



Fachhochschule Köln  
Cologne University of Applied Sciences

# WPF Compiler und Interpreter: Java-Hardener

Projektdokumentation über einen Java-Postprozessor  
zur automatisierten Bytecode-Manipulation  
zur Reduzierung von `NullPointerExceptions`.

Dozent: Prof. Dr. Erich Ehse  
Fachhochschule Köln

ausgearbeitet von  
Christoph Jerolimov, Matrikelnr. 11084742  
Sommersemester 2013

# Inhaltsverzeichnis

<b>1</b>	<b>Abstrakt</b>	<b>1</b>
<b>2</b>	<b>Analyse</b>	<b>2</b>
2.1	Problemstellung . . . . .	2
2.2	Bytecode-Analyse . . . . .	3
2.2.1	Ausgangsbasis . . . . .	3
2.2.2	Bedingungsoperator ?: . . . . .	4
2.2.3	Try-Catch . . . . .	4
<b>3</b>	<b>Umsetzung</b>	<b>6</b>
3.1	Maven . . . . .	6
3.2	ASM . . . . .	6
3.2.1	Visitor Pattern . . . . .	6
3.2.2	Tree / DOM API . . . . .	7
3.3	Umsetzung automatisierte IFNULL-Prüfung . . . . .	7
3.3.1	Iteration 1: Grundsätzliches Vorgehen . . . . .	8
3.3.2	Iteration 2: Verfielfältigung . . . . .	9
3.3.3	Iteration 3: Generalisierung . . . . .	9
3.3.4	StackSize und Labels . . . . .	10
3.4	Umsetzung ClassLoader . . . . .	11
3.5	TODO . . . . .	11
<b>4</b>	<b>Fazit</b>	<b>12</b>
4.1	Projektergebnis . . . . .	12
4.2	Erweiterungsmöglichkeiten . . . . .	12
	<b>Eidesstattliche Erklärung</b>	<b>13</b>

# 1 Abstrakt

`NullPointerException` (NPE) sind ein klassisches Problem der Softwareentwicklung und treten in der Programmiersprache Java auf wenn Methoden- oder Attribut-Zugriffe auf `null`-Object erfolgen<sup>1</sup>.

Die Behandlung solcher ungültiger Aufrufe ist grundsätzlich abhängig von der Programmiersprache und der Laufzeitumgebung. So können entsprechende Zugriffe zum Absturz des Programms führen, wie in Java zum werfen einer entsprechender Ausnahme oder, wie etwa in Objective-C<sup>2</sup>, ignoriert werden.

Diese fehlertolerantere Version von Objective-C soll hier nachgebildet werden und durch eine automatisierte manipulation des Java-Bytecodes erreicht werden. Wie in der Vorlage müssen entsprechende Methoden immer einen Rückgabewert liefern, hier werden, analog zu Objective-C, möglichst neutrale Werte gewählt: `False` für boolsche Ausdrücke, `Null` für Zahlen und `NULL`-Referenzen für Objekte

Die beiden folgenden zwei Anwendungsfälle (vgl. Listing 1.1 und 1.2) verdeutlichen die Einfachheit für den Programmier und würden ohne Bytecode-Manipulation zu `NullPointerException` führen.

```
1 List nullList = null;
2 System.out.println("List size: " + nullList.size());
```

Listing 1.1: Beispiel für einen Null-Zugriff mit erwartetem Integer-Ergebnis

```
1 List nullList = null;
2 if (!nullList.isEmpty()) {
3     // Will run this code also if the nullList is null...
4 }
```

Listing 1.2: Beispiel für einen Null-Zugriff mit erwartetem Boolean-Ergebnis

Im folgendem werden verschiedene technische Möglichkeiten untersucht sowie deren prototypische Umsetzung mithilfe der OpenSource Bibliothek ASM (siehe Kapitel 3) beschrieben. Zur besseren Lesbarkeit wird auf Java-Packagenamen (etwa `java.lang.`) sowie Generics in allen Java und Bytecode (Assembler) Darstellungen verzichtet. Alle angesprochenen Dateien finden sich innerhalb dem diesem Projekt zugehörigem Sourcecode-Archiv.

<sup>1</sup>Dadüber hinaus kann eine NPE auch noch in anderen Fällen geworfen werden. Vgl. <http://www.java-blog-buch.de/0503-nullpointerexception/>

<sup>2</sup>Vgl. <http://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/>

## 2 Analyse

### 2.1 Problemstellung

Wie in der Einführung beschrieben, können Objectaufrufe, z.B. durch Methoden- und Variablenaufrufe (lesend und schreibend), auf NULL durch vorheriges Prüfen gesichert werden. Auch andere Fälle, etwa der Zugriff auf Arrays (`[index]`-Zugriff oder `.length`) kann zu NPE-Ausnahmefehlern führen. Nicht alle diese Anwendungsfälle werden in diesem Prototypem umgesetzt sollen aber wenigstens in dieser Einführung angesprochen werden.

Problematisch sind insbesondere verkettete Aufrufe (vgl. Listing 2.1). So müssen die zwischen Ergebnisse etwa in lokalen Variablen gespeichert werden (vgl. Listing 2.2) oder die Aufrufe wiederholt werden wenn diese in umgebende Bedingungen einzubauen (vgl. Listing 2.3). Letzteres würde jedoch nicht nur die Performance negativ beeinflussen, sondern könnte bei inmutablen Zugriffen auch zu Fehlerhaften Programmläufen führen.

```
1 Deque<Map<String, Integer>> example = null;
2 int size = example.getFirst().get("size");
```

Listing 2.1: Beispiel für verkettete Aufrufe

```
1 Deque<Map<String, Integer>> example = null;
2 Map v1 = example.getFirst();
3 Integer v2 = v1.getSize("size");
4 int size = v2 != null ? v2.intValue() : 0;
```

Listing 2.2: Umwandlung verketteter Aufrufe in lokale Variablen

```
1 Deque<Map<String, Integer>> example = null;
2 int size = 0;
3 if (example != null &&
4     example.getFirst() != null &&
5     example.getFirst().get("size") != null) {
6     size = example.getFirst().get("size");
7 }
```

Listing 2.3: Verkettete Aufrufe umfasst mit NULL-Prüfungen

Autoboxing bezeichnet die mit Java 1.5 eingeführte automatische Umwandlung zwischen primitiver Datentypen sowie deren Wrapper-Typen. Diese implizite Umwandlung wird durch

zusätzliche Methodenaufrufe durch den Compiler eingewebt und ist für den Java-Interpreter nicht von normalen Aufrufen zu unterscheiden.

Für die manipulation des Bytecodes zur Verbesserung der Fehlertoleranz sollte dies ebenfalls keinen Unterschied bieten.

## 2.2 Bytecode-Analyse

Mithilfe des im ASM enthaltenen Textifier Programms können verschiedene Lösungswege deassembliert und analysiert werden. Zum einfacheren Aufruf wurde ein kleines Shell-Script (siehe textifier) erstellt. Mit dessen Hilfe wurden etwa für das in Listing 2.4 angegebene Java-File die in 2.5 angegebene Ausgabe erzeugt.

Der Aufruf erfolgt über den Scriptnamen gefolgt von einer Java-Bytecode-Datei:

```
./textifier target/test-classes/de/.../testcases/Test1.class
```

### 2.2.1 Ausgangsbasis

```
1 public class Test1 {
2     public int getStringLength(Map map, String key) {
3         return ((String) map.get(key)).length();
4     }
5 }
```

Listing 2.4: Beispiel Sourcecode mit Null-Prüfung

```
1 public class de/fhkoeln/gm/cui/javahardener/testcases/Test1 {
2     public getStringLength(Ljava/util/Map;Ljava/lang/String;)I
3         ALOAD 1
4         ALOAD 2
5         INVOKEINTERFACE java/util/Map.get (Ljava/lang/Object;)Ljava/lang/Object;
6         CHECKCAST String
7         INVOKEVIRTUAL String.length ()I
8         IRETURN
9         MAXSTACK = 2
10        MAXLOCALS = 3
11 }
```

Listing 2.5: Auszug ASM Assembler-Ausgabe für Listing 2.4

Im folgenden sollen die Unterschiede aufgezeigt werden, wenn man diese ursprüngliche Version mit gegen NPE gesicherte Versionen vergleicht. Die dafür angelegten Klassen befinden sich im test-Ordner innerhalb des Java-Packages **analysebytecode**.

### 2.2.2 Bedingungsoperator ?:

Durch eine NULL-Prüfung mit einem Bedingungsoperator (etwa `e != null ? e.toString() : null`) fügt der Compiler zwei Labels (Ziele für Sprungmarken) ein und prüft anschließend die aktuell auf dem Stack liegende entry Variable (vgl. Listing 2.6 Zeile 1) auf null (Z. 2). Ergebnis die NULL-Prüfung wahr springt die Ausführung zur angegebenen Sprungmarke (hier L0) und fügt eine NULL-Referenz auf den Stack hinzu. Falls die NULL-Prüfung falsch ergibt wird die Ausführung fortgesetzt und der eigentliche Methodenaufruf durchgeführt (INVOKEVIRTUAL in Zeile 4). Um anschließend den nicht benötigten Alternativen Zweig der Anwendung zu gehen wird dieser mithilfe eines GOTOs (hier zur Sprungmarke L1) übersprungen.

```
1      ALOAD 2 /* entry */
2      IFNULL L0
3      ALOAD 2 /* entry */
4      INVOKEVIRTUAL String.toString ()LString;
5      GOTO L1
6  L0
7      ACONST_NULL
8  L1
```

Listing 2.6: Auszug ASM für Null-Prüfung mit Bedingungsoperator

### 2.2.3 Try-Catch

Eine weitere Möglichkeit wäre die mögliche Ausnahmebehandlung von dem eingebauten try-catch Mechanismus behandeln zu lassen und einen entsprechenden Block um den möglicherweise zu fehlern führenden Aufruf zu erstellen.

Für dieses Vorgehen wird eine zusätzliche lokale Variable benötigt, welche im Fehlerfall mit einem Defaultwert gefüllt wird (vgl. Listing 2.7).

```
1  int l;
2  try {
3      l = entry.length();
4  } catch (NullPointerException e) {
5      l = 0;
6  }
```

Listing 2.7: Beispiel Null-Prüfung mit try-catch

Der darauf entstehende Bytecode speichert das Ergebnis des Originalaufrufs in einer lokalen Variable (Listing 2.8 Zeile 4 und 5). Sollte es während dieses Aufrufs zu einer Fehlerbehandlung kommen wird diese Variable mit einer NULL-Referenz überschrieben (Zeile 10 und 11).

```

1      TRYCATCHBLOCK L0 L1 L2 NullPointerException
2      L0
3      ALOAD 2
4      INVOKEVIRTUAL String.length ()I
5      ISTORE 3
6      L1
7      GOTO L3
8      L2
9      ASTORE 4
10     ICONST_0
11     ISTORE 3
12     L3

```

Listing 2.8: Auszug ASM für Null-Prüfung mit try-catch

Insgesamt fällt auf, dass dieser Code bereits bei diesem einfachen Beispiel deutlich mehr Instruktionen beinhaltet als die zuvor genannte Bedingungsoperator-Variante. Gleichzeitig wird für quasi jeden Methodenaufruf eine zusätzliche lokale Variable benötigt. (Ggf. könnten diese auf eine Variable je Datentyp kombiniert werden.)

Dadrüber hinaus würde diese Variante nicht nur unmittelbare `NullPointerException`s abfangen, sondern auch Fehler, welche innerhalb der Methode ausgeführt werden und ggf. gar nicht vom Java-Compiler manipuliert wurden.

## 3 Umsetzung

### 3.1 Maven

Um die Abhängigkeiten mit Maven runterzuladen kann ein entsprechenden IDE-maven-plugin verwendet werden oder die IDE Konfiguration mit den folgenden Befehlen erzeugt werden:

```
mvn eclipse:clean eclipse:eclipse -DdownloadSources
mvn idea:clean idea:eclipse
```

Zum runterladen der Ressourcen und compilieren des Projektes kann anschließend die IDE verwendet werden oder einer der folgenden Befehle zum bauen bzw. paketieren der Klassen als JAR-Datei:

```
mvn compile
mvn test      # Beinhaltet compile
mvn package   # Beinhaltet test
```

### 3.2 ASM

Zur Manipulation von Java Bytecode bietet sich die leichtgewichtige und speziell dafür entwickelte OpenSource-Bibliothek ASM an. Während der Entwicklung wurden drei ASM-Libraries mithilfe von Maven eingebunden:

- Die Kernbibliothek ASM (asm-4.x.jar) bietet Schnittstellen zum Einlesen und Schreiben von Class-Dateien mithilfe des Visitor-Patterns.
- Optional kann ASM durch eine Library zum DOM-basierten Zugriff auf den Bytecode erweitert werden (asm-tree-4.x.jar).
- Häufig verwendete Methoden, etwa zum Ausgeben von Assembler-Code finden sich in der ebenfalls optionalen Utility-Erweiterung (asm-util-4.x.jar).

#### 3.2.1 Visitor Pattern

Zur Manipulation des Bytecodes verwendet ASM das Visitor Pattern und verschachtelt dabei drei verschiedene Visitor Schnittstellen (jeweils als Abstrakte Klassen):



- **ClassVisitor** für den Header einer Klasse, Annotations, etc. Diese Klasse delegiert den Visitor für Methoden und Klassenvariablen (Fields) an neue Instanzen der folgenden Klassen.
- **MethodVisitor** bietet visitor Methoden für die Methoden deklaration sowie die enthaltene Implementation (Operationsaufrufe für den virtuellen Java-Prozessor).
- **FieldVisitor** bietet ausschließlich die Möglichkeit auf die deklarierte, und ggf. annotierte Klassenvariable zu reagieren.

Zum schreiben von Klassen bietet ASM mit der Klasse **ClassWriter** eine Implementierung des **ClassVisitor** welche sein Ergebnis in einen entsprechenden Ausgabekanal schreibt.

Zur Visualisierung des Assembler-Codes bietet sich die Klasse **TraceClassVisitor** an welche eine menschenlesbare Ausgabe produziert.

### 3.2.2 Tree / DOM API

Alternativ zum Visitor Pattern bietet die ASM-Tree Bibliothek einen darauf aufbauenden wahlfreien (DOM-basierten) Zugriff auf den Klassencode.

Dies hat den Vorteil das deutlich komplexere Analysen möglich sind und der Kontext eines Befehles mit betrachtet werden kann. Jedoch sind solche Analysen deutlich komplexer als diese etwa auf einem Quellcode-DOM wären da viele Informationen beim Reduzieren auf Assembler-Bytecode verloren gehen.

## 3.3 Umsetzung automatisierte IFNULL-Prüfung

Nach einer Testumsetzung und verschiedenen Analysemöglichkeiten findet sich das Ergebnis in den beiden Klassen **CheckNullClassVisitor** sowie **CheckNullMethodVisitor**. Während ersetzter die nötige Schnittstelle für die **ClassReader.accept(ClassVisitor classVisitor, int flags)** Methode implementiert hat diese jedoch keine manipulierende Auswirkung auf den Bytecode. Ihre einzige Funktion ist es für jede zu prüfende Methode (**visitMethod**) eine neue Instanz der Klasse **CheckNullMethodVisitor** zurück zu geben.

Der Methoden-Visitor kümmert sich anschließend um die Prüfung aller **INVOKE\_** Assembler aufrufe. Hierfür muss die Methode folgende Methode überladen werden:

```
visitMethodInsn(int opcode, String owner, String name, String desc)
```

Für nicht behandelte Anwendungsfälle reicht es die Implementierung der Elternklasse aufzurufen. Wenn stattdessen andere **visit\*** Methoden der Elternklasse aufgerufen werden, werden diese Methoden an den im Konstrukt übergebenen Visitor übergeben.

Auf diese Art können verschiedene **MethodVisitor** ineinander geschachtelt (chaining) werden und die jeweiligen Teilaufgaben übernehmen. Eine übergebene **ClassWriter** Instanz kann etwa die veränderten visit-Aufrufe in Bytecode umwandeln. Vgl. hierzu auch die Debug-Möglichkeiten im Kapitel Umsetzung ClassLoader.

### 3.3.1 Iteration 1: Grundsätzliches Vorgehen

Die erste prototypische Umsetzung<sup>1</sup> der Klasse `CheckNullMethodVisitor` behandelte ausschließlich den Methodenaufruf `String.length()`. Alle anderen Aufrufe wurden in dieser Version nicht beachtet. Für die Null-Prüfung wurde der Original Aufruf in eine Bedingung mit Sprungbefehlen gekapselt.

Um den Original Aufruf nicht zu verändern muss die aktuelle Stackreferenz auf das Objekt welches die Methode ausführt mittels `DUP` (vgl. Listing 3.1 Zeile 5) verdoppelt werden. Diese neue Referenz wird bei der `NULL`-Prüfung mit `IFNULL` wieder vom Stack gelöscht. Ergibt die Prüfung das es sich um eine `NULL`-Referenz handelt springt die Laufzeitumgebung zur angegebenen Springmarke (hier `Label fallback`, vgl. Zeile 2, 6 und 10). Wenn die `NULL`-Prüfung ergibt das es zu keiner `NullPointerException` kommen wird wird in der nächsten Instruktion der Original Aufruf durchgeführt und hier die `super` Methode aufgerufen welche die Argumente an den jeweils nachgeschalteten `MethodVisitor` übergibt. Um den im folgenden beschriebenen alternativen Anwendungspfad nicht zu durchlaufen wird dieser mit einem `GOTO` und der Zielsprungmarke übersprungen.

Falls der Aufruf nicht ausgeführt werden soll, da die aktuelle Pointerreferenz `NULL` ist muss dieser Aufruf mithilfe von `POP` vom Stack entfernt werden. (Während die duplizierte Adresse von `IFNULL` aufgebraucht wurde, würde der eigentliche Aufruf einer Methode die Objektreferenz löschen und durch das Ergebnis ersetzen.)

Damit die nächsten Instruktionen mit dem erwarteten Ergebnis auf dem Stack rechnen können muss anschließend nur noch ein Standard-Ergebnis auf den Stack geschrieben werden. Für die aktuelle Methode (`String.length()`) bietet sich hierfür die Instruktion `ICONST_0` an. Dies fügt ein `int 0` dem Stack hinzu.

```
1 public void visitMethodInsn(...) {
2     Label fallback = new Label(); Label behind = new Label();
3
4     super.visitInsn(Opcodes.DUP);
5     super.visitJumpInsn(Opcodes.IFNULL, fallback);
6     super.visitMethodInsn(opcode, owner, name, desc);
7     super.visitJumpInsn(Opcodes.GOTO, behind);
8
9     super.visitLabel(fallback);
10    super.visitInsn(Opcodes.POP);
11    super.visitInsn(Opcodes.ICONST_0);
12    super.visitLabel(behind);
13 }
```

Listing 3.1: Erste Umsetzung einer automatischen Null-Prüfung mit ASM

<sup>1</sup>Vgl. Projektsourcen - Rev 951f48 `CheckNullMethodVisitor.java` Zeile 43-66

### 3.3.2 Iteration 2: Verfielfältigung

Bei der zweiten Iteration<sup>2</sup> wurde versucht dieses Vorgehen auch auf andere Methoden anzuwenden und die jeweiligen Unterschiede zu beleuchten.

Problematisch dabei ist die Reihenfolge des Stacks für den jeweiligen Methodenaufruf. So liegen auf oberster Position die Argumente und erst unter dieser die eigentliche Referenz auf das Objekt wessen Methode aufgerufen werden soll. Um die Objektreferenz einer NULL-Prüfung mit IFNULL unterzuziehen zu können muss jedoch diese jedoch oben auf dem Stack aufliegen.

Für Methoden mit nur einem Argument konnte dies noch einfach über das Hintereinander schalten der beiden Instruktionen DUP2 sowie POP sein. Während der erste Befehl (bei nur einem Argument) die Referenz des Objektes und des Arguments kopiert, wird die des Arguments anschließend wieder entfernt.

Für eine beispielhafte Implementierung für die Methode `Map.get(Object)` muss schließlich nicht nur eine Referenz sondern ebenfalls zwei Referenzen vom Stack gegen eine NULL-Referenz (anstatt eines `int 0`) ersetzt werden (Vgl. Listing 3.2).

```
1 super.visitInsn(OpCodes.DUP2);
2 super.visitInsn(OpCodes.POP);
3 [...]
4 super.visitInsn(OpCodes.POP2);
5 super.visitInsn(OpCodes.ACONST_NULL);
```

Listing 3.2: Auszug für eine automatischen Null-Prüfung mit einem Argument

Dieser Mechanismus funktioniert jedoch nur Argumente mit maximal einem Parameter. Gleichzeitig darf dieser Parameter weder ein `long` noch ein `double` sein, da die DUP2 Instruktion den Stack bit-orientiert kopiert<sup>3</sup>.

### 3.3.3 Iteration 3: Generalisierung

Mit der dritten Iteration<sup>4</sup> wurden schließlich die Sonderbehandlungen für `String.length()` und `Map.get(Object)` gegen eine allgemeingültigere Implementierung ersetzt. Entsprechend der vorhergenannten Einschränkungen können aktuell nur Methoden ohne bzw. maximal einem Argument so manipuliert werden das es zu keiner NPE kommen kann (vgl. Listing 3.3).

Über den Rückgabotyp der Funktion kann zusätzlich erkannt werden von welchem Typ ein entsprechender Eintrag auf dem Stack sein muss. Auch wenn dieser Unterschied zur Laufzeit vermutlich keine Relevanz hat wird bei laden der Klasse der Bytecode auf innere Korrektheit geprüft was den korrekten Datentyp erforderte (vgl. Listing 3.4).

<sup>2</sup>Vgl. Projektsourcen Rev 749111 - CheckNullMethodVisitor.java Zeile 42-131

<sup>3</sup>Vgl. [http://en.wikipedia.org/wiki/Java\\_bytecode\\_instruction\\_listings](http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings)

<sup>4</sup>Vgl. Projektsourcen Rev c6e6bd - CheckNullMethodVisitor.java Zeile 42-169

```

1 Type type = Type.getType(desc);
2 int argumentCount = type.getArgumentTypes().length;
3 int argumentSize = type.getArgumentTypes()[0].getSize();
4
5 if (argumentCount == 0) {
6     invokeMethodWithoutArguments(opcode, owner, name, desc);
7 } else if (argumentCount == 1 && argumentSize < 2) {
8     invokeMethodWithOneArgument(opcode, owner, name, desc);
9 } else {
10     super.visitMethodInsn(opcode, owner, name, desc);
11 }

```

Listing 3.3: Einschränkung der manipulierbaren Methodenaufrufe

```

1 private void pushDefault(Type type) {
2     switch (type.getSort()) {
3     case Type.VOID: break;
4     case Type.BOOLEAN: super.visitIntInsn(Opcodes.BIPUSH, 0); break;
5     case Type.INT: super.visitInsn(Opcodes.ICONST_0); break;
6     [...]
7     default: super.visitInsn(Opcodes.ACONST_NULL); break;
8     }
9 }

```

Listing 3.4: Typabhängiges setzen von Defaultwerten auf den Stack

### 3.3.4 StackSize und Labels

Die maximale Stacksize kann durch dieses Vorgehen ändern und sollte im ungünstigsten Fall  $1 \times \text{Anzahl Methoden ohne Argumente} + 2 \times \text{Anzahl Methoden mit einem Argument}$  größer werden. Dieser unwahrscheinliche Fall tritt jedoch nur auf wenn alle Methoden Ergebnisse (ohne Lokale Zwischenspeicherung in Variablen) aufeinander aufbauen.

Zur Optimierung innerhalb der Java-Laufzeitumgebung beinhalten Methoden die Informationen wieviele elemente sich maximal auf dem Stack befinden können. Diese Variable ( $\text{MAXSTACK} = n$ ) muss entsprechend der oben gekannten Berechnungen angepasst werden.

Durch das setzen des `ClassWriter.COMPUTE_MAXS` flags beim erzeugen einer `ClassWriter` Instanz kann die ASM Bibliothek diese Berechnung übernehmen. Da der Java-Bytecode beim laden entsprechende Konsistenz-Prüfungen vornimmt ist es dringend erforderlich das dieses erneute Berechnung stattfinden.

## 3.4 Umsetzung ClassLoader

Die ClassLoader-Implementierung `JHClassLoader` lädt die nötigen Ressourcen von ihrem Parent-ClassLoader und manipuliert den erhaltenen Bytecode-Datenstrom mithilfe der oben beschriebenen `CheckNullClassVisitor` Implementierung.

Durch passendes Zusammenstecken der ClassVisitor-chain kann der gegebene Bytecode-Datenstrom bzw. das manipulierte Bytecode-Ergebnis mithilfe der Klasse `TraceClassVisitor` auch ausgegeben werden.

## 3.5 TODO

- Shell-Script das Class-Dateien bearbeitet.
- Ein kleines Shell-Script welches den Classloader setzt (für bestimmte Klassen?)
- zB `javahardener -Dharden=methodcalls -cp ... Main?`
- oder `jarh -Dharden=methodcalls beispiel.jar?`

## 4 Fazit

### 4.1 Projektergebnis

Im Rahmen dieses Prototypes wurde gezeigt das auf Basis des Visitor-Patterns eine bequeme Anpassung von Java-Bytecode möglich ist. Allerdings bringt dieses Vorgehen auch entsprechende Probleme mit sich und der fehlende Ausführungskontext verhindert u.a. das dieses Vorgehen auf Methodenaufrufe mit mehr als einem Argument angewendet werden kann.

### 4.2 Erweiterungsmöglichkeiten

Auf Basis der gewonnen Erkenntnis könnte dieses Vorgehen auch in kurzes Zeit für weitere NullPointerExceptions-Situationen erweitert werden: Etwa das Lesen (Instruktion GETFIELD) oder Schreiben (SETFIELD) von Attributen.

Auch der Zugriff auf Arrays ([B,C,S,I,F,L,D,A]ALOAD bzw. [B,C,S,I,F,L,D,A]ASTORE)) bzw. die Längenprüfung (ARRAYLENGTH) gegen NULL-Zugriff gesichert werden.

Während die zusätzlichen Instruktionen die Performance einer Anwendung negativ beeinflussen können wären mögliche Optimierungspotenzialle oft nur über einen größeren Kontext (Vorherige Zuweisungen, Schleifen, etc.) zu fassen. Für eine Bewertung der Performance wären Änderungsstatiken sowie Performanceanalysen nötig.

Dadrüber hin wären Optionen zum aktivieren/deaktivieren des manipulation auf Package, Klassen oder Methoden Basis. Etwa über Umgebungsvariablen. So können etwa Zugriffe auf häufige, aber im generellen als sicheregelte Aufrufe ungeprüft bleiben (etwa `System.[in,out,err].*-Aufrufe`).

Ob eine NULL-Prüfung etwa bereits durch den Programmier sichergestellt wurde (oder durch ein doppeltes laden durch die hier entwickelten Klassen) läßt sich aktuell nicht entscheiden. Auch das in gewissen Situationen eine weitere Prüfung nicht nötig wäre könnte auf der Basis der aktuellen Entwicklung nicht nachgerüstet werden. (Variablen Zuweisungen aus einem Konstruktor, Konstante Zuweisungen (etwa eines primitiven Datentypes oder eines Strings) oder die für das Autoboxing verwendeten Methoden `valueOf` sind etwa Garantien dafür dass eine Variable nicht NULL sein kann.)

# Eidesstattliche Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben.

Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Gummersbach, 31. Juli 2013

Christoph Jerolimov