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, wobei während der Compile-Time bereits klar ist, dass Fehler beziehungsweise allgemeine Ungereimtheiten vorhanden sind, jedoch selbst keinen Compile-Fehler erzeugen. 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. Weiter soll dieses Plugin Fehler der Art: {WEITERE FEHLEARTEN EINFÜGEN}

{EBENSO DAS VORGEHEN INNERHALB DER ARBEIT ERLÄUTERN}

# 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 werden ausgewählte Design Patterns erläutert und der Zusammenhang von Design Patterns und Compilern erklärt.

## 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.