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 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