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 Die Idee

`NullPointerException`s (NPE) sind ein klassisches Problem 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 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[2], 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 NullPointerExceptions 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

Für die Umsetzung bietet sich die ASM[3] Bibliothek an welche für das manipulieren von Java-Bytecodes verschiedene technische Möglichkeiten an, diese werden im folgendem Untersucht und deren prototypische Umsetzung beschrieben wird.

[1] Dadrüber hinaus kann eine NPE auch noch in anderen Fällen geworfen werden. Vgl. http://www.java-blog-buch.de/0503-nullpointerexception/
[2] Vgl. http://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/
[3] Vgl. http://asm.ow2.org/

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 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 enthaltenenen Textifier Programms 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/gm/cui/javahardener/testcases/Test1.class`

2.2.1 Ausgangsbasis

```
1 package de.fhkoeln.gm.cui.javahardener.testcases;
2 public class Test1 {
3         public int getStringLength(Map<String, String> map, String key) {
4                 return map.get(key).length();
5         }
6 }
```

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 java/lang/String
7      INVOKEVIRTUAL java/lang/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 aufgezeiugt 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 `de.fhkoeln.gm.cui.javahardener.analysebytecode`.

2.2.2 Bedingungsoperator ?:

Durch die Null-Prüfung mit einem Bedingungsoperator (etwa `entry != null ? entry.toString() : null`) fügt der Compiler zwei Labels (Ziele für Springmarken) 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 `GOTO`s (hier zur Sprungmarke L1) übersprungen.

```
1     ALOAD 2 /* entry */
2     IFNULL L0
3     ALOAD 2 /* entry */
4     INVOKEVIRTUAL java/lang/String.toString ()Ljava/lang/String;
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:
`int l; try  l = entry.length();  catch (NPE e)  l = 0`
Der dadrauf entstehende Bytecode speichert das Ergebnis des Originalaufrufs in einer lokalen Variable (Listing 2.7 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 java/lang/NullPointerException
2    L0
3     ALOAD 2
4     INVOKEVIRTUAL java/lang/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 2.7: Auszug ASM für Null-Prüfung mit try-catch

Insgesamt fällt auf das dieser Code bereits bei diesem einfachem 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.)
Dadrüber 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

- Stack size anpassen?
- Labels anpassem
- Toolchain
- 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`?

3.1 ASM

Zur Manipulation von Java Bytecode bietet sich die leichtgewichtige und speziel 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).

3.1.1 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
```

Supported maven 2+ (tested with maven 3) commands:

```
mvn compile # download the dependencies and compile the sources
mvn package # compiles, test and package the java-hardener sources
```

3.1.2 Visitor Pattern

Siehe asm

3.1.3 Tree / DOM API

Siehe asm-tree

3.2 ClassLoader

Siehe JHClassLoader, CheckNullClassVisitor und CheckNullMethodVisitor.
It shall use the ASM (Vgl. http://asm.ow2.org/) bytecode manipulation and analysis framework and will be implemented as a bytecode-to-bytecode post-processor and maybe also as java `ClassLoader`.

3.3 IFNULL Bytecode manipulation

Version 1: https://github.com/jerolimov/java-hardener/blob/951f48194f53baebd0915c01e0ed3cc2596bd0db/s
66
Version 2: https://github.com/jerolimov/java-hardener/blob/749111f5dcc3f71a1d1db5a669591288245e912b/s
131
Version 3: https://github.com/jerolimov/java-hardener/blob/c6e6bdc7d081eae5e47d2c926073aa3715d908f6/s
169

4 Erweiterungsmöglichkeiten

4.1 Möglichkeite Laufzeitprobleme

- Statistiken ausgeben
- Anaylse Umsetzungsmöglichkeiten (aus `variable.doAnything()` wird z.b.)
- Springmarke mit `label` / `goto`?
- Kann man das ggf. Erkennen (vgl. Optimierung `Integer.valueOf()`)?
- Was ist wenn dies Eingebunden in Schleifen ist?
- Was ist wenn sie mehrmals hintereinander aufgerufen wird?
- Wenn Variable zuletzt gesetzt wurde ist mit new, kann sie nicht null sein.
- Wenn Variable zuletzt gesetzt wurde mit einem "String" oder einem primitiven Typen, kann sie nicht null sein.
- Nicht für `System.[in,out,err].*`-Aufrufe.
- Nicht wenn Ergebnis von `Integer.valueOf()`, `Integer.toString()`, etc.
- Nicht wenn Feld `final` ist

5 Fazit

Zum Ende eines jeden Projektes sollte die Entwicklung und der Abschluss noch einmal kritisch hinterfragt und den vorher gesetzten Zielen gegenübergestellt werden.

5.1 Projektstatus

Positive Aspekte

Negative Aspekte

5.2 Reflektion und Ausblick

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