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.

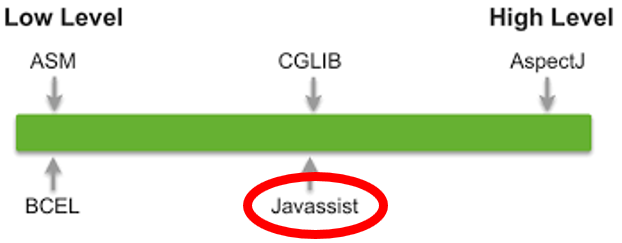
The main goal of the NullSpy application takes a step to that ideal vision. Anytime developers are facing a NullPointerException they don’t have to spent time on debugging finding where and why a reference was set to null. With NullSpy the exact location of the null assignment is shown next to the ordinary 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.

**Technical Background**

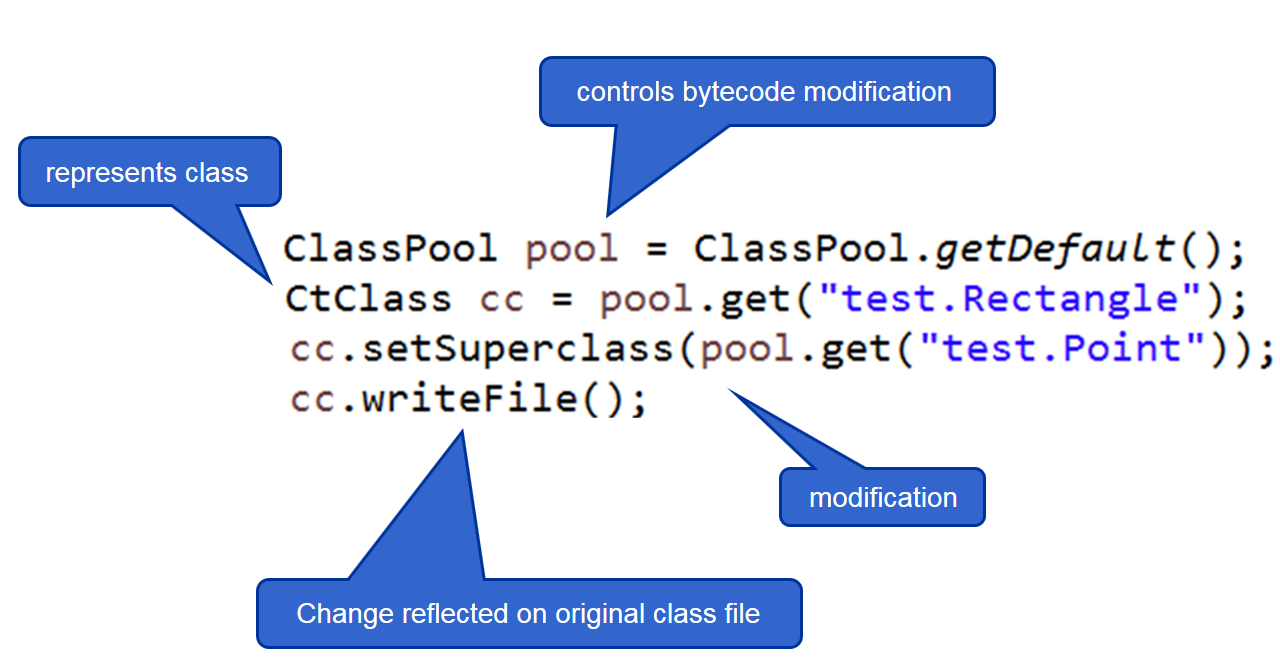
This chapter provides a short overview of works/technologies used in this project.

**Javassist**Javassist or Java Programming Assistant, 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.

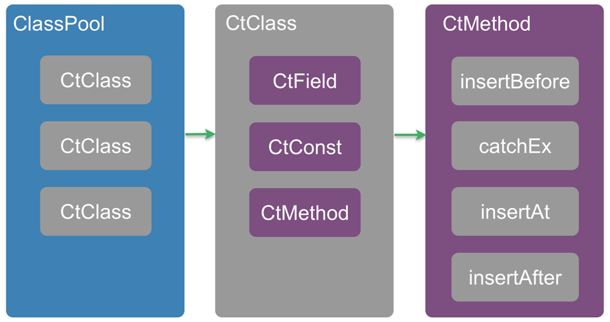


Unlike many other libraries Javassist offers two levels of API: source level and bytecode level. Using the source-level API, the user can edit a class file without any familiarity with the specifications of the Java bytecode. Only knowing the Java language is enough because the API is designed only with 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.

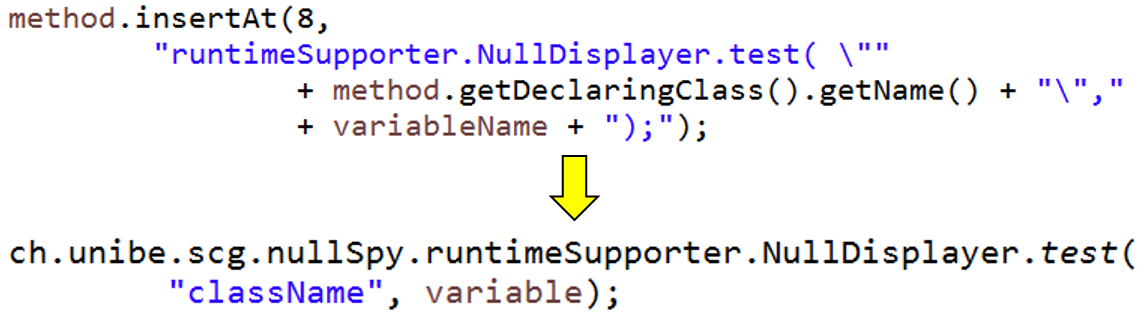
At this point, let us look at a small example to give you an idea how the bytecode manipulation works.



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 ClassPoola 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.



This image 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 method is manly meant. It is possible to insert source code at the beginning of the method body, at the end or at a specific line. Next to these options a “catchBlock” can even be added.



**JHotDraw**To check whether the logic of the bytecode manipulation in this project NullSpy is working as desired, we had to perform them on a large working project. Thanks to Nevena Milojkovic and her experience with the combination Javassist and JHotDraw we as well decided to test NullSpy on the project JHotDraw.

It is an open-source Java GUI framework for technical and structured Graphics. Its original authors have been Erich Gamma and Thomas Eggenschwiler.

JAD  
JAva Decompiler is a decompiler for the Java programming language. 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.

JAD is used in NullSpy since after running NullSpy on JHotDraw only modificated bytecodes are available. But to simplify the check if the modification by Javassist, e.g. inserting source code, has succeeded, a decompiler is needed.

t**NullSpy**

As earlier explained in the introduction, 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 (ref chapter future work).

**High level overview/Rough Scheme**

The general approach of NullSpy is to statically analyze and add additional bytecode to a project. After reading the “Technical Background” section 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.

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.

pic

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. To know the exact location of the null assignment of course we also have to collect information about the variable assignments.

The main idea behind NullSpy actually is to get information about method receivers and information about the variables on one side and comparing those together if a NullPointerException occurs on the other side. If there is a hit on the matching the null assignment location can be obtained easily.

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 the chapter [CHAPTER REF]

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 invocation opcode: invokedynamic, invokeinterface, invokespecial, invokestatic, invokevirtual. The bytecode instruction invokedynamic facilitates the dynamic-typed languages(footnote) through dynamic method invocation. In our case it can be ignored because NullSpy only supports the static-typed language(footnote) Java.

http://www.javaworld.com/article/2860079/learn-java/invokedynamic-101.html

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.

Variable Data Collection

While going through the bytecode attention is paid to some opcodes (footnote). 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] 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 [REFERENCE].

*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.

pic

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. Storing opcodes 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…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 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 x) 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.

Pic

We had to pay attention to be sure to get the right local variable. Every time when we bumped into the opcode “.\*store.\*” we could only get its slot and the pc 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 used by the next instantiated local variable. This way, the local variable table could contain multiple entries with the same local variable slot.

Pic

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 it the line number can be easily obtained.

There was a big problem about inserting additional source code after each assignment. It is caused by the limitation of the class library Javassist (see SUBSEC).

*Instance or class/static variables*

Once we have gone through the bytecode of a Java class, the modified class files are stored in a destination directory. To the modified project we also added an additional supporter which contains the test method mentioned above.

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 the altered project does not include Javassist. This way the user does not have to download Javassist and integrate it to the project.

Only storing data about assigned variables is not enough, therefore we also have to store information about the method receiver. This one is stored as an external file due to performance and minimizing of the overhead.

NullSpy only handles the unhandled NullPointerException, meaning the main method of a project is wrapped with a “catchBlock” where the NPE is treated when it crops up. The idea in this “catchBlock” is to load the data about the assignments and the data 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 happened 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 stacktrace printed after an exception occures. Thanks to that link that points directly to the NPE root, debugging will be easier or this process can even be skipped or avoided.

**Low level overview**

**Challenges**

*Storing data about assigned variables*

To do some information comparisons we first have to obtain some data about where a variable is actually set to null. To be able to get those data we load the class files with Javassist and then create CtClasses out of them, just as in chapter “Related Work” described. With the CtClass object we finally have access to look at and editing the bytecode of a loaded class file, which means that we for example adding code to the loaded Java class. That is exactly what we were doing.

For each class file we obtained and went through its bytecode and looked for opcodes like  
🡪 aastore  
astore  
putfield  
putstatic  
.

* BILD

All these opcodes represent some variable assignments. After bumping into one of those enumerated keywords the test method of the supporter class we added to the project is inserted right after the assignment. The idea behind this insertion is to collect some information about the variable which was just set to a value. We only store them if the value is the culprit null. So what kind of information are stored? The role of those information is to allow the identification of a variable and to get the exact location of its instantiation. That means the name, type, line number, the class name or method name and lot more. It depends on what kind of variable the examined one is, it could be a local variable or a class member, static or non-static one, a variable of a superclass etc.

Challenges/Problems

Javassist already provides a way to look up all fields in a class but not for local variables which means getting information about those local variables took quite a long time. Looking up all fields means Javassist offers a method that iterates through bytecode and looks for keywords for method calls (invokestatic, invokeinterface, invokevirtual), field accesses (getfield, getstatic, putfield, putstatic), object creation (new), e.g. constructor calls (invokespecial) etc. Finding those opcodes Javassist returns

How did we solve this

Slot, stack explanation, word -> byte

As mentioned Javasssist provides the source level to modify an existing code. Due to the easiness to only operate on the higher level we first inserted the testmethod at that level. Everything worked fine until we were faced with some situations, where we want to insert additional code right before a closing bracket. code first source then bytecode level  
source level: syntax, everything in string  
bytecode level: creating right bytecode is harder than thought -> different types of variable, direct, indirect, etc.

**General Problem/Challenge**

At the beginning of a project or even before starting it,

In this section I want to introduce some hurdles I had to overcome during the implementation of th

Get used to how Javassist works, tutorial, forgot things actually could take in, javassist problem not updating codeAttr after insertion after field assignments

Architect of project, first own project, without thinking started to code, naming, complex methods without any documentations, refactored many thimes, problem with concentrating on one task -> abschweifen, try to solve many things in parallel

**Limitations**

**Validation**

**Conclusion and Future work**