out.println("--------------------" + classInfo + " " + instanceValue + " " + instanceClassType);
                InstanceInfo instance = new InstanceInfo(classInfo, instanceValue, instanceClassType);
                int numOfParas = Integer.parseInt(props.getProperty(designation + ".instance" + index + ".parameter.num"));
                if (numOfParas > 0) {
                        List<ParameterInfo> parameters = instance.getParameters();
                        for (int i = 0; i < numOfParas; i++) {
                                resolveParameter(designation + ".instance" + index, i, parameters);
                        }
                }
                classInfo.addInstance(instance);
                if (instanceToClassInfoMap.put(instance.getValue(), classInfo) != null) {
                        System.err.println("Fatal error: duplicated mock definition of " + classInfo);
                        System.exit(1);
                }
        }

        private void resolvInitializations() throws ClassNotFoundException, NoSuchMethodException {
                int numOfInitializations = 0;
                String numInString = props.getProperty("initialization.num");
                try {
                        numOfInitializations = Integer.parseInt(numInString);
                } catch (NumberFormatException e) {
                        e.printStackTrace();
                }
                if (numOfInitializations == 0) {
                        System.out.println("No initializations defined!");
                } else {
                        for (int i = 0; i < numOfInitializations; i++) {
                                resolveInitialization(i);
                        }
                }
```

```
	}

	private void resolveInitialization(int index) throws ClassNotFoundException, NoSuchMethodException {
		String instanceName = props.getProperty("initialization" + index + ".instance");
		ClassInfo mock = instanceToClassInfoMap.get(instanceName);
		if (mock == null) {
			System.err.println("Fatal error: no such mock instance declared: " + instanceName);
			System.exit(1);
		}
		Class<?> instanceClass = null;
		InstanceInfo instance = mock.getInstance(instanceName);
		if (instance.getCollectionClass() != null) {
			instanceClass = instance.getCollectionClass();
		} else {
			instanceClass = Class.forName(mock.getFullyQualifiedClassName());
		}
		String methodName = props.getProperty("initialization" + index + ".name");
		Method method = null;
		int numOfParas = Integer.parseInt(props.getProperty("initialization" + index + ".parameter.num"));
		List<ParameterInfo> parameters = null;
		if (numOfParas > 0) {
			parameters = new ArrayList<ParameterInfo>();
			for (int i = 0; i < numOfParas; i++) {

	resolveParameter("initialization" + index, i, parameters);
			}
			List<Class<?>> parameterTypes = new ArrayList<Class<?>>();
			for (ParameterInfo p : parameters) {

	parameterTypes.add(p.getClassType());
			}
//			for (Method m : instanceClass.getMethods()) {
//				if (m.getName().contains("add")) {
//
	System.out.println(m.getParameterTypes()[0]);
//				}
//			}
			method = instanceClass.getMethod(methodName, parameterTypes.toArray(new Class[parameterTypes.size()]));
		} else {
			method = instanceClass.getMethod(methodName, (Class[])null);
		}
//		MethodInfo methodInfo = new MethodInfo();
//		methodInfo.setMethod(method);
//		methodInfo.setParameters(parameters);
//		mock.add(methodInfo);
		Invocation invocation = new Invocation(new Invoker(instanceClass, instanceName), method);
		if (parameters != null) {
			for (ParameterInfo p : parameters) {

	invocation.addMethodParameterName(p.getValue());
			}
		}
		if (!initializations.contains(invocation)) {
			initializations.add(invocation);
		} else {
			System.err.println("Duplicated initializations!");
		}
	}

	private void resolvExpectations() throws ClassNotFoundException, NoSuchMethodException {
		int numOfExpectations = 0;
		String numInString = props.getProperty("expectation.num");
		try {
			numOfExpectations = Integer.parseInt(numInString);
		} catch (NumberFormatException e) {
			e.printStackTrace();
		}
		if (numOfExpectations == 0) {
			System.err.println("Fatal error: no expecation defined!");
			System.exit(1);
		} else {
```

```
                for (int i = 0; i < numOfExpectations; i++) {
                        resolveExpectation(i);
                }
        }
    }

    private void resolveExpectation(int index) throws ClassNotFoundException, NoSuchMethodException {
        String instanceName = props.getProperty("expectation" + index + ".instance");
        ClassInfo mock = instanceToClassInfoMap.get(instanceName);
        if (mock == null) {
            System.err.println("Fatal error: no such mock instance declared: " + instanceName);
            System.exit(1);
        }
        Class<?> instanceClass = Class.forName(mock.getFullyQualifiedClassName());
        String methodName = props.getProperty("expectation" + index + ".name");
        Method method = null;
        int numOfParas = Integer.parseInt(props.getProperty("expecation" + index + ".parameter.num"));
        List<ParameterInfo> parameters = null;
        if (numOfParas > 0) {
            parameters = new ArrayList<ParameterInfo>();
            for (int i = 0; i < numOfParas; i++) {

    resolveParameter("expectation" + index, i, parameters);
            }List<Class<?>> parameterTypes = new ArrayList<Class<?>>();
            for (ParameterInfo p : parameters) {

    parameterTypes.add(p.getClassType());
            }
            method = instanceClass.getMethod(methodName, parameterTypes.toArray(new Class[parameterTypes.size()]));
        } else {
            method = instanceClass.getMethod(methodName, (Class[])null);
        }
        MethodInfo methodInfo = new MethodInfo();
        methodInfo.setMethod(method);
        methodInfo.setParameters(parameters);
        mock.add(methodInfo);
        Invocation invocation = new Invocation(new Invoker(instanceClass, instanceName), method);
        expectations.add(invocation); // expectations can have duplications
    }

    private void resolveParameter(String header, int index, List<ParameterInfo> parameters) throws ClassNotFoundException {
        String className = props.getProperty(header + ".parameter" + index + ".type");
        String parameterName = props.getProperty(header + ".parameter" + index + ".value");
        parameters.add(new ParameterInfo(Class.forName(className), parameterName));
    }

    /**
     * @return the mockClassInfoMap
     */
    public Hashtable<String, ClassInfo> getMockClassInfoMap() {
        return instanceToClassInfoMap;
    }

    /**
     * @return the mockClassInfoMap
     */
    public List<ClassInfo> getMockClassInfo() {
        return testCase.getMocks();
    }

    /**
     * @return the expectations
     */
    public List<Invocation> getExpectations() {
        return expectations;
    }

    /**
     * @return the cuts
     */
    public List<ClassInfo> getCuts() {
        return cuts;
    }
```

```
	/**
	 * @return the testCase
	 */
	public TestCase getTestCase() {
		return testCase;
	}

	public static void main(String[] args) {
		TestingCaseReader reader = new TestingCaseReader(args[0]);
		reader.process();
		TestingComponentResolver resolver = new TestingComponentResolver(reader.getProps());
		resolver.resolve();

	System.out.println(resolver.getMockClassInfoMap());

	System.out.println(resolver.getExpectations());
	}
}
```

MockClassGenerator.java

```
package edu.leiden.msc.limock.control;

import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.UnsupportedEncodingException;
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import java.lang.reflect.Type;
import java.util.List;

import edu.leiden.msc.limock.model.ClassInfo;
import edu.leiden.msc.limock.model.MethodInfo;
import edu.leiden.msc.limock.model.ModelFomatter;

/**
 * This class generates the mock class files as defined
 * in the testing case XML file.
 *
 * @author Xuan Li
 *
 */
public class MockClassGenerator {

	private List<ClassInfo> mockClassInfos;

	public MockClassGenerator(List<ClassInfo> infos) {
		this.mockClassInfos = infos;
	}

	public void generate() {
		try {
			generateMockClasses();
		} catch (Exception e) {
			e.printStackTrace();
			System.exit(1);
		}
	}

	private void generateMockClasses() throws IOException, UnsupportedEncodingException {
		for (ClassInfo mock : mockClassInfos) {
			String path = mock.getSrcPath();
			String packagePath = mock.getPackageName();
			if (packagePath != null && packagePath.contains(".")) {
				packagePath = packagePath.replace('.', File.separatorChar);
			} else {
				packagePath = "";
			}
			String mockClassName = ModelFomatter.getMockClassName(mock.getClassName());
			File mockClassFile = new File(path + File.separator + packagePath, mockClassName + ".java");

	System.out.println(mockClassFile.getAbsolutePath());
			if (mockClassFile.exists()) {
				mockClassFile.delete();
			}
			mockClassFile.createNewFile();
			PrintWriter writer = new PrintWriter(mockClassFile, "UTF-8");
			generateClassHeader(writer, mock);
			generateClassBody(writer, mock);
			writer.close();
		}
```

```
    }

    private void generateClassHeader(PrintWriter writer, ClassInfo mock) {
        writer.println("package " + mock.getPackageName() + ";");
        writer.println();
        writer.println("import java.lang.reflect.Method;");
        writer.println();
        writer.println("import edu.leiden.msc.limock.model.ReturnValue;");
        writer.println("import edu.leiden.msc.limock.model.Trace;");
        writer.println("import edu.leiden.msc.limock.model.Invocation;");
    }

    private void generateClassBody(PrintWriter writer, ClassInfo mock) {
        writer.println();
        writer.println("public class " + ModelFomatter.getMockClassName(mock.getClassName()) + " extends " + mock.getClassName() + " {");
        for (Constructor<?> constructor : mock.getConstructors()) {
            writer.println();
            generateConstructor(writer, constructor);
        }
        for (MethodInfo methodInfo : mock.getMethods()) {
            writer.println();
            generateMethod(writer, methodInfo);
        }
        writer.println("}");
    }

    private void generateConstructor(PrintWriter writer, Constructor<?> constructor) {
        writer.print("\t");
        StringBuilder builder = new StringBuilder();

        builder.append(Modifier.toString(constructor.getModifiers()));
        builder.append(" ");
        String constructorName = constructor.getName();
        String[] tokens = constructorName.split("\\.");

        builder.append(ModelFomatter.getMockClassName(tokens[tokens.length-1]));
        builder.append('(');
        int index = 0;
        Type[] parameterTypes = constructor.getGenericParameterTypes();
        if (parameterTypes != null && parameterTypes.length > 0) {
            for (Type type : parameterTypes) {

                builder.append(ModelFomatter.getGenericReturnType(type));
                builder.append(" arg");
                builder.append(index);
                builder.append(", ");
                index++;
            }
            builder.delete(builder.length()-2, builder.length());
        }
        builder.append(')');
        Type[] exceptionTypes = constructor.getGenericExceptionTypes();
        if (exceptionTypes != null && exceptionTypes.length > 0) {
            builder.append(" throws ");
            for (Type type : exceptionTypes) {

                builder.append(ModelFomatter.getGenericReturnType(type));
                builder.append(", ");
            }
            builder.delete(builder.length()-2, builder.length());
        }
        builder.append(" {");
        writer.println(builder.toString());
        writer.print("\t\t");
        // clear the string builder
        builder.delete(0, builder.length());
        builder.append("super(");
        if (index > 0) {
            for (int i = 0; i < index; i++) {
                builder.append("arg");
                builder.append(i);
                builder.append(", ");
            }
```

```
                builder.delete(builder.length()-2,
builder.length());
            }
            builder.append(");");
            writer.println(builder.toString());
            writer.print("\t");
            writer.println("}");
        }

        private void generateMethod(PrintWriter writer,
MethodInfo methodInfo) {
            Method method = methodInfo.getMethod();
            // annotation
            writer.print("\t");
            writer.println("@Override");
            // method header
            writer.print("\t");
            writer.println(

        Modifier.toString(method.getModifiers()) + " " +

        ModelFomatter.getGenericReturnType(method.getRetur
nType()) + " " +
                        method.getName() + "(" +

        getMethodParameterListDeclaration(method) + ")" +

        getMethodExceptionList(method) + " {");
            // method body
            writer.print("\t\t");
            writer.println("Method thisMethod =
null;");writer.print("\t\t");
            writer.println("Invocation invocation =
null;");writer.print("\t\t");
            writer.println("try {");writer.print("\t\t\t");
            writer.println("thisMethod =
this.getClass().getMethod(\"" + method.getName() + "\", " +
getMethodParameterTypes(method) + ");");writer.print("\t\t\t");
        //TODO take care of the parameter lists
            writer.println("invocation =
Trace.getInstance().getInvocationMap().get(this.hashCode()+this
Method.getName());");writer.print("\t\t\t");
//          writer.println("invocation = new
RuntimeInvocation(this.getClass(), thisMethod,
this.hashCode());");writer.print("\t\t\t");

        writer.println("invocation.setReturnValue(new
ReturnValue(thisMethod.getGenericReturnType()));");writer.print
("\t\t");
            writer.println("} catch (Exception e)
{");writer.print("\t\t\t");
        writer.println("e.getMessage();");writer.print("\t\t\t");

        writer.println("System.exit(1);");writer.print("\t\t");
            writer.println("}");writer.print("\t\t");

        writer.println("Trace.getInstance().notify(invocation);");
writer.print("\t\t");
            writer.println("try {");writer.print("\t\t\t");

        writer.println(ModelFomatter.getSuperCall(methodInfo)
);writer.print("\t\t");
            writer.println("} catch (Exception e)
{");writer.print("\t\t\t");

        writer.println("e.printStackTrace();");writer.print("\t\t");
            writer.println("}");writer.print("\t\t");

        writer.println(ModelFomatter.getReturnStatement(meth
od));
            // method close
            writer.print("\t");
            writer.println("}");
//          writer.println();writer.print("\t\t");
        }

        private                                 String
getMethodParameterListDeclaration(Method method) {
            return "";
        }

        private String getMethodExceptionList(Method
method) {
            return "";
        }

        private String getMethodParameterTypes(Method
method) {
            return "(Class[])null";
        }

        public static void main(String[] args) {
            TestingCaseReader reader = new
TestingCaseReader(args[0]);
            reader.process();
            TestingComponentResolver resolver = new
TestingComponentResolver(reader.getProps());
            resolver.resolve();
```

```
                MockClassGenerator gen = new MockClassGenerator(resolver.getMockClassInfo());
                gen.generate();
        }
}
```

TestingClassGenerator.java

```
package edu.leiden.msc.limock.control;

import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.UnsupportedEncodingException;
import java.lang.reflect.Method;

import edu.leiden.msc.limock.model.ClassInfo;
import edu.leiden.msc.limock.model.InstanceInfo;
import edu.leiden.msc.limock.model.Invocation;
import edu.leiden.msc.limock.model.MethodInfo;
import edu.leiden.msc.limock.model.ModelFomatter;
import edu.leiden.msc.limock.model.TestCase;

/**
 * This class generates the test class which takes
 * Advantage of JUnit to check the results.
* @author Xuan Li
*/
public class TestClassGenerator {

        private TestCase testCase;

        public TestClassGenerator(TestCase testCase) {
                this.testCase = testCase;
        }

        public void generate() {
                try {
                        generateTestClass();
                } catch (Exception e) {
                        e.printStackTrace();
                        System.exit(1);
                }
        }

        private void generateTestClass() throws IOException, UnsupportedEncodingException {
                String path = testCase.getSourcePath();
                String packagePath = testCase.getPackageName();
                if (packagePath != null && packagePath.contains(".")) {
                        packagePath = packagePath.replace('.', File.separatorChar);
                } else {
                        packagePath = "";
                }
                String testClassName = testCase.getClassName();
                File testClassFile = new File(path + File.separator + packagePath, testClassName + ".java");

        System.out.println(testClassFile.getAbsolutePath());
                if (testClassFile.exists()) {
                        testClassFile.delete();
                }
                testClassFile.createNewFile();
                PrintWriter writer = new PrintWriter(testClassFile, "UTF-8");
                generateClassHeader(writer);
                generateClassBody(writer);
                writer.close();
        }

        private void generateClassHeader(PrintWriter writer) {
                writer.println("package " + testCase.getPackageName() + ";");
                writer.println();
                writer.println("import java.util.Iterator;");
                writer.println();
//              writer.println("import junit.framework.Assert;");
//              writer.println();
                writer.println("import org.junit.Before;");
                writer.println("import org.junit.Test;");
                writer.println();
                writer.println("import edu.leiden.msc.limock.model.Invoker;");
                writer.println("import edu.leiden.msc.limock.model.Trace;");
                writer.println("import edu.leiden.msc.limock.model.Invocation;");
                for (ClassInfo cut : testCase.getCuts()) {
```

```
                writer.println("import " + cut.getFullyQualifiedClassName() + ";");
            }
            for (ClassInfo mock : testCase.getMocks()) {
                writer.println("import " + mock.getFullyQualifiedClassName() + ";");
                writer.println("import " + mock.getPackageName() + "." + ModelFomatter.getMockClassName(mock.getClassName()) + ";");
            }
        }

        private void generateClassBody(PrintWriter writer) {
            writer.println();
            writer.println("public class " + testCase.getClassName() + " {");
            generateFields(writer);
            writer.println();
            generateSetUpMethod(writer);
            writer.println();
            generateTestMethod(writer);
            writer.println("}");
        }

        private void generateFields(PrintWriter writer) {
            writer.println();
            writer.print("\t");
            writer.println("private Trace trace;");
            for (ClassInfo cut : testCase.getCuts()) {
                writer.print("\t");
                writer.println("private " + cut.getClassName() + " " + cut.getClassName().toLowerCase() + ";");
            }
            for (ClassInfo mock : testCase.getMocks()) {
                for (InstanceInfo i : mock.getinstances()) {
                    writer.print("\t");

        writer.println(ModelFomatter.getInstanceDeclaration(i));
                }
            }
        }

        private void generateSetUpMethod(PrintWriter writer) {
            // annotation
            writer.print("\t");
            writer.println("@Before");
            // method header
            writer.print("\t");
            writer.println("public void setUp() throws Exception {");
            // method body
            writer.print("\t\t");
            writer.println("trace = Trace.getInstance();");
            // instantiates CUTs
            for (ClassInfo cut : testCase.getCuts()) {
                writer.print("\t\t");

        writer.println(cut.getClassName().toLowerCase() + " = new " + cut.getClassName() + "();");
            }
            // instantiates mocks
            for (ClassInfo mock : testCase.getMocks()) {
                for (InstanceInfo i : mock.getinstances()) {
                    writer.print("\t\t");

        writer.println(ModelFomatter.getInstantiation(i, true));
                }
            }
            // initialize cuts and mocks
            // TODO this needs to be refined
            for (Invocation initialization : testCase.getInitializations()) {
                writer.print("\t\t");

        writer.println(initialization.getInvoker().getInvokerNam e() + "." +

        initialization.getMethod().getName() + "(" +

        initialization.getMethodParametersAsStringInCall() + ");");
            }
            // generate expectations
            writer.print("\t\t");
            writer.println("try {");
            for (Invocation expectation : testCase.getExpectations()) {
                writer.print("\t\t\t");
                writer.println("trace.expect(new Invoker(" +
```

```
		expectation.getInvoker().getInvokerName() +
".getClass(), \"" +

		expectation.getInvoker().getInvokerName() + "\", " +

		expectation.getInvoker().getInvokerName() +
".hashCode()), \"" +

		expectation.getMethod().getName() + "\", " +

		getMethodParameterTypes(expectation.getMethod()) +
");");
//				trace.expect(new
Invoker(voter1.getClass(), "voter1", voter1.hashCode()), "vote",
(Object[])null);
			}
			writer.print("\t\t");
			writer.println("} catch
(NoSuchMethodException e) {");
			writer.print("\t\t\t");
			writer.println("e.printStackTrace();");
			writer.print("\t\t\t");
			writer.println("System.exit(1);");
			writer.print("\t\t");
			writer.println("}");
			// method close
			writer.print("\t");
			writer.println("}");
		}

		private void generateTestMethod(PrintWriter writer) {
			// annotation
			writer.print("\t");
			writer.println("@Test");
			// method header
			writer.print("\t");
			writer.println("public void test" +
testCase.getName() + "() {");
			// method body
			for (ClassInfo cut : testCase.getCuts()) {
				for (MethodInfo methodInfo :
cut.getMethods()) {
					writer.print("\t\t");

		writer.println(ModelFomatter.getMethodCallIndication(
cut.getClassName().toLowerCase(), methodInfo));
					writer.print("\t\t");

		writer.println(ModelFomatter.getMethodCall(cut.getCla
ssName().toLowerCase(), methodInfo));
//
		writer.println(cut.getClassName().toLowerCase() + "."
+
//
		methodInfo.getMethod().getName() + "(" +
//
		ModelFomatter.getParametersAsStringInCall(methodIn
fo.getParameters()) + ");");
				}
			}
			writer.print("\t\t");
			writer.println("Iterator<Invocation> expected
= trace.expectationSequence().iterator();");
//			writer.print("\t\t");
//			writer.println("Iterator<Invocation> results =
trace.resultSequence().iterator();");
			writer.print("\t\t");
			writer.println("int index = 1;");
			writer.print("\t\t");
			writer.println("while (expected.hasNext()) {");
			writer.print("\t\t\t");
			writer.println("Invocation expectedNext =
expected.next();");
//			writer.print("\t\t\t");
//			writer.println("Invocation resultNext =
results.next();");
			writer.print("\t\t\t");

		writer.println("System.out.println(\"expected@\" +
index++ + expectedNext.toString());");
//			writer.print("\t\t\t");
//			writer.println("System.out.println(\"result@\"
+ index++ + resultNext.toString());");
//			writer.print("\t\t\t");
//			writer.println("Assert.assertEquals(\"@\" +
index++, expectedNext.toString(), resultNext.toString());");
			writer.print("\t\t");
			writer.println("}");
			writer.print("\t\t");
			writer.println("System.out.println(\"Expected
sequence size: \" + trace.expectationSequence().size());");
			writer.print("\t\t");
			writer.println("System.out.println(\"Result
sequence size: \" + trace.resultSequence().size());");
//			writer.print("\t\t");
```

```
//
        writer.println("Assert.assertEquals(trace.expectationSeq
uence().size(), trace.resultSequence().size());");
                // method close
                writer.print("\t");
                writer.println("}");
        }


        private String getMethodParameterTypes(Method
method) {
                return "(Object[])null";
        }
                public static void main(String[] args) {
                TestingCaseReader reader = new
TestingCaseReader(args[0]);
                reader.process();
                TestingComponentResolver resolver = new
TestingComponentResolver(reader.getProps());
                resolver.resolve();
                MockClassGenerator mockGen = new
MockClassGenerator(resolver.getMockClassInfo());
                mockGen.generate();
                TestClassGenerator testGen = new
TestClassGenerator(resolver.getTestCase());
                testGen.generate();
        }
}
```

NOTE: LiMock MODEL PART IS OMITTED. FOR FURTHER INFORMARION PLEASE CONTACT AUTHOR XUAN LI (WELCOMEMARIE@GMAIL.COM [after graduation contact email])