



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 `NullPointerException`s are very serious in Java projects. When a `NullPointerException` occurs the developer is provided only with a stack trace to where the exception happened. This only gives insight into the effect of the fault but not into its cause. So we have to ask the question when and why this reference was set to null.

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

By tracking the null assignments the debugging after a `NullPointerException` will be simplified.

Contents

1	Introduction	1
2	Technical Background	3
2.1	Javassist	3
2.2	JAD	6
3	NullSpy	7
3.1	High level overview/Rough Scheme	7
3.2	Low level overview	8
3.2.1	Method Receiver Data Collection	9
3.2.2	Variable Data Collection	10
3.2.2.1	Local Variable	10
3.2.2.2	Instance and Class/Static variables (Fields)	13
3.2.3	Bytecode Adaptation	15
3.3	Challenges	16
3.3.1	Obtaining Method Receiver Data Difficulties	16
3.3.2	Obtaining Variable Data Difficulties - Fields	21
3.3.3	Bytecode Adaptation Difficulties	22
3.4	Limitations	23
4	Validation	25
4.1	JHotDraw	25
4.2	Execution Time Difference	25
5	Conclusion and Future Work	28
5.1	NullSpy	28
5.2	Future Work	29
5.2.1	Support unsupported method receivers and variable assignments	29
5.2.2	Track NullPointerException root for all projects	29
5.2.3	Plug-in for Eclipse	30

CONTENTS

iii

6 Anleitung zu wissenschaftlichen Arbeiten

31

1

Introduction

Nowadays, certainly every programmer is confronted with `NullPointerExceptions` in big Java Projects, whether it is for an enterprise or for private purposes. Not to mention even in small Java Projects they are also heavily present.

So what are those `NullPointerExceptions`? This thesis is going to attach importance to Java that is a concurrent, class-based, object-oriented programming language. We chose Java because `NullPointerExceptions` are more serious in this language than in others, e.g. Smalltalk. `NullPointerException` is a `RuntimeException`. In Java, an object reference can be assigned with a special null value. The exception is thrown when an application attempts to use an object reference that has the null value. (There are multiple ways this exception can be thrown, like: Calling an instance method on the object referred by a null reference; Accessing or modifying an instance field of the object referred by a null reference and so on.) In Java Projects developers always have to deal with a huge amount of references which means avoiding these `NullPointerExceptions` is as good as impossible.

On regular meetings among programmers they report what they have been doing and what they are planning to do for the next few weeks. But all too often it is stated that they are trying to fix bugs or have spent a lot of time fixing them. If there would be a way to minimize the time fixing exceptions and allow to work more efficiently, projects would progress much faster.

The main goal of the NullSpy application takes a step to that ideal vision. Anytime

developers are facing a `NullPointerException` they don't have to spent time on debugging finding where and why a reference was set to null. With `NullSpy` the exact location of the null assignment is shown next to the ordinary stack trace the Java virtual machine produces.

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 works/technologies used in this project.

2.1 Javassist

Javassist or Java Programming Assistant¹, a subproject of Jboss, is a class library which allows you 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.

¹<http://jboss-javassist.github.io/javassist/>

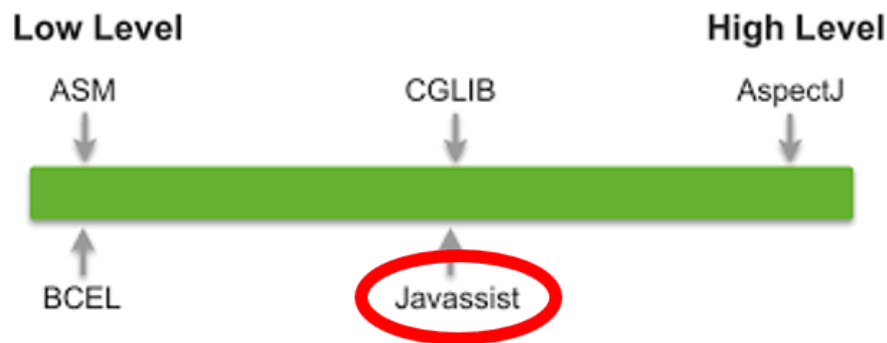


Figure 2.1: Bytecode modification levels

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 normal 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 to give you an idea how the bytecode manipulation works.

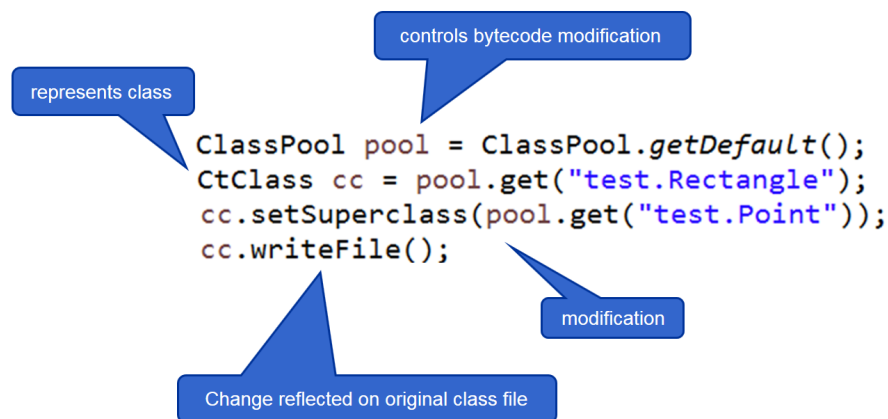


Figure 2.2: Javassist example

First a `ClassPool` object is obtained that controls bytecode modification with Javassist. With the `ClassPool` a class file can be read on demand for constructing a `CtClass` object. The class `CtClass` (compile-time class) is just an abstract representation of a class file

which means all manipulations are performed on the CtClass. With the method invocation `get()` on `ClassPool` a reference to the class file `test.Rectangle` is obtained. In this example the superclass of `test.Rectangle` is just changed to `test.Point`. If the changes are done, the method call `writeFile()` on `CtClass` is necessary to make sure that the changes are reflected on the original class file.²

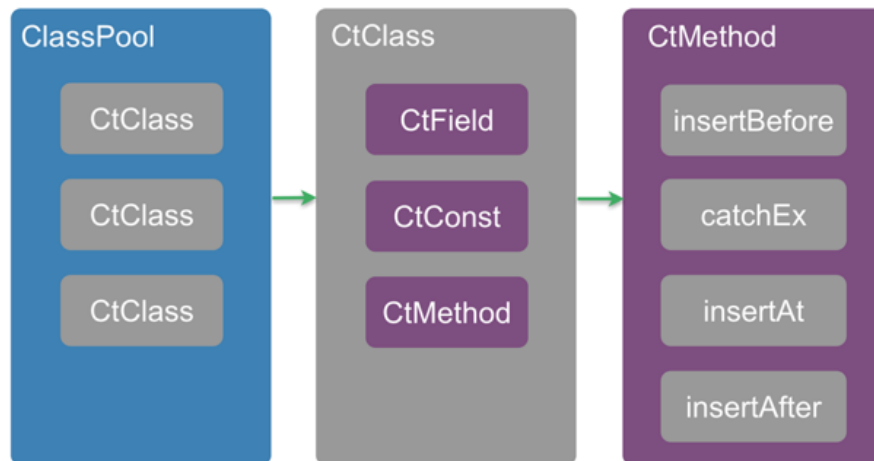



Figure 2.3: Javassist Modules

Figure 2.3 gives you 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` is just the abstract representation of a class file on which modifications are done. Like typical classes, it can hold several compile-time fields, constants or methods. While speaking about bytecode manipulation all the time, nothing but editing methods is mainly meant. 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 `catchBlock` can be added.

²Getting started with Javassist: <http://jboss-javassist.github.io/javassist/tutorial/tutorial.html>

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



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

Figure 2.4: Inserting code example

2.2 JAD

Java Decompiler³ is a decompiler and a Eclipse plugin for the programming language Java. A short explanation what a decompiler is: a computer program that takes an executable file as input, and attempts to create a high level, compatible source file that does the same thing. So it is used in software reverse engineering.

```
// Method descriptor #28 ([Ljava/lang/String;)V
// Stack: 2, Locals: 13
public static void main(java.lang.String[] args);
0  getstatic java.lang.System.out : java.io.PrintStream
3  ldc <String "\nMethod main starts."> [35]
5  invokevirtual java.io.PrintStream.println(java.lang.
8  aconst_null
9  putstatic isFieldOrLocalVariableNullExample.Main.
12 aconst_null
13 astore_1 [d]
14 aconst_null
15 astore_1 [d]
16 getstatic isFieldOrLocalVariableNullExample.Main.
19 putstatic isFieldOrLocalVariableNullExample.Main.
22 aload_1 [d]
23 putstatic isFieldOrLocalVariableNullExample.Main.
26 getstatic isFieldOrLocalVariableNullExample.Main.
29 astore_1 [d]
30 aconst_null
31 astore_2 [d2]
32 aload_2 [d2]
33 astore_1 [d]
34 new isFieldOrLocalVariableNullExample.Person [45]
37 dup
38 invokespecial isFieldOrLocalVariableNullExample.Person() [47]
41 astore_3 [p]
42 new isFieldOrLocalVariableNullExample.Person [45]
45 dup
46 invokespecial isFieldOrLocalVariableNullExample.Person() [47]
49 putstatic isFieldOrLocalVariableNullExample.MainAssignToNull.o : isFieldOrLocalVariableNullExample.Person [48]
52 invokestatic isFieldOrLocalVariableNullExample.Person.say() : java.lang.Object [50]
55 checkcast isFieldOrLocalVariableNullExample.Person [45]
58 astore_4 [p2]
```

```
3 public class MainAssignToNull {
4     public static Object o = null;
5     public static Object p;
6     public static Object c;
7     public static Integer i;
8     public static NullObject nullObject;
9     public static Person o;
10
11     public static void main(String[] args) {
12         // long startTime = System.nanoTime();
13
14         System.out.println("\nMethod main starts.");
15
16         o = null;
17         Object d = null;
18         d = null;
19
20         o = o;
21         o = d;
22         d = o;
23         Object d2 = null;
24         d = d2;
25
26         Person p = new Person();
27         o = new Person();
28         Person p2 = (Person) Person.say();
29         p.o = null; // aload, aconst, putfield Person.o
```

Figure 2.5: Decompile example

JAD is used in NullSpy since after running NullSpy on a project only the modified bytecodes are available. To simplify the check whether the modification by Javassist, e.g. inserting source code, has succeeded, a decompiler is needed.

³<https://sourceforge.net/projects/jadclipse/>

3

NullSpy

As earlier explained in the introduction (1), this project is about providing the user with additional stack trace where the origin of a `NullPointerException` is actually rooted. Briefly worded, it shows the developer the exact location of where a method receiver, which causes the NPE, was assigned to null.

This is the main chapter of the thesis. Here we would like to give you 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 let you know what challenges we were encountering during the implementation and about the limitations we planned for future work (??).

3.1 High level overview/Rough Scheme

The general approach of NullSpy is to statically analyze and add additional bytecode to a project. After reading the section Javassist (2.1) you should be more familiar with how bytecode manipulation with Javassist works.

What NullSpy first does is loading the project you want it to 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 “Low level overview”.

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 do additional stuff like tracking the null assignment. Because only the class files are accessed previously, the result which is stored in the destination folder is as expected only the modified bytecode.

The reason why we store the modified project in another directory than overwriting the existing one is because we would like to only involve Javassist library in the loading and editing part. This means the execution of the altered project does not include Javassist. This way the user does not have to download Javassist and integrate it to the project.

How do we check whether the instrumentation worked and the project really tracks the null assignment? The answer is wrapping the modified project into a jar file with which the modified project can be executed in the terminal or in Eclipse.

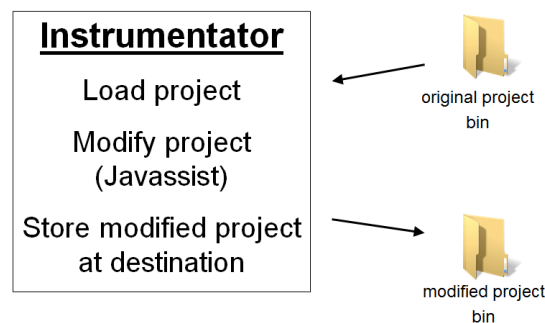


Figure 3.1: Modification overview

3.2 Low level overview

NullSpy only handles the unhandled `NullPointerException`, meaning the main method of a project is wrapped with a `catchBlock` where the NPE is treated. The idea in this `catchBlock` is to load the data about the assignments and about the method receiver to compare them with each other. With the stacktrace the line number and the name of the Java class of where the NPE occurred can be figured out. We look them up in the method receiver information and get some hold points with which we look up in the assignment information. If there is a match we provide the user with an additional stacktrace next to the usual one. Thanks to that link that points directly to the NPE root, debugging will

be easier or it can even be skipped or avoided completely. To be able to provide this function data about the method receiver and variable assignment 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 only be thrown if a method call was performed on a method receiver which is null. That means we have to gather information about the method receiver. Next to this also information about the variable assignments has to be collected to know the exact location of the null assignment. 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.

The implementation behind the `NullSpy` concept will be explained more fully.

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

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 you will learn more in chapter 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*, *invokespe-*

cial, *invokestatic*, *invokevirtual*. The invocation opcode *invokedynamic* facilitates the dynamic-typed languages¹ through dynamic method invocation. In our case it can be ignored because NullSpy only supports the static-typed language² 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.

3.2.2 Variable Data Collection

While going through the bytecode attention is paid to some opcodes³. 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 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 subsection 3.3.

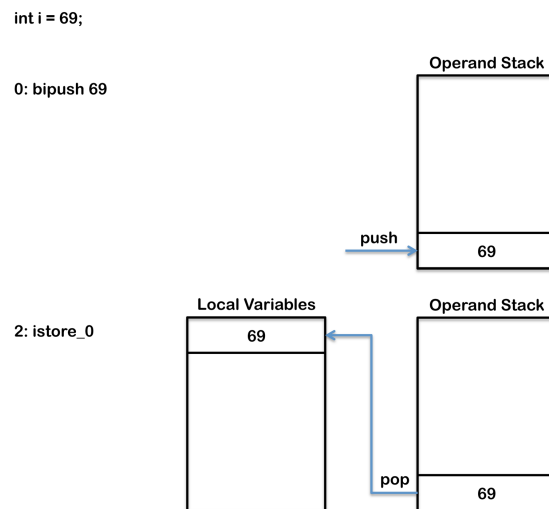
3.2.2.1 Local Variable

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 you a small bytecode introduction.

¹Language whose type checking is usually performed at runtime.

²Language whose type checking is performed at compile time.

³Operation code: Machine language instruction.

Figure 3.2: Local variable creation bytecode⁴

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 figure 3.3) 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.javadraw.JavaDrawApp
[pc: 0, pc: 31] local: newDrawing index: 1 type: org.jhotdraw.framework.Drawing
[pc: 5, pc: 31] local: d index: 2 type: java.awt.Dimension
[pc: 22, pc: 31] local: newDrawingView index: 3 type: org.jhotdraw.framework.DrawingView
```

Figure 3.3: 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 pc⁵ 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.

```
Local variable table:
[pc: 0, pc: 456] local: args index: 0 type: java.lang.String[]
[pc: 16, pc: 456] local: m index: 1 type: isFieldOrLocalVaria
[pc: 18, pc: 456] local: i index: 2 type: int
[pc: 31, pc: 38] local: k index: 3 type: java.lang.Object
[pc: 44, pc: 456] local: d index: 3 type: java.lang.Object
[pc: 63, pc: 456] local: d2 index: 4 type: java.lang.Object
```

Figure 3.4: Local variable table entries with same slot

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 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: 27]
[pc: 8, line: 28]
[pc: 16, line: 29]
[pc: 18, line: 30]
```

Figure 3.5: Line number table

⁵Program counter/instruction pointer: A processor register that indicates where a computer is in its program sequence.

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.expr.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:

- (l...l: put value on operand stack for assigning to a field)
1. aload_0, l...l, putfield
 2. l...l, putstatic
 3. aload.*, l...l, putfield
 4. aload_0, (getfield)+, l...l, putfield
 5. getstatic, (getfield)*, l...l, putfield

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

- 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)

In NullSpy we call these fields *"directFields"*. After reading *"direct"* you surely think of 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 localVar or a field)

```

public class A {

    private B b = new B();
    private static B b2;

    public void x() {

        // directFields
        this.b = ...; // aload_0, |...|, putfield
        b2 = ...; // |...|, putstatic

        B b3 = new B();

        // indirectFields
        b3.c = ...; // aload_*, (getfield)*, |...|, putfield
        b3.c.d = ...
        b.c = ...; // aload_0, (getfield)+, |...|, putfield
        b.c.d = ...;
        b2.c = ...; // getstatic, (getfield)*, |...|, putfield
        b2.c.d = ...;
    }
}

public class B {
    public C c = new C();
    ...
}

public class C {
    public D d = new D();
    ...
}

public class D {
    ...
}

```

Once included more bytecode, getting the data of fields needed some adaptation (see 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.

The added bytecode represents a static method of a class named *ch.scg.nullSpy.runtimeSupporter.Variable* 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 (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 3.1. Next to the instrumentation supplementary supporter classes are added to the project. The most important ones are *ch.unibe.scg.nullSpy.runtimeSupporter.VariableTester* which tests whether a variable is null or not and *ch.unibe.scg.nullSpy.runtimeSupporter.NullDisplayer* 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 you an understanding of few difficulties occurred during the implementation of NullSpy.

3.3.1 Obtaining Method Receiver Data Difficulties

Aforementioned in subsection *Method Receiver Data Collection* 3.2.1 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 3.6: 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 3.7: Bytecode to figure 3.6

```

[pc: 0, line: 31]
[pc: 3, line: 32]
[pc: 11, line: 31]
[pc: 15, line: 33]
[pc: 17, line: 34]

```

Figure 3.8: Line number table/interval to figure 3.6

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 (3.5) has to be consulted too. There is no method receiver in the figure because the method receiver would contain a method invocation itself ("*Iconkit.instance()*"), 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 you how the bytecode and the line number attribute for this looks like. Applying the algorithm (3.2.1) 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

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 (3.8) 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 3.9: 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 3.10: Bytecode to figure 3.9

```
[pc: 32, line: 124]
[pc: 38, line: 125]
[pc: 47, line: 128]
[pc: 51, line: 127]
[pc: 54, line: 128]
[pc: 57, line: 127]
[pc: 59, line: 130]
[pc: 61, line: 131]
```

Figure 3.11: Line number table/interval to figure 3.9

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 3.12: 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 3.13: Bytecode to figure 3.12

```

[pc: 0, line: 148]
[pc: 8, line: 149]
[pc: 13, line: 150]
[pc: 14, line: 151]
[pc: 18, line: 152]
[pc: 22, line: 153]
[pc: 23, line: 154]
[pc: 27, line: 155]
[pc: 28, line: 151]
[pc: 31, line: 150]
[pc: 34, line: 149]
[pc: 44, line: 158]

```

Figure 3.14: Line number table/interval to figure 3.12

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 3.15: Uncomplete interval example

```

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

```

Figure 3.16: Bytecode to figure 3.15

```

[pc: 0, line: 17]
[pc: 16, line: 21]
[pc: 18, line: 22]
[pc: 19, line: 23]
[pc: 22, line: 24]
[pc: 26, line: 25]
[pc: 28, line: 26]
[pc: 29, line: 25]
[pc: 32, line: 23]
[pc: 35, line: 28]
[pc: 46, line: 201]

```

Figure 3.17: Line number table/interval to figure 3.15

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.

```
i = index in line number attribute which contains the invoke.* opcode
if (lineNrAttribute[j>i] <= lineNrAttribute[i]) {

    if (isAlternating(i, j))
        return getAlternatingInterval(i, j);

    // non-alternating
    b = j which has the smallest line number after j;
    a = corresponding index to b; // index < k which has the same line number as b // k
    return getNonAlternatingInterval(a, b);

} else if (lineNrAttribute[j<i] == lineNrAttribute[i]) {
    b = i;
    a = j;
    return getInterval(a, b);
}
```

This pseudocode will give you the main or a very rough idea how the invocation interval can be gained. If you are interested in knowing the exact way, please see the implementation of the class `ch.unibe.scg.nullSpy.instrumentator.controller.methodInvocation.MultipleLineManager` and `ch.unibe.scg.nullSpy.instrumentator.controller.methodInvocation.MethodInvocationAnalyzer`.

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` (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 startPc 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 starting pc of the field access can be find out easily, but in our case not due to bytecode alternation. This method returns the starting pc of the field as if there was no changes, however it actually has to return a bigger starting pc than it does. The starting pc is needed to distinguish in what category the field has to be assigned to (3.2.2.2).

To find the right starting pc of the field assignment we compare the starting pc which a method of Javassist returns with the *afterStorePc* of the previously found field assignment. Every time an assignment is found we store it as a reference for obtaining right data. Even for the storing pc of an assignment ("*put.** " the last found field is essential. If there is more interest how those pcs are obtained, please see the class *ch.unibe.scg.nullSpy.instrumentator.controller.FieldAnalyzer*.

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 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 2.4 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.

```
Object fontName;  
if (fName.startsWith("A")) {  
    fontName = new Arial();  
} else {  
    fontName = new Calibri();  
}
```

Figure 3.18: Bytecode adaptation example

```

Object fName;
if (fName.startsWith("A")) {
    fName = new Arial();
} else {
    ch.unibe.scg.nullSpy.runtimeSupporter.VariableTester.testLocalVar("", "", "", "", "", fName, 0, 0, 0, 0);
    fName = new Calibri();
}

```

Figure 3.19: Wrong Adaptation to 3.18

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.controller.ByteCodeAdapter* 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 *NullPointerException*s 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 (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 3.2.1) 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 (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 JHotDraw¹

This project 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 look 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 tell you 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 of tracking the `NullPointerException`s 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 `NullPointerException`s. To achieve this, we developed an algorithm (3.2.1) 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 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 you have read earlier (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.

6

Anleitung zu wissenschaftlichen Arbeiten

This consists of additional documentation, e.g. a tutorial, user guide etc. Required by the Informatik regulation.

Bibliography

”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.”