b
UNIVERSITÄT
BERN

Where does this null come from ?

An Approach to show the exact location where a value was referenced to null

Bachelor Thesis

Lina Tran

from

Biel/Bienne BE, Switzerland

Faculty of Science
University of Bern

31. July 2016

Prof. Dr. Oscar Nierstrasz

Research assistant Nevena Milojković

Research assistant Boris Spasojević

Software Composition Group

Institute for Computer Science

University of Bern, Switzerland

Abstract

A previous study found out that NullPointerExceptions are the most frequently occurring furthermore very hard to be debugged exception in Java projects. It is hard to debug because the developer is only provided with a stack trace to where the exception happened when a NullPointerException is thrown. This only gives insight into the effect of the fault but not into its cause. That is why the developer often asks where this null reference is coming from.

The aim of the project is to provide a developer with an additional stack trace of where the value was actually set to null, next to the default stack trace of a NullPointerException. We attempt to achieve this goal by instrumenting java source code ideally with a minimal execution overhead.

By tracking the null assignments by statically analyzing bytecode and inserting additional code the debugging after a NullPointerException will be simplified.

Contents

1 Introduction

Nowadays, certainly every programmer is confronted with NullPointerExceptions in both big and small Java projects, whether it is for an enterprise or for private purposes. The seriousness of null dereferencing can be stated since it has been found out that in projects 35% of the conditional statements are null checks which has a negative impact on performance [3]. It is also considered as the number one error Java programmers make[1].

So what are those NullPointerExceptions? NullPointerException is a commonly occurring RuntimeException in object-oriented languages. There is a special value *null* that is a pointer which does not point to a valid object, so it points to nowhere. The main cause is a method invocation on a null object or an attempt to access a field. This object reference can be a local variable, a instance field, an element of a collection, a return value of a function, an object has not be initialized and so on. In Java Projects developers always have to deal with a huge amount of references which means there is a high probability that there will be a NullPointerException.

Now we will present two different situations in which a NullPointerException can be thrown.

```
1 public void drop(java.awt.dnd.DropTargetDropEvent dtde) {
2     //...
3     try {
```

[1] http://www.javacoffeebreak.com/articles/toptenerrors.html

```
4           //...
5           DNDFigures ff = (DNDFigures)DNDHelper.processReceivedData(
                DNDFiguresTransferable.DNDFiguresFlavor, dtde.
                getTransferable());
6           //...
7           Point theO = ff.getOrigin();
8           //...
9       }
10      catch (NullPointerException npe) {
11          npe.printStackTrace();
12          dtde.dropComplete(false);
13      }
14      //...
15  }
```

Code 1.1: NullPointerException example - Method receiver

Lets say in Code 1.1 the variable at line 5 is assigned with the special null value thanks to the return value of the function `DNDHelper.processReceivedData()` . So the method receiver (`ff` at line 7) is null and when a method is called on this object a NullPointerException is thrown. Because the the NullPointerException occurred in the `try{}` block the `catch{}` block right after it handles the exception. The stack trace only takes the developer to line 7 where the exception is caused but not to the real culprit which is the assignment at line 5. This is a very simple example and the easiest way to produce a NullPointerException. The root of the bug is located in the same method. But in case a field access is performed it can be a lot more complicated.

```
1  public class DrawApplication extends JFrame implements
       DrawingEditor,
2          PaletteListener, VersionRequester {
3      //...
4      private IconkitManager fManager;
5
6      public void open() {
7          open(createInitialDrawingView());
8      }
9
10     protected void open(final DrawingView newDrawingView) {
11         //...
12         setIconkit(createIconkit());
13         //...
14         setTool(new NullTool(this), "");
15         //...
16     }
17
18     protected Iconkit createIconkit() {
19         fManager = getIconkitManager();
20         //...
```

```
21        }
22
23        public void setTool(Tool t, String name) {
24            //...
25            fManager.getComponent();
26            //...
27        }
28        //...
29    }
```

Code 1.2: NullPointerException example - Field access

With the flow-sequence of the method `open()` at line 10 the class instance `fManager` (line 4) is initialized and at a later point it is accessed. Suppose the instance field `fManager` is set to null by the function `getIconkitManager()` at line 19when the method `createIconkit()` is performed at line 12. Later at line 14 the function `setTool()` is invoked which attempts an access to the field `fManager` that points to nowhere. That means a NullPointerException is thrown at line 25 and with the stack trace we only can track back the exception like :

```
Exception in thread "main" java.lang.NullPointerException
        at org.jhotdraw.application.DrawApplication.setTool(DrawApplication.java:25)
        at org.jhotdraw.application.DrawApplication.open(DrawApplication.java:14)
        at org.jhotdraw.application.DrawApplication.open(DrawApplication.java:7)
        ...
```

Box 1.1: NullPointerException

In Box 1.1 can be seen that the stack trace does not track to the root of the NullPointerException. That means the Java developer has to debug to find the exception root.

All too often programmers states that they are trying to fix bugs or have spent a lot of time fixing them. Of course not all of the bugs are NullPointerExceptions but there could be few of them. If there would be a way to minimize the time fixing exceptions and allow to work more efficiently, projects would progress much faster. NM ►*Rewrite: yes, developers deal with a lot of problems caused by null pointer exception, and you should find some paper to reference that.*◄ LT ►*can't find papers!*◄

At this point we would like to introduce our project named ***NullSpy*** which supports the developers in situations discussed previously, in other words its main goal is to take a step to that ideal vision. The intention behind NullSpy is to present the developers the

exact location of the null assignment next to the ordinary stack trace, in order to help him find the place of dereference.

In this thesis it is explained how the goal mentioned above is achieved step by step, by using a class library Javassist (Java Programming Assistant) which allows us to deal with Java bytecode.

2 Technical Background

This chapter provides a short overview of technologies used in this project that were essential for the implementation of NullSpy.

2.1 Javassist

Javassist or *Java Programming Assistant*[1][1][2], a subproject of Jboss, is a class library which allows to deal with Java bytecode. Since 1999 it is used as an engineering toolkit in a broad domain, and is still being extended by Shigeru Chiba. It enables developers to manipulate Java bytecode in a simplified way like defining a new class at runtime or modifying a class file when it is loaded by the JVM. All manipulations are performed at load-time through a provided class loader. In case of more interest how Javassist works, we recommend to read the following tutorial: `http://jboss-javassist.github.io/javassist/tutorial/tutorial.html`.

[1] `http://jboss-javassist.github.io/javassist/`

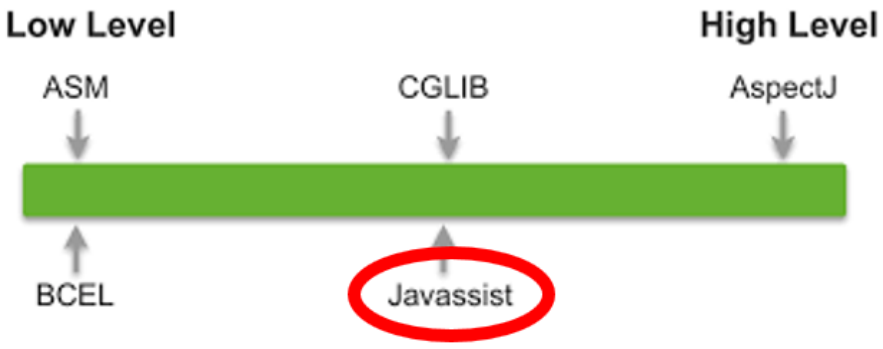

Figure 2.1: Bytecode modification levels

Unlike many other libraries Javassist offers two levels of API: **source**-level and **bytecode**-level (See Figure 2.1). Using the source-level API, the user can edit a class file without any familiarity with the specifications of the Java bytecode. Only knowing the Java language is enough because the API is designed only with the vocabulary of Java. On this level the programmer just has to write regular source code and Javassist compiles it automatically. The bytecode level allows the user to modify classes directly in binary form like other editors, e.g. ASM.

At this point, let us look at a small example of how the bytecode manipulation works.

```
1 ClassPool pool = ClassPool.getDefault();
2 CtClass cc = pool.get("test.Rectangle");
3 cc.setSuperclass(pool.get("test.Point"));
4 cc.writeFile();
```

Code 2.1: A polymorphic call site

1. First a *ClassPool* object that controls bytecode modification with Javassist is obtained. With the ClassPool a class file can be read on demand for constructing a CtClass object. By class file the binary file which has the extension *.class* and represents either a Java class or an interface is meant.
2. The class *CtClass* (compile-time class) represents the class file, that means all manipulations are performed on the CtClass. With the method invocation `get()` on ClassPool a reference to the CtClass object which represents the class `test.Rectangle` is obtained.
3. In this example the superclass of `test.Rectangle` is just changed to `test.Point` .

4. If the bytecode modification is done, the method call `writeFile()` on CtClass is necessary to make sure that the changes are reflected on the original class file.

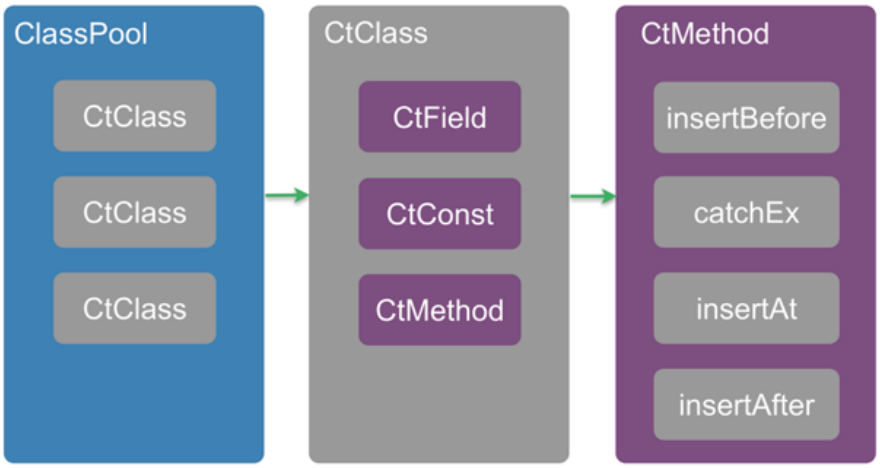

Figure 2.2: Javassist Modules

Figure 2.2 gives an understanding/overview how the main part of bytecode manipulation with Javassist is built up. The ClassPool is nothing else than a container of multiple CtClasses. As described before CtClass represents those class files on which modifications are done. Like typical classes, it can hold compile-time fields, constants or methods. Javassist is capable of adding or modifying classes, behaviors, fields, method invocations, local variables etc. But in our case we mainly address the manipulation of behaviors. It is possible to insert additional source code at the beginning of the method body, at the end or at a specific line. Next to these options even a *catch block* can be added. The difference between `insertAfter()` and `addCatch()` is insertAfter inlines some code just before every return instruction in the method body and addCatch adds a catch clause to the method which handles an exception thrown in the body. So the catch clause is always positioned at the end of the method body. In Figure 2.3 below a small insertion example is demonstrated.

```
method.insertAt(8,
        "runtimeSupporter.NullDisplayer.test( \""
                + method.getDeclaringClass().getName() + "\","
                + variableName + ");");
```

```
ch.unibe.scg.nullSpy.runtimeSupporter.NullDisplayer.test(
        "className", variable);
```

Figure 2.3: Inserting code example

2.2 JAD

Java Decompier (JAD)[2] is a decompiler and a Eclipse plugin for the programming language Java. A decompiler is a computer program that takes an executable file as input, and attempts to create a high level, compatible source file. If the source file is compiled again, it will produce an executable program that behaves the same way as the original one. It is used in software reverse engineering.

The example Bytecode 2.2 is decompiled by JAD and the result is the source code right under the example. It is also a good opportunity to present how bytecode looks like.

```
// Method descriptor #10 (Lorg/jhotdraw/applet/DrawApplet;)V
// Stack: 2, Locals: 2
// Method descriptor #22 (Ljava/awt/event/ItemEvent;)V
// Stack: 2, Locals: 2
public void itemStateChanged(java.awt.event.ItemEvent e);
  0  aload_1 [e]
  1  invokevirtual java.awt.event.ItemEvent.getStateChange()...
  4  iconst_1
  5  if_icmpne 22
  8  aload_0 [this]
  9  getfield org.jhotdraw.applet.DrawApplet$1.this$0 : ...
 12  aload_1 [e]
 13  invokevirtual java.awt.event.ItemEvent.getItem() : ...
 16  checkcast java.lang.String [33]
 19  invokevirtual org.jhotdraw.applet.DrawApplet....
 22  return
Line numbers:
 [pc: 0, line: 213]
 [pc: 8, line: 214]
 [pc: 22, line: 216]
Local variable table:
 [pc: 0, pc: 23] local: this index: 0 type: new org...
 [pc: 0, pc: 23] local: e index: 1 type: java.awt.event...
Stack map table: number of frames 1
 [pc: 22, same]
```

Code 2.2: Bytecode example

```
1 public void itemStateChanged(ItemEvent e) {
2     if(e.getStateChange() == 1)
3     loadDrawing((String)e.getItem());
4 }
```

Code 2.3: Decompiled bytecode

[2] https://sourceforge.net/projects/jadclipse/

In NullSpy itself, JAD is not used but it was a big help during the implementation phase. Since after running NullSpy on a project only the modificated bytecode is available. The use of JAD is for the sake of debugging through Nullspy; to check whether the modification by Javassist, e.g. inserting source code, has succeeded. Another way to check the result for correctness is looking at bytecode itself what would have taken a lot of effort and time.

3
NullSpy

As earlier explained in Chapter 1, this project is about providing the user with additional stack trace with the information of the origin or line of code of where null is assigned to a variable that caused the NullPointerException. In short, it shows the developer the exact location of where a method receiver or a field (in a field access case) was assigned to null.

Here we would like to give a short insight of how we managed to successfully implement the core of the project NullSpy. Next to how it is built up, we will also address the challenges we were encountering during the implementation and the limitations we planned for future work (Chapter 5).

3.1 High Level Overview/Rough Scheme

The general approach of NullSpy is to statically analyze and add bytecode to a project. We add bytecode to check on every variable for its value and extract information (name, line number etc.) about it if necessary. Another reason is to get information about method receivers (Box 3.3). To know where exactly they are placed in source code we need to statically examine the code and insert the bytecode at the right place.

What NullSpy first does is loading the project that should be able to track the null assignment if a NullPointerException is thrown. By loading the project to NullSpy, the

compiled class files of the project are addressed only, which means the project itself does not have to be imported to the programming environment, e.g. Eclipse. Simultaneously at load time each class file is modified with help of Javassist. In what way will be discussed in the following Section 3.2.

Once the project modification is done it is stored in a destination folder that the user has chosen before. This means after the changes there will be another version of the project which can perform additional work like tracking the null assignment. Since only the class files are accessed previously, the result which is stored in the destination folder is as expected only the modified bytecode.

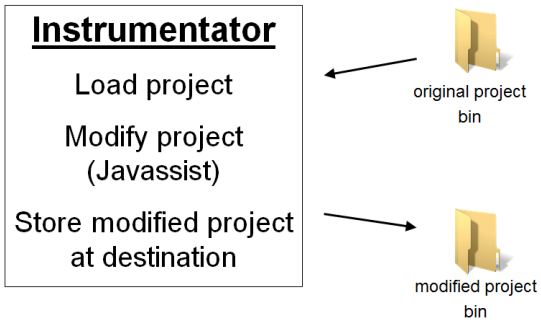

Figure 3.1: Modification overview

Of course the original project can be replaced by the modificated one too. But the developer has to think carefully about it because once replaced, the additional bytecode stays there and the execution time is slightly different than it was before.

3.2 Low Level Overview

NullSpy only handles the unhandled NullPointerException, meaning the main method of a project is wrapped with a *catch block* where the NPE is treated. The idea in this *catch block* is to load the data about the assignments and about the method receiver to compare them with each other. NullSpy will add other classes to the project which manage the added bytecode. One of those classes keeps *HashMaps* in which data about assignments is stored. They are stored in HashMaps because each variable has a unique key. Next time if a variable is assigned with another value than null, its entry in the HashMap is deleted since it cannot cause a NullPointerException anymore. The data about the method receiver which is loaded is stored in an external *csv* file for performance reasons.

In a NullPointerException situation the developer is provided with the stack trace with which he/she can extract the information about the line number and the Java class name of where the NullPointerException occurred. Those data serves as a search criteria for further actions. We look these two information up in the file about the method receiver data. On a match more details about the method receiver can be read out. With this method receiver entry information we search for its assignment location which can be found in the HashMaps containing the details about all null assignments. And finally, if an entry in the HashMaps is found which matches the information extracted before, the exact location of the null assignment is found. The result is providing the user with an additional stack trace link next to the usual one. Clicking on that link will take one directly to the NullPointerException root. So to be able to provide this function, data about the method receivers and variable assignments have to be collected initially.

So we first load the project which should be modified and then go through all class files of it. A NullPointerException can be thrown if a method call was performed on an object respectively an access on a field which is null. To keep it simple for further explanations we will only mention method receiver from now on instead of method receiver and field.

To reach the goal of NullSpy we have to gather information about the method receiver. A list of what kind of information is needed about the method receiver can be found in Subsection 3.2.1. Next to this also information about the variable assignments has to be collected to know the exact location of the null assignments. The main idea behind NullSpy actually is to get information comparing those together when a NullPointerException occurs and if there is a hit during the comparison the location of the null assignment can be obtained easily.

To explain the approach we will go through an example. The Code 1.1 serves as a good example. One method receiver is located at line 7 with the name `ff` . We extract information from bytecode about it and store the result in an external csv file. Data like name, type, line number, behavior, class etc. are stored (full list: Box 3.3). The next step is to find all assignments in the code and insert new bytecode right after the assignments. In the example two assignments are present, in line 5 and 7. To do something with them obviously we have to know them better, that is why we again extract data about the variables `ff` and `theO` and pass the gathered information further to the inserted code right before line 6 and line 8 starts. The new bytecode checks whether the variables are null and stores the information about the variable with null value in a HashMap. The name and type of the variable `ff` , the behavior name and its signature and much more are stored in the HashMap.

If a NullPointerException is thrown the `main()` method will catch it. From stack trace info like the line number 7, the method name *“drop"* and its signature, and the

class name are read out. The `main()` method loads the csv file about the method receivers and looks up those details extracted from the stack trace. There is an entry (data about `ff`) which matches those search criteria. From the entry we need the name, type etc. from the variable. Now the `main()` method looks up the data about the variable in the HashMaps of the variable assignments. In the HashMaps the `main()` method finds the record with the same information as the method receiver `ff` . This record contains the exact location where null was assigned to the variable `ff` . With the line number 5, the behavior name drop and class name X the link which points to the assignment location can be created and is added to the origin stack trace. With only a click on that link the root of the NullPointerException is exposed.

The implementation behind the NullSpy concept will be explained more thoroughly.

3.2.1 Method Receiver Data Collection

Unfortunately, Javassist does not provide the function to directly get the method receiver. We got a suggestion to use AST (Abstract Syntax Tree footnote wiki) to get it but we decided to not go deeper into this and implement our own algorithm.

The algorithm contains following steps (abstract):

1. Getting pc-interval of method invocation
2. Storing all possible method receiver interval within the interval of step 1 into an ArrayList
3. Getting the number of parameters, the method invocation takes
4. Traversing back the ArrayList the amount of parameters obtained in step 3
5. Result: method receiver interval
6. Store variable name, type etc. into an external csv file

Algorithm 3.2: Method receiver algorithm

In step 1 we had big troubles getting the right interval of the method receiver because only by statically analyzing the bytecode it is unapparent where the method receiver is situated exactly. But more about the challenges in Subsection 3.3.1.

Statically analyzing bytecode for method receiver means looking for certain opcodes which matches all opcodes that matches with the regex *"invoke.*"*. There are

exactly five kinds of bytecode instruction: *invokedynamic*, *invokeinterface*, *invokespecial*, *invokestatic*, *invokevirtual*. The invocation opcode invokedynamic facilitates the dynamic-typed languages[1] through dynamic method invocation. In our case it can be ignored because NullSpy only supports the static-typed language[2] Java.

In case of the *invokestatic* instruction we do not have a method receiver. That is why NullSpy treats it extraordinary like ignoring it completely or wrap it as a possible method receiver when it is actually a parameter of a method invocation. In all other cases we normally use the algorithm to get the method receiver. The gathered data about method receivers are stored in an external *csv* file due to performance and overhead minimization. The following list will list the information which is stored about the method receiver.

- nr (method receivers are counted)
- lineNumber
- variableID (whether it is a local variable or a field)
- variableType
- isVariableStatic
- ClassWhereVariableIsAccessed
- behavior (method)
- behaviorSignature
- localVariableAttributIndex
- fieldDeclaringClassName

Box 3.3: Method receiver information

3.2.2 Variable Data Collection

While going through the bytecode attention is paid to some opcodes[3]. Right after each keyword that indicates a variable assignment we insert some bytecode. The inserted code represents a test method which tests whether the value of the assigned variable is null or not and store some information about it. Unlike in getting information about

[1] Language whose type checking is usually performed at runtime.
[2] Language whose type checking is performed at compile time.
[3] Operation code: Machine language instruction.

the method receiver in Subsection 3.2.1 the data about the variables are stored in a HashMap.

What kind of opcodes were NullSpy looking for? For instance or class/static variables the bytecode instruction *getfield* and *getstatic* were essential, for local variables the important opcodes were those which matches the regex *"aload.*"*. Due to different types of variables and the limitation of Javassist gathering information about them was performed differently. Again getting the necessary data about the variables we encountered many difficulties which will be discussed in the Section 3.3.

3.2.2.1 Local Variable

Unfortunately, Javassist does not provide any support for gaining information about local variables that is why getting the needed data we had to understand how bytecode is constructed. At this point we would like to give a small bytecode introduction.

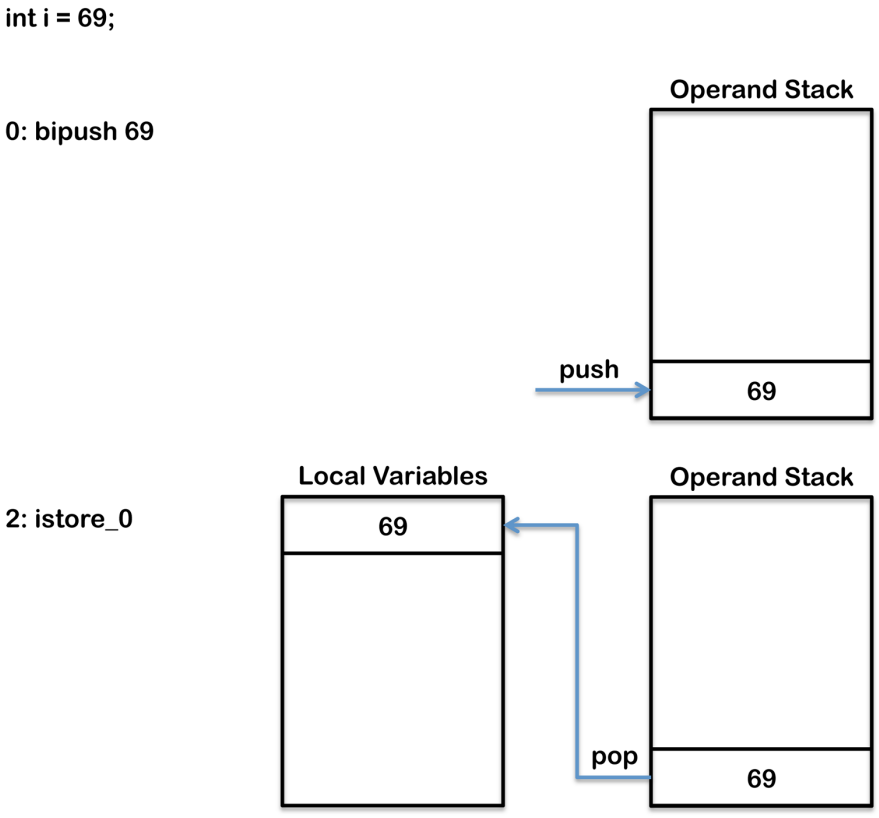

Figure 3.4: Local variable creation

If a local variable is created, the value assigned to it is pushed onto the operand stack. With the bytecode instruction *".*store.*"* the local variable is popped from the operand stack and stored into a local variable array slot. In which slot it is stored can be extracted from the instruction. Opcodes for storing local variables is composed of one, or in some cases two bytes. There are reserved machine commands for the first four local variables, index-linked from 0 to 3 and each of them contains one byte (astore_0, astore_1, astore_2, astore_3). If there is no slot number visible in the instruction, it indicates that the slot number is stored in the second byte from where it can be extracted. Next to storing the local variable loading it from the local variable array is possible to, but only with the local variable slot number.

With this short introduction understanding the local variable table should be easier. Each method of a class file contains a local variable table (see Bytecode 3.1) with which many information can be read out of it, e.g. the lifespan of the local variable, what it is called, in which slot it is stored and what type it has.

```
Local variable table:
 [pc: 0, pc: 31] local: this index: 0 type: org.jhotdraw.samples...
 [pc: 0, pc: 31] local: newDrawing index: 1 type: org.jhotdraw...
 [pc: 5, pc: 31] local: d index: 2 type: java.awt.Dimension
 [pc: 22, pc: 31] local: newDrawingView index: 3 type: org.jhot...
```

Code 3.1: Local variable table

We had to pay attention to be sure to get the right local variable. Every time when we bumped into the opcode *".*store.*"* we could only get its slot and the pc[4] where it is situated in the bytecode sequence. In the earlier paragraph the lifespan of the local variable was mentioned, the importance behind this is as soon as the lifespan of one ends, the slot can be reused by the next instantiated local variable. This way, the local variable table could contain multiple entries with the same local variable slot.

```
1 Local variable table:
2   [pc: 0, pc: 510] local: args index: 0 type: java.lang.String[]
3   [pc: 24, pc: 510] local: m index: 1 type: isFieldOrLocalVar...
4   [pc: 26, pc: 510] local: i index: 2 type: int
5   [pc: 39, pc: 46] local: k index: 3 type: java.lang.Object
6   [pc: 52, pc: 510] local: d index: 3 type: java.lang.Object}
7   [pc: 71, pc: 510] local: d2 index: 4 type: java.lang.Object
```

Code 3.2: Local variable table entries with same slot (line 5&6)

After extracting the slot of the local variable we will get the first local variable table entry which contains that slot. If the pc of the local variable assignment is not included

[4]Program counter/instruction pointer: A processor register that indicates where a computer is in its program sequence.

in the lifespan-pc-interval of the entry, the next entry with the same slot will be checked until both criteria (slot and pc) fits. Once those criteria are met we can be positive about having got the right local variable table entry to extract the information needed.

Next to the local variable table each methods of a class file also holds another attribute called line number attribute. This is just the mapping table from pc to source code line number. Since encountering the storing keyword the pc is available , with help of it the line number can be easily obtained.

```
Line numbers:
  [pc: 0, line: 26]
  [pc: 4, line: 27]
  [pc: 8, line: 29]
  [pc: 16, line: 30]
```

Code 3.3: Line number table

3.2.2.2 Instance and Class/Static variables (Fields)

Although Javassist does not support the access to local variable it provides a way for fields. Javassist allows modifying an expression in a method body with the class `Javassist.expt.ExprEditor` . In our case we only want to extract some information about the fields instead of any modification, nevertheless this class can be used appropriately. What it does is to scan the bytecodes on instructions like *"putfield"* or *"putstatic"*.

There are only two instructions that indicates an access to a field, but there are actually many different types. To get the meaning of different types see the following list:

(|. . . |: put value on operand stack for assigning to a field)

1. aload_0, |. . . |, putfield
2. |. . . |, putstatic
3. aload.*, |. . . |, putfield
4. aload_0, (getfield)+, |. . . |, putfield
5. getstatic, (getfield)*, |. . . |, putfield

Box 3.5: Field keywords

Depending on the category the field belongs, different kind of information is stored. For fields 1-2 following information is needed and stored:

(|. . . |: put value on operand stack for assigning to a field)

- fieldID (which is field or localVar)
- fieldname
- fieldType
- fieldDeclaringClassName (in which class it is instantiated)
- isFieldStatic
- fieldLineNumber
- startPosition (nonstatic: this; static: value loading bytecode for assigning to the field)
- storePosistion (putfield/putstatic)
- afterStorePosition (right before this pos additional bytecode is inserted)
- classWhereFieldIsAccessed (in this case the same as fieldDeclaringClassName)
- behavior (method where it is accessed)
- indirectVariable (null - explained immediately)

Box 3.6: Variable information (I)

In NullSpy we call these fields *"directFields"*. Reading *"direct"* implicates that there must be something like *"indirectFields"*, and that is right. As *"indirectFields"* the fields 3-4 are numbered among. Everything before the instruction *putfield* is termed as *"indirectVariable"*. For those fields more data are needed to store for their identification. Nearly the same as above, except:

- classWhereFieldIsAccessed (can be differend than fieldDeclaringClassName)
- indirectVariable:
  - indirectVariableName
  - indirectVariableType
  - indirectVariableDeclaringClassName
  - isIndirectVariableStatic
  - indirectVariableOpcode (to distinguisch if it is a localVariable or a field)

Box 3.7: Variable information (II)

An example where the instructions *"putfield"* and *"putstatic"* are used:

```
1  public class A {
2
3      private B b = new B();
4      private static B b2;
5
6      public void x() {
7
8          // |...|: value assigned
9
10         // directFields
11         this.b = ...; // aload_0, |...|, putfield
12         b2 = ...; // |...|, putstatic
13
14         B b3 = new B();
15
16         // indirectFields
17         b3.c = ...; // aload.*, (getfield)*, |...|, putfield
18         b3.c.d = ...
19         b.c = ...; // aload_0, (getfield)+, |...|, putfield
20         b.c.d = ...;
21         b2.c = ...; // getstatic, (getfield)*, |...|, putfield
22         b2.c.d = ...;
23     }
24 }
25
26 public class B {
27     public C c = new C();
28     ...
29 }
30
31 public class C {
32     public D d = new D();
33     ...
34 }
35
36 public class D {
37     ...
38 }
```

Code 3.4: Example: *"putfield"* and *"putstatic"*

Once the bytecode is modified, the way to get the data of fields needs some adaptation (see Subsection 3.3.2).

3.2.3 Bytecode Adaptation

Each time encountering a variable assignment we first extract the needed data and directly after this we adapt the class file by adding extra bytecode right after the assignment bytecode to check whether the variable is null or not. If it is null, the information collected before is stored either into the *"localVariableMap"* or into a *"fieldMap"* which are HashMaps.

`ch.scg.nullSpy.runtimeSupporter.VariableTesterddddddddddddddddddddddd`

The added bytecode represents a static method of a class named `ch.scg.nullSpy.runtimeSupporter.VariableTester` which will be added to the modified project after class file modification is done. Depending on the kind of variable analyzed at the moment different bytecode is constructed, meaning different method with different parameter/variable data is added.

There were again some challenges we had to get over, like getting the wanted data after an instrumentation and entering the additional code in the right position of the bytecode sequence (Subsection 3.3.3).

Once we have gone through the bytecode of a Java class, the modified class files are stored in a destination directory as mentioned in Section 3.1. Next to the instrumentation supplementary supporter classes are added to the project. The most important ones are `ch.unibe.scg.nullSpy.runtimeSupporter.Variabl` which tests whether a variable is null or not and `ch.unibe.scg.nullSp` which matches data and prints the location of a null assignment when a NullPointerException is thrown.

3.3 Challenges

In this section we could like to give present some of the difficulties which occurred during the implementation of NullSpy.

3.3.1 Obtaining Method Receiver Data Difficulties

Aforementioned in Subsection 3.2.1 (*Method Receiver Data Collection*) we were encountered with a persistent problem, namely getting the pc-interval of the method receiver when the interval covers multiple lines in source code. In many development environment the written code can be formatted automatically and as well manually. See following figures:

```
31          Image image = Iconkit.instance().registerAndLoadImage(
32              (Component)view, imageName);
```

Figure 3.8: Method invocation split in two lines example

```
0  invokestatic org.jhotdraw.util.Iconkit.instance() : org.jhotdraw.util.Iconkit [22]
3  aload_1 [view]
4  checkcast java.awt.Component [28]
7  aload_0 [this]
8  getfield org.jhotdraw.samples.minimap.MiniMapDesktop.imageName : java.lang.String [14]
11 invokevirtual org.jhotdraw.util.Iconkit.registerAndLoadImage(java.awt.Component, java.lang.String) : java.awt.Image [30]
14 astore_2 [image]
```

Figure 3.9: Bytecode to Figure 3.8

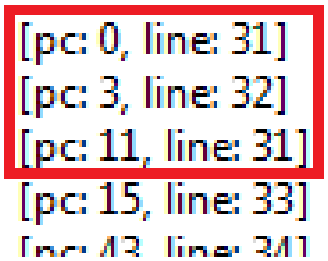

Figure 3.10: Line number table/interval to Figure 3.8

Figure shows a normal method invocation which is split into two lines in source code. This cannot be figured out by just looking at the bytecode therefore the line number attribute (Code 3.3) has to be consulted too. There is no method receiver in the figure because the method receiver would contain a method invocation itself, what is not supported in NullSpy. Another method receivers NullSpy does not support are elements of collections due to complex structures that can be

stored in the collections. But still it is a good example to show how the bytecode and the line number attribute for this looks like. Applying the (Algorithm 3.2) for gathering data about method receivers in this situation looks as follows:

1. pc-interval:
   0-11 (cut everything off what has nothing to do with the method invocation)
2. ArrayList:
   [0,[3,4],[7,8]]
3. Number of parameters:
   2
4. Traversing back the amount of parameters:
   [7,8] (1) ; [3,4] (2)
5. Result:
   0 (invokestatic) = possible method receiver
6. Not storing it because NullSpy does not support this kind of method receiver

Box 3.11: Extract method receiver example

So how did we get the pc-interval? Just looking at bytecode is not enough, so we are looking at the line number attribute. Line number *31* (Figure 3.10) is listed twice, this indicates that the method invocation is split into multiple line in source code. The former declares the starting point of the interval and the latter the end of it. How to get the pc-interval will be presented shortly. But first take a look at more examples:

```
127         Connector oldConnector = ((ChangeConnectionHandle.UndoActivity)
128             getUndoActivity()).getOldConnector();
```

Figure 3.12: Alternating line number example

```
47  aload_0 [this]
48  invokevirtual org.jhotdraw.standard.ChangeConnectionHandle.getUndoActivity() : org.jhotdraw.util.Undoable [72]
51  checkcast org.jhotdraw.standard.ChangeConnectionHandle$UndoActivity [76]
54  invokevirtual org.jhotdraw.standard.ChangeConnectionHandle$UndoActivity.getOldConnector() : org.jhotdraw.framework.Connector [142]
57  astore 7 [oldConnector]
```

Figure 3.13: Bytecode to Figure 3.12

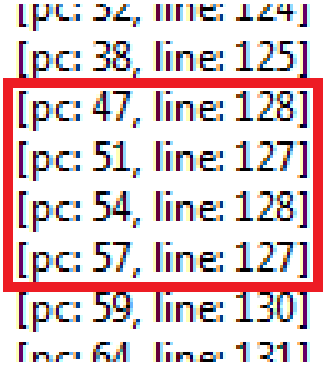

Figure 3.14: Line number table/interval Figure 3.12

Special about this example is that the multiple line interval does not start with the smaller line number and end with the bigger one, apart from that it is alternated stored in the line number attribute.

```
149         for (int i = 0; i < ColorMap.size(); i++)
150             choice.addItem(
151                 new ChangeAttributeCommand(
152                     ColorMap.name(i),
153                     attribute,
154                     ColorMap.color(i),
155                     this
156                 )
157             );
```

Figure 3.15: Nested interval example

```
 8  iconst_0
 9  istore_3 [i]
10  goto 37
13  aload_2 [choice]
14  new org.jhotdraw.standard.ChangeAttributeCommand [213]
17  dup
18  iload_3 [i]
19  invokestatic org.jhotdraw.util.ColorMap.name(int) : java.lang.String [246]
22  aload_1 [attribute]
23  iload_3 [i]
24  invokestatic org.jhotdraw.util.ColorMap.color(int) : java.awt.Color [252]
27  aload_0 [this]
28  invokespecial org.jhotdraw.standard.ChangeAttributeCommand(java.lang.String, org.jhotdraw.framework.FigureAttributeConstant, java.lang.Object, org.jhotdraw.framework.DrawingEditor) [222]
31  invokevirtual org.jhotdraw.util.CommandChoice.addItem(org.jhotdraw.util.Command) : void [225]
34  iinc 3 1 [i]
37  iload_3 [i]
38  invokestatic org.jhotdraw.util.ColorMap.size() : int [256]
41  if_icmplt 13
```

Figure 3.16: Bytecode to Figure 3.15

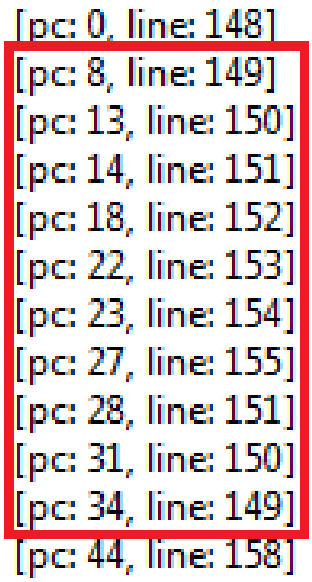

Figure 3.17: Line number table/interval to Figure 3.15

Neither in bytecode nor in line number attribute we can extract the exact interval of method invocations. Generally, we cannot distinguish if the interval represents a method invocation or a loop. Another interesting thing is that in the figure multiple corresponding pcs are included.

```
22              fooTestSupporter
23                      .bla(getObject(),
24                              staticMethod(o),
25                              fooTestSupporter2.bla2(null,
26                                      o));
```

Figure 3.18: Uncomplete interval example

```
18  aload_1 [fooTestSupporter]
19  invokestatic isFieldOrLocalVariableNullExample.testMethodCall.FooTest.getObject() : java.lang.Object [26]
22  aload_3 [o]
23  invokestatic isFieldOrLocalVariableNullExample.testMethodCall.FooTest.staticMethod(java.lang.Object) : java.lang.Object [30]
26  aload_2 [fooTestSupporter2]
27  aconst_null
28  aload_3 [o]
29  invokevirtual isFieldOrLocalVariableNullExample.testMethodCall.FooTestSupporter.bla2(java.lang.Object, java.lang.Object) : java.lang.Object [34]
32  invokevirtual isFieldOrLocalVariableNullExample.testMethodCall.FooTestSupporter.bla(java.lang.Object, java.lang.Object, java.lang.Object) : void [38]
```

Figure 3.19: Bytecode to Figure 3.18

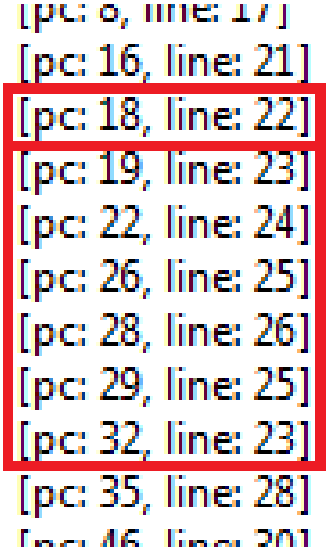

Figure 3.20: Line number table/interval to Figure 3.18

By using the algorithm we only get the interval from pc 19-32, but actually pc 18 also belongs to the interval. That the missing pc is missing will be detected in step 4&5 when traversing back the ArrayList by the number of parameters. If it is not a static method invocation and there is not enough possible method receivers stored in that ArrayList to traverse back, the missing pc is added in retrospect.

```
1  i = index in line number attribute which contains the invoke.*
       opcode
2
3  if (lineNrAttribute[j>i] <= lineNrAttribute[i]) {
4
5      if (isAlternating(i,j))
6          return getAlternatingInterval(i, j);
7
8      // non-alternating
9      b = j which has the smallest line number after j;
10     a = corresponding index to b; // index<k which has the same
           line number as b||k
11     return getNonAlternatingInterval(a, b);
12
13 } else if (lineNrAttribute[j<i] == lineNrAttribute[i]) {
14
15         b = i;
16         a = j;
17         return getInterval(a, b);
```

```
18
19  }
```

Code 3.5: Multiple line interval algorithm

This pseudocode will give a very rough idea how the invocation interval can be read out. The detailed implementation can be seen in the classes `ch.unibe.scg.nullSpy.instrumentator.contro` and `ch.unibe.scg.nullSpy.instrumentator.controller.methodInvocation.M` .

3.3.2 Obtaining Variable Data Difficulties - Fields

We previously mentioned that NullSpy first collects data about fields and we met some difficulties regarding this. By using the method `loopBody()` from class `Javassist.expt.ExprEditor` (Subsection 3.2.2.2) for gathering information caused those troubles because it loops through the GIVEN `methodInfo` which contains the bytecode sequence of a method. Even after finding a field assignment and adding some additional bytecode to it, the method still iterates through the unchanged `methodInfo` it got as parameter without the additional bytecode. Getting the right start-pc of a field assignment is not a straight forward process as it may seem.

After entered extra instructions the method `loopBody()` iterates onwards until it finds a key which indicates an access to a field. Normally, Javassist provides a method with which the start-pc of the field access can be find out easily, but in our case not due to bytecode alternation. This method returns the start-pc of the field as if there was no changes, however it actually has to return a bigger start-pc than it does. The start-pc is needed to distinguish in what category the field has to be assigned to (Subsection 3.2.2.2).

To find the right start-pc of the field assignment, we compare the start-pc returned by a method of Javassist with the `afterStorePc` of the previously found field assignment. Every time an assignment is found we store it as a reference for the next assignment to obtain the right data. Even for the store-pc of an assignment ("*put.**" the last found field is es-

sential. If there is more interest how those pcs are obtained, please see the class `ch.unibe.scg.nullSpy.instrumentator.controller.F` .

3.3.3 Bytecode Adaptation Difficulties

The reason why we decided to use Javassist for our NullSpy project is because the thought of only using the source-level API Figure 2.1 to implement everything. It would have been much easier to only operate at source-level instead of learning how to read bytecode or extract data from it or enter extra code into it, yet at the end we still had to do everything at bytecode-level.

One big problem encountered while inserting the test method between an assignment and a closing bracket `}` . We tried to insert additional code as shown in Figure 2.3 by specifying the exact line number where it should be entered in source code. Unfortunately Javassist first checks the specified line whether it contains some code (only symbols excluded). If there is no code at that line it computes the next line that contains some and inserts the test method right before it. Please visualize a situation where for example a local variable is created/instantiated at the end of a `if-body` . In this situation Javassist adds the extra code right before the next code line which is outside the existing scope of the local variable.

```
1 Object fontName;
2 if (fName.startsWith("A")) {
3     fontName = new Arial();
4 } else {
5     fontName = new Calibri();
6 }
```

Code 3.6: Bytecode adaptation example

```
1 Object fontName;
2 if (fName.startsWith("A")) {
3     fontName = new Arial();
4  } else {
5     ch.unibe.scg.nullSpy.runtimeSupporter.VariableTester.
          testLocalVar("", "", "", "", "", "", fontName, 0, 0, 0, 0,
          0);
```

```
6       fontName = new Calibri();
7   }
```

Code 3.7: Wrong Adaptation to Code 3.6

That is why changed the way to insert the test method at bytecode-level. Like this we first have to build up the bytecode sequence and then enter it before a specific pc. Please take a look at the class `ch.unibe.scg.nullSpy.instrumentator.controll` how the bytecode sequence is created.

There are of course many other small problems during the implementation of NullSpy but these mentioned are the most troublesome ones.

3.4 Limitations

During the implementation of NullSpy we had to change the concept few times due to limitations or an overhead that could have grown immense.

Our first idea of how NullSpy could track the NullPointerExceptions is to gather information about variable assignment, which is the case now, and also inject another test method right before each method invocations. In small projects this way could have worked fine but in larger projects which could contain hundreds of classes with a lot of method invocations the execution time would be strongly influenced.

Being able to collect data about variables we still are not able to avoid injecting bytecode even it affects the runtime. Related to this issue a limitation about Javassist is mentioned before namely inserting bytecode right before a closing bracket (Subsection 3.3.3). Because of this it highly depends on the location where additional code should be inserted whether using the source-level API is possible or not. Avoiding checking all the location where something new should be added it is more secure to do this on bytecode-level. But in cases like entering something at the beginning or at the end of a method body the source-level is just fine.

Another limitation of Javassist to mention is that it does not provide anything to get information about method receivers. It only allows one to extract information about local variables, fields and method invocations itself. The programming language Java also does not provide any information about the method receiver, since the exception object or the stack trace element just contains the class, file, method name and the line number. Nonetheless making possible to gather data about them the algorithm discussed before (see Algorithm 3.2) fulfills the missing task.

The last limitation we want to discuss here is that unfortunately NullSpy is not capable to track the root of NullPointerExceptions that are originated in a null which was returned of a method call or in an element of a collection. Why those situations are not supported in NullSpy is because of the impossibility way to store them, e.g. imagine a nested ArrayList or a HashMap and a never ending return value of method invocations. So we lack something tangible to compare with each other, get a hit and read the location out of the hit (Section 5.2).

4 Validation

This chapter will provide some numbers to compare the execution time each project takes, the original and the instrumented one. To get the numbers we instrumented the example project JHotDraw.

4.1 JHotDraw

JHotDraw[1] also served to check whether the logic of the bytecode manipulation behind NullSpy is working as desired. JHotDraw is an open-source Java GUI framework for technical and structured Graphics. It is big enough to get reliable numbers and it provides many different cases we had to take care of in NullSpy. Also thanks to Nevena Milojković and her experience with the combination Javassist and JHotDraw we as well decided to test NullSpy on it.

[1] http://www.jhotdraw.org/

4.2 Execution Time Difference

How did we get the numbers? First of all, of course another executable jar file of the modified project has to be created. The steps to it are followings: load project, modify project, store modified project, build project that creates an executable jar file of the original project and one of the modified one. We then simulate the terminal with a Java class to run each jar thirty times and calculate the average. The next table lists each execution time and the average:

	Original project	Modified project
1	7.223	7.442
2	7.427	7.738
3	7.171	7.893
4	7.035	7.379
5	7.488	7.458
6	7.194	7.691
7	6.849	7.472
8	7.286	8.068
9	7.083	7.519
10	7.27	7.55
11	7.16	7.177
12	7.161	7.55
13	7.225	7.223
14	7.037	7.316
15	7.067	7.54
16	6.975	7.77
17	7.287	7.117
18	7.52	7.488
19	7.303	7.35
20	6.942	7.307
21	7.147	7.535
22	7.222	7.644
23	7.145	7.32
24	7.334	8.187
25	7.364	7.488
26	7.269	7.942
27	7.441	7.943
28	7.223	7.467
29	6.912	7.647
30	7.363	7.784

Table 4.1: Execution time

Original project	Modified project
7.2041	7.566 834

Table 4.2: Average time

```
The runtime of the modified project takes 0.362734s longer
```

than the original one, this means after adding additional code to the project results in approximately **5%** overhead. This small overhead is quite negligible. But this numbers have to be interpreted with caution because the overhead is only measured on JHotDraw. It only has few variable assignments but how much would it go up if there are tons of them?

5

Conclusion and Future Work

Now we have come so far to retrospect (step back and have a critical loot at) the entire project for summarizing what goals we have achieved so far and for proposing further aims that could be completed in the future. In a small section we also want to talk about the gained experience during the whole project.

5.1 NullSpy

Happily, we could say here that we successfully managed to meet the main purpose we have set at the beginning of the project. NullSpy is now capable to tracking the NullPointerExceptions to its root and provide the user with more information about its origin without spending much time on finding it. The most important steps which lead to the success of NullSpy are listed below:

1. Extract and gather information about all method receivers because method invocations on these causes NullPointerExcetptions. To achieve this, we developed an algorithm (Al-

gorithm 3.2) with the function of finding method receivers and extracting data from it by only doing static bytecode analysis. The information about it is then stored as an external csv file. It is needed for a comparison in in a later step.

2. Again collecting data, but this time about variable assignments namely local variables and fields. Here next to the static bytecode analysis additional instructions are inserted to the class files to check right after the assignment if the variable is assigned to the special value `null` . If this is the case everything about the variable is stored in a HashMap which serves for a comparison too. It is stored in a HashMap because if a variable is assigned to another object than null, it will be deleted from the HashMap.

3. NullSpy does only handle the uncaught NullPointerException which means we can wrap up the main method with a catch block. In this catch block the class name and the line number where the NullPointerException occurred is extracted from the stack trace. This information is passed to a method in which the parameter helps to find the guilty method receiver. With the hit the exact location where that variable was set to null can be derived from the HashMap.

4. All the additional needed classes are added to the project after it is modified and stored in a folder the user has chosen. Being able to run the modified project of course a jar file is created. In our case JHotDraw already provides a *build.xml* which we had to alternate a little bit.

During the implementation we were encountered with many difficulties. Some of those we were able to solve and some not unfortunately. Those unsolved ones could be proposed as goals of future work. The next section will list some of them.

5.2 Future Work

5.2.1 Support unsupported method receivers and variable assignments

As mentioned in (Section 3.4), if the cause of a NullPointerException lies in an element of a collection or in an object that is returned by a method invocation, it cannot be tracked to its assignment location. Come up with a way to extract and store information could be a future aim. Of course gathering information about the method receivers has to be improved too.

5.2.2 Track NullPointerException root for all projects

As now our goal in this project was tracking null assignments of JHotDraw. First of all, we had to make sure that building JHotDraw works properly with the additional classes. To fulfil this task manual changes on the build.xml was necessary. If this can be done automatically by NullSpy it would be much better.

Next to this we only looked at the assignment and method receiver types which appear in JHotDraw itself. There could be other types that are not covered in NullSpy depending on projects. In case NullSpy is used on projects that contain not supported issues improving it to cover them can be added to the toDo list.

5.2.3 Plug-in for Eclipse

Last target for future work to mention here is transform NullSpy into an Eclipse compatible plug-in project. After integrated it with Eclipse the null tracking can be started without any expenses on how to manipulate the build.xml if there is one or even bother to create an executable jar file out of it.

For now, we can only think of these future work that could improve NullSpy to give it more value.

5.3 Personal Experience

At the beginning, after reading the description of the project I had no clue how to approach its goal at all. Since this is my first big project on my own but with help from two experienced research assistants I learned a lot, especially in the matter of programming.

As the project is all about manipulating class files, here with Javassist, we first had to learn how bytecode is constructed and get familiar with Javassist. Luckily there is a very good tutorial that teaches one how to use it. This class library however does not cover everything we needed. Thanks to this we had to deal with the lack extensively and learned quite a lot about working on bytecode-level.

Next to getting familiar with bytecode we also had to invent algorithms like that of extracting method receivers from the bytecode or that of getting the pc interval of a method receiver. It was quite interesting to invent those as these are the first ones that fulfill more complex tasks.

Debugging: It is not as easy as it seems. Sometimes it took hours to find the cause of a small bug, and after fixing it another occurs. The reason why it took that long to debug is because along with checking whether the logic of our code was right we also had to find the right bytecode position to ensure the implementation does what it is created for.

Coding beautifully the way so that it will not smell was another challenge due to lack of experience. Sometimes I tended to put everything in one class instead of abiding by the single responsibility principle. Therefore, refactoring the whole project multiple times was necessarily which also took some time. Best before starting to code is to clearly thinking through what is needed, how the structure should look like and what kind of responsibility each class of the project should take for gaining time for other things.

We also came in contact with the building XML file and the jar file. Concerning the XML file, we gained experience by learning a new language by modifying it that it also creates a jar file out of the modified project.

6

Anleitung zu wissenschaftlichen Arbeiten

NullSpy is a program which helps Java developers to find the root of NullPointerExceptions by providing an additional link next to the common stack trace. The key idea behind NullSpy is to help developers to save time fixing bugs which are caused by NullPointerExcpetions. Of course this approach tries to provide this service by keeping the overhead to a minimum. To demonstrate how NullSpy works, this chapter serves as a small tutorial that takes JHotDraw as the testing project to show the feature of NullSpy.

6.1 Installation

For this tutorial JHotDraw is needed. In order to download it, visit the following site and download "JHotDraw6.0 beta 1" (version we worked with during the implementation of NullSpy):
www.jhotdraw.org
Unpack the archive and store it to a location of your choice.

To be able to run JHotDraw additional libraries are used, that are not included in JHotDraw itself. Following libraries are to be downloaded:
“batik-all-1.7": www.java2s.com/Code/Jar/b/Downloadbatikall17jar
“jdo": www.java2s.com/Code/Jar/j/Downloadjdojar
Again unpack those archives and store them in a lib folder in your JHodDraw project.

Running the tests of JHotdraw with the commando line the location of libraries “hamcrest" and “junit" are needed, which are normally also downloaded when the programming environment Eclipse[1] is downloaded.

[1]Can be downloaded here: www.eclipse.org/downloads

Bibliography

[1] Shigeru Chiba. Load-time structural reflection in Java. In *Proceedings of ECOOP 2000*, volume 1850 of *LNCS*, pages 313–336, 2000.

[2] Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In *In Proceedings of the second International Conference on Generative Programming and Component Engineering (GPCE'03)*, volume 2830 of *LNCS*, pages 364–376, 2003.

[3] Haidar Osman. Null check analysis. In *Extended Abstracts of the Eighth Seminar on Advanced Techniques and Tools for Software Evolution (SATToSE 2015)*, pages 86–88, July 2015.

”Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.âĂIJ