UNIVERSITÄT
BERN

NullSpy

An approach to pinpoint the origin location of a null reference

Bachelor Thesis

Lina Tran

from

Biel/Bienne BE, Switzerland

Faculty of Science
University of Bern

20. December 2016

Prof. Dr. Oscar Nierstrasz

Nevena Milojković, Boris Spasojević

Software Composition Group

Institute for Computer Science

University of Bern, Switzerland

Abstract

A previous study found that null pointer exceptions are the most frequently occurring exceptions in Java projects. Also, it is difficult to debug because a developer is only provided with a stack trace to the line of code where the exception was thrown. This only gives insight into the effect of the fault but not into its cause.

The aim of the project is to provide the developer with an additional stack trace. It shows the location where the variable that caused the null pointer exception was initially assigned to null. We attempt to achieve this goal by instrumenting Java source code while striving for minimal execution overhead.

By tracking the null assignments through static analysis and bytecode instrumentation we can achieve a more efficient debugging process after an occurrence of a null pointer exception.

Acknowledgements

First, I would like to show gratitude to *Prof. Dr. Oscar Nierstrasz* and the *Software Composition Group* for the opportunity to make this bachelor project possible.

My special thanks to my tutors *Nevena Milojković* and *Boris Spasojević* who invested a lot of effort to support me and the project. With their help I was able to finish this project successfully.

Finally, I appreciate the help of the people who have taken their time in providing good suggestions and giving me the help required to improve this project.

Contents

1 Introduction

Null pointer exceptions are a commonly occurring run-time exception in object-oriented programming (OOP) languages. In most OOP languages there is a special **null** value that is assigned to references in order to indicate that the reference does not refer to an object. A null pointer exception is caused by invoking a method or accessing a field through a null value reference, *i.e.,* a reference with the null value.

Previous research has found that 35% of conditional checks in Java projects are null checks. This reduces the readability of source code and has a negative impact on performance [6]. It is also considered the number one error Java programmers make[1].

In order to understand the origin of null pointer exceptions we present two situations in which they may occur.

Assume that in the example shown in Code 1.1[2] the variable `ff` at line 5 is assigned the null value as the result of assigning the return value of the method `DNDHelper.processReceivedData(...)` to the variable. This means that the same variable `ff` is null at line 7 and when a method is invoked on this object a null pointer exception is thrown. Because the null pointer exception is raised in the

[1] `http://www.webcitation.org/6lNPzbIyy`

[2] The code snippet is a modified version of code taken from JHotDraw project: `http://www.jhotdraw.org/`

```
1  public void drop(...) {
2      ..
3      try {
4          ..
5          DNDFigures ff = DNDHelper.processReceivedData(...);
6          ..
7          Point theO = ff.getOrigin();
8          ..
9      }
10     catch (NullPointerException npe) {
11         npe.printStackTrace();
12         ..
13     }
14     ..
15 }
```

Code 1.1: Null pointer exception example (I). The null assignment happens in line 5 and the null pointer exception is thrown in line 7.

try-block the following catch-block will handle the exception by printing the stack trace. The stack trace takes the developer to line 7 where the exception was thrown but not to the real culprit which is the assignment at line 5.

A more complicated way a null pointer exception can be triggered is when it involves variables with larger scopes such as class members. Code 1.2[3] shows such a situation. This code snippet defines the class named `DrawApplication` which has the field `fManager` (declared at line 3). This class contains four relevant methods: `open(...)` , `createIconkit()` , `getIconkitManager()` and `setTool(...)` .

The `open(...)` method invokes the methods `createIconkit()` at line 7 and `setTool(...)` at line 9. By calling `createIconkit()` at line 7 the field `fManager` is assigned null. The field is assigned null by performing the method `getIconkitManager()` at line 14, which just returns the null value at line 19.

Afterwards, when the method `open(...)` calls `setTool()` , the attempt to call the method `getComponent()` on field `fManager` at line 24 causes the null pointer exception. The resulting null pointer exception stack trace is shown in Stack trace 1.1:

[3]The code snippet is a modified version of code taken from JHotDraw project: `http://www.jhotdraw.org/`

```
1  public class DrawApplication {
2      ..
3      private IconkitManager fManager;
4
5      protected void open(...) {
6          ..
7          createIconkit();
8          ..
9          setTool(...);
10         ..
11     }
12
13     protected Iconkit createIconkit() {
14         fManager = getIconkitManager();
15         ..
16     }
17
18     protected getIconkitManager() {
19         return null;
20     }
21
22     public void setTool(...) {
23         ..
24         fManager.getComponent();
25         ..
26     }
27     ..
28 }
```

Code 1.2: Null pointer exception example (II). The null assignment happens in line 14 and the null pointer exception is thrown in line 24.

```
Exception in thread "main" java.lang.NullPointerException
        at org.jhotdraw.application.DrawApplication.setTool(DrawApplication.java:24)
        at org.jhotdraw.application.DrawApplication.open(DrawApplication.java:9)
        ...
```

Stack trace 1.1: Common stack trace of a null pointer exception.

In Stack trace 1.1 we can see that the stack trace points only to the line 24, where null pointer exception occured, and it does not track to the location where the variable is assigned the null value [7]. We call this location the ***root of an exception***. The execution stack holds basically no information about the cause of the exception [3]. That means the Java developer has to debug their way to the exception root.

At this point we introduce our project named ***NullSpy*** which supports the developers in situations discussed previously. The main goal of NullSpy is to take a step towards minimizing the time spent debugging null pointer exceptions. NullSpy presents the developer the exact location of the null assignment after an unhandled null pointer exception has occurred. Stack trace 1.2 shows an example of the output.

```
Field this.fManager at line 14 is null: (DrawApplication.java: 14)
Exception in thread "main" java.lang.NullPointerException
        at org.jhotdraw.application.DrawApplication.setTool(DrawApplication.java:24)
        at org.jhotdraw.application.DrawApplication.open(DrawApplication.java:9)
        ...
```

Stack trace 1.2: Stack trace of a null pointer exception with NullSpy.

There are other tools which can help developers debug and find the root cause of a bug. Most notably the Back-in-time debuggers [5]. The main differences between NullSpy and Back-in-time debuggers are:

1. NullSpy targets a particular problem, namely the null pointer exception, while back-in-time debugging is more general.
2. Back-in-time debugging might require the developer to step back many times before finding the root of the exception while NullSpy finds it directly.

Another approach to which could help find the root cause of a null pointer exception is the Object Flow Analysis *i.e.,* tracking the object flow [4]. The main difference here to NullSpy is that NullSpy tracks the null assignments instead of the object flow.

This thesis explains how NullSpy traces the root of a null pointer exception. It also discusses the challenges, limitations and performance impact of this approach.

2 Technical Background

This chapter provides a short overview of technologies used in the implementation of NullSpy.

2.1 Bytecode

The backbone of NullSpy is analysis and modification of Java bytecode. Java bytecode is an abstract machine language that the stack-based Java virtual machine (JVM) can understand and execute. A JVM keeps an operand stack which is modified every time the JVM executes an instruction. The instruction is represented by an operation code (opcode) which also has a string representation.

Since Java bytecode plays an important role in our project, a short introduction to the basics follows. We will also explain some terms that are often used in this thesis.

We start with an easy *HelloWorld* source code example (Code 2.1). The code snippet contains the definition of the class `HelloWorld` with an instance field named `hello` and method `say()` which prints the value of the field. The bytecode of the method `say()` is represented in Bytecode 2.2. We omit the bytecode of the class and field definition since it is not relevant for our example.

The green words in Bytecode 2.2 are the string representations of Java bytecode

```java
public class HelloWorld {
    private String hello = "Hello World!";

    public void say() {
        String result = hello;
        System.out.println(result);
    }
}
```

Code 2.1: The use of the *Hello World* example to explain bytecode.

```
public void say();
     0  aload_0 [this]
     1  getfield HelloWorld.hello : java.lang.String [14]
     4  astore_1 [result]
     5  getstatic java.lang.System.out : java.io.PrintStream [21]
     8  aload_1 [result]
     9  invokevirtual java.io.PrintStream.println(java.lang.String) :
          void [27]
    12  return
        Line numbers:
            [pc: 0, line: 5]
            [pc: 5, line: 6]
            [pc: 12, line: 7]
        Local variable table:
            [pc: 0, pc: 13] local: this index: 0 type: HelloWorld
            [pc: 5, pc: 13] local: result index: 1 type: java.lang.
               String
```

Bytecode 2.2: Bytecode from method `say()` in Code 2.1.

instructions. We will also use the term ***opcode*** when referring to individual instructions. Multiple opcode instructions are referred to as ***bytecode***.

On the left-side of these instructions we see the ***program counter*** (pc) which is a processor register that contains the address (location) of the instruction being executed at the current time. It is also called an **instruction pointer**.

The instructions from pc 0 to pc 4 (lines 2-4 in Bytecode 2.2) represent the source code at line 5 in Code 2.1. Therefore, we use the term ***pc-interval*** to describe a set of instructions that represents a single Java expression. This term will play an important role in later sections. We use the syntax **"<x,y>"** to represent an pc-interval, *e.g.,* the mentioned pc-interval <0,4> represents the source code at line 5 in Code 2.1. The first number indicates where the pc-interval starts, thus we name it ***start-pc*** and the latter one where it ends, hence the name ***end-pc***, both inclusive.

If the opcode represents a variable, on its right side we can see the information about the name and type of the variable as shown in line 3. In case of an invocation opcode, it shows the behavior/method name, the parameter types and the return type as seen in line 7.

Each method bytecode representation holds a ***line number table*** and a ***local variable table*** which are listed underneath the instructions of the method (lines 9-15 in Bytecode 2.2).

The line number table maps the source code line of the method to the pc that indicates the beginning of the bytecode to the corresponding source code line. The line number table in Bytecode 2.2 is represented between the lines 9 and 12: source code line 5 starts with the pc 0 (as shown on line 10), source code line 6 starts with the pc 5 (in line 11) *etc.* This mapping does not apply to cases where a statement is separated into multiple lines.

If we look back to the Stack trace 1.1, we see that the stack trace provides us with a part of the call-chain that led to the null pointer exception. This includes the line numbers in the source code where the invocations happened. The information the line number table holds can be used to recover the information about the actual point in the code where the null was assigned to a reference.

As mentioned before, the bytecode of each method also holds a local variable table which is a list of information about local variables (lines 13-15 in Bytecode 2.2). This table has the information about the *this* reference (if the method is an instance method or a constructor), about method parameters and method local variables, respectively. An example of the *this* reference representation can be seen in the line 14. Each line of this table represents one variable and contains the following:

```
1  private void loadDrawing(String filename) {
2          try {
3              URL url = new URL(getCodeBase(), filename);
4              InputStream stream = url.openStream();
5              StorableInput reader = new StorableInput(stream);
6              fDrawing = (Drawing)reader.readStorable();
7          }
8          catch (IOException e) {
9              fDrawing = createDrawing();
10             System.err.println("Error when Loading: " + e);
11             showStatus("Error when Loading: " + e);
12         }
13     }
```

Code 2.3: The source code for which the local variable table of its bytecode contains two entries , `url` and `e` , that share the same slot.

```
1  private void loadDrawing(java.lang.String filename);
2      0  new java.net.URL [98]
3      ...
4      12  astore_2 [url]
5      ...
6      40  goto 94
7      ...
8      94  return
9  Local variable table:
10  [pc: 0, pc: 95] local: this index: 0 type: org.jhotdraw.samples.
       javadraw.JavaDrawViewer
11  [pc: 0, pc: 95] local: filename index: 1 type: java.lang.String
12  [pc: 13, pc: 40] local: url index: 2 type: java.net.URL
13  [pc: 18, pc: 40] local: stream index: 3 type: java.io.InputStream
14  [pc: 28, pc: 40] local: reader index: 4 type: org.jhotdraw.util.
       StorableInput
15  [pc: 44, pc: 94] local: e index: 2 type: java.io.IOException
```

Bytecode 2.4: Bytecode of Code 2.3.

1. the pc-s which represent the lexical scope of the variable, *i.e.,* the pc-s in between the square brackets
2. the name of the variable
3. the index/slot at which the variable is stored (starting at index 0) and
4. the type of the variable.

In our example, *this* reference has the lexical scope of the whole method, thus pc starts at 0 and ends with 13. *This* reference is the first one in the table, hence it is indexed with 0, and its type is the `HelloWorld` class. The variable `result` has the lexical scope [pc: 5, pc: 13] because it is created from pc 0 to pc 4 and is accessible after it is created, so from pc 5 onwards until the method ends at pc 13.The indexes of the local variable table are important because we use them to get the right local variable we are interested in.

Let us now look at a more complicated example of the local variable table. Consider the source code example Code 2.3 which contains a method `void loadDrawing(String filename)` . The bytecode of this example is shown in the Bytecode 2.4. This method consists of a try-catch-block. We see that the variable `url` (declared at line 3 in Code 2.3, stored at line 4 in Bytecode 2.4 and contained in the local variable table entry at line 12 within the same bytecode) is only accessible within the try-block. The try-block starts with pc 0 and ends with pc 40 and the variable `url` is accessible from pc 13 onwards until pc 40 ([pc: 13, pc: 40]), as it can be seen in the local variable table in Bytecode 2.4.

The local variable `url` is stored at the index 2 in the local variable table in Bytecode 2.4, since the first two indexes occupy *this* reference and the method parameter. As soon as the lifespan of a local variable ends, the slot can be reused by the next instantiated local variable. So, after pc 40, slot 2 is free again. The freed slot can be immediately used by the next declared variable, which is the variable `e` in our example. Looking at line 15 in Bytecode 2.4 we see that the variable `e` also has the slot number 2. This is why we can find local variable tables that contain multiple entries with the same local variable slot.

2.2 Javassist

Javassist or *Java Programming Assistant*[1] [1] [2], a subproject of JBoss, is a library which enables manipulation of the Java bytecode. Since 1999 it is used as an engineering

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

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

Code 2.5: A code snippet that alternates the super class of a class by using the Javassist library.

toolkit in a broad domain, and is still being extended by Shigeru Chiba. It enables developers to manipulate Java bytecode in a simplified way. Examples of this manipulation include 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. A tutorial for using Javassist[2] is available online and was used as a starting point for this work.

Unlike many other bytecode manipulation libraries, Javassist offers two levels of API: **source**-level and **bytecode**-level. Using the source-level API, the user can edit a class file without any familiarity with the specifications of the Java bytecode. Knowledge of the Java language is enough since the API is designed only with the vocabulary of Java. On this level the programmer has to write Java source code and Javassist compiles it automatically. The bytecode-level allows the user modify classes by modifying the bytecode directly.

Let us look at the small example shown in Code 2.5[3] of how the bytecode manipulation with Javassist works. We go through the example line by line.

1. First a *ClassPool* object that controls bytecode modification is obtained. With the *ClassPool* object a class file (".class") can be read on demand for constructing a *CtClass* object.
2. The class *CtClass* (compile-time class) represents the class file. This means that all manipulations are performed on the *CtClass* object. We obtain a reference to the *CtClass* object representing the `test.Rectangle` class by invoking the `get()` method on the *ClassPool* instance.
3. In this example the only bytecode modification done is changing the superclass of `test.Rectangle` to `test.Point`. This change serves only as illustration.
4. Once the bytecode modification is done, the method call `writeFile()` on

[2] `http://jboss-javassist.github.io/javassist/tutorial/tutorial.html`
[3] Example taken from Javassist tutorial.

```
1 public static void main(String[] args) {
2     System.out.println("This is an example class.");
3 }
```

Code 2.6: Initial code.

the instance of *CtClass* is necessary to make sure that the changes are reflected on the original class file. The method `writeFile()` converts the modified *CtClass* object into a class file and stores it on a local disk.

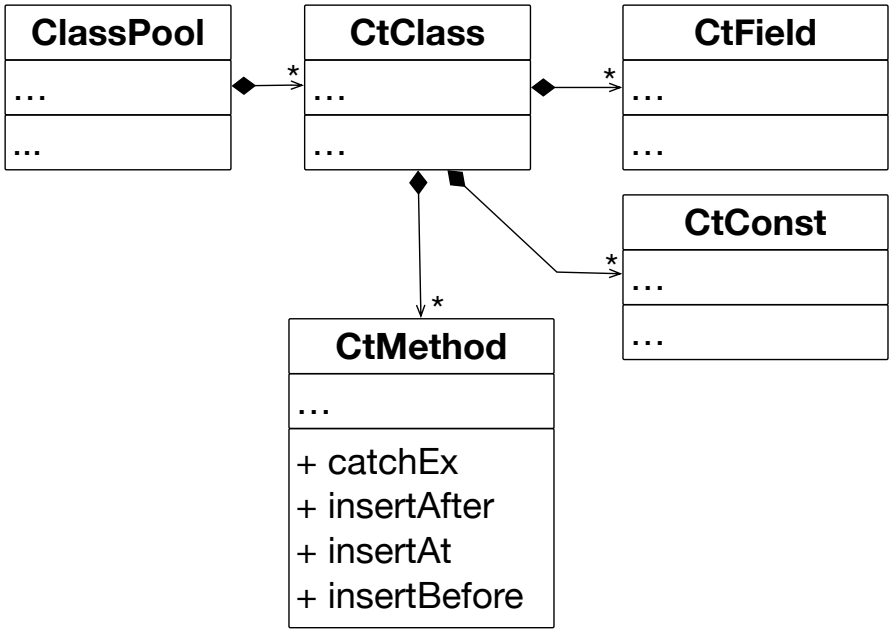

Figure 2.1: Javassist modules.

Figure 2.1 gives an overview of how the main part of bytecode manipulation with Javassist is built up. The *ClassPool* is simply a container of multiple *CtClasses*. As described before *CtClass* represents a class file 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/methods, 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 a method body, at the end or at a specific line.

The next example (Code 2.8) shows how to add code by using Javassist. We want to

```
1  public static void main(java.lang.String[]);
2    0: getstatic     #16  // Field java/lang/System.out:Ljava/io/
       PrintStream;
3    3: ldc           #22  // String This is an example class.
4    5: invokevirtual #24  // Method java/io/PrintStream.println:(Ljava/
       lang/String;)V
5    8: return
```

Bytecode 2.7: Initial bytecode.

```
1  public class BytecodeModifier {
2      public static void main(String[] args) ... {
3          ClassPool pool = ClassPool.getDefault();
4          CtClass cc = pool.get("insertJavaCodeExample.ExampleClass");
5
6          CtBehavior behavior = cc.getDeclaredMethod("main");
7          behavior.insertBefore("System.out.println(\"This is the
               inserted code.\");");
8
9          cc.writeFile();
10         ...
11     }
12 }
```

Code 2.8: Bytecode modifier.

add one line of code at the beginning of the `main()` method in Code 2.6. Bytecode 2.7 is the corresponding bytecode of Code 2.6.

We first obtain a *CtClass* object `cc` (Code 2.8 line 4) which represents the class to be modified. At line 6 we get the *CtMethod* object `behavior` which represents the method we want to modify (in our case, method `main()`). We modify the method by adding the source code `System.out.println("This is the inserted code.");` at the beginning of the method. Because we want to insert code at the beginning of the method we invoke the method `insertBefore()` at line 7. If we wanted to enter extra code at the end of the method we could have called the analogue method `insertAfter()`. Both methods `insertBefore()` and `insertAfter()` expect an argument of type String, which Javassist then compiles, and adds into the bytecode at the specified location (in our situation at the beginning of the `main()` method).

If the modified bytecode (Bytecode 2.9) were to be decompiled, its source code

```
1 public static void main(java.lang.String[]);
2   0: getstatic     #16  // Field java/lang/System.out:Ljava/io/
        PrintStream;
3   3: ldc           #35  // String This is the inserted code.
4   5: invokevirtual #24  // Method java/io/PrintStream.println:(Ljava/
        lang/String;)V
5   8: getstatic     #16  // Field java/lang/System.out:Ljava/io/
        PrintStream;
6  11: ldc           #22  // String This is an example class.
7  13: invokevirtual #24  // Method java/io/PrintStream.println:(Ljava/
       lang/String;)V
8  16: return
```

Bytecode 2.9: Modified bytecode. The modification is visible from line 2 to 4.

```
1 public static void main(String args[]) {
2     System.out.println("This is the inserted code.");
3     System.out.println("This is an example class.");
4 }
```

Code 2.10: Modified code: Decompiled with JAD (Section 2.3)

representation would be as Code 2.10. At line 2 of this code, we can see the changes, *i.e.,* the inserted line of code. However, the actual result is the inserted bytecode represented by the pc-interval <0,5> in the modified bytecode (Bytecode 2.9).

2.3 JAD

Java Decompier (JAD)[4] is a decompiler and an Eclipse plugin for the Java programming language. A decompiler is a 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 often used for software reverse engineering.

As a JAD illustration we decompile the bytecode shown in Bytecode 2.7 of the source code example Code 2.6 and indicate the differences between the decompiled and the original version of the source code. The result of the decompilation is shown in Code 2.11.

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

```
1  import java.io.PrintStream;
2
3  public class HelloWorld {
4
5      public HelloWorld() { }
6
7      public static void main(String args[]) {
8          System.out.println("This is an example class.");
9      }
10 }
```

Code 2.11: The result of the decompiled bytecode shown in Bytecode 2.7.

JAD imported the class `java.io.PrintStream` at line 1 and generated a default constructor at line 5. As already mentioned, we see that the decompiled version of the class `HelloWorld` behaves the same way as the original version, even the decompiled source code does not look identically as the original one.

JAD is in no way a dependency of NullSpy but it was a big help during the implementation phase. After running NullSpy on a project only the modified bytecode is available. Thus, JAD was used to check whether the modification NullSpy made with Javassist was successful. Without JAD one would have to manually look through and compare the resulting byte code. This would have required a lot more effort and time.

3

NullSpy: The Null Pointer Tracking Tool

This project is about providing the user with an additional piece of information next to the common stack trace containing the location of a *null* assignment to the variable that caused the null pointer exception. We use the term ***link*** to describe the additional stack trace line which points to the source code line where the variable that caused the null pointer exception was assigned null.

In this chapter we give an overview of how we implemented NullSpy. We will also address the challenges (Section 3.3) we encountered during the implementation, as well as the limitations of NullSpy (Section 3.4).

3.1 High Level Overview

The general approach of NullSpy is to statically analyze a project in which a null pointer exception occurs and instrument the bytecode. NullSpy statically analyzes the project to extract data from bytecode about variables, *i.e.,* the name of the variable, the source code line number where it is located *etc.* We instrument the code to get the value stored in those variables at run time.

NullSpy is a console application that takes two arguments. The first argument to NullSpy is the local path to the folder containing the compiled class files of the original

project and the second one is the path to the folder in which the user wants to store the modified class files.

We provide the option to choose the destination in case the developer does not want to overwrite the original project with the modified one. This means that the updated bytecode can be stored in another location, keeping the original class files intact. Of course, the original project can also be replaced by the modified one. To remove the instrumented bytecode, the developer just has to recompile the source code and get the original class files when needed.

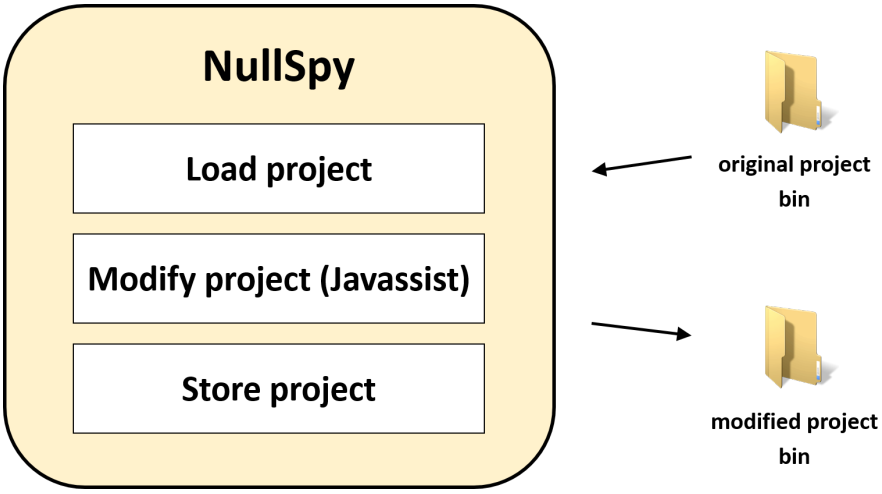

Figure 3.1: High level overview.

NullSpy contains three basic steps which we can see in Figure 3.1: *load, modify, store*. Those three steps of NullSpy carry out the following actions:

1. Load compiled class files of the project into NullSpy.
2. The modification part deals with the bytecode instrumentation and adding modules to the project that support the inserted code.
3. Store modified class files to the chosen output folder.

Loading is done by traversing through all the subfolders of the folder whose local path was given as a first parameter, and extracting all the *.class* files. NullSpy modifies the class file directly when it is loaded. As soon as the modification is done, the class file is stored at the destination folder *i.e.,* the folder whose path is the second parameter of NullSpy. The structure of the source folder is preserved at the destination folder.

3.2 Low Level Overview

In this section we will focus on the bytecode modification. Figure 3.2 shows that the modification part of NullSpy contains three different modules. Each class file of the loaded project will be run through the first two modules. After the instrumentation process (module one and two) is done, we add the last module to the modified project.

The module *"Bytecode data extractor"* extracts data from bytecode. The second module *"Instrumentation"* inserts bytecode into the original code. Lastly, the module *"Run-time supporter"* adds the package that implements the functionality to which the previously inserted bytecode refers to.

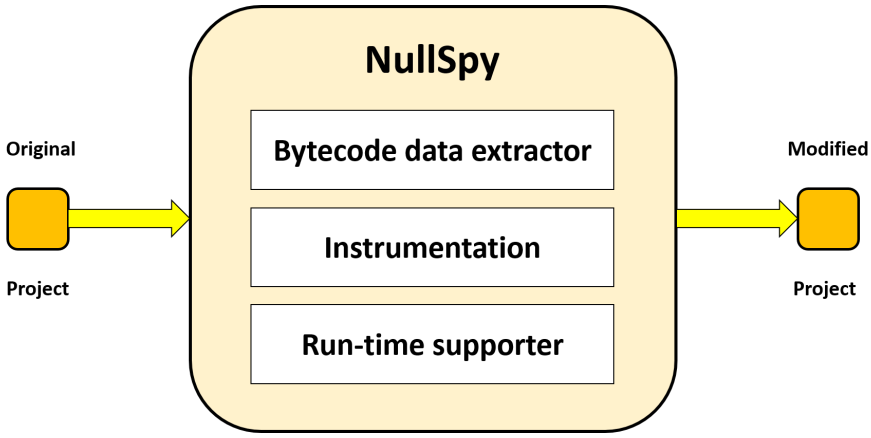

Figure 3.2: NullSpy modification modules.

We will look at these modules in more detail.

3.2.1 NullSpy Process Overview

In this overview of the entire process we explain the reason why we have to extract data from bytecode and what it is used for. Details are discussed in the following subsections.

From the stack trace of a null pointer exception, we can obtain three kinds of information: the class name, method name and the line number of where the null pointer exception happened. The aim of NullSpy is to reveal the location of the null assignment to the variable which caused the null pointer exception. To achieve this goal we need to extract information from the bytecode about **target variables** and **null assignments**.

The first step in the modification part is to statically gather information about the target variables and the variable assignments. We use the term ***target variable*** to refer to variables on which methods are invoked, or whose fields are accessed. These variables are candidates for triggering a null pointer exception. For each target variable we extract the information needed to create its unique identifier (details about this are explained in Subsection 3.2.2). These identifiers are used later on to track the assignments to the aforementioned variable, and to find the null assignment entry for this variable, if any. For every non-primitive variable assignment we also perform static analysis to collect information about the variable. The information again is needed to create its unique identifier.

The second step is to instrument the assignments. After the assignment we insert code which represents a call to the **run-time supporter** package with the variable's identifier. The run-time supporter package checks at run-time whether the variable is assigned null. If this is the case we store the information about this variable.

At the end of the bytecode modification part we wrap the `main()` method into a try-catch-block. The catch-block extracts from the null pointer exception stack trace the information about the location where the exception happened (class name, method name, line number). This is the only information we can get from the stack trace. This information is passed on to the run-time supporter package which compares it with the stored identifiers about the target variables. If there is a match, NullSpy uses this identifier to reveal the information stored about the assignments to this variable. If there is an entry about the null assignment of that variable, we extract from it the line number of the assignment and create the additional link that reveals the location of the null assignment. This link is then added to the common null pointer exception stack trace.

The following sections describe the process of extracting and storing this data.

3.2.2 Target Variable Data Collection

In this section we explain how to get information about the target variable needed to construct a unique identifier. Target variables in the example Code 3.1 are the variables `fToolButton` at line 7, `fToolButton.myIcon` at line 8 and `activePanel` at line 11.

We use the library Javassist to partially extract this data from bytecode. Unfortunately, Javassist does not provide the functionality to directly get all the information we need about the target variable, thus we need to do it manually.

```
1  public class DrawApplication {
2
3      private ToolButton fToolButton;
4
5      protected void open(final DrawingView newDrawingView) {
6          ...
7          fToolButton.tool();                // field access
8          fToolButton.myIcon.toString();     // field access
9
10         JPanel activePanel = new JPanel();
11         activePanel.add((Component)getDesktop(),BorderLayout.CENTER);
                 // local variable as a target variable
12         ...
13     }
14 }
```

Code 3.1: Code snippet to demonstrate the target variable extracting algorithm.

```
1  144  aload_3 [activePanel]
2  145  aload_0 [this]
3  146  invokevirtual org.jhotdraw.application.DrawApplication.
       getDesktop() : org.jhotdraw.contrib.Desktop [271]
4  149  checkcast java.awt.Component [274]
5  152  ldc_w <String "Center"> [276]
6  155  invokevirtual javax.swing.JPanel.add(java.awt.Component, java.
       lang.Object) : void [254]
```

Bytecode 3.2: Bytecode of line 11 from Code 3.1.

```
Line numbers:
    ...
    [pc: 136, line: 10]
    [pc:  144, line:  11]
    [pc: 158, line: 12]
    ...
```

Bytecode 3.3: Part of the line number table of the example bytecode shown in Bytecode 3.2.

3.2.2.1 Finding the Invocation Opcode/End-pc

Let us suppose that we are interested in the target variable `activePanel` of the method invocation `add(...)` at line 11 in the example Code 3.1. Its bytecode is presented in Bytecode 3.2.

To get data about target variables we iterate through the bytecode looking for the instructions that represent a method invocation. To do this, NullSpy looks for the opcodes that match the regex *"invoke.*"*. Each of these opcodes represents one method invocation. There are four invocation bytecode instructions in Java bytecode: *invokeinterface*, *invokespecial*, *invokestatic*, *invokevirtual*.

For each invocation we want to find the target variable. In our example the bytecode instruction that matches the invocation of the method `add(...)` is at line 6 in Bytecode 3.2. Since the bytecode lists the target variable and all the method parameters before the *"invoke.*"* instruction, the pc of this *"invoke.*"* opcode will represent the **end-pc** of the method invocation pc-interval. In our example, the end-pc of the method invocation `add(...)` is 155.

3.2.2.2 Finding the Start-pc

The **start-pc** of the method invocation pc-interval is the pc at which the bytecode representing the source code line containing the invocation begins. The start-pc is found by using the line number table of the method. We compare the **end-pc** with the entries of the line number table and take the last entry that has a smaller pc than the **end-pc**. The line number table of the example bytecode is shown in Bytecode 3.3. The **start-pc** in our example is 144.

We define the ***outermost pc-interval*** as the interval starting with the **start-pc** and ending with the **end-pc**. We know that the target variable is located in this interval. In our example, this is <144,155>.

3.2.2.3 Extracting Candidate Target Variable Pc-intervals

The outermost pc-interval <144,155> could contain other invocations, which could have target variables of their own. This is visible in our example, where the `getDesktop()` invocation (pc 146) is embedded in our outermost pc-interval. For this reason, we explored all possible bytecode combinations of instructions that can represent target variables and developed a system for identifying such embedded call sites. More about this system can be found in the Appendix (B.1). This system allows us to split the outermost pc-interval into candidate target variable pc-intervals. In our example, the outermost pc-interval is divided into {<144,144>,<145,149>,<152,152>}. The candidate pc-intervals represent:

1. <144,144>: variable `activePanel`
2. <145,149>: first parameter `(Component) getDesktop()`
3. <152,152>: second parameter `BorderLayout.CENTER`

3.2.2.4 Finding the Actual Target Variable

The set of candidate pc-intervals ends with the arguments of the method. Javassist allows us to obtain the number of parameters of a method invocation by analyzing the method signature. Thus, if the method expects N arguments, we can ignore the last N candidate pc-intervals. The one before them is the actual target variable pc-interval. In our example, since the method takes two arguments, the actual target variable pc-interval is <144,144>.

If the target variable is actually the return value from another method invocation, it is not apparent where exactly the target variable is situated in bytecode. More about the challenges we encountered and why we cannot directly assume that the first target variable candidate is the one we are actually looking for can be found in Subsection 3.3.1.

In case of the simple stand-alone *invokestatic* instruction we do have a target variable but NullSpy ignores these instructions because static method invocations are called on classes which can never be null. But if the static method call is a parameter of another method call, NullSpy still treats it as a target variable candidate. As for the other *invoke* instructions, they are treated normally as explained.

With the target variable pc-interval we extract the information needed to create a unique target variable identifier from the bytecode by using Javassist API. We get the index of the target variable by using Javassist (in our example the index is 3), and with this index and pc information of the target variable, we can obtain the needed information from the local variable table shown in Bytecode 3.4.

```
Local variable table:
    ...
    [pc: 100, pc: 340] local: activePanel index: 3 type: javax.swing.
       JPanel
    ...
```

Bytecode 3.4: Part of the local variable table of the example bytecode shown in Bytecode 3.2.

The following shows a part of the extracted information needed to create the unique identifier for a target variable:

- source code line number of the target variable
- name of the class and signature of the method containing the method invocation
- full variable name
- statically declared variable type

The exact information we need can be found in the Appendix B.2.

3.2.3 Assignments Collection

To find the assignments in the bytecode we search for the instructions matching regexes *"astore.*"* and *"put.*"*. The former refers to local variable assignments and the latter to field assignments.

In the next two subsections we will look at how the information about the assigned variables is extracted. Due to different variable types and the limitation of Javassist, the ways of gathering information about the local variables and fields are performed differently.

3.2.3.1 Local Variable

Unfortunately, Javassist does not provide any support for gaining information about local variables, thus we needed to do it manually.

Every time we encounter the opcode *"astore.*"* (which represents an assignment to a local variable) we get the index of the variable and the pc of the assignment instruction. A local variable has a scope. As we have already seen in Section 2.1, multiple entries

```
1 Local variable table:
2  [pc: 0, pc: 95] local: this index: 0 type: org.jhotdraw.samples.
      javadraw.JavaDrawViewer
3  [pc: 0, pc: 95] local: filename index: 1 type: java.lang.String
4  [pc:  13, pc:  40] local:  url index:  2 type:  java.net.URL
5  [pc: 18, pc: 40] local: stream index: 3 type: java.io.InputStream
6  [pc: 28, pc: 40] local: reader index: 4 type: org.jhotdraw.util.
      StorableInput
7  [pc:  44, pc:  94] local:  newTool index:  2 type:
      org.jhotdraw.framework.Tool
```

Bytecode 3.5: Local variable table entries with same slot 2 (lines 4 and 7).

```
1 Local variable table:
2     ...
3     37  aload_0 [this]
4     38  invokevirtual org.jhotdraw.samples.javadraw.JavaDrawViewer.
         getCodeBase() : java.net.URL [100]
5     41  aload_1 [filename]
6     42  invokespecial java.net.URL(java.net.URL, java.lang.String)
         [104]
7     45 astore_2 [url]
8     ...
```

Bytecode 3.6: Local variable assignment at pc 45.

can have the same local variable slot (one variable takes the index of the other whose lifespan has expired).

We solve the problem of reused slots by looking at the lifespan in the variable table. Let us assume that we have a local variable assignment where its *"astore_2"* opcode is located at pc 45 (Bytecode 3.6). In this example we can extract slot 2 from the opcode *astore_2*. After extracting the slot (index: 2) we iterate through the local variable table and find the first entry that contains that slot (line 4 in Bytecode 3.5). If the pc of the assignment is not included in the local variable lifespan of the entry (pc 45 $\notin$ [pc: 13, pc: 40], line 4), the next entry with the same slot (pc 45 $\in$ [pc: 44, pc: 94], line 7) will be examined. Checking the entries with the same slot goes on until both the slot and pc criteria fit. Once those criteria are met we can be positive about having the right local variable table entry. We now extract from local variable table the information we need to create a unique identifier for a variable. The unique identifiers are used for the comparisons explained in Subsection 3.2.1. A part of the extracted information is listed below:

```
1  Line numbers:
2        [pc: 0, line: 58]
3        [pc: 13, line: 59]
4        [pc: 18, line: 60]
5        [pc: 28, line: 61]
6        [pc: 40, line: 62]
7        [pc: 43, line: 63]
8        [pc:  44, line:  64]
9        [pc: 52, line: 65]
10       [pc: 74, line: 66]
11       [pc: 94, line: 68]
```

Bytecode 3.7: Line number table.

- source code line number of the assignment
- name of the class and method signature containing the local variable
- local variable name
- statically declared local variable type

The exact list of the information we extract can be found in the Appendix B.3.

Using the pc of the assignment (45) and line number table (shown in Bytecode 3.7) we obtain the source code line number of the assignment. As we can see in the line number table, the local variable assignment starts at source code line 64.

3.2.3.2 Instance and Class/Static Variable (Field)

Although Javassist does not support the access to information about local variables it provides a way to access information about fields. Javassist allows us to modify an expression in a method body by using the class `Javassist.expt.ExprEditor` . It scans the bytecode for instructions like *"putfield"* for instance fields or *"putstatic"* for static fields, and allows us to directly get the information we need to create a unique identifier for the field. Next we list a part of the information we are interested in:

- source code line number of the assignment
- name of the class and method signature containing the field assignment
- full field name
- statically declared field type

The full list of the information we extract can be found in the Appendix B.4.

During the data collection of the fields we encountered some difficulties which we will discuss in Subsection 3.3.2.

As already mentioned earlier, every time it encounters an assignment either to a local variable or a field, NullSpy inserts bytecode that checks at run time whether the variable is assigned null. If the variable is assigned null, NullSpy stores all the data about the variable.

3.2.4 Bytecode Adaptation

Each time we encounter a variable assignment we first extract the data about the variable and then we add our own bytecode right after the assignment bytecode. The inserted bytecode checks at run time whether the value being assigned to the variable is null. If this is the case, we store the previously explained information about this assignment. The explanation of the way we do it can be found in the Appendix B.5.

Once we have gone through the bytecode of all the class files, the modified class files are stored in the destination directory as mentioned in Section 3.1. After the instrumentation we add our supplementary supporter classes to the project. The most important ones are `VariableTester` [1] which tests whether a variable is null and `NullDisplayer` [1] which matches data and prints the location of a null assignment when a null pointer exception is thrown.

3.3 Challenges

In this section we present some of the difficulties we encountered during the implementation of NullSpy.

3.3.1 Obtaining Target Variable Data Difficulties

In Subsection 3.2.2 we have explained how to extract the information about target variable when the whole method invocation is contained within one line of code. However, we have encountered a persistent problem, namely getting the pc-interval of the target variable when the method invocation is split throughout multiple lines of code, thus when method invocation pc-interval covers multiple lines in source code.

[1] In package *ch.scg.nullSpy.runtimeSupporter*

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

Code 3.8: Method invocation split in two lines example.

```
1 Line number table:
2     [pc:  0, line:  31]
3     [pc: 3, line: 32]
4     [pc:  11, line:  31]
5     ...
```

Bytecode 3.9: Line number table of Code 3.8.

In order to understand the difficulties in situations where method invocations are split into multiple lines, we look at three different examples. In the first example shown in Code 3.8, we are interested in the method invocation `registerAndLoadImage(...)` . The invocation starts at line 31 and ends at line 32. The second line only contains the arguments of the method call. These situations cannot be resolved by just looking at the bytecode, therefore the line number table (Bytecode 3.9) has to be consulted.

Please note that there is no target variable (in the terms we defined it) in Code 3.8 because the target variable would be a method invocation itself. NullSpy does not support target variables which are return objects of method invocations. Enabling the coverage of such situations is a part of the future work. Other target variables NullSpy does not support are elements of collections since a collection can hold other collections, *etc.*

The line number *31* is listed twice in the line number table (Bytecode 3.9). This indicates that the method invocation is split into multiple lines in source code.

The line number table entries in Bytecode 3.9 indicate the followings:

- line 2: execution of the static method invocation `Iconkit.instance()`
- line 3: starting point where the parameters are loaded onto the operand stack
- line 4: execution of the method `registerAndLoadImage(...)`

To obtain the method invocation pc-interval we had to perform further analyses. By using the algorithm described in Subsection 3.2.2 we would get <0,0> as the method invocation pc-interval, which is incorrect. The correct method invocation pc-interval in this example is <0,11>.

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

Code 3.10: Alternating line number example.

```
1 Line number table:
2     ...
3     [pc: 47, line: 128]
4     [pc: 51, line: 127]
5     [pc: 54, line: 128]
6     [pc: 57, line: 127]
7     ...
```

Bytecode 3.11: Line number table/interval to Code 3.10.

A more complicated situation such as the one shown in the example Code 3.10 can arise. In this example the method invocation is situated in the second line unlike the invocation in the previous example. This is interesting because line numbers 127 and 128 are stored in an alternating way in the line number table shown in Bytecode 3.11. The line number table indicates the followings:

- line 3: the method invocation `getUndoActivity()`
- line 4: the cast to `ChangeConnectionsHandle.UndoActivity`
- line 5: the second method invocation `getOldConnector()`
- line 6: the assignment to the variable `oldConnector`

First, the method invocation `getUndoActivity()` at line 128 is executed and then the execution jumps back to line 127 to perform the cast to `ChangeConnectionsHandle.UndoActivity` . After the cast, again execution jumps to line 128 where the method `getOldConnector()` is invoked.

With the default analyses we would obtain <47,48> as the invocation pc-interval, whereas the correct one is <47,54>. Another interesting thing is that the start-pc of the invocation is mapped to line 128. So the invocation does not start with the smaller line number, *i.e.,* line 127 what normally would be expected.

The third situation we encountered is presented in the example (Code 3.12). In the line number table, there are several pairs of entries corresponding to the same source code line number (line number table shown in Bytecode 3.13), *i.e.,* lines 3 and 12 both

```
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     );
```

Code 3.12: Nested interval example.

```
1  Line number table:
2      ...
3      [pc: 8, line: 149]
4      [pc: 13, line: 150]
5      [pc: 14, line: 151]
6      [pc: 18, line: 152]
7      [pc: 22, line: 153]
8      [pc: 23, line: 154]
9      [pc: 27, line: 155]
10     [pc: 28, line: 151]
11     [pc: 31, line: 150]
12     [pc: 34, line: 149]
13     ...
```

Code 3.13: Line number table/interval to Code 3.12.

correspond to the source code line 149, lines 4 and 11 both correspond to the source code line number 150 and lines 5 and 10 both correspond to the source code line 151. The first line of these pairs indicates the start-pc of an expression and the latter represents the end-pc of the expression. Again, we examine the line number table:

1. line 3: start-pc of the for-loop
2. line 4: start-pc of the method invocation `addItem(...)`
3. line 5: `new ChangeAttributeCommand(...)` start-pc which is actually argument loading onto the stack
4. line 6-8: argument loading onto the operand stack
5. line 9-10: object creation `new ChangeAttributeCommand(...)`
6. line 11: call to `addItem(...)`
7. line 12: for-loop incrementation

By using the algorithm of extracting the pc-interval of a target variable (described in Subsection 3.2.2) we would again get the wrong outermost pc-interval of the method invocation. To handle this mistake we implemented a supporting class called `MultipleLineManager` [2]. Its responsibility is to extract the outermost pc-interval of invocations in split line situations. For complexity reasons we only explain the key idea behind this system. For each invocation we look for the end-pc. With that we go back and forth through the line number table of the method including the invocation and check whether there is another entry with the same source code line number as the end-pc. In examples like the one shown in Code 3.10 we also have to pay attention that the end- and start-pc of the outermost pc-interval do not have to match the same source code line number.

Another difficulty with the invocations split into multiple lines is that even with the system of getting the outermost pc-interval we sometimes still get a smaller one than the one representing the invocation. This happens because we cannot get more information out of the line number table. For this reason we have to take the number of arguments the invocation expects for help complete the outermost pc-interval.

3.3.2 Obtaining Assignment Data Difficulties - Fields

We mentioned that NullSpy collects data about fields with help of the class `Javassist.expt.ExprEditor` . This class uses the method

[2] In package *ch.unibe.scg.nullSpy.instrumentator.controller.methodInvocation*.

`loopBody(...)` to iterate through bytecode of a method and scans for specific opcodes, in our place we are only interested in the opcodes which matches the regex *"put.*"*. One parameter passed to the method `loopBody(...)` is the `mehodInfo` object which contains the bytecode of the method we want to instrument.

Since the `loopBody(...)` is invoked within a `while-loop` , it iterates through the given `methodInfo` until it reaches the end of the bytecode sequence. When it goes through the bytecode sequence we instrument the bytecode. But because the changes made on the bytecode is updated after the `while-loop` , the method `loopBody(...)` still finds the specific opcodes at locations within the origin bytecode sequence instead of the modified one.

Due to this situation we had difficulties finding the right **start-pc** of field assignments. The class `FieldAnalyzer` [3] serves as the explanation of how we still managed to find the **start-pc** of a field assignment.

3.3.3 Bytecode Adaptation Difficulties

The reason we decided to use Javassist for NullSpy was that it allows us to instrument code using the source-code-level API.

One big problem we encountered was inserting code when a variable assignment is the last line of code before a closing curly bracket. We tried to insert the code by specifying the exact line number that we want to instrument, *i.e.,* the line after the assignment. Unfortunately Javassist first checks whether the specified line contains some code (standalone symbols and Java keywords excluded). If there is no code at the specified line, Javassist computes the next line containing a code and inserts our code right before that line.

Let us look at an example where the local variable `var` is assigned a value at the end of the *if-block* at line 3 in Code 3.14. We want to insert the assignment checking code for assignment check just after, at line 4. Since line 4 does not contain any source code, except the closing curly bracket, Javassist adds the code right before the next line which contains the code, which, in this case, is the beginning of the `else-body` , line 5 in Code 3.15). So our added code which serves to check the value of the local variable `var` at run time is added at the wrong place.

Due to this problem we needed to work on the bytecode-level. How we build the bytecode sequence insert is explained in Appendix B.5.

[3] In package *ch.unibe.scg.nullSpy.instrumentator.controller*.

```
1 Object var;
2 if (...) {
3     var = ...;
4     // We want to insert code before this line.
5 } else {
6     var = ...;
7 }
```

Code 3.14: Bytecode adaptation example.

```
1 Object var;
2 if (...) {
3     var = ...;
4 } else {
5     // We accidentally inserted code here.
6     var = ...;
7 }
```

Code 3.15: Wrong Adaptation to Code 3.14.

3.4 Limitations

During the implementation of NullSpy we had to change the concept a few times due to limitations of Javassist or too much computational overhead resulting in serious performance degradation.

NullSpy is not capable of tracking null assignment in three circumstances: if the culprit of a null pointer exception is caused by

- an element of a collection and
- an object that is a return value of a method invocation and
- when the null pointer exception is triggered in a multithreaded environment.

In situations where an element of a collection causes a null pointer exception, we did not develop a way to refer to that element. We have a rough idea of how to store this information, but it would involve effort that is beyond the scope of this thesis. Hence, we cannot identify collection elements.

To be able to track null assignment of target variables we have to statically analyze bytecode and obtain information about that variable. We are not able to statically extract information about an object that is the return value of a method call. For this issue we

also have a rough idea of how to deal with it. But again, the effort required would be beyond the scope of this thesis.

Another situation NullSpy does not support yet is when a null pointer exception is triggered in another thread than the one including the `main()` method, the null pointer exception cannot go up the call hierarchy until the `main()` method since each thread has its own stack. Because the null pointer exception does not reach the `main()` method our inserted catch-block of the `main()` method cannot be reached. Thus the null pointer exception cannot handled by NullSpy. The usage of multiple threads is another problem that is not handled in the scope of this work.

4 Validation

We have evaluated NullSpy on the JHotDraw project. We compared the time it takes to execute both original and the instrumented version of the project.

4.1 JHotDraw

JHotDraw[1] is an open-source Java GUI framework for technical and structured Graphics. It was used to check whether the logic of the bytecode manipulation behind NullSpy is working as desired. It is big enough to get reliable numbers and it provides many varied and complex examples for testing NullSpy's utility.

4.2 Execution Time Difference

JHotDraw already provides an Ant buildfile *build.xml* that packs the project into an executable jar file. We modified the buildfile so that it also creates a jar file out of the class files instrumented by NullSpy. The steps to create an executable jar of the modified project are followings: load the project, modify the project, store the modified project,

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

create a jar file of the original project and one of the modified one. We then run the tests included in each jar file thirty times, recording the execution time. At the end we calculate the average time for the original and the modified project. The execution times are listed in Table 4.1 and the average times are shown in Table 4.2.

	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 run time of the modified project takes rounded 0.363s longer than the original one. Thus after instrumenting the code, the project results in approximately **5%** performance overhead. We claim this overhead is acceptable for the advantages NullSpy offers.

4.3 NullSpy Demonstration

We want to demonstrate how NullSpy helps the user find the real culprit that leads to a null pointer exception. The relevant example used for the demonstration is shown in Code 4.1.

The `main()` method at line 6 invokes three methods (`makeReader()` , `readNumbers()` , `closeReader`) and prints out the field `numbers` . In the `makeReader()` method it tries to read a file named “file.txt”. If the example does not find that file, it will execute the catch-block and assign null to the field `reader` at line 18.

The call to `readNumbers()` at line 8 is not important for our demonstration since the null pointer exception triggered at line 27 is handled by the catch statement at line 31. NullSpy presents the developer the exact location of the null assignment only for an unhandled null pointer exception. That is why we skip it in our analysis.

The null pointer exception is triggered when the method `closeReader()` is called. There is an attempt at line 39 to close the reader. But because the field `reader` is null, the method call `close()` causes a null pointer exception. Since only the *IOException* is caught with the catch-block, the null pointer exception goes up the call hierarchy until the `main()` method. As discussed in Subsection 3.2.1, in the catch-block of `main()` we extract information from the null pointer exception stack trace, multiple comparisons are performed and finally the link to the null assignment and the common null pointer exception stack trace are displayed. The resulting stack trace is presented in Stack trace 4.1. Hence, we can see how the origin of the null pointer exception can be indicated by NullSpy even when it is far away from the actual occurrence of the null pointer exception.

```java
public class Ideone {

    public static List<Integer> numbers;
    public static BufferedReader reader;

    public static void main(String[] args) {
        makeReader();
        readNumbers();
        closeReader();
        System.out.println(numbers);
    }

    // Buggy, reader should not be null if there is no file to read
    private static void makeReader() {
        try {
            reader = new BufferedReader(new FileReader(new File("file
                .txt")));
        } catch (FileNotFoundException e) {
            reader = null;
        }
    }

    // Not really important.
    private static List<Integer> readNumbers() {
        try {
            List<Integer> numbers = new ArrayList<Integer>();
            String text = null;
            while ((text = reader.readLine()) != null) {
                numbers.add(Integer.parseInt(text));
            }
            return numbers;
        } catch (Exception e) {
                return new ArrayList<Integer>();
        }
    }

    // NPE triggered here!
    private static void closeReader() {
        try {
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Code 4.1: NullSpy demonstration example.

```
Field this.reader at line 18 is null: (Ideone.java: 18)
Exception in thread "main" java.lang.NullPointerException
        at Ideone.closeReader(Ideone.java:39)
        at Ideone.main(Ideone.java:9)
```

Stack trace 4.1: Stack trace of the demonstration example Code 4.1.

5 Conclusion and Future Work

In this chapter we reflect on the implementation process and the results of the project. We summarize what goals we have achieved so far and propose further work.

5.1 Conclusion

In short, we successfully managed to meet the main goal we have set at the beginning of the project. NullSpy is now capable of tracking a null pointer exception to its root and provide the user with more information about its origin without a significant overhead. The most important steps that lead to the success of NullSpy are listed below:

1. Extracting the information about target variables was crucial, because they are the candidates for causing null pointer exceptions. To achieve this, we developed an algorithm which finds the target variable, statically extracts the needed information about the target variable to create its unique identifier and stores the details.
2. Collecting data about variable assignments: local variables and fields. For this purpose we statically analyze and instrument the bytecode. We use Javassist to scan bytecode for instructions that represent variable assignments (regexes: *"astore.*"* and *"put.*"*). We extract information about those assignments and pass the collected information to the code we insert after each variable assignment.

During run time we check whether the assignment value is null. If it is the case, we store the information about this assignment.

3. NullSpy handles the uncaught null pointer exception. We wrap up the `main()` method with a catch-block. In this catch-block we extract details of where the null pointer exception occurred from the exception stack trace. With this data and the identifier for target variables and variable assignments we perform several comparisons to find the location of the null assignment.

During the implementation we encountered many difficulties. For most of the problems we came across we managed to find a solution. Those unsolved ones remain as future work.

After the implementation of NullSpy we used the project JHotDraw to validate the implementation. First we measured the time how long JHotDraw takes to execute all its tests without modifying the bytecode. After that we instrumented the project and again measured the execution time. The result is a performance overhead of approximately 5%. So NullSpy could help the developer find the null assignment without a big impact on performance. While 5% may be too much for some time-critical applications, we believe that for most common Java applications such as JHotDraw, this overhead is justifiable by the improved ability to debug null pointer exceptions.

5.2 Future Work

5.2.1 Support Unsupported Target Variables and Variable Assignments

As mentioned in Section 3.4, if the culprit that caused the null pointer exception is an element of a collection or the return object of a method invocation, NullSpy cannot track the null assignment of the culprit variable. This can be overcome by additional instrumentation of the bytecode to keep track of individual elements of collections and by inter-procedural analysis of the bytecode.

5.2.2 Track Null Pointer Exception Root for all Projects

For now NullSpy works only on projects that have a clear starting point in the `main()` method. NullSpy can be improved to also be applicable to projects that do not have a `main()` method, *e.g.,* Java-Applets or web applications. For this, a suitable or even customizable way on where to place the try-catch has to be implemented.

5.2.3 Plug-in for Eclipse

Another possible future work is to transform NullSpy into an Eclipse plug-in project, which would allow a simpler way to start tracking null assignments.

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] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*, pages 15–26, New York, NY, USA, 2005. ACM.

[4] Adrian Lienhard. *Dynamic Object Flow Analysis*. PhD thesis, University of Bern, December 2008.

[5] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical object-oriented back-in-time debugging. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *LNCS*, pages 592–615. Springer, 2008. ECOOP distinguished paper award.

[6] Haidar Osman, Manuel Leuenberger, Mircea Lungu, and Oscar Nierstrasz. Tracking null checks in open-source Java systems. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, March 2016.

[7] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, October 2005.

A

Anleitung zu wissenschaftlichen Arbeiten

NullSpy is a program which helps Java developers find the root of null pointer exceptions by providing an additional link next to the common stack trace. The key idea behind NullSpy is to help developers save time when fixing bugs that manifest themselves as null pointer exceptions. This approach tries to provide the service mentioned while keeping the overhead at a minimum. To demonstrate how NullSpy works, this chapter serves as a small tutorial that takes the example shown in Code 4.1 as the testing code to show the features of NullSpy.

A.1 Installation

1. Take the example shown in Code 4.1 and import it into Eclipse.
2. Checkout NullSpy from `https://github.com/litran8/nullpointer-javassist` and import it into eclipse
3. Look for the class `MainProjectModifier` [1] and run it with two parameters:
   (a) path to the bin folder of the example source code
   (b) path to where the modified project should be stored

[1] In package *ch.unibe.scg.nullSpy.run*.

4. Look for the class `ExecutableJarCreator` [1] and run it with three parameters:
   (a) path to the folder containing the modified project
   (b) path to where the executable jar file should be stored
   (c) the name of the class containing the `main()` method
5. Look for the class `ModifiedProjectLauncher` [1] and run it with two parameters:
   (a) path to the executable jar file
   (b) the name of the class containing the `main()` method

If these steps are followed as explained, the result is similar to the stack trace shown in Stack trace 4.1.

B

Additional Explanations

B.1 Bytecode Combinations of Target Variables

In the algorithm that identifies the target variable pc-interval we have to split the outermost invocation pc-interval into candidate target variable pc-intervals. Here we discuss how the extraction of all possible combinations of target variables that are embedded in the outermost pc-interval are extracted.

We developed a system that checks whether the combinations of specific opcodes are valid. Bear in mind that *aload_0* represents *this* if we are analyzing a non-static method. If we are analyzing a static method the opcode *aload_0* refers to the first argument of the method, if there are any, or to the first declared local variable of the method. The combinations for identifying the candidate target variable pc-intervals are listed below:

- non-static field
  - **(aload.* | getstatic).getfield**
    If the instruction before *getfield* is *"aload.*"*, the combination refers to an access to a public or a private field of *this* object, or access to public field of argument/local variable.
    If the instruction before *getfield* is *getstatic*, the combination refers to an access to a field of a static field.

  - **(aload.* | getstatic).getfield.getfield**
    Access to a public field declared in another class then the currently analyzed one.
    We support only up to two consecutive *getfield* instructions; we have not encountered any case with more than two instructions in row.

- static field
  - **getstatic**
    Access to a static field of any class
- local variable
  - **aload.***
    Access to a local variable of the currently analyzed method

To illustrate how this list is used we go through the synthetic bytecode example shown in Bytecode B.1. This bytecode represents the source code shown in Code B.2.

The outermost pc-interval of the method invocation `move(...)` is <26,33>. When we iterate through the outermost invocation pc-interval we encounter the opcode *"aload.*"* at line 1. We check what comes after this opcode. Again, there is the opcode *"aload.*"* at line 2. With that we look at the list of the possible target variable bytecode combinations and search for *"aload.*"* followed by *"aload.*"*. This expression does not fit into our template which indicates that those two opcodes do not have any relationship with each other. So we can be sure that they are two separate target variable candidates. After that we check the opcode at line 3 and see if the opcodes from lines 2 and 3 fit into our template. For the opcode at line 4 it is the same situation.

But when we check the opcode *"getfield"* at line 5 we see that the opcodes *"aload.*"* followed by *"getfield"* fits into our template. With that we know that the combination of the opcodes at line 4 and line 5 represents one target variable candidate.

After this process, the outermost pc-interval is split into {<26,26>,<27,27>,<28,28>,<29,30>}.

We continue this process until the following opcode does not fit into the allowed pattern of the template. As long as the checked opcode fits into our template, we consider them as one possible target variable.

For other non-primitive types (*e.g.,* Strings) and inlined variable creation we do a similar process. If there is an embedded method invocation, we take the outermost pc-interval of that as a target variable candidate of the external method invocation. The end-pc of the embedded method invocation always comes before the external one in the bytecode sequence. So we will have already figured out the outermost pc-interval of the

```
1 26  aload_1 [figure]
2 27  aload_2 [x]
3 28  aload_3 [y]
4 29  aload_0 [this]
5 30  getfield AppendixMethodReceiverBytecodeCombi.forwardDirection :
      java.lang.Integer [17]
6 33  invokevirtual Figure.move(java.lang.Integer, java.lang.Integer,
      java.lang.Integer) : void [27]
```

Bytecode B.1: Synthetic bytecode example to illustrate the system to split the outermost invocation pc-interval into candidate target variable pc-intervals.

```
figure.move(x, y, forwardDirection);
```

Code B.2: The source code of Bytecode B.1.

embedded method invocation before we examine the external one.

B.2 Target Variables

We explained that we extract target variable information to create its unique identifier for the several comparisons which have to be done to find out the location of the null assignment of the target variable. In Subsection 3.2.2 where we describe the algorithm for the extraction of target variable pc-interval, we gave a hint of what kind of information we need to create this unique identifier. The complete list of what data we store about the target variable is listed next:

- target variable id (simple counter which increments on every occurrence)
- source code line number of the target variable
- name of the class containing the method invocation
- name of the method which contains the invocation
- parameter types and return type of the method which contains the invocation
- is the variable a local variable or a field
- is the variable static or not
- variable name

- statically declared variable type
- name of the class declaring the variable, if the variable is a field, else empty string
- attribute index, if the variable is a local variable

B.3 Local Variable Assignments

To find the location of the null assignment NullSpy has to perform several comparisons of the identifiers representing local variables or fields with the identifiers of target variables. To create those identifiers of local variables we have to collect information about them. We listed part of the information needed to create a identifier of a local variable in Subsubsection 3.2.3.1. The full list contains following information:

- source code line number of the local variable
- name of the class containing the local variable
- name of the method which contains the local variable
- parameter types and return type of the method which contains the invocation
- is the variable a local variable or a field
- local variable name
- statically declared local variable type
- attribute index
- local variable slot
- start-pc: the pc where the local variable assignment actually starts
- store-pc: the pc of the *"astore.*"* opcode
- after-pc: the following pc after store-pc, before which we insert our code

B.4 Field Assignments

Bytecode has two instructions that indicate field assignments, namely *putfield* and *putstatic*. However, we distinguish more than two *"different"* kinds of field assignments.

By *"different"* we mean that there are different kinds of instruction sets that represent a field assignment. The following list contains all *"different"* kinds of field assignments. The placeholder |...|indicates a set of instructions which represents the value that is assigned to a field, *i.e.,* the object which is loaded onto the operand stack for the assignment.

1. **aload_0, |...|, putfield**
   represents an assignment to a field of the currently analyzed class,
   e.g., this.field = |...|;
2. **|...|, putstatic**
   represents an assignment to a static field of any class
   e.g., Class.field = |...|;
3. **aload.*, |...|, putfield**
   represents an assignment to a field declared in another class than the analyzed one
   e.g., memberObject.memberField = |...|;
   We use the term ***member field*** to describe a field declared in another class than the class under analysis.
   Thus we call the object used to access the member field **member object**.
4. **aload.*, (getfield)+, |...|, putfield**
   represents an assignment to a member field with a public modifier
   e.g., this.field.memberField... = |...|;
   e.g., memberObject.memberField... = |...|;
5. **getstatic, (getfield)*, |...|, putfield**
   represents an assignment to a member field with a public modifier and a static member object
   e.g., Class.memberObject...memberField = |...|;

In Subsubsection 3.2.3.2 we listed a part of the data which we actually extracted from field assignment bytecode to create the unique identifier for the field.

If the assignment is to a field of the currently analyzed class, we extract following information about it:

- source code line number of the local variable
- name of the class containing the field assignment
- name of the method which contains the field assignment
- parameter types and return type of the method which contains the field assignment
- is the variable a local variable or a field

- class name in which the field is declared
- is the field static or not
- field name
- statically declared field type
- start-pc: the pc where the field assignment actually starts
- store-pc: the pc of the *"put.*"* opcode
- after-pc: the next pc after store-pc; before that pc we insert our code

If we detect an assignment to a member field we additionally have to store the information about the member object:

- is the member object a local variable or a field
- class name in which the member object is declared
- is the member field static or not
- member object name
- statically declared member object type

B.5 Bytecode Instrumentation

The aim of NullSpy is to find the location of the null assignment of the variable which caused the null pointer exception. We achieve this goal by instrumenting bytecode with the help of the bytecode-level API of Javassist.

After each non-primitive variable assignment we insert bytecode that represents a method invocation. The invoked method is part of the class named `VariableTester` [1]. Depending on the variable (local variable or field) being analyzed at the moment, a different method is invoked.

If we take the source code snippet Code B.3 which is the source code at line 16 from the demonstration example Code 4.1 and decompile the instrumented version of this snippet, it looks the way as shown in Code B.4.

We mentioned that Javassist provides the way to modify bytecode either with the source- or bytecode-level API. Due to a limitation of Javassist we modify the class files

[1] In package *ch.scg.nullSpy.runtimeSupporter*. This package is added to the modified project when the bytecode modification is done.

```
16 reader = new BufferedReader(new FileReader(new File("file.txt")));
```

Code B.3: Source code at line 16 of the NullSpy demonstration example shown in Code 4.1.

```
16 reader = new BufferedReader(new FileReader(new File("file.txt")));
   VariableTester.testDirectField("Ideone", "makeReader", "()V", "
   field", "reader", "Ljava/io/BufferedReader;", "Ideone", 1, reader,
    25, 0, 23, 26);
```

Code B.4: Decompiled instrumented version of Code B.3.

at bytecode-level by constructing a bytecode sequence and including it into the file. How we build this sequence is shown in Code B.5.

The code snippet Code B.5 creates the method invocation which we insert at bytecode and which tests a field assignment for null value. The header of that method is presented in Code B.6.

This method takes seven java.lang.String objects, one integer, one Object and four integers as arguments. The code snippet Code B.5 reproduces these arguments.

The instruction *"addLdc"* from line 16 to 22 is used to add strings to the bytecode. For adding a boolean we use the integers *"1"* and *"0"* to represent the boolean value *true* and *false* respectively, as used in Java bytecode. Hence, line 25 to 29 adds an integer to the bytecode sequence.

The object, in our case a field, is added to the bytecode sequence by using the method `addAload(...)` at line 31 and the method `addGetfield(...)` at lines 32 and 33. These lines add the instruction set which loads a field onto the operand stack: *"aload_0", "getfield"*. NullSpy takes the field itself as an argument to evaluate its value in the check-method.

The line number and pcs are added to the bytecode sequence from line 35 to 38.

Finally all classes of the arguments the check-method takes are prepared from line 40 to 43 and the instruction *"invokestatic"* at line 45 is added to the bytecode. If we add a method invocation to the bytecode we have to specify what kind of arguments the invocation takes. For this reason we prepared the classes.

With the classes *CodeAttribute* and *CodeIterator* from the Javassist API we can enter our bytecode sequence into existing ones. CodeAttribute represents the byte-

```
1  private byte[] getInsertCodeByteArray(Variable var) {
2      Bytecode bytecode = new Bytecode(cp);
3
4      Field field = (Field) var;
5      CtBehavior behavior = field.getBehavior();
6
7      String varName = field.getVarName();
8      String varType = field.getVarType();
9      String varID = field.getVarID();
10     String fieldDeclClassName = field.getFieldDeclaringClassName();
11     int varLineNr = field.getVarLineNr();
12     int varStartPc = field.getStartPc();
13     int varStorePc = field.getStorePc();
14     int varAfterPc = field.getAfterPc();
15
16     bytecode.addLdc(behavior.getDeclaringClass().getName());
17     bytecode.addLdc(behavior.getName());
18     bytecode.addLdc(behavior.getSignature());
19     bytecode.addLdc(varID);
20     bytecode.addLdc(varName);
21     bytecode.addLdc(varType);
22     bytecode.addLdc(fieldDeclClassName);
23
24     // int 1 -> static, 0 -> nonStatic
25     if (field.isStatic()) {
26         addIntegerToBytecode(bytecode, 1);
27     } else {
28         addIntegerToBytecode(bytecode, 0);
29     }
30
31     bytecode.addAload(0);
32     bytecode.addGetfield(
33         fieldDeclaringClassName varName, varType);
34
35     addIntegerToBytecode(bytecode, varLineNr);
36     addIntegerToBytecode(bytecode, varStartPc);
37     addIntegerToBytecode(bytecode, varStorePc);
38     addIntegerToBytecode(bytecode, varAfterPc);
39
40     CtClass variableTester = ClassPool.getDefault().get(
41             "ch.unibe.scg.nullSpy.runtimeSupporter.VariableTester");
42     CtClass str = ClassPool.getDefault().get("java.lang.String");
43     CtClass object = ClassPool.getDefault().get("java.lang.Object");
44
45     bytecode.addInvokestatic(variableTester,
46         "testDirectField", CtClass.voidType, new CtClass[] {
47             str, str, str, str, str, str, str,
48             CtClass.intType, object, CtClass.intType,
49             CtClass.intType, CtClass.intType,
50             CtClass.intType });
51
52     byte[] bytecodeArray = bytecode.get();
53     return byteCodeArray;
54 }
```

Code B.5: This bytecode is inserted after each assignment to a field of the currently analyzed class. It represents the bytecode added after a field assignment.

```
31 public static void testDirectField(
32     String classNameInWhichVarIsAccessed,
33     String behaviorName,
34     String behaviorSignature,
35     String varID, String varName, String varType,
36     String varDeclaringClassName,
37     int isStatic,
38     Object varValue,
39     int varLineNr,
40     int startPc, int storePc, int afterPc
41     ) {...}
```

Code B.6: Check field method header.

code of a behavior and with CodeIterator we can iterate through that bytecode. We obtain a CodeIterator object from the CodeAttribute. So we first move the CodeIterator to the position in the bytecode with `codeIterator.move(pc)` where we want to enter our bytecode and then enter our sequence with a call to `codeIterator.insert(ourByteCode)` .

”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.”