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 `NullPointerException`s.

Dozent: Prof. Dr. Erich Ehses
Fachhochschule Köln

ausgearbeitet von

Christoph Jerolimov, Matrikelnr. 11084742

Sommersemester 2013

Inhaltsverzeichnis

1 Abstrakt

`NullPointerException`s (im Folgendem `NPE`) sind ein (üblicherweise leicht lösbares) Standardproblem der Softwareentwicklung und treten in der Programmiersprache Java auf, wenn Methoden- oder Attribut-Zugriffe auf `null`-Object erfolgen[1].
Die Behandlung solcher ungültiger Aufrufe ist grundsätzlich abhängig von der Programmiersprache (Definition) bzw. 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, ignoriert werden[2].
Diese fehlertolerantere Version von Objective-C soll hier für die Programmiersprache Java nachgebildet werden und durch eine automatisierte manipulation des Java-Bytecodes erreicht werden. Wie in der Vorlage müssen, soweit dies deklariert wurde, Methoden einen Rückgabewert liefern. Hier werden, analog zu Objective-C, möglichst neutrale Werte gewählt: `false` für boolsche Ausdrücke, *Null (0)* für Zahlen sowie `NULL`-Referenzen für Objekte.
Die beiden folgenden zwei Anwendungsfälle (vgl. Listing 1.1 und 1.2) verdeutlichen die vereinfachte Lesbarkeit und würden ohne Bytecode-Manipulation jeweils zu einer zu `NPE` 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 Folgenden werden verschiedene technische Möglichkeiten untersucht, sowie deren prototypische Umsetzung beschrieben.
Zur besseren Lesbarkeit wird auf Java-Packagenamen (etwa `java.lang`) sowie Generics in allen Java und Bytecode (Assembler ähliche Schreibweise) Darstellungen verzichtet. Alle angesprochenen Dateien sind Bestandteil des zugehörigem Quellcodeprojekts[3].

[1] Dadrüber hinaus kann eine NPE auch noch in anderen Fällen geworfen werden. Vgl. http://www.java-blog-buch.de/0503-nullpointerexception/, abgerufen am 31.07.2013

[2] Weitereführende Informationen zur Sprache Objective-C: http://developer.apple.com/library/mac/documentation/Cocoa/Con abgerufen am 31.07.2013

[3] U.a. zu finden auf GitHub: https://github.com/jerolimov/java-hardener

2 Analyse

2.1 Problemstellung

Wie in der Einführung beschrieben, können Objektaufrufe, 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 Prototyp umgesetzt sollen aber wenigstens in dieser Einführung angesprochen werden.
Problematisch sind insbesondere verkettete Aufrufe (vgl. Listing 2.1). So müssen die Zwischenergebnisse etwa in lokalen Variablen gespeichert werden (vgl. Listing 2.2) oder die Aufrufe wiederholt werden, wenn diese in umgebende Bedingungen eingebaut werden (vgl. Listing 2.3). Letzteres würde jedoch nicht nur die Performance negativ beeinflussen, sondern könnte bei inmutablen Zugriffen auch zu Fehlerhaften Programmabläufen führen.

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

Listing 2.1: Beispiel für verkette 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` Anwendung können verschiedene Lösungswege deassembliert und analysiert werden. Zum einfacheren Aufruf wurde ein kleines Shell-Script (siehe textifier) erstellt. Mitdessen 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/fhkoeln/.../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 ohne NULL-Prüfung

```
1  public class de/fhkoeln/.../testcases/Test1 {
2    public getStringLength(LMap;LString;)I
3      ALOAD 1
4      ALOAD 2
5      INVOKEINTERFACE Map.get (LObject;)LObject;
6      CHECKCAST String
7      INVOKEVIRTUAL String.length ()I
8      IRETURN
9      MAXSTACK = 2
10     MAXLOCALS = 3
11 }
```

Listing 2.5: Bytecode Auszug für Listing 2.4

Im Folgenden sollen die Unterschiede aufgezeigt werden, wenn man diese ursprüngliche Version mit einer `NPE`-gesicherte Versionen vergleicht. Die dafür angelegten Klassen befinden sich im Quellcode-Ordner `test` innerhalb des Java-Packages `de.fhkoeln...analysebytecode`.

2.2.2 Bedingte Ausführung

Durch eine NULL-Prüfung, etwa mit einem Bedingungsoperator ?:, fügt der Compiler zwei Labels (Ziele für Springmarken) ein und prüft anschließend die aktuell auf dem Stack liegende Variable auf NULL (vgl. Listing 2.6 Zeile 2). Ergibt etwa die Prüfung von `e != null ? e.toString() : null`, dass die Variable NULL ist, so springt die Ausführung zur angegebenen Sprungmarke (hier L0) und fügt eine NULL-Referenz auf den Stack hinzu (Zeile 7). Falls die NULL-Prüfung jedoch negativ ausviel, 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 daraufhin 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 Intruktionen beinhaltet als die zuvor genannte Bedingungsoperator-Variante. Gleichzeitig wird für quasi jeden Methodenaufruf eine zusätzliche lokale Variable benötigt. (Ggf. könnten diese auf eine Varaible je Datentyp kombiniert werden.)
Darüber hinaus würde diese Variante nicht nur unmittelbare NullPointerExceptions abfangen sondern auch Fehler welche innerhalb der Methode ausgeführt werden und ggf. gar nicht vom java-hardener manipuliert wurden.

3 Umsetzung

3.1 Maven

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

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

Zum Herrunterladen der Resourcen 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 speziel dafür entwickelte OpenSource-Bibliothek ASM[1] 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).

[1] Weitereführende Informationen zu ASM auf der offiziellen Homepage: http://asm.ow2.org/, besucht am 31.08.2013

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 deligiert den Visitor für Methoden und Klassenvariablen (Fields) an neue Instanzen der beiden folgenden Klassen.
- `MethodVisitor` Instanzen werden für jede Methodendeklaration, sowie die enthaltene Implementation (jede Instruktion 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.
Mithilfe der Klasse `TraceClassVisitor` können Bytecode Daten in für menschen deutlich bessere, Assembler-artige Ausgaben umgewandelt werden.

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 Erstere 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-Vistor kümmert sich anschließend um die Prüfung aller `INVOKE_` Assembler Aufrufe. Hierfür muss die Methode `visitMethodInsn` überschrieben werden.
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 Konstruktur 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 prototische Umsetzung[2] 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 Sprungmarke (hier `Label fallback`, vgl. Zeile 2, 6 und 10). Wenn die `NULL`-Prüfung ergibt, dass es zu keiner `NPE` kommen wird, wird in der nächsten Intruktion der Original Aufruf durchgeführt und hier die `super`-Methode aufgerufen, welche die Argumente an den jeweils nachgeschalteten `MethodVisitor` übergibt. Um den im folgendenen 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 erwartetem 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);
```

[2] Vgl. Projektsourcen - Rev 951f48 CheckNullMethodVisitor.java Zeile 43-66

```
11          super.visitInsn(Opcodes.ICONST_0);
12          super.visitLabel(behind);
13  }
```

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

3.3.2 Iteration 2: Verfielfältigung

Bei der zweiten Iteration[3] wurde versucht, dieses Vorgehen auch auf andere Methoden anzuwenden und die jeweiligen Unterschiede zu beleuchten.
Problematisch ist dabei die Reihenfolge des Stacks für den jeweiligen Methodenaufruf. So liegen auf oberster Position des Stacks die nötigen Argumente, und erst unter diesen die eigentliche Objektreferenz dessen Methode aufgerufen werden soll. Um die Objektreferenz einer `NULL`-Prüfung mit `IFNULL` unterziehen zu können, muss jedoch diese Referenz jedoch oben auf dem Stack aufliegen.
Für Methoden mit nur einem Argument konnte dies noch einfach über das Hintereinanderschalten der beiden Intruktionen `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 wider 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 für Methoden mit maximal einem Argument. Gleichzeitig darf dieser Parameter weder ein `long` noch ein `double` sein, da die `DUP2` Instruktion den Stack bit-orientiert kopiert[4].

3.3.3 Iteration 3: Generalisierung

Mit der dritten Iteration[5] 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

[3] Vgl. Projektsourcen Rev 749111 - CheckNullMethodVisitor.java Zeile 42-131
[4] Vgl. http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings
[5] Vgl. Projektsourcen Rev c6e6bd - CheckNullMethodVisitor.java Zeile 42-169

einem Argument so manipuliert werden, dass es zu keiner `NPE` kommen kann (vgl. Listing 3.3).
Über den Rückgabetyp 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 beim Laden der Klasse der Bytecode auf innere Korrektheit geprüft was den korrekten Datentyp erforderte (vgl. Listing 3.4).

```
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 sich durch dieses Vorgehen ändern und sollte im ungünstigsten Fall *1 x Anzahl Methoden ohne Argumente + 2 x Anzahl Methoden mit einem Argument* größer werden. Dieser unwahrscheinliche Fall tritt jedoch nur auf, wenn alle Methoden (ohne Lokale Zwischenspeicherung in Variablen) unmittelbar aufeinander aufbauen.
Zur Optimierung innerhalb der Java-Laufzeitumgebung beinhalten Methoden die Informationen wieviele Elemente sich maximal auf dem Stack befinden können. Dieser Wert (`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 zwigend erforderlich, dass diese erneute Berechnung stattfindet.

3.4 Umsetzung ClassLoader

Die ClassLoader-Implementierung `JHClassLoader` lädt die nötigen Resourcen 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, dass 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., dass dieses Vorgehen auf Methodenaufrufe mit mehr als einem Argument angewendet werden kann.

4.2 Erweiterungsmöglichkeiten

Auf Basis der gewonnenen Erkenntnis könnte dieses Vorgehen auch in kurzer Zeit für weitere Situationen, in dennen `NullPointerExceptions` auftreten können, erweitert werden: Beispielsweise 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. das Auslesen der Array-Länge (`ARRAYLENGTH`) kann so gegen `NULL`-Zugriff gesichert werden.
Da die zusätzlichen Instruktionen die Performance einer Anwendung negativ beeinflussen können, müssten noch weitere mögliche Optimierungspotenzialle erkannt werden. Dieser Kontext, etwa Schleifen oder eine vorherige Zuweisung, ist auf Basis des Visitor Patterns nicht oder nur sehr schwer zu fassen. Für eine Bewertung der Performance wären Änderungsstatiken sowie Performanceanalysen nötig.
Darüber hinaus wären Optionen zum Aktivieren/Deaktivieren der Manipulationen auf Package, Klassen oder Methoden Basis sinnvoll, etwa über Umgebungsvariablen. So könnten Zugriffe auf häufige, aber im generellen als sicher geltene Aufrufe unverändert bleiben (Z.B. `System.[in,out,err].*` - Aufrufe).
Ob eine `NULL`-Prüfung bereits durch den Programmier sichergestellt wurde (oder durch ein doppeltes laden durch die hier entwickelten Klassen), läßt sich aktuell nicht entscheiden. Auch, dass in gewissen Situationen eine weitere Prüfung nicht nötig ist, kann auf der Basis der aktuellen Entwicklung nicht nachgerüstet werden. So müssen Variablen Zuweisungen aus einem Konstruktur, Konstante Zuweisungen (etwa eines primitiven Datentypes oder eines Strings) oder die für das Autoboxing verwendeten Methoden *WrapperKlasse*`.valueOf()` nicht geprüft werden. Diese Zuweisungen Garantieren, 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 Prüfungsgbehörde vorgelegen.

Gummersbach, 31. Juli 2013

Christoph Jerolimov