Inhalt:

1. Motivation
2. Codeanalyse
   1. Statisch
   2. Dynamisch
   3. Vergleich
3. Compilerpatterns
   1. Insbesondere Visitor
4. Java Bytecode
   1. Opcodes
   2. Stack
5. Erweiterte Motivation
   1. Code Reusability
6. Technologien
   1. Spotbugs
7. Praktische Umsetzung
8. Schluss
9. Literaturverzeichnis

# 1. Motivation

In vielen Projekten kommt es heutzutage zu unzähligen Wiederholungen von Code. Dieser Umstand ist nicht nur kurzfristig von Nachteil, da für jede neue Iteration desselben Codes ein Entwickler sich die Zeit nehmen muss diesen nach bestem Wissen und Gewissen zu implementieren, sondern auch langfristig von Nachteil, da man bei jeder Änderung des Codes auch daran denken muss die anderen Codewiederholungen zu überarbeiten.

Ein erster Schritt hierfür ist es sich wiederholende Codefragmente in Methoden auszulagern und diese zu verwenden. Hierbei kann es jedoch passieren, dass jedes der vorliegenden Codefragmente Kleinigkeiten anders macht. Diese Unterschiede sollten möglichst verringert werden, damit die jeweilige Methode an möglichst vielen (bestenfalls allen) Stellen verwendet werden kann.

Des Weitern kann sich das Problem ergeben, dass der Methodenkopf, der ausgelagerten Methode, nicht an jede Stelle passt, da beispielsweise an der einen Stelle eine „ArrayList“ und an der anderen Stelle eine „LinkedList“ verwendet wird und diese Typen beispielsweise in der Programmiersprache Java nicht implizit aufeinander gecastet werden können.

Dieses Problem soll von dieser Arbeit aufgegriffen und gelöst werden. Das Ziel soll somit sein, dass ein Programm geschrieben wird, welches den Sourcecode nach Methoden durchsucht, welche beispielsweise als Übergapeparameter eine „ArrayList“ hat, jedoch keine der Methoden benutzen, welche nicht auch schon durch eine allgemeinere Klasse oder ein Interface, wie zum Beispiel „List“, welches sowohl von einer „ArrayList“ als auch von einer „LinkedList“ implementiert wird, implementiert werden.

Als Hilfsmittel für die Codeanalyse wird „SpotBugs“ verwendet. SpotBugs ist ein Programm, um Fehler zu finden. Bei diesen Fehlern ist allerding während der Compile-Time bereits klar ist, dass ein Fehler beziehungsweise eine allgemeine Ungereimtheit vorliegt. Dieser Fehler selbst jedoch keinen Fehler bei der Übersetzung des Codes erzeugt. Des Weiteren kann man für SpotBugs eigene Plugins entwickeln, welches spezifische Fehler entdeckt. Das Endprodukt dieser Arbeit soll somit ein fertig entwickeltes SpotBugs-Plugin sein, dass Fehler, wie im Absatz vorher beschrieben entdeckt und ausgibt. Hierfür wird auf die mächtige SpotBugs API zur Entwicklung von Plugin zugegriffen.

# 2. Design Patterns

Bevor tiefer in die Materie eingestiegen werden kann, muss ein kleiner Absprung in die Welt der Design Patterns und spezieller Design Pattern im Umfeld von Compilern gemacht werden.

Hierbei wird zuerst erklärt, was Design Patterns sind. Im Anschluss daran die verschiedenen Arten von Design Patterns aufgezeigt und für jede dieser Arten ein beispielhaftes Design Pattern erläutert.

## 2.1 Erklärung Design Patterns

Ein Design Pattern soll den Software Entwickler dabei unterstützen wiederkehrende Probleme schnell und einfach zu lösen. Hierbei muss jedoch gesagt werden, dass ein Design Pattern keine endgültige Lösung ist. Ein Design Pattern ist ausschließlich eine Art Bauplan, um ein Problem auszubauen. [Shvets 2020] Somit gibt ein Design Pattern nur eine Richtung der Implementierung und nicht eine genaue Implementation vor. Ein Design Pattern hat nicht den Anspruch dauerhaft gültig zu sein. Design Patterns können und sollen mit der Zeit und den Problemen abgewandelt und verbessert werden. [Shvets 2020]

Durch diesen Bauplan, welcher von Design Patterns geliefert wird, können Problemlösungen beschleunigt werden. Dies liegt unter anderem daran, dass die Implementierungen sowohl bewiesen als auch getestet sind. Des Weiteren können Design Patterns dabei helfen Probleme zu beheben bevor sie entstehen. Ebenso wird durch den Einsatz von Design Patterns die Lesbarkeit des Codes erhöht, wenn alle beteiligten Entwickler das jeweilige Design Pattern kennen. [Shvets 2020] Zusätzlich ist ein Vorteil von Design Patterns, dass diese sprachenunabhängig fungieren. Somit können Problemlösungen problemlos von einer Sprache in die nächste übersetzt werden. [Wikipedia Entwurfsmuster]

Allerdings haben Design Patterns nicht nur Vorteile. Ein Nachteil ist beispielsweise, dass in den Köpfen der Entwickler das Design Pattern zu sehr befolgt wird, ohne, dass offensichtliche Verbesserungen an den jeweiligen Implementierungen ignoriert werden. Hierdurch kann sowohl in der Entwicklungszeit ineffizient gearbeitet werden als auch die Laufzeiteffizient des Programms beeinträchtigt werden. [Shvets 2020] Allein das Verwenden eines Design Patterns sorgt nicht für guten Code.

Schlussendlich kann man jedoch sagen, dass es für jeden Entwickler von Vorteil sein kann Design Patterns zu beherrschen. Design Patterns zu verwenden ist so lange vorteilhaft, wie sie nicht in zu großer Anzahl verwendet werden oder vom Entwickler zu genau befolgt werden.

## 2.2 Arten von Design Pattern

Im folgenden Kapitel werden die verschiedenen Arten aufgezeigt, in welche sich Design Patterns einordnen lassen. Zu jeder dieser Arten werden Beispiel Design Patterns erläutert. Die drei Arten von Design Patterns umfassen:

1. Strukturelle Pattern
2. Verhaltensorientierte Pattern
3. Erzeugungs Pattern

[Shvets 2020]

Es gibt allerding auch Design Patterns, welche sich nicht in die drei eben genannten Arten einteilen lassen. Diese werden in dieser Arbeit nicht berücksichtigt.

### 2.2.1 Strukturelle Pattern

Ein Design Pattern dieser Kategorie regeln Beziehungen zwischen einzelnen Entitäten und sollen somit den Entwurf erleichtern. Strukturell und objektbasierte Pattern definieren Wege Objekte so zusammenzufügen, dass neue Funktionalitäten entstehen. [Shvets 2020]

Ein Beispiel für ein Strukturelles Pattern ist das Decorator Pattern, welches im Folgenden erläutert wird.

Mit Hilfe des Decorator Patter kann die Funktionalität einer Methode erweitert werden, ohne dass die zu Erweiternde Methode direkt erweitert wird. Dadurch bleibt die originale Funktionalität dauerhaft erhalten. [Baeldung 2019]

Klassen, die zur Implementierung des Decorator Patterns benötigt werden:

Eine Klasse, welche eine Methode enthält, die dekoriert werden soll:

**public class** ChirstmasTreeImpl **implements** ChristmasTree {  
 @Override  
 **public** String decorate() {  
 **return "Weihnachtsbaum"**;  
 }  
}

Ein Interface für die zu dekorierende Methode:

**public interface** ChristmasTree {  
 String decorate();  
}

Eine Klasse, welche die Dekorier Funktion zur Verfügung stellt:

**public class** TreeDecorator **implements** ChristmasTree {  
 **private** ChristmasTree **christmasTree**;  
 **public** TreeDecorator(ChristmasTree christmasTree) {  
 **this**.**christmasTree** = christmasTree;  
 }  
 @Override  
 **public** String decorate() {  
 **return christmasTree**.decorate();  
 }  
}

Diese Klasse implementiert das „ChristmasTree“ Interface und besitzt ein Objekt des Interfaces.

Eine Klasse, welche von der eben genannten Klasse ableitet und die „decorate()“ Methode entsprechend modifiziert.

**public class** Star **extends** TreeDecorator {  
 **public** Star(ChristmasTree christmasTree) {  
 **super**(christmasTree);  
 }  
  
 **public** String decorate() {  
 **return super**.decorate() + decorateWithStar();  
 }  
  
 **private** String decorateWithStar() {  
 **return " mit Stern"**;  
 }  
}

Wie das Decorator Pattern benutzt werden kann, zeigt diese Klasse.

**public class** Main {  
 **public static void** main(String[] args) {  
 ChristmasTree christmasTree1 = **new** Balls(**new** ChirstmasTreeImpl());  
 System.***out***.println(christmasTree1.decorate());  
  
 ChristmasTree christmasTree2 = **new** Balls(**new** Star(**new** ChirstmasTreeImpl()));  
 System.***out***.println(christmasTree2.decorate());  
 }  
}

Die Klasse „Balls“ hat hierbei eine ähnliche Implementierung wie die Klasse Star.

Wenn diese Methode ausgeführt wird, erhält man folgende Ausgabe:

*Weihnachtsbaum mit Kugeln  
Weihnachtsbaum mit Kugeln mit Stern*

Mit Hilfe des Decorator Patterns wurde nun erfolgreich die Methode „decorate“ der Klasse „ChristmasTreeImpl“ dekoriert. Die Funktionalität der Methode wurde erweitert, ohne dass die Methode an sich verändert wurde. Somit ergibt sich eine stark modulare Struktur.

Die Codebeispiele dieses Kapitel sind von [Baeldung Decorator 2019] inspiriert.

### 2.2.2 Verhaltensorientierte Pattern

Ein Verhaltensorientiertes Design Pattern hilft bei der Entwicklung von der Kommunikation zwischen verschiedenen Objekten. Mit diesen Design Patterns kann komplexes Verhalten modelliert werden. Hiermit wird die Flexibilität der Software, in Bezug auf ihr Verhalten, erhöht. [Shvets 2020]

Als Beispiel für Verhaltensorientierte Pattern wird das Visitor Pattern herangezogen.

Das Ziel des Visitor Patterns ist es neue Funktionalitäten zu implementieren, ohne dass Modifizierungen an der bisherigen Objekt Struktur durchgeführt werden müssen. [Baeldung Visitor 2019]

Dis wird durch das Einführen von sogenannten Visitors und sogenannten Elementen erreicht. Die Visitors sind die Teile des Codes, in welchen die Geschäftslogik ablaufen soll. Die Elemente sind Daten, auf welchen Logik stattfinden soll. [Baeldung Visitor 2019]

Klassen, um das Visitor Pattern zu realisieren:

Die Elemente:

**class** FOO **implements** Element {  
 **public void** accept(Visitor v) {  
 v.visit(**this**);  
 }  
 **public** String getFOO() {  
 **return "FOO"**;  
 }  
}  
  
**class** BAR **implements** Element {  
 **public void** accept( Visitor v ) {  
 v.visit( **this** );  
 }  
 **public** String getBAR() {  
 **return "BAR"**;  
 }  
}

Das zugehörige Interface:

**public interface** Element {  
 **void** accept(Visitor v);  
}

Die Visitors:

**class** UpVisitor **implements** Visitor {  
 **public void** visit(FOO foo) {  
 System.***out***.println(**"do Up on "** + foo.getFOO());  
 }  
 **public void** visit(BAR bar) {  
 System.***out***.println(**"do Up on "** + bar.getBAR());  
 }  
}  
  
**class** DownVisitor **implements** Visitor {  
 **public void** visit(FOO foo) {  
 System.***out***.println(**"do Down on "** + foo.getFOO());  
 }  
 **public void** visit(BAR bar) {  
 System.***out***.println(**"do Down on "** + bar.getBAR());  
 }  
}

Das zugehörige Interface:

**public interface** Visitor {  
 **void** visit(FOO foo);  
 **void** visit(BAR bar);  
}

Benutzung des Visitor Patterns:

**public class** Main {  
 **public static void** main( String[] args ) {  
 Element[] list = {**new** FOO(), **new** BAR()};  
 UpVisitor up = **new** UpVisitor();  
 DownVisitor down = **new** DownVisitor();  
 **for** (Element element : list) {  
 element.accept(up);  
 }  
 **for** (Element element : list) {  
 element.accept(down);  
 }  
 }  
}

In diesem Beispiel wird jeweils ein Element von „Foo“ und „Bar“ erzeugt. Es wird sowohl ein UpVisitor als auch ein DownVisitor erzeugt. Diese werden für jedes Element durchlaufen. Die Ausgabe für diese Methode lautet:

*do Up on FOO  
do Up on BAR  
do Down on FOO  
do Down on BAR*

Mit Hilfe dieses Patterns kann durch das Erstellen eines neuen Visitors problemlos neues Verhalten eingebaut werden. Hierfür muss die bisherige Struktur nicht angepasst werden.

### 2.2.3 Erzeugung Pattern

Der Hauptnutzen von Konzeptionellen Pattern ist die Initialisierung von Klassen. Diese Art von Design Pattern kann grob in zwei Untergruppen geteilt werden.

1. Patterns, um Klassen zu erstellen.
2. Patterns, um Objekte zu erstellen

Ersteres möchte die Vererbung effektiv benutzen, um Klassen zu instanziieren. Letzteres behandelt die Objekterzeugung, wobei Aufgaben anderen Objekten zugeteilt werden. [Shvets 2020]

Ein Beispiel für das Erzeugungsmuster ist das Builder Pattern, welches im Folgenden genauer betrachtet wird.

Das Builder Pattern soll es ermöglichen, dass komplexe Objekte mithilfe von simplen Strukturen und einem Schritt für Schritt Verfahren gebaut werden können. Die Verbindung zu den Erzeugungspattern ist somit leicht aufgebaut, da das Builder Pattern Objekte erzeugt. [Tutorialspoint 2020]

Klassen, welche benötigt werden, um das Builder Pattern zu realisieren:

Eine Klasse, welche initialisiert werden soll:

**public class** Task {  
 **public final long id**;  
 **public** String **summary** = **""**;  
 **public** String **description** = **""**;  
 **public boolean done** = **false**;  
 **public** Date **dueDate**;  
 **public** Task(**long** id) {  
 **this**.**id** = id;  
 }  
 **public** Task(**long** id, String summary, String description, **boolean** done,  
 Date dueDate) {  
 **this**.**id** = id;  
 **this**.**summary** = summary;  
 **this**.**description** = description;  
 **this**.**done** = done;  
 **this**.**dueDate** = dueDate;  
 }  
}

(Um den Code nicht aufzublähen wurde auf Getter- und Setter-Methoden verzichtet)

Eine Builder Klasse, welche die Komplexität der Klasseninitialisierung abstrahiert:

**public class** TaskBuilder {  
 **private final long id**;  
 **private** String **summary** = **""**;  
 **private** String **description** = **""**;  
 **private boolean done** = **false**;  
 **private** Date **dueDate**;  
 **public** TaskBuilder(**long** id) {  
 **this**.**id** = id;   
 }  
 **public** TaskBuilder setSummary(String summary) {  
 **this**.**summary** = summary;  
 **return this**;  
 }  
 **public** TaskBuilder setDescription(String description) {  
 **this**.**description** = description;  
 **return this**;  
 }  
 **public** TaskBuilder setDone(**boolean** done) {  
 **this**.**done** = done;  
 **return this**;  
 }  
 **public** TaskBuilder setDueDate(Date dueDate) {  
 **this**.**dueDate** = **new** Date(dueDate.getTime());  
 **return this**;  
 }  
 **public** Task build() {  
 **return new** Task(**id**, **summary**, **description**, **done**, **dueDate**);  
 }  
}

Wenn nun ein Objekt der Klasse Task erstellt werden möchte, kann dies über die TaskBuilder Klasse erreicht werden.

**public class** Main {  
 **public static void** main(String[] args) {  
 TaskBuilder taskBuilder = **new** TaskBuilder(1);  
 taskBuilder.setSummary(**"Zusammenfassung"**);  
 taskBuilder.setDescription(**"Beschreibung"**);  
 taskBuilder.setDone(**false**);  
 taskBuilder.setDueDate(**new** Date());  
 Task task = taskBuilder.build();  
 }  
}

Das Endergebnis ist ein Objekt der Klasse Task, welche die vom TaskBuilder Objekt spezifizierten Werte besitzt.

Da jede der „set“-Methoden das zugehörige TaskBuilder Objekt zurückliefert, kann der Code weiter auf eine Anweisung gekürzt werden. Hierdurch wird der Code noch schlanker.

**public class** Main {  
 **public static void** main(String[] args) {  
 Task task = **new** TaskBuilder(1)  
 .setSummary(**"Zusammenfassung"**)  
 .setDescription(**"Beschreibung"**)  
 .setDone(**false**)  
 .setDueDate(**new** Date())  
 .build();  
 }  
}

Mithilfe der TaskBuilder Klasse kann problemlos mit Teilinformationen eine Instanz der Klasse Task erstellt werden, obwohl diese zur Instanziierung fünf verschiedene Parameter benötigt. Gleichzeitig wird die Lesbarkeit des Codes erhöht.

## 2.3 Verbleib

Design Pattern können die Implementierungsdauer stark verkürzen, indem sie die groben Konstrukte bereits vorgeben. Gleichzeitig kann durch diese die Wartung erleichtert werden. Des Weiteren können durch Design Patterns Projekte leichter zwischen verschiedenen Entwicklern gewechselt werden, vorausgesetzt die jeweiligen Design Patterns sind den Entwicklern bekannt. Jedoch muss hierbei darauf geachtet werden, dass Design Pattern nicht gedankenleer übernommen werden, sondern ausschließlich diese zur Inspiration herangezogen werden.

Für diese Arbeit ist vor allem das Visitor Pattern im Kopf zu behalten.

Abschließend ist zu sagen, dass Design Patterns in der Programmierung ein nicht zu unterschätzendes Hilfsmittel ist.

# 3. Codeanalyse

Die Codeanalyse ist eine Analyse des Source Codes, welche Fehler entdecken soll. Hierbei handelt es sich um eine Fülle von Fehlern. Bei diesen Fehlern handelt es sich hauptsächlich um Fehler, welche vom Compiler nicht erkannt werden können, jedoch zur Laufzeit einen Fehler produzieren können.

Mit Hilfe der Codeanalyse kann die Anzahl an Bugs, welche bei späteren Tests hervorkommen, reduziert werden. Dies sorgt wiederum dafür, dass der vorgegebene Zeitrahmen eingehalten werden kann. Dieser Vorteil kommt vor allem deswegen zut Geltung, da ach einem Bericht des Project Management Institute von 2017 nur 51 Prozent der IT-Projekte in ihrem vorgegebenen Zeitrahmen beendet werden. [PMI 2017]

Grundlegend kann man die Codeanalyse in zwei Arten unterteilen:

1. Statische Codeanalyse
2. Dynamische Codeanalyse

Diese verschiedenen Arten werden Im Folgenden erklärt. Im Anschluss daran werden die statische und die dynamische Codeanalyse miteinander verglichen.

## 3.1 Statische Codeanalyse

Das Ziel der statischen Codeanalyse ist es Fehler zu finden bevor sie auftreten.

Als Beispiel:

**public class** StaticCodeAnalysis {  
 **public static void** main(String[] args) {  
 Object object = **null**;  
 **if** (object.equals(**new** Object())) *// Erzeugt Exception* {  
 *// Kann nicht erreicht werden!* }  
 }  
}

Der Compiler kann bei diesem Code keine Fehler entdecken, da rein syntaktisch der Code keine Fehler enthält. Allerdings wird der Code zur Laufzeit in Zeile 4 immer zu einer *NullPointerException* führen, da das Objekt „object“ auf null ist.

Ein Tool, welches statische Codeanalyse benutzt, kann diesen Fehler bereits zur Design-Time erkennen und den Entwickler zum Handeln auffordern und somit verhindern, dass zu einem späteren Zeitpunkt sich noch einmal mit dieser Codestalle befasst werden muss.

Die verschiedenen Fehlertypen, welche im Normalfall von einem Tool der statischen Codeanalyse erkannt werden, sind:

* Race-Conditions
* Pufferüberläufe
* Speicherlecks
* *Out of Bounds* Fehler bei der Verwendung von Arrays
* Mathematische Fehler (Division durch 0)
* Schwachstellen bei der Nutzung von Formatstring
* Usw.

[Wikipedia Statische Code-Analyse]

Bei einigen Tools zur statischen Codeanalyse beschränkt sich diese nicht nur auf Fehler. Bei diesen Tools wird sie auch dazu eingesetzt gewisse Regeln zur Gestaltung des Sourcecodes zu überprüfen und durchzusetzen.

Als Beispiel für diese Art von Gestaltungregel wird die Platzierung der geschweiften Klammern herangezogen.

Hierbei gibt es einerseits die Variante mit der geschweiften Klammer in derselben Zeile:

**public class** CurlyInlineNextLine {  
 **public void** method() {  
 *// Aktion* }  
}

Und andererseits die Variante mit der geschweiften Klammer in der nächsten Zeile:

**public class** CurlyInlineNextLine  
{  
 **public void** method()   
 {  
 *// Aktion* }  
}

Ein Tool der statischen Codeanalyse kann den Sourcecode nach Abweichungen dieser Gestaltungsregel durchsuchen und die entsprechenden Stellen hervorheben, um den Entwickler zum Handeln aufzufordern.

### 3.1.1 Techniken der statischen Codeanalyse

Um Sourcecode mittels der statischen Codeanalyse zu durchleuchten, existieren verschieden Techniken. Ein einzelnes Tool der statischen Codeanalyse verwendet meist mehrere oder gar alle der im Folgenden genannten Techniken, um den Code zu analysieren. Diese Techniken sind meist von verschiedenen Compilertechniken abgeleitet. Die verschiedenen Techniken sind:

1. Datenflussanalyse
2. *Taint-*Analyse
3. Lexikalische Analyse

[Owasp]

**Datenflussanalyse**

Die Datenflussanalyse wird benutzt, um Informationen der Laufzeit zu bekommen, während das Programm sich in seinem statischen Zustand befindet. Dies bedeutet, dass das Programm nicht ausgeführt wird. [Owasp]

In der Datenflussanalyse gibt es drei verschiedene Teile:

1. Code Block
2. Erstellung eines Kontrollflussgraphen
3. Durchlaufen des Kontrollflussgraphen

Der Code Block bildet hierbei das zu analysierende Stück Sourcecode.

Das Ergebnis von Punkt 2 ist ein Kontrollflussgraph. Ein beispielhafter Kontrollflussgraph ist im Folgenden abgebildet.



Abbildung : Kontrollflussgraph mit Schleife [Wikipedia Kontrollflussgraph]

Ein Kontrollflussgraph ist eine abstrakte graphische Representation von Software in der Form von Knoten, welche verschiedene Code Blöcke abbilden. Hierbei sind die gerichteten Kanten einfache Sprünge von Codeblock zu Codeblock. Wenn ein Knoten ausschließlich Ausgangskanten besitzt ist dies ein Eingangscodeblock. Das heißt, dass dieser beispielsweise durch eine Nutzerinteraktion oder durch den Programmstart aufgerufen wird. Wenn ein Knoten keine Ausgangskante besitzt ist dies ein Endknoten. Wird dieser erreicht, beendet sich der Thread nach dem Verlassen des Codeblocks. [Owasp]

Beim Durchlaufen des Kontrollflussgraphen wird dieser darauf untersucht, ob beispielsweise eine endlose Schleife vorhanden ist. Wird ein Fehler dieser Art gefunden wird dieser gemeldet.

***Taint-*Analyse**

Die *Taint*-Analyse versucht herauszufinden, welche Variablen durch Userinput beeinflusst werden und in potenziell verwundbare Methoden gegeben werden. Sollte diese Variable vorher nicht auf ihre Ungefährlichkeit überprüft worden sein und trotzdem in eine solche Methode übergeben werden, wird diese als Verwundbarkeit eingestuft. [Owasp]

**Lexikalische Analyse**

Das Ziel der lexikalischen Analyse ist es den Sourcecode in einen sogenannten Tokenstrom umzuwandeln. Dieser Tokenstrom kann benutzt werden, um den Sourcecode auf Fehler zu überprüfen. Der Tokenstrom ist meist auch ein Zwischenschritt des Compilers, um den Sourcecode in Maschinensprache zu übersetzen. [Owasp]

### 3.1.2 Vor- und Nachteile der statischen Codeanalyse

Im Folgenden werden die verschiedenen Vor- und Nachteile der statischen Codeanalyse erörtert.

#### 3.1.2.1 Vorteile der statischen Codeanalyse

Die wichtigsten Vorteile der statischen Codeanalyse sind:

* Leicht einzupflegen
* Hohe Trefferrate bei einfachen Fehlern

[Owasp]

**Leicht einzupflegen:**

Es existieren für nahezu jede Sprache Tools der statischen Codeanalyse. Beispiele hierfür sind:

* Spotbugs im Java-Umfeld
* Resharper im .NET-Umfeld
* Cppcheck für C und C++

Diese Tools sind bereits so ausgereift, dass sie problemlos in jedes Entwicklungsprojekt der jeweiligen Umgebung eingepflegt werden können. Des Weiteren können diese Tools ohne weiteres in sogenannte *Nightly Builds* integriert werden. Ein Nightly Build ist ein Build, welcher nächtlich läuft und aus dem aktuellen Sourcecode Stand eine Programmversion baut. [Owasp]

Diese Einfachheit in der Einpflegung von Tools der statischen Codeanalyse ist ein klarer Vorteil für diese. Daher lohnt es sich solche Tools zu verwenden.

**Hohe Trefferrate bei einfachen Fehlern:**

Bei vielen Fehlertypen, wie zum Beispiel Endlosschleifen oder einer definitiven *NullPointerException,* weißen Tools der statischen Codeanalyse eine sehr hohe Trefferrate auf. Diese würden zwar auch schnell durch einen gewöhnlichen Entwickler erkannt werden, jedoch arbeitet die statische Codeanalyse um ein Vielfaches schneller als der gewöhnliche Entwickler. Des Weiteren kann diese ohne das Eingreifen eines Entwicklers den Code überprüfen und bei Bedarf den Entwickler auf einen Fehler aufmerksam machen.

Durch diese hohe Trefferrate bei einfachen Fehlern in Verbindung mit der schnellen und einfachen Verwendung der statischen Codeanalyse ist es zu empfehlen die statische Codeanalyse zu benutzen.

#### 3.1.2.2 Nachteile der statischen Codeanalyse

Zwei Nachteile der statischen Codeanalyse sind:

* Vielzahl von Falsch Positiven
* Falscher Sinn von Sicherheit

[Owasp] [Kumar 2015]

**Vielzahl von Falsch Positiven:**

Als erstes folgt eine kurze Erklärung, was ein Falsch Positiv im Rahmen der statischen Codeanalyse bedeutet. Ein Falsch Positiv ist ein vom Tool gemeldeter Fehler, welcher in Wahrheit gar kein Fehler ist oder spezifisch im Fall des Schreibens von Sourcecode ein Fehler, welcher durch eine Regel ausgelöst wurde, welche vom Entwickler als nicht relevant eingeschätzt wird. [Hicken 2016]

Vor allem der zuletzt genannte Fall von Falsch Positiven kann eine Vielzahl von „Fehlern“ erzeugen. Hierbei muss jedoch gesagt werden, dass viele Tools die Möglichkeit bieten bestimmte Regeln auszuschalten.

Ein weiterer Fall eines Falsch Positiven ist, wenn Objekte *disposed* werden müssen, um einen Speicherleck zu verhindern. Hierbei kann es passieren, dass Verweise auf dieses Objekt extern gespeichert werden, was von der statischen Codeanalyse nicht entdeckt werden kann. Dies führt wiederum zu einem Falsch Positiven.

Da die Stellen, an welchen Falsch Positive auftreten, einzeln deaktiviert werden müssen, kann es von Nachteil sein, Tools, welche den Code statisch analysieren, zu benutzen.

**Falscher Sinn von Sicherheit:**

Wenn statische Codeanalyse benutzt wird, kann es sein, dass Entwickler davon ausgehen, dass jedes Problem von aufgedeckt wird. Hierbei kann es also passieren, dass Code nicht mehr manuell auf Fehler überprüft wird. Somit kann eine Art Gottkomplex entstehen. Dies kommt vor allem an sicherheitskritischen Stellen zum Tragen.

Daher kann es, um dies zu verhindern, von Vorteil sein auf die statische Codeanalyse zu verzichten.

### 3.1.3 Fazit zur statischen Codeanalyse

Zur statischen Codeanalyse lässt sich abschließend sagen, dass ihre Verwendung dem Programmierer massive Vorteile erbringt. Gleichzeitig sind die Nachteile der statischen Codeanalyse entweder rein subjektiv oder können durch entsprechende Konfigurationen des Tools am Anfang der Verwendung entgegengewirkt werden.

Daher sollte aus diesem Kapitel mitgenommen werden, dass jedes größere Programmierprojekt ein Tool zur statischen Codeanalyse verwenden sollte.

## 3.2 Dynamische Codeanalyse

Das Ziel der dynamischen Codeanalyse ist es Fehler zu entdecken, während sich das Programm in Ausführung befindet. Die dynamische Codeanalyse läuft hierbei meist parallel zu den Tests ab. [Viva]

Diese Art von Analyse lässt sich in mehrere Schritte gliedern:

1. Vorbereitung der Eingangsdaten
2. Ausführen des Tests
3. Abfangen der zu analysierenden Daten
4. Analysieren der Daten

[Viva]

Hierbei spielt es keine Rolle, ob das Programm auf einem echten oder einem virtuellen Prozessor ausgeführt wird. [Viva]

Die dynamische Codeanalyse wird ausgeführt, indem Daten dem zu testenden Programm als Eingabe zur Verfügung gestellt werden. Daher ist die Qualität der eingegebenen Daten von hoher Bedeutung, da mit schlechten Daten auch schlechte Ergebnisse erzielt werden. Sind die Daten nicht gut genug können dementsprechend keine qualitativ hochwertigen Ergebnisse erzielt werden. [Viva]

Mittels der dynamischen Codeanalyse sein Programm zu testen ist dort am wichtigsten, wo die Zuverlässigkeit des Programms, Antwortzeit oder die benutzten Ressourcen von besonderer Bedeutung sind. [Viva]

Es gibt verschiedene Methoden mit welchen Tools der dynamischen Codeanalyse arbeiten. Diese umfassen:

1. Injektion von speziellem Code in den Source Code bevor der Source Code übersetzt wird. (*source code instrumentation*)
2. Injektion von speziellem Code direkt in die ausführbare Datei. (*object code instrumentation*)
3. Prüfungscode wird mittels vom Compiler gegebenen Switches injiziert. Diese Technik wird nicht von jedem Compiler unterstützt. (compilation stage instrumentation)
4. Live-Überprüfung des Programms über spezielle Laufzeit-Bibliotheken.

[Viva]

Typische Metriken, welche durch die dynamische Codeanalyse extrahiert werden, sind:

1. Benötigte Rechnerressourcen (CPU-Zeit/Arbeitsspeicher/usw.)
2. Die zyklomatische Komplexität
3. Programmfehler (Division durch null, Speicherlecks, Race Condition)
4. Verwundbarkeiten im Programm

[Viva]

Ein Beispiel für ein Tool der dynamischen Codeanalyse befindet sich im Debugger von Visual Studio 2017.

{Bild vom Debugger VS 2017}

Mithilfe dieses Debuggers können live die benötigte CPU-Auslastung und der benötigte Arbeitsspeicher ausgelesen werden.

### 3.2.1 Vor- und Nachteile der dynamischen Codeanalyse

In diesem Kapitel werden die Vorteile und die Nachteile der dynamischen Codeanalyse erörtert.

#### 3.2.1.1 Vorteile der dynamischen Codeanalyse

Einige wichtige Vorteile der dynamischen Codeanalyse sind:

1. Kaum falsch Positive
2. Keine Notwendigkeit von Sourcecodes zum Testen

[Viva]

**Kaum falsch Positive:**

Einer der Nachteile der statischen Codeanalyse war es, dass durchaus viele Falsch Positive entstehen können. Die dynamische Codeanalyse hat den Vorteil, dass Fehler, welche entstehen auch tatsächlich Fehler sind, da vom Code durch die Eingabe der Testdaten ein Fehler erzeugt wurde. Somit ist der Fehler durch sein Auftreten real und keine reine Projektion von den Regeln eines Analysetools. [Viva]

**Keine Notwendigkeit von Sourcecodes zum Testen:**

Da bei der dynamischen Codeanalyse das Programm als solches getestet wird, benötigt man zur Analyse ausschließlich eben dieses Programm. Der dynamischen Codeanalyse ist es somit egal, wie der Sourcecode dahinter aussieht.

Dies hat den Vorteil, dass die Analyse nicht von den Entwicklern selbst durchgeführt werden muss. Diese Personen können beispielsweise speziell auf das Erstellen von Daten für eine solche Analyse trainiert sein. Die Ergebnisse können die Analysten dann an die Entwickler zurückgeben, damit eventuelle Fehler ausgebaut werden können.

#### 3.2.1.2 Nachteile der dynamischen Codeanalyse

Nachteile der dynamischen Codeanalyse sind:

1. Erkennung von Fehlern ausschließlich am von den Daten vorgegebenen Pfad
2. Hoher Ressourcenverbrauch

[Viva]

**Erkennung von Fehlern ausschließlich am von den Daten vorgegebenen Pfad:**

Da bei der dynamischen Codeanalyse das Programm mit gewissen Daten durchlaufen wird, kann, sobald das Programm auch nur eine Alternativroute besitzt, es in einem Lauf nicht vollständig analysiert werden.

Daher müssen, um das Programm vollständig zu analysieren, mehrere Datensätze erstellt werden. Die Menge der benötigten Daten ist hierbei relativ zur Höhe der zyklomatischen Komplexität.

**Hoher Ressourcenverbrauch:**

Die Überprüfung von Laufzeitressourcen eines Programms ist eine Zeitintensive Herausforderung. {Begründung}

Da bei der dynamischen Codeanalyse mehrmals mit verschiedenen Daten das Programm durchlaufen werden muss, um alle Zweige analysieren zu können, wird das vorher genannte Problem weiter verschlimmert.

### 3.2.2 Fazit zur dynamischen Codeanalyse

Abschließend lässt sich zur dynamischen Codeanalyse sagen, dass auch sie neben der statischen Codeanalyse gute und wichtige Vorteile erbringt, welche auch den Nachteilen der statischen Codeanalyse entgegenwirkt. Die Nachteile sind im Gegensatz zu den Nachteilen der statischen Codeanalyse jedoch etwas schwerwiegender. Der hohe Ressourcenaufwand und die Menge an Vorarbeit, welche zu tun sind, sind schlichtweg nicht zu unterschätzen.

Letztendlich ist die dynamische Codeanalyse eine gute Ergänzung zur statischen Codeanalyse.

## 3.3 Vergleich zwischen statischer und dynamischer Codeanalyse

In diesem Abschnitt werden die statische und die dynamische Codeanalyse gegenübergestellt.

|  |  |  |
| --- | --- | --- |
| Aspekte | Statische Codeanalyse | Dynamische Codeanalyse |
| Ausführungszeitpunkt | Designtime | Runtime |
| Rate an Falsch Positive | Hoch | Keine |
| Fehlererkennungsrate | 85% [Brain 2010] | 85% [Brain 2010] |
| Testverfahren | White-Box | Black-Box oder White-Box |
| Anzahl an benötigten Durchläufen | 1 | Relativ zur zyklomatischen Komplexität |
| Überprüfungsteil | Komplettes Programm | Nur der von den Daten vorgegebene Pfad |
| Ressourcenaufwand | Gering | Hoch |

Abschließend noch eine kleine Allegorie, um den Unterschied zwischen statischer und dynamischer Analyse klarzustellen.

Man stelle sich vor man ist in einer Küche und möchte etwas kochen. Dies lässt sich grob in zwei Phasen aufteilen. Erstens die Phase, in welcher das Rezept durchgelesen, analysiert und verstanden wird und zweitens die Phase, in der man tatsächlich kocht. In der ersten Phase analysiert man das Rezept statisch. Man überlegt, ob das Rezept mit den mir zur Verfügung stehenden Mitteln möglich ist. In beziehungsweise nach der zweiten Phase kann das Ergebnis des Kochens dynamisch analysiert werden und eventuell aufgetretene Fehler aufgeschrieben werden. Mit Hilfe der nun bekannten Fehler kann das Rezept angepasst werden, damit diese Fehler in Zukunft nicht mehr auftreten werden. [Dietrich]

# 4. Java Virtual Machine

Da das Ergebnis dieser Arbeit ein Plugin für ein statisches Analysetool, welches die Programmiersprache Java analysiert, sein soll und die Basis des ausgewählten Tools der zugrundeliegende Java Bytecode ist, werden dieser und allgemein die gesamte Java Virtual Machine, mit ihren Einzelteilen den OpCodes, dem Instruction Set und dem Stack, im Folgenden Kapitel genauer betrachtet.

Zunächst folgt ein grober Überblick über die Java Virtual Machine, auch JVM genannt.

Die JVM ist eine Laufzeitumgebung die ursprünglich für die Ausführung von Programmen der Programmiersprache Java entwickelt wurde. Hierfür wird Java-Code in einen Bytecode übersetzt, welcher im Anschluss von der JVM ausgeführt werden kann.

Durch die JVM, welche Betriebssystem- beziehungsweise Hardwareschnittstellen abstrahiert, entstehen Sicherheitsvorteile, welche dafür sorgen, dass Programme innerhalb der JVM nicht in Hauptspeicherbereiche gelangt, auf welche das jeweilige Programm keinen Zugriff haben sollte.

Die JVM benutzt einen sogenannten JIT-Compiler (*Just-In-Time Compiler*), um den Bytecode zur Ausführungszeit in Maschinencode, welcher von dem Computer ausgeführt werden kann, zu übersetzen. Dieser übersetzte Maschinencode bleibt für die Ausführungszeit des Programms in der JVM erhalten, wodurch der betroffene Bytecode ab der zweiten Ausführung sehr schnell ausgeführt werden kann. Dies hat den Vorteil, dass das Kompilieren des Programms schneller erfolgt, als wenn der Code direkt in Maschinencode übersetzt wird. Gleichzeitig liegt der Code bereits in einem für die Maschine leicht verdaulichem Zustand vor, weswegen die Laufzeit Performanz besser ist als die Laufzeit Performanz von Interpretierten Sprachen.

Des Weiteren ermöglicht es die JVM eine gewisse Plattformunabhängigkeit herzustellen. Diese entsteht dadurch, dass die JVM, wie bereits genannt, die Betriebssystemschnittstellen abstrahiert und vereinheitlicht. Hierdurch ist es ausschließlich von Nöten, dass die JVM auf dem Zielsystem installiert ist. Somit muss sich nur um die Entwicklung der JVM für das jeweilige Betriebssystem gekümmert werden. Diese Entwicklung wurde anfangs von Sun Microsystems durchgeführt. Nachdem Sun Microsystems im Jahre 2010 von Oracle aufgekauft wurde, wurde die Weiterentwicklung der JVM von Oracle übernommen. Somit muss sich der Anwendungsentwickler nicht um die Plattformunabhängigkeit seines Programms kümmern, da dies bereits von der JVM übernommen wird. Dies kann sehr viel Zeit einsparen.

Die Eigenschaft, dass die JVM einen standardisierten Bytecode verwendet, wurde sich von verschiedenen anderen Programmiersprachen zu Nutze gemacht. Dies geschieht, indem der Code von diesen Programmiersprachen ebenfalls in diesen standardisierten Bytecode kompiliert wird.

Beispiele für solche Programmiersprachen sind:

* Kotlin
* Groovy
* Scala

## 4.1 Bytecode

Eine knappe Erklärung für Bytecode lautet: „Bytecode bezeichnet einen Zwischencode, der bei der Compilierung[sic!] einer Programmiersprache entsteht. Damit wird der Quelltext eines Programmes nicht direkt in eine maschinenlesbare Form transformiert. Der Bytecode ist somit maschinenunabhängig und es entsteht eine kompaktere Darstellung vom ursprünglichen Quelltext. Die endgültige Ausführung des Bytecodes wird dann durch eine von der Rechnerplattform abhängige virtuelle Maschine realisiert.“ [ITWissen]

Die ersten Ideen für eine Art Zwischencode zwischen Sourcecode und Maschinencode entstanden bereits in den 60er und 70er Jahren. Man wollte es schaffen, dass Code unabhängig vom Zielsystem kompiliert werden kann und dass diese Programme auch zwischen den verschiedenen Zielsystemen ausgetauscht werden können. Ein erster Vorläufer, welcher dieses Konzept umgesetzt hat, ist die Programmiersprache Basic Combined Programming Language (BCPL). Diese Programmiersprache konnte sich jedoch nicht durrchsetzen, da zu ihrer Zeit die Computer noch nicht schnell genug waren, um diesen Ansatz zu benutzen. [ITWissen]

Um den Bytecode auszuführen wird der Bytecode interpretiert und in einer virtuellen Maschine emuliert. Durch diese Emulation können Performanceeinbrüche entstehen. Dies ist einer der größten Nachteile von bytecodebasierten Programmiersprachen. Um dies zu umgehen wird der JIT-Compiler verwendet, welcher im vorigen Unterkapitel bereits erwähnt wurde.

Beispiele für Bytecodes sind:

* Java-Bytecode
* Common Intermediate Langugage (C#, VB)

Ein Vorteil von bytecodebasierten Programmiersprachen ist es, dass durch die Emulation, in welcher der Code ausgeführt wird, eine deutlich höhere Kontrolle über diesen herrscht. Dieser Vorteil kommt vor allem bei dem bereits genannten Sicherheitsvorteil zu Gute. Ebenso lässt sich das Programm während der Entwicklung einfacher debuggen.

## 4.2 Primitive Typen und Werte der JVM

Die Gruppe der primitiven Typen, welche von der JVM unterstützt werden, lassen sich in drei Untergruppen zusammenfassen.

1. Numerische Typen
2. Boolescher Wert
3. *returnAdress*

[Oracle Java Spec]

Die numerischen Typen lassen sich weiter in zwei Untergruppen einteilen:

1. Integrale Typen
2. Gleitkommatypen

[Oracle Java Spec]

Zur Abbildung von Integralen Typen existieren fünf verschiedene primitive Typen, welche alle einen korrespondierenden Sprachtypen in der Programmiersprache Java besitzen. [Oracle Java Spec]

Diese sind:

* *byte:* Kann Werte zwischen -27 und 27 -1 abbilden. Die Abbildung erfolgt mittels Zweierkomplement. Der Standardwert für diesen Typen ist 0.
* *short:* Kann Werte zwischen -215 und 215 -1 abbilden. Die Abbildung erfolgt mittels Zweierkomplement. Der Standardwert für diesen Typen ist 0.
* *int:* Kann Werte zwischen -231 und 231 -1 abbilden. Die Abbildung erfolgt mittels Zweierkomplement. Der Standardwert für diesen Typen ist 0.
* *long:* Kann Werte zwischen -263 und 263 -1 abbilden. Die Abbildung erfolgt mittels Zweierkomplement. Der Standardwert für diesen Typen ist 0.
* *char:* Kann Werte zwischen 0und 216 -1 abbilden. Die Werte repräsentieren einen Unicode Code Punkt im sogenannten Basic Multilingual Plane, welcher in UTF-16 codiert ist. Der Standardwert für diesen Typen ist ‚\u0000‘, welcher den null Code Punkt wiederspiegelt.

(Alle Wertangaben mit „zwischen“ sind inklusive der Ränder) [Oracle Java Spec]

Zur Abbildung von Gleitkommazahlen existieren innerhalb der JVM zwei primitive Typen. Diese besitzen jeweils einen korrespondierenden Sprachtypen in der Programmiersprache Java. [Oracle Java Spec]

Diese sind:

* *float*: Kann Werte gemäß der IEEE-754 mit 32 Bit abspeichern. Der Standardwert für diesen Typen ist die positive 0.
* *double*: Kann Werte gemäß der IEEE-754 mit 64 Bit abspeichern. Der Standardwert für diesen Typen ist die positive 0.

[Oracle Java Spec]

Der Boolesche Wert bildet einen einfachen *true* oder *false* Wert ab. Hierbei ist jedoch zu erwähnen, dass es in JVM keine direkte Abbildung eines Booleschen Wertes gibt. Jeder Boolesche Wert wird in den numerischen Typ *int* übersetzt. *Arrays* von Booleschen Werten werden von der JVM jedoch direkt unterstützt und unter Haube mit den Befehlen für ein *bytearray* angesprochen und modifiziert. [Oracle Java Spec] Im Quellcode gibt es den korrespondierenden Sprachtypen *boolean*, mit welchem ein Boolescher Wert erzeugt werden kann.

Der *returnAdress* Typ ist ein Zeiger auf den jeweiligen OpCode eines Befehls innerhalb der JVM. Er wird beispielsweise verwendet, um die Stelle des Methodenaufrufs zu speichern. Dieser Typ besitzt innerhalb der Programmiersprache Java keinen korrespondierenden Sprachtypen. [Oracle Java Spec]

## 4.3 Referenztypen und Werte

Innerhalb der JVM gibt es drei verschiedene Arten von Referenztypen:

* Klassentypen
* *Array-*Typen
* Interfacetypen

[Oracle Java Spec]

Diese verschiedenen Typen referenzieren auf eine dynamisch generierte Klasseninstanz, ein Feld oder eine Klasseninstanz oder Feld, welches ein Interface implementiert. [Oracle Java Spec]

Eine Referenz kann die spezielle *null-*Referenz sein. Dies bedeutet, dass kein Objekt über die Referenz referenziert wird. Diese Referenz hat keinen Typen zur Laufzeit und kann somit auf jeden Typen gecastet werden. Der Standardwert für eine Referenz ist die Referenz auf *null.* [Oracle Java Spec]

## 4.4 JVM OpCodes

Ein Java OpCode ist ein Teil eines Stückes Bytecode.

Ein Opcode ist eine Beschreibung, wie die folgenden Operanden zu bewerten sind. Diese OpCodes können in verschiedene Kategorien eingeteilt werden:

* Laden und Speichern (z.B. aload\_0, istore)
* Arithmetik und Logik (z.B. iadd)
* Typenwandlung (z.B. i2b)
* Objekterstellung und -manipulation (z.B. new)
* Operandenstackmanagement (z.B. swap)
* Kontrollentransfer (z.B. goto)
* Methodenaufruf und Rückgabe (z.B. invokespecial, return)

[Wikipedia Java Bytecode]

Des Weiteren gibt es für eine Vielzahl von Befehlen Prä- beziehungsweise Suffixe. Diese sind:

|  |  |
| --- | --- |
| Präfix/Suffix | Operanden Typ |
| i | integer |
| l | long |
| s | short |
| b | byte |
| c | character |
| f | float |
| d | double |
| a | reference |

[Wikipedia Java Bytecode]

Eine komplette Liste aller in der JVM enthaltenen OpCodes ist unter: <https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings> zu finden.

Einige für diese Arbeit wichtige OpCodes werden im Folgenden hervorgehoben.

Zu diesen Befehlen zählen:

* Alle Return OpCodes
* OpCodes, um Methoden aufzurufen
* OpCodes, um Felder zu bearbeiten

**Return OpCodes:**

Zu den verschiedenen OpCodes, welche ein *return* enthalten, zählen:

* RETURN
* ARETURN
* IRETURN
* DRETURN
* FRETURN
* LRETURN

Die Befehle definieren einen Absprungpunkt in einer Methode. Das Präfix (A/I/D/F/L) zeigen an, was von der Methode zurückgegeben wird. Der zugehörige Typ kann der vorigen Tabelle entnommen werden.

**OpCodes, um Methoden aufzurufen:**

Die OpCodes, um Methodenaufrufe zu erkennen, lauten:

* INVOKEVIRTUAL
* INVOKESPECIAL

Diese OpCodes zeigen an, dass eine Methode an dieser Stelle aufgerufen wird. INVOKESPECIAL verweist hierbei auf besondere Methoden, wie zum Beispiel einen Konstruktoraufruf.

**OpCodes, um Felder zu bearbeiten:**

Wenn mit Feldern interagiert wird, werden im Bytecode folgende OpCodes erzeugt.

* GETFIELD
* PUTFIELD

Die OpCodes zeigen an, ob die Daten eines Feldes geholt werden (Präfix GET) oder, ob Daten in ein Feld geschrieben werden (Präfix PUT).

## 4.5 JVM Instruction Set

Bei dem Instruction Set der JVM handelt es sich um den Java Bytecode. Jedes Stück Bytecode beinhaltet einen OpCode und die Operanden, welche beliebig viele sein können, zu dem OpCode.

Um das Ganze zu veranschaulichen wird im Folgenden eine Methode mit ihrem zugehörigen Bytecode erstellt.

**public void** bytecode() {  
 outer:  
 **for** (**int** i = 2; i < 1000; i++) {  
 **for** (**int** j = 2; j < i; j++) {  
 **if** (i % j == 0)  
 **continue** outer;  
 }  
 System.***out***.println (i);  
 }  
}

[Wikipedia Java Bytecode]

Diese Methode verwendet einige der zur Verfügung stehenden OpCodes und ist somit gut geeignet die Übersetzung von Quellcode zu Bytecode zu zeigen.

Der zugehörige Bytecode lautet wie folgt:

0: iconst\_2

1: istore\_1

2: iload\_1

3: sipush 1000

6: if\_icmpge 44

9: iconst\_2

10: istore\_2

11: iload\_2

12: iload\_1

13: if\_icmpge 31

16: iload\_1

17: iload\_2

18: irem

19: ifne 25

22: goto 38

25: iinc 2, 1

28: goto 11

31: getstatic #84;//Field java/lang/System.out:Ljava/io/PrintStream;

34: iload\_1

35: invokevirtual #85;//Method java/io/PrintStream.println:(I)V

38: iinc 1, 1

41: goto 2

44: return

[Wikipedia Java Bytecode]

Die Zahlen, welche vor dem jeweiligen OpCode stehen, stehen für das Offset im Verhältnis zum Methodenstart. Wie groß der Unterschied zwischen den einzelnen Zahlen ist, hängt davon ab, wie viele Operanden der jeweilige Befehl mit sich trägt. Der *goto* Befehl trägt beispielsweise einen Operanden mit sich. Dieser Operand steht für den Befehl zu welchem gesprungen werden soll. Somit werden durch Befehl und OpCode insgesamt zwei Plätze belegt. Daher kann der nächste Befehl erst bei n+2 anfangen.

## 4.6 JVM Stack

Jeder Thread innerhalb der JVM besitzt seinen eigenen privaten *Stack*. Dieser wird mit der Threaderstellung erzeugt. Dieser Stack beinhaltet Frames, welche im nachfolgenden Kapitel erläutert werden. Man kann den Stack der JVM mit dem Stack der Programmiersprache C vergleichen. Er hält verschiedene lokale Variablen und temporäre Resultate. Des Weiteren spielt er eine wichtige Rolle im Aufruf von Methoden und deren Ende. Da der Stack nicht durch den Programmierer direkt bearbeitet werden kann, können die Frames auf dem allgemeinen Speicher (*Heap*) abgelegt werden. [Oracle Java Spec]

Die Spezifikation der Java Virtual Machine erlaubt es, dass die Stackgröße entweder dynamisch erweitert oder komprimiert werden kann oder die Stackgröße einen bei der Initialisierung des Threads festgelegte Größe besitzt.

Es gibt zwei Ausnahmebedingungen, welche mit den Stacks der Java Virtual Machine zusammenhängen:

* Wenn die Berechnung eines Threads einen größeren Stack benötigt als ihm erlaubt ist, wird ein *StackOverflowError* geworfen.
* Wenn der Stack dynamisch erweitert werden kann und dies versucht wird allerdings nicht genug Arbeitsspeicher zur Verfügung steht oder nicht genug Arbeitsspeicher für die Erstellung eines neuen Thread vorhanden ist, wird ein *OutOfMemoryError* geworfen.

[Oracle Java Spec]

## 4.7 JVM Frames

Ein Frame wird benutzt um Daten und Teilergebnisse zwischenzuspeichern, Fehler zu werfen und Rückgabewerte von Methoden handzuhaben. [Oracle Java Spec]

Ein neuer Frame wird jedes Mal erstellt, wenn eine neue Methode aufgerufen wird. Wenn diese Methode beendet wird, wird das Frame zerstört. Hierbei ist es egal, ob das Beenden über einen standardmäßigen Weg durchgeführt wurde oder, ob es durch einen anderen Wege beendet wurde. Das Frame wird mittels dem JVM Stack des jeweiligen Threads erstellt. Jedes Frame besitzt sein eigenes Feld an lokalen Variablen, seinen eigenen Operanden Stack und eine Referenz zur Klasse der aktuellen Methode des Konstanten Pools. [Oracle Java Spec]

Die jeweilige Größe des Feldes und des Operanden Stacks werden zum Zeitpunkt der Übersetzung festgelegt. Die Datenstruktur eines Frames ist von der jeweiligen Implementation der Java Virtual Machine abhängig. Der Speicher für ein Frame wird während des Methodenaufrufs allokiert. [Oracle Java Spec]

## 4.8 Schlussfolgerung

Abschließend lässt sich sagen, dass die Java Virtual Machine einige Vorteile, welche sowohl den Anwender als auch den Entwickler betreffen, mit sich bringt.

Mit ihrer Struktur und der Einfachheit diese zu analysieren bietet sie sich hervorragend, um diese statisch zu analysieren.

Des Weiteren wurde in diesem Kapitel der Bytecode angesprochen. Dieser kann als einer Art Zwischenform zwischen Maschinen- und Quellcode gesehen werden. Über diesen wird ein Kompromiss zwischen Geschwindigkeit bei der Ausführung und Geschwindigkeit während des Übersetzens geformt. Ebenso bietet auch der Bytecode massive Vorteile beim Debuggen von Code.

# 5. Spotbugs

SpotBugs ist ein Programm, welches mittels statischer Codeanalyse, nach Bugs in Java Code sucht. [Github SpotBugs]

Standardmäßig sucht SpotBugs nach über 400 verschiedenen Bugmustern. SpotBugs lässt sich in mehrere Entwicklungswerkzeuge integrieren. [Github Spotbugs] Zu diesen zählen:

* Maven
* Gradle
* Eclipse
* Ant

[Github Spotbugs]

Des Weiteren ist es möglich selbst Plugins für SpotBugs zu schreiben, um spezielle auf den eigenen Code zugeschnittene Bugpattern zu finden. Diese Möglichkeit wird in dieser Arbeit ausgenutzt. [SpotBugs Docs]

In diesem Kapitel wird zunächst die Geschichte von SpotBugs besprochen, im Anschluss daran wird kurz die Funktionsweise von SpotBugs erklärt, danach wird SpotBugs aus der Anwendersicht gezeigt und zuletzt wird SpotBugs aus der Entwicklersicht beschrieben. Hierbei ist die Erweiterungsmöglichkeit von SpotBugs gemeint.

## 5.1 Geschichte

SpotBugs entsprang aus dem ehemaligen FindBugs. FindBugs wurde ursprünglich von Bill Pugh und David Hovemeyer an der *University of Maryland* entwickelt. Die erste Vollversion von FindBugs wurde am 10. Juni 2006 veröffentlicht. Es wurde mittels der GNU Lesser General Public License (LGPL) lizenziert. [Wikipedia Findbugs]

Nachdem der zuständige Instandhalter Bill Pugh sich nicht mehr um das Projekt gekümmert hat, wurde von Andrey Loskutov und seinem Team das Projekt SpotBugs ins Leben gerufen. Die neueste Version, welche von FindBugs existiert, ist die Version 3.0.1 vom 6. März 2015. SpotBugs sollte einen Nachfolger zu FindBugs darstellen. Die erste Version des Nachfolgers SpotBugs ist mit der Version 3.1.0 vom Oktober 2017 in Betrieb gegangen. [Wikipedia FindBugs] Die aktuelle Version von SpotBugs ist die Version 4.0.1.

## 5.2 SpotBugs funktionsweise

SpotBugs ist ein Tool zur statischen Analyse von Code für die Java Virtual Machine. SpotBugs untersucht hierbei ausschließlich den Bytecode, welchen es Schritt für Schritt durchgeht. Dies bedeutet, dass SpotBugs nicht wissen muss, wie der Code in seiner ursprünglichen Form ausgesehen hat. Gleichzeitig muss das Programm nicht ausgeführt werden, um die Analyse durchführen zu können.

## 5.2 SpotBugs aus Anwendersicht

In diesem Unterkapitel wird SpotBugs aus der Sicht des Anwenders beschrieben.

Wie bereits erwähnt, lässt sich SpotBugs in mehreren Entwicklungsabläufen integrieren.

**Eclipse:**

Um SpotBugs in Eclipse zu integrieren, kann der Eclipse Marketplace verwendet werden. Dieser kann unter dem Eintrag **Help** 🡪 **Eclipse Marketplace…** geöffnet werden. In der Suchleiste kann nach „SpotBugs“ gesucht werden. Mit dem Button *Install* kann das Plugin installiert werden. SpotBugs kann nun auf jedem im Workspace enthaltenen Projekt ausgeführt werden.

Um eine Fehlerdurchsuchung mittels SpotBugs zu starten, muss das Kontextmenü auf einem Projekt geöffnet werden. Im Kontextmenü gibt es den Eintrag **SpotBugs** 🡪 **Find Bugs**. Wird dieser Eintrag angeklickt, wird der gesamte Code nach den entsprechenden Fehlertypen durchsucht. Diese Funktion muss nicht für das gesamte Projekt ausgeführt werden. Der Kontextmenüeintrag steht auch für einzelne Dateien beziehungsweise Packages zur Verfügung.

**Maven:**

Um SpotBugs in Maven zu integrieren, muss das SpotBugs Plugin in der *pom.xml* eingetragen werden. Hierzu muss im <plugins> Abschnitt folgendes hinzugefügt werden.

<plugin>

<groupId>com.github.spotbugs</groupId>

<artifactId>spotbugs-maven-plugin</artifactId>

<version>4.0.0</version>

<dependencies>

<dependency>

<groupId>com.github.spotbugs</groupId>

<artifactId>spotbugs</artifactId>

<version>4.0.1</version>

</dependency>

</dependencies>

</plugin>

Nun kann bei einem sogenannten *maven build* das Ziel einen SpotBugs Durchlauf durchzuführen definiert werden. Dies wird mit dem gleichnamig benannten *goal* spotbugs erreicht. Wenn dieses Ziel definiert wurde, wird, während des Builds, der gesamte Code mit SpotBugs nach Fehlern durchsucht. Falls hierbei Fehler entstehen, werden diese in der Konsole, in welcher der *maven build* durchgeführt wurde, angezeigt. Genaueres zu diesem Ziel kann unter <https://spotbugs.github.io/spotbugs-maven-plugin/spotbugs-mojo.html> nachgelesen werden.

**Gradle:**

Die Integration in Gradle ist deutlich kürzer als die Integration in maven. In der *build.gradle* Datei muss hierzu folgendes Stück Code hinzugefügt werden.

plugins {

id "com.github.spotbugs" version "4.0.5"

}

Falls der „plugins“ Bereich bereits existiert kann „id "com.github.spotbugs" version "4.0.5"“ diesem hinzugefügt werden.

Dies ist alles, was für die Integration von SpotBugs in Gradle geschehen muss. Ab dem nächsten Gradle-Build wird eine Analyse mittels SpotBugs durchgeführt und die Fehler ausgegeben.

**Ant:**

Die integration von SpotBugs in Ant erfolgt in mehreren Schritten.

Zunächst muss der Ant Installation im „lib“ Ordner folgendes hinzugefügt werden:

$SPOTBUGS\_HOME/lib/spotbugs-ant.jar

Als nächstes muss eine Task Definition erstellt werden. Diese sollte wie folgt aussehen:

<taskdef

resource="edu/umd/cs/findbugs/anttask/tasks.properties"

classpath="path/to/spotbugs-ant.jar" />

Als letztes muss der *build.xml* ein Stück Code hinzugefügt werden, welches SpotBugs aktiviert werden soll.

<property name="spotbugs.home" value="/export/home/daveho/work/spotbugs" />

<target name="spotbugs" depends="jar">

<spotbugs home="${spotbugs.home}"

output="xml"

outputFile="bcel-sb.xml" >

<auxClasspath path="${basedir}/lib/Regex.jar" />

<sourcePath path="${basedir}/src/java" />

<class location="${basedir}/bin/bcel.jar" />

</spotbugs>

</target>

Hierbei muss erwähnt werden, dass die Variable „spotbugs.home“ auf das Installationsverzeichnis von SpotBugs gesetzt werden muss.

SpotBugs analysiert nun die in der Datei „bcel.jar“ enthaltenen Klassen nach Fehlern. Diese Datei ist die Standard Ausgabedatei vom Ant-Buildscript.

Wenn alles korrekt eingefügt wurde, kann in der Konsole, in welcher das Ant-Buildscript ausgeführt wurde, ein Abschnitt von SpotBugs entdeckt werden. Dieser Abschnitt sollte wie folgt aussehen:

spotbugs:

[spotbugs] Running SpotBugs...

[spotbugs] Bugs were found

[spotbugs] Output saved to bcel-sb.xml

Wenn Fehler mittels SpotBugs entdeckt wurden, werden diese in der Datei „bcel-sb.xml“ ausgegeben.

**Beispiel:**

In Folgendem Beispiel ist in Zeile 3 ein einfacher Fehler schnell entdeckt. In jedem Fall wird der Code an dieser Stelle eine *NullPointerException* erzeugen.

**public** **static** **void** main(String[] args) {

Object object = **null**;

**if** (object.equals(args))

{

System.***out***.println("aasda");

}

}

SpotBugs zeigt diesen Fehler in Eclipse wie folgt an:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Description | Resource | Path | Location | Type |
| Null pointer dereference of object in test.Main.main(String[]) [Scary(5), High confidence] | Main.java | /src/main | line 3 | SpotBugs Problem (Scary) |

Man kann an dieser Fehlermeldung schnell sehen, wo sich der Fehler befindet. Des Weiteren teilt SpotBugs auch mit, wie gefährlich der Bug ist (z.B. Scary, für einen sehr gefährlichen Fehler) und, wie sicher sich SpotBugs ist, dass der Fehler auch tatsächlich ein Fehler ist (z.B. High Confidence, für eine sehr große Sicherheit, dass der Fehler ein Fehler ist).

Abschließend lässt sich als Anwender von SpotBugs sagen, dass es als statisches Codeanalysetool massive Vorteile während der Entwicklung bringen kann. Des Weiteren wird einem die Integration von SpotBugs in den Laufenden Entwicklungsprozess, durch die verschiedenen Integrationsmöglichkeiten in den beliebtesten Build-Tools und einer weit verbreiteten Entwicklungsumgebung in Eclipse, sehr einfach gemacht.

## 5.3 SpotBugs aus Entwicklersicht

Nun kommen wir für den aus der Sicht dieser Arbeit wichtigeren Teil. Da es nicht möglich ist für jedes spezifische Problem ein allgemeines Fehlermuster zu erstellen, auf welches abgefragt werden kann, hat man mit SpotBugs die Möglichkeit eigene Fehlermuster zu erstellen, welche in SpotBugs injiziert werden können.

Anwendungsszenarien für solche Plugins können sein:

* Spezielle Abhängigkeiten von eigens erstellten Klassen, welche berücksichtigt werden sollten.
* Fehlertypen, welche (noch) nicht in SpotBugs integriert sind.
* Usw.

Die Injektion von neuen Fehlermustern findet in SpotBugs über sogenannte Plugins statt. Für die Entwicklung solcher Plugins bietet SpotBugs entsprechende Bibliotheken an. Des Weiteren wird auch eine vorbereitete Entwicklungsvorlage, in welcher eine der benötigten Methoden bereits angesprochen wird, angeboten.

Im Folgenden Abschnitt wird erklärt, wie ein Plugin für SpotBugs geschrieben werden kann. Im Anschluss daran wird aufgezeigt, wie ein solches Plugin in SpotBugs eingespielt werden kann.

Wenn Maven auf der lokalen Maschine vorhanden ist, kann mittels des folgenden Befehles ein beispielhaftes Plugin erzeugt werden:

mvn archetype:generate

-DarchetypeArtifactId=spotbugs-archetype

-DarchetypeGroupId=com.github.spotbugs

-DarchetypeVersion=0.2.3

Maven verlangt hierbei eine groupId, eine artifactId, eine Versionsnummer und einen Namen für das erste Package.

**Projektstruktur:**

Die Projektstruktur besteht aus den Maven Standardordnern *main* und *test*.

Im Ordner *main* befinden sich sowohl ein Ordner namens *resources*, in welchem verschiedene XML-Dateien, auf welche später eingegangen wird, enthalten, als auch das Package, dessen Name im vorigen Schritt angegeben wurde. In diesem Package befindet sich die Klasse *MyDetector.* In dieser Klasse findet die Fehlermustererkennung statt. In der Standardversion wird nach allen *System.out* Anweisungen gesucht. Wenn eine solche Anweisung gefunden wurde, wird an dieser Stelle ein Bug angezeigt. Behoben wäre dieser Fehler, indem alle *System.out* Anweisungen mit *System.err* ersetzt werden.

Nun zurück zum Ordner *resources*. In diesem befinden sich insgesamt zwei XML-Dateien.

* *findbugs.xml*
* *messages.xml*

In der *findbugs.xml* wird angegeben, wie die Detektorklasse heißt und nach welchem Bug sie sucht. Hier können auch mehrere solcher Detektorklassen angegeben werden. Des Weiteren werden die verschiedenen von dem Plugin gesuchten Bugpattern initialisiert. Das heißt es wird eine Abkürzung für dieses Bugpattern festgelegt, es wird ein Typ (oder Name des Bugpatterns) festgelegt und es wird die Kategorie des Bugpatterns festgelegt (z.B. Korrektheit des Codes). Wie auch bei den Detektoren können mehrere Bugpattern initialisiert werden.

In der *messages.xml* wird zunächst eine allgemeine und eine kurze Beschreibung des Plugins angegeben. Dies wird mit dem XML-Tag *<plugin>* begonnen. Des Weiteren werden sowohl die einzelnen Detektoren und die verschiedenen Bugpatterns genauer beschrieben. Für die Detektoren wird der XML-Tag *<Detector>* und der darunter liegende XML-Tag <Details> verwendet. Für das Bugpattern wird der XML-Tag *<BugPattern>* verwendet. Für die Beschreibung wird einerseits der XML-Tag *<ShortDescription>* andererseits der XML-Tag *<LongDescription>* verwendet. Als letztes wird ein Bugcode anhand der Abkürzung des Bugpatterns vergeben.

Im Ordner *test* befindet sich ausschließlich das am Anfang benannte Package. In diesem Package sind standardmäßig drei Dateien enthalten.

* *BadCase.java*
* *GoodCase.java*
* *MyDetectorTest.java*

Die Datei *BadCase.java* beinhaltet das Fehlermuster, welches von dem zu entwickelnden Plugin entdeckt werden soll.

Die Datei *GoodCase.java* beinhaltet Java Code, welcher explizit nicht dem Fehlermuster entspricht. Bei der Erstellung dieses Java Codes sollte darauf geachtet werden, dass entweder der korrigierte Code aus der *BadCase.java* Datei enthalten ist oder ein Muster, welches dem zu findenden Fehlermuster ähnlich ist, jedoch explizit kein Fehler ist.

In der Datei *MyDetectorTest.java* werden die einzelnen Dateien durch das in der Klasse *MyDetector* festgelegten Fehlermuster gejagt. Wenn in der *BadCase.java* genau ein Fehler und in der GoodCase.java genau kein Fehler gefunden wurde laufen beide Tests auf grün.

**Entwickeln des Plugins:**

Um nun selbst ein eigenes Fehlermuster, nach welchem Code durchsucht werden soll, zu schreiben muss die Klasse *MyDetector* angepasst werden. Der Name kann angepasst werden. In diesem Beispiel wird jedoch der Name *MyDetector* weiterhin verwendet.

Zunächst sollte hierbei erwähnt werden, dass die Klasse *MyDetector* von der abstrakten Klasse *OpcodeStackDetector* erbt. Über diese werden verschiedene Methoden zur Verfügung gestellt, welche benutzt werden können, um den Code zu durchsuchen. Des Weiteren wird von SpotBugs das Visitor Pattern verwendet. Die Klasse *MyDetector* bildet hierbei den Visitor im Pattern.

Die Methode, welche bereits in der Starterversion benutzt wird, ist die Methode *sawOpCode(int)*. Diese Methode wird jedes Mal aufgerufen, wenn ein OpCode gefunden wird. Der Interger, welcher übergeben wird, ist die Zahl, welche dem OpCode zugewiesen ist. Auf diese OpCode kann nun entsprechend reagiert werden. Beispielsweise kann somit erkannt werden, wann eine Methode aufgerufen wird. Auch ein Aufruf eines Konstruktors oder der Aufruf einer statischen Methode kann somit abgefangen werden. Als Referenz für diese verschiedenen OpCodes kann <https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings> verwendet werden.

Neben der standardmäßigen *sawOpCode(int)-*Methode gibt es auch die Methoden *afterOpcode(int)* und *beforeOpCode(int): Boolean.* Das Besondere der *beforeOpCode(int): Boolean-*Methode ist es, dass, wenn die Methode *false* zurückliefert der OpCode, nicht die Methode *sawOpCode(int)* ausgeführt wird. Wird *true* zurückgegeben werden die restlichen Methoden normal aufgerufen. Die Methode *afterOpcode(int)* wird ausgeführt, nachdem die Methode *sawOpCode(int)* ausgeführt wird. Diese hat jedoch keine Auswirkung auf den restlichen Verlauf des Programms. Beide dieser Methoden könnten auch in der *sawOpCode(int)-*Methode simuliert werden. Für die Methode *afterOpcode(int)* müsste Code am Ende der *sawOpCode(int)-*Methode hinzugefügt werden und für die Methode *beforeOpCode(int): Boolean* müssteCode am Anfang der Methode hinzugefügt werden. Dieser Code sollte für eine vollständige Simulation einen optionalen Absprungpunkt über einen *return*-Befehl beinhalten.

Neben den OpCode-Methoden gibt es noch einige weitere Saw-Methoden. Zu diesen zählen:

* *sawMethod()*
* *sawIMethod()*
* *sawClass()*
* *sawField()*
* *sawDouble(double)*
* *sawInteger(int)*
* *sawLong(long)*
* *sawString(String)*
* *sawRegister(int)*
* *sawBranchTo(int)*

Diese Methoden werden aufgerufen, wenn die entsprechende Sache im Code gefunden wurde.

Neben den Methoden, welche selbstständig implementiert werden müssen, gibt es von SpotBugs auch noch einige Hilfsmethoden, welche bestimmte Informationen zur aktuellen Position im Code oder den aktuell vom Code geladenen Daten geben.

Die Grundlage für diese Hilfsmethoden ist die Fülle an Hilfsklassen, wie *MethodDescriptor* oder *ClassDescriptor*. In diesen Klassen sind entsprechende Informationen zu diesen hinterlegt. Diese können über die Methoden *getMethodDescriptor()* beziehungsweise *getClassDescriptor()* angesprochen werden.

Analog zu diesen Methoden gibt es das gleiche Muster auch für Felder. Das bedeutet, dass die Methode *getFieldDescriptor()* und die zugehörige Klasse *FieldDescriptor* vorhanden sind.

Wie in einem vorherigen Kapitel bereits erwähnt, besitzt die Java Virtual Machine ihren eigenen Operanden Stack. Auch dieser kann abgerufen werden. Dieser kann mit der Methode *getStack()* abgerufen werden. Diese Methode liefert ein Objekt der Klasse *OpodeStack* zurück. Die Datenstruktur ähnelt einem Stack. Dieser besteht aus mehreren *Items*, welche einzeln angesprochen werden können. Somit kann beispielsweise herausgefunden werden, mit welchen Übergabeparametern eine Methode aufgerufen wurde.

Neben diesen Methoden gibt es noch einige weitere Methoden, welche für die Informationsbeschaffung verwendet werden können. Diese können folgenden Dokumentationen entnommen werden:

* <https://javadoc.io/doc/com.github.spotbugs/spotbugs/latest/edu/umd/cs/findbugs/visitclass/PreorderVisitor.html>
* <https://javadoc.io/doc/com.github.spotbugs/spotbugs/latest/edu/umd/cs/findbugs/bcel/OpcodeStackDetector.html>
* <https://javadoc.io/doc/com.github.spotbugs/spotbugs/latest/edu/umd/cs/findbugs/visitclass/DismantleBytecode.html>

**Einspielen des Plugins:**

Wie ein solches Plugin in SpotBugs eingespielt wird, wird im Folgenden am Beispiel der SpotBugs Integration von Eclipse kurz erläutert.

Mittels Maven kann das Plugin gebaut werden. Dies funktioniert über einen einfachen Maven install Befehl. Dieser kann, wenn man sich mit der Kommandozeile in dem Ordner der entsprechenden *pom.xml* befindet, mit der Instruktion *mvn install* gestartet werden. Das Ergebnis dieser Instruktion ist eine jar-Datei.

Die jar-Datei kann nun in Eclipse in das SpotBugs Plugin injiziert werden. Hierzu muss der Eintrag **Window 🡪 Preferences** ausgewählt werden. Dieser öffnet ein Fenster, in welchem Einstellungen zu SpotBugs enthalten sind.

Um diese Einstellungen zu erreichen muss der Abschnitt *Java* geöffnet werden. Wenn das SpotBugs Plugin über den Eclipse Marketplace installiert wurde, befindet sich hier ein Eintrag namens *SpotBugs*. Wird dieser angeklickt, öffnet sich auf der rechten Seite eine View, in welcher neben verschiedenen Einstellungen zu SpotBugs auch ein Reiter namens *Plugins and misc. Settings* vorhanden ist.

In dem zu dem Reiter gehörigen Abschnitt befindet sich ein Fenster, in welchem die verschiedenen eingespielten Plugins von SpotBugs angezeigt werden. Wenn noch keine Einstellungen diesbezüglich getroffen worden sind, sollte dort ausschließlich der Eintrag *Core SpotBugs plugin [edu.umd.cs.findbugs.plugins.core]* vorhanden sein.

Auf der rechten Seite befinden sich Buttons, um weitere Plugins hinzuzufügen. Durch den *Add…* Button öffnet sich eine Dateiauswahl. In dieser Dateiauswahl muss nun die durch die *mvn install-*Instruktion erstellte jar-Datei ausgewählt werden. Wenn der Dialog beendet ist, sollte der Name des Plugins angezeigt werden, Ist dies abgeschlossen muss eventuell die IDE neugestartet werden, um die Änderungen zu aktivieren.

Wenn nun eine Prüfung mittels SpotBugs durchgeführt wird, wird der Code auch nach dem Muster, welches durch das geschriebene Plugin definiert wird, abgesucht. Ein gefundenes Fehlermuster wird entsprechend den Angaben in der *messages.xml* ausgegeben.

## 5.4 Schlussfolgerung

SpotBugs ist ein vielfältiges Tool. Dies trifft einerseits durch die Fülle an durch SpotBugs gefundenen Fehlermustern, welche bereits von Anfang an in SpotBugs enthalten sind, andererseits trifft dies auch durch die vielseitige API zu. Durch diese ist es möglich jedes Fehlermuster, das einem einfällt, in SpotBugs einzuspielen.

Ebenso ist es sehr einfach SpotBugs in mehrere Entwicklungsabläufe einzuspielen. Daher kann für SpotBugs nur eine Empfehlung ausgesprochen werden.

# 6. Problemstellung

Das Problem, welches mittels dem in dieser Arbeit entwickelten Plugin, soll nun noch einmal genauer vorgestellt werden.

Das Plugin soll Fehler einen Fehler ausgeben, wenn eine Methode einen Übergabeparameter genauer fordert als es für die Methode nötig ist. Ein Beispiel hierfür wäre folgende Methode:

**public void** method(ArrayList<String> couldBeIterable) {  
 **for** (String toOut: couldBeIterable) {  
 System.***out***.println(toOut);  
 }  
}

Der Übergabeparameter *couldBeIterable* könnte auch vom Typ *Iterable* sein, ohne dass dieser Code einen Fehler ausgeben würde. Korrigiert würde der Code folgendermaßen aussehen:

**public void** method(Iterable<String> iterable) {  
 **for** (String toOut: iterable) {  
 System.***out***.println(toOut);  
 }  
}

Der Vorteil von der zweiten Methode ist, dass sie nun mit weitaus mehr Klassen aufrufbar ist. Natürlich wäre es möglich jedes *Iterable* in eine *ArrayList* zu transformieren, jedoch wäre dies in diesem Fall schlichtweg nicht nötig, wenn die Methode auch ein *Iterable* akzeptieren würde, was in diesem Fall problemlos möglich ist.

Eine weitere Aufgabe ist es, dies auch für lokale Variablen zu erkennen. Ein Beispiel hierzu ist dieser Code, welcher den Fehlertypen enthält:

**public boolean** method() {  
 ArrayList<String> strings = **new** ArrayList<>();  
 *// List manipulations* **return** strings.contains(**"Hallo Welt!"**);  
}

Man könnte diesen Code auch folgendermaßen erstellen:

**public boolean** method() {  
 Collection<String> strings = **new** ArrayList<>();  
 *// List manipulations* **return** strings.contains(**"Hallo Welt!"**);  
}

Dieser Code ist insgesamt schöner und besser zu verwenden als der ursprüngliche Code.

Das Ziel ist nun genau diese Fehlertypen zu erkennen und anzuzeigen, dass dies verbessert werden sollte.

# 7. Praktische Umsetzung

Im Folgenden Kapitel wird der Weg von der Problemstellung zur finalen Lösung beschrieben. Hierbei wird sowohl auf die Philosophie hinter der Lösung als auch auf den handfesten Code der Lösung eingegangen.

Begonnen wird hierbei mit der Projektstruktur des Plugins. Im darauffolgenden Teil wird der Code unter Einbeziehung der dahinterstehenden Philosophie vorgestellt und erklärt.

## 7.1 Projektstruktur

Bei der Projektstruktur hat sich im Vergleich zur der Standardprojektstruktur von SpotBugs wenig verändert. Es wurden lediglich einige Hilfsklassen hinzugefügt, um spezielle Informationen beziehungsweise Informationskombinationen zu speichern. Des Weiteren wurden Hilfsklassen hinzugefügt, um die Funktionalität des Codes zu überprüfen.

### 7.1.1 Hilfsklassen des *main-*Pfades

Zu den Hilfsklassen, welche Informationen speichern, zählen:

* Das Interface: *Usage*
* *FieldUsage*
* *MethodUsage*
* *FieldWrapper*
* *MethodWrapper*
* *Parameter*

***Usage:***

Bei dem Interface *Usage* handelt es sich ausschließlich um ein sogenanntes Marker Interface ohne Inhalt, welches dafür sorgt, dass sowohl die Klasse *FieldUsage* als auch die Klasse *MethodUsage* in eine Liste gepackt werden können. Dies spielt später eine wichtige Rolle.

***FieldUsage:***

Die Klasse *FieldUsage* implementiert das Interface *Usage* und wird somit markiert. In der Klasse selbst werden Informationen wie ein Übergabeparameter in Verbindung mit einem Feld in einer Untersuchten Methode verwendet wurde. Sie enthält zwei Felder. Einerseits den *ClassDescriptor*, welcher die Klasse enthält, welche das Feld besitzt, und andererseits einen *FieldDescriptor*, welcher den Typen und den Namen des Feldes beschreibt.

***MethodUsage:***

Die Klasse *MethodUsage* ähnelt der Klasse *FieldUsage* in vielen Dingen. Allerdings speichert sie, wie der Name schon sagt, Informationen zur Verwendung von Methoden eines Übergabeparameters. Hierzu wird ein *ClassDescriptor* analog dem *ClassDescriptor* der *FieldUsage* verwendet. Daneben ist in der Klasse *MethodUsage* anstatt dem *FieldDescriptor* ein *MethodDescriptor* vorhanden, welcher Informationen zu den Aufrufen von Methoden von dem Übergabeparameter enthält.

***FieldWrapper:***

Der *FieldWrapper* speichert genauso wie die Klasse *FieldUsage* ebenfalls Informationen zu einem Feld ab. Allerdings funktioniert dieser allgemeiner und benutzt ausschließlich im Konstruktor Hilfsklassen von SpotBugs. Er wird verwendet, um eine Gleichheit zwischen einem Feld einer Oberklasse und einem verwendeten Feld festzustellen.

Hierfür verwendet der *FieldWrapper* die deklarierende Klasse des Feldes, den Namen des Feldes und den Typen des Feldes.

***MethodWrapper:***

Der *MethodWrapper* ist der Klasse *MethodUsage*, was die Klasse *FieldWrapper* der Klasse *FieldUsage* ist.

Um die gewünschte Funktionalität zu erfüllen verwendet der *MethodWrapper* insgesamt vier Felder:

* Die deklarierende Klasse der Methode
* Den Rückgabewert der Methode
* Den Namen der Methode
* Die Übergabeparameter der Methode

***Parameter:***

Die Klasse Parameter wird verwendet, um Übergabeparameter einer zu Untersuchenden Methode zwischenzuspeichern. Hierfür verwendet es grundlegend zwei Felder:

1. Den Typen des Übergabeparameters
2. Die sogenannte *registerNumber*

Die sogenannte *registerNumber* ist ein Integer, welcher dafür sorgt, dass jede Variable einzigartig ist. Mit Hilfe dieses Integers können Variablen desselben Typs unterschieden werden.

### 7.1.2 Hilfsklassen des *test*-Pfades

Zu den Hilfsklassen, welche die Funktionalitätsüberprüfung erleichtern, zählen:

* Das Interface *ITester*
* Eine implementierende Klasse *Tester*

***ITester:***

Das Interface definiert eine Methode Namens *dosmth()*. Diese Methode soll keinen größeren Sinn haben. Sie wird ausschließlich benötigt, um die Funktionalität des Codes zu überprüfen.

***Tester:***

Die Klasse Tester implementiert das Interface *ITester.* Zusätzlich zu der vom Interface vorgegebenen Methode, ist eine weitere Methode enthalten. Diese Klasse wird ebenfalls benötigt, um die Funktionalität des Codes zu überprüfen.

## 7.2 Die {Hauptklasse}

In diesem Abschnitt wird das Herzstück dieser Arbeit besprochen. Die Klasse {Hauptklasse} beinhaltet die gesamte Logik des Plugins.

Bei der Erklärung dieser Klasse wird systematisch vorgegangen. Die Reihenfolge hierbei lautet:

* Felder der Klasse
* Die überschriebene *sawOpcode(int)*- Methode
* Die überschriebene Methode *sawMethod()* und *sawField()*
* Die Methoden zur Bestimmung, ob Fehler vorhanden sind

### 7.2.1 Felder der Klasse

Um Daten zwischenzuspeichern werden verschiedene Felder benötigt. Zu diesen Feldern zählen:

* Eine Instanz des *BugReporter*
* Map zur Speicherung von Variablen einer zu untersuchenden Methode
* Map zur Speicherung der Verwendungen von Variablen einer Methode
* *OpcodeStackItem*
* Integer zur Speicherung der Menge an Übergabeparametern
* Ein Objekt zur Speicherung der aktuellen Methode

***BugReporter-*Instanz:**

Dieses Feld entsteht automatisch bei der Erstellung eines SpotBugs Projekt. Es wird benötigt, um einen Bug zu erstellen, welcher dem Nutzer angezeigt wird. Sein Nutzen beschränkt sich auf diesen Teil.

**Map zur Speicherung von Variablen einer zu untersuchenden Methode:**

Um die Variablen einer Methode zu speichern, wurde eine *HashMap* angelegt. Diese *HashMap* beinhaltet als Schlüssel die entsprechende Methode und als Wert eine Liste von *Parameter*. Diese *HashMap* wird während der Methode *sawOpcode(int)* befüllt. Diese *HashMap* wurde „*parametersPerMethod“* getauft.

**Map zur Speicherung der Verwendungen von Variablen einer Methode:**

Um die Verwendung von Variablen zu speichern, wurde eine *HashMap* erstellt. Diese *HashMap* beinhaltet als Schlüssel die zutreffende Methode und als Wert eine weitere *HashMap*, welche als Schlüssel einen *Parameter* und als Wert eine Liste von *Usage* enthält. Diese *HashMap* wird während den Methoden *sawMethod()* und *sawField()* befüllt. Diese *HashMap* heißt „*UsagesPerAttributePerMethod“.*

***OpcodeStackItem:***

Damit es in den Methoden *sawMethod()* und *sawField()* möglich ist zu erkennen, welche Variable verwendet wird, muss klar sein, mit welchem Objekt der Aufruf stattfindet, wenn die jeweiligen Methoden aufgerufen werden. Hierfür wurde eine globale Variable der Typs *OpcodeStack.Item* angelegt. In dieser Variable wird, je nach gesehenem OpCode, das erste oder zweite Item des Operandenstacks abgelegt. Dieser Inhalt wird in den Methoden sawField() und sawMethod() mit den vorhanden Variablen abgeglichen. Wenn ein Match gefunden wurde, wird in der HashMap *UsagesPerAttributePerMethod* ein entsprechender Eintrag erzeugt. Dieses Opcodestackitem heißt „*item“.*

**Integer zur Speicherung der Menge an Übergabeparametern:**

Damit eine Unterscheidung zwischen einer lokal erstellten Variable und einem Übergabeparameter möglich ist, wird ein Integer erstellt, welcher für die aktuelle Methode die Zahl von Übergabeparametern speichert. Dieser Integer erhielt den Namen „*paras*“.

***Method-*Objekt:**

Damit ein eindeutiger Schnitt zwischen zwei verschiedenen Methoden stattfinden kann, muss die aktuelle Methode zwischengespeichert werden. Dies wird mit Hilfe dieses Feldes gewährleistet. Dieses Feld wurde *method* genannt.

### 7.2.2 Die Methode *sawOpcode(int)*

Zunächst wird in dieser Methode überprüft, ob es sich bei dem vorgefundenen OpCode um einen OpCode handelt, welcher auf einen Methodenaufruf oder eine Feldoperation schließen lässt. Erfolgt diese Prüfung positiv, wird das Feld *item* mit dem aktuell obersten Item des Operandenstacks befüllt.

Im Folgenden wird abgeprüft, ob es sich bei dem aktuellen OpCode, um einen der *return-*OpCodes handelt. Trifft dies zu wird zusätzlich eine komplexere Prüfung anhand des Program-Counters durchgeführt, da ein Return auch mitten in einer Methode vorkommen kann. Wenn dies ebenfalls zutrifft wird die Methode *onLeaveMethod(Method)* aufgerufen. In dieser Methode befindet sich die Logik, die überprüft, ob ein Parameter verbessert werden kann.

Hiernach wird überprüft, ob die Methode gewechselt wurde. Hierfür wird die aktuelle Methode mit der letzten Methode verglichen, welche beim letzten gesehen OpCode in das Feld *method* abgelegt wurde. Wenn eine neue Methode gefunden wurde, werden zunächst alle Übergabeparameter in das Feld *parametersPerMethod* geschrieben. Um dies zu gewährleisten, werden zu diesem Zeitpunkt die obersten n Items des OpCode Stacks eingelesen und in die Klasse Parameter transformiert.

Im Anschluss wird der sogenannte *LocalVariableTable* ausgelesen. In diesem sind die lokalen Variablen, welche in der Methode erstellt werden, enthalten. Diese lokalen Variablen werden ebenfalls in Objekte der Klasse Parameter transformiert.

### 7.2.3 Die Methoden *sawMethod()* und *sawField()*

Die Methode *sawMethod()* wird aufgerufen, wenn ein Methodenaufruf innerhalb einer Methode festgestellt wird. Die Methode *sawField()* wird aufgerufen, wenn eine Feldoperation stattfindet.

Zunächst wird in diesen Methoden mit Hilfe des in der *sawOpcode(int)* Methode eingespeicherten Item der entsprechende Parameter aus der *HashMap* *parametersPerMethod* gesucht.

Wird ein passender Parameter gefunden, wird eine entsprechende *Usage* erstellt. Dies bedeutet, dass in der Methode *sawMethod()* eine *MethodUsage* und in der Methode *sawField()* eine *FieldUsage* erstellt.

Als letztes wird diese *Usage* in die *Map* *UsagesPerAttributePerMethod* eingetragen und das Item des *Opcode-Stacks* auf *null* zurückgesetzt.

### 7.2.4 Fehlerbestimmung