OTTO VON GUERICKE
UNIVERSITÄT
MAGDEBURG

INF

FAKULTÄT FÜR
INFORMATIK

Masterarbeit

im Studiengang Informatik

Erhöhung von Quellcode-Wartbarkeit durch Entwurfsmusterautomatisierung

eingereicht von: Michael Fritsch

eingereicht am: 14.11.2014

Betreuer: Sebastian Nielebock M. Sc.
Erstgutachter: Prof. Dr. Frank Ortmeier
Zweitgutachter: Dr.-Ing. Eike Schallehn

Inhaltsverzeichnis

Inhaltsverzeichnis **I**

Abkürzungsverzeichnis **IV**

Abbildungsverzeichnis **V**

Tabellenverzeichnis **VI**

1 Motivation und Zielsetzung der Entwurfsmusterautomatisierung **1**

1.1 Wartbarkeit als wichtiger Faktor für Software-Entwicklung 1

1.2 Entwurfsmusterautomatisierung als Ansatz zur Wartbarkeitsverbesserung 1

1.3 Zielsetzung 2

1.4 Thematische Abgrenzung und Rahmenbedingungen 2

1.5 Aufbau der Arbeit 4

2 Grundlagen der Wartbarkeitsmessung und Entwurfsmusterautomatisierung **6**

2.1 Wartbarkeit 6

2.1.1 Definition 6

2.1.2 Kriterien für die Wartbarkeit 7

2.1.3 Quellcodeeigenschaften als Wartbarkeitskriterien 11

2.2 Metriken 13

2.2.1 MOOD-Metrik-Suite 14

2.2.2 Weitere Einzel-Metriken 17

2.2.3 Eignung von Metriken zum Messen der Wartbarkeitskriterien . . 19

2.3 Entwurfsmuster und ihr Einfluss auf die Wartbarkeit 19

2.3.1 Entwurfsmuster im Allgemeinen 19

2.3.2 Erläuterung der GoF-Entwurfsmuster 20

2.3.3 Übersicht der Auswirkungen der Muster auf die Wartbarkeit . . . 22

2.4 Refactorings 24

2.4.1 Verwendete Refactorings 24

2.5 Bisherige Arbeiten 28

2.5.1 Bewertung von Entwurfsmustern bezüglich Wartbarkeit 28

2.5.2 Entwurfsmusterautomatisierung 28

3 Umsetzung der Entwurfsmusterautomatisierung 32
3.1 Auswahl verwendeter Entwurfsmuster 32
3.1.1 Vorauswahl anhand von erwartetem Einfluss auf die Wartbarkeit von Quellcode 33
3.1.2 Vorauswahl anhand bereits ermittelter Automatisierbarkeit 33
3.1.3 Vorauswahl anhand von vollständiger Automatisierbarkeit 34
3.1.4 Vorauswahl anhand von erwartetem Einfluss auf die Wartbarkeit . 38
3.2 Verfahren zur Ermittlung geeigneter Spots für Entwurfsmuster 40
3.2.1 Spot-Detektierungsansatz von Jeon 40
3.2.2 Erweiterung des Spot-Detektierungsansatzes 43
3.3 Verfahren für entwurfsmusterbildende Refactorings 44
3.3.1 Ausgangssituationen 45
3.3.2 Minipatterns 45
3.3.3 Komplexe Refactorings 46
3.4 Implementierungsentscheidungen 50

4 Ergebnisse der Entwurfsmusterautomatisierung 51
4.1 Vorbemerkungen zu den Experimenten 51
4.1.1 Verwendete Metrik-Messwerkzeuge 51
4.1.2 Verwendetes Qualitätsmodell 51
4.1.3 Testprojekte 52
4.1.4 Allgemeiner experimenteller Ablauf 54
4.2 Übersicht über detektierte und transformierbare Spots in Testprojekten . . 54
4.2.1 Abstrakte Fabrik 55
4.2.2 Kompositum 56
4.2.3 Zustand 56
4.2.4 Konsequenzen der Spotdetektierungsergebnisse 57
4.3 Durchführung und Ergebnisse der Entwurfsmusterautomatisierung 57
4.3.1 Abstrakte Fabrik 58
4.3.2 Kompositum 63
4.3.3 Zustand 68

5 Schlussfolgerungen und Ausblick für die Wartbarkeit durch Entwurfsmusterautomatisierung 74
5.1 Ergebnisse der Entwurfsmusterautomatisierung 74
5.1.1 Anwendbarkeit der Entwurfsmusterautomatisierung 74
5.1.2 Einfluss auf die Wartbarkeit 75
5.1.3 Aussagekraft der Ergebnisse 75

5.2 Vor- und Nachteile der Entwurfsmusterautomatisierung 76
5.2.1 Vorteile und Chancen 76
5.2.2 Nachteile und Risiken 76
5.3 Vergleich mit den Ergebnissen anderer Studien 77
5.4 Möglichkeiten zur weiteren Forschung 78

Literaturverzeichnis 79

A Anhang 83
A.1 Beschreibung der einzelnen Entwurfsmuster 83
A.2 Automatisierbarkeit der einzelnen Entwurfsmuster 91
A.2.1 Erzeugungsmuster 91
A.2.2 Strukturmuster 93
A.2.3 Verhaltensmuster 94
A.3 Erkannte und transformierbare Spots für Entwurfsmuster-Refactorings . . 98

Danksagung 100

Eidesstattliche Erklärung 101

Abkürzungsverzeichnis

AHF .... Attribute Hiding Factor

AIF .... Attribute Inheritance Factor

AST .... Abstract Syntax Tree

CF .... Coupling Factor

GoF .... Gang of Four

JDT .... Java Development Tools

LCOM .... Lack of Cohesion in Methods

LOC .... Lines of Code

MHF .... Method Hiding Factor

MIF .... Method Inheritance Factor

MOOD .... Metrics for Object-Oriented Design

RF .... Reuse Factor

UML .... Unified Modeling Language

Abbildungsverzeichnis

2.1 Qualitätsmerkmale nach ISO 9216 7
2.2 Taxonomie für Objekt-orientierte Software-Metriken nach [AS95] 14
2.3 Extract Interface Refactoring 25
2.4 Extract Method Refactoring 25
2.5 Move Method Refactoring 26
2.6 Replace Conditional with Polymorphism Refactoring 27
2.7 Replace Constructor with Factory Method Refactoring 27
2.8 Replace Type Code with State/Strategy Refactoring 28

3.1 Refactoring zum Kompositum-Entwurfsmuster 37

A.1 Refactoring zum Brücken-Entwurfsmuster 94
A.2 Refactoring zum Befehl-Entwurfsmuster 95

Tabellenverzeichnis

2.1 Abbildung von Wartbarkeitskriterien nach ISO9126 auf Quellcodeeigenschaften . . . 11
2.2 90%-Konfidenzintervall für Werte der MOOD-Metriksammlung . . . 16
2.3 Übersicht über Metriken und ihren Fokus . . . 19
2.4 Übersicht über Entwurfsmuster . . . 21
2.5 Einfluss von Entwurfsmustern auf Wartbarkeitskriterien . . . 23

3.1 Automatisierbarkeit von Entwurfsmustern laut [ÓCN01] . . . 34
3.2 Übersicht über autonom automatisierbare Entwurfsmuster . . . 38

4.1 Wichtung der einzelnen Mood-Metriken für das Qualitätsmodell . . . 52
4.2 Wartbarkeit des „JChessThreesome“-Projekts vor den Refactorings . . . 60
4.3 Wartbarkeit des „JChessThreesome“-Projekts nach dem Refactoring des ersten Spots . . . 60
4.4 Wartbarkeit des „JChessThreesome“-Projekts nach dem Refactoring des zweiten Spots . . . 60
4.5 Metrikergebnisse für das Abstrakte Fabrik-Refactoring des ersten Spots des „JChessThreesome“-Testprojekts . . . 61
4.6 Wartbarkeit des „de.ovgu.cse.vecs.saml“-Projekts vor dem Refactoring . . . 66
4.7 Wartbarkeit des „de.ovgu.cse.vecs.saml“-Projekts nach dem Refactoring . . . 66
4.8 Metrikergebnisse für das Kompositum-Refactoring des „de.ovgu.cse.vecs.saml“-Testprojekts . . . 66
4.9 Wartbarkeit des „h2o“-Projekts vor dem Refactoring . . . 70
4.10 Wartbarkeit des „h2o“-Projekts nach dem Refactoring . . . 70
4.11 Metrikergebnisse für das Zustand-Refactoring des „h2o“-Testprojekts . . . 71

A.1 Übersicht über erkannte und transformierbare Spots für Entwurfsmuster-Refactorings . . . 99

1 Motivation und Zielsetzung der Entwurfsmusterautomatisierung

1.1 Wartbarkeit als wichtiger Faktor für Software-Entwicklung

Keine Software ist bei der Auslieferung perfekt oder beinhaltet bereits alle Funktionalitäten, so dass Kunden auf Jahre wunschlos glücklich sind. Die Ansprüche und die Anforderungen der Kunden an eine Software ändern sich mit der Zeit – sie entwickeln sich weiter (Wunsch nach neuartigen Funktionen) oder passen sich an neue Gegebenheiten (beispielsweise neue gesetzliche Vorschriften) an. Somit sind Nachbesserungen an der Software – in Form von Fehlerbehebungen, Optimierungen oder Erweiterungen – fast immer nötig. Entwickler müssen ihre Software warten, um die Kundenzufriedenheit und damit dauerhafte wirtschaftliche Erfolge zu gewährleisten, oder einfach um vertragliche Verpflichtungen zu erfüllen. Dabei übersteigen die Kosten der Wartung oftmals die reinen Entwicklungskosten. In vielen Organisationen macht die Wartung einen Anteil von 40-70% der Gesamtkosten des kompletten Software-Lebenszyklus aus [GT03].
Um diese Kosten zu senken ist eine gute Wartbarkeit des Quellcodes wichtig. In gut wartbarem Code können Änderungen einfach, relativ schnell und somit günstig umgesetzt werden. In schlecht wartbarem Code hingegen ist es bei Anpassungen oft nötig einen großen Teil des Codes umzuschreiben, was sowohl zeit- und ressourcenintensiv als auch mit Risiken (beispielsweise ungewollte Seiteneffekte in bestehenden Funktionalitäten) behaftet ist.

1.2 Entwurfsmusterautomatisierung als Ansatz zur Wartbarkeitsverbesserung

Als eine Möglichkeit zur Erhöhung der Wartbarkeit von Quellcode werden Entwurfsmuster angesehen [GHJV95; JLB02; EGY97; ÓC01; GPP+02; Jen10; AMAL06]. Das Anwenden der Entwurfsmuster setzt jedoch entsprechende Erfahrung voraus. Ohne Kenntnisse über die Struktur und die möglichen Anwendungsbereiche der Muster kann ein Software-Entwickler sie nicht implementieren. Ohne das Wissen über Konsequenzen der Implementierung der Muster kann nicht entschieden werden, ob das Muster den Quellcode verbessert oder nicht. Werden die Muster mangels dieser Kenntnisse nicht angewandt,

so wird die Chance verpasst den Quellcode wartbar zu halten.
Entwurfsmuster sind jedoch nicht trivial und können, falsch angewendet, die Wartbarkeit von Code sogar verschlechtern. Erfahrung hilft zu entscheiden, ob ein Entwurfsmuster sinnvoll angewendet werden kann oder ob es die Komplexität des Quellcodes unnötig erhöht. Doch oft fehlen Entwicklern diese Erfahrungen noch.
Eine automatische Implementierung von Entwurfsmustern hätte den Vorteil, dass Quellcode auch dann um Entwurfsmuster erweitert werden könnten, wenn die Entwickler noch keine bzw. nur wenige Erfahrungen mit Entwurfsmustern haben. Genauso könnten erfahrenere, mit der Wartung beauftragte Entwickler bestehenden Quellcode effizienter warten, wenn der Code automatisiert um Entwurfsmuster erweitert wird, so dass der Entwickler dies nicht selbst tun muss.

1.3 Zielsetzung

Aus den genannten Gründen soll untersucht werden, ob die Wartbarkeit von Quellcode Objekt-orientierter Anwendungen durch das automatisierte Einfügen von Entwurfsmustern erhöht werden kann. Es sollen solche Entwurfsmuster in bestehenden Quellcode implementiert werden, die anhand von statischer Quellcodeanalyse situationsbedingt als zur Wartbarkeitsverbesserung geeignet erachtet werden. Dazu soll ermittelt werden, von welchen Kriterien die Wartbarkeit abhängt und ob diese Kriterien durch das automatisierte Einfügen von Entwurfsmustern in positiver Weise beeinflusst werden können. Dabei gilt es zu beachten, dass das Einfügen der Entwurfsmuster zwei grundlegende Folgen hat:

- bestehende Klassen werden verändert
- es können neue Klassen und Interfaces hinzu kommen

Daher gilt es zum Einen zu prüfen, ob bestehende Klassen wartbarer werden. Zum Zweiten gilt es zu untersuchen, ob das Gesamtsystem wartbarer wird, denn durch neu erstellte Klassen wird ein System wieder komplexer, wodurch die Wartbarkeit ggf. auch wieder abnehmen kann. Daher kann der Ansatz nur als der Wartbarkeit dienlich angesehen werden, wenn nicht nur bestehende Klassen wartbarer werden, sondern auch die Komplexität der neuen Klassen das Gesamtsystem nicht schwerer wartbar macht, als es ohne sie wäre.

1.4 Thematische Abgrenzung und Rahmenbedingungen

Die Wartbarkeit von Quellcode kann durch verschiedenste Methoden verbessert werden. Um den Einfluss von Entwurfsmustern auf die Wartbarkeit isoliert betrachten zu können,

ist es nötig sich von anderen Ansätzen abzugrenzen. Daher beschränkt sich diese Arbeit, trotz der Vielfalt an Methoden, auf das automatisierte Einfügen von Entwurfsmustern in bestehenden Quellcode. Die Verbesserung der Wartbarkeit durch andere Ansätze als Entwurfsmuster, wie beispielsweise Code auf Einhaltung von Programmierrichtlinien bzw. Namenskonventionen hin zu prüfen und anzupassen oder Methoden aus dem Bereich des "Clean Code Development" [Rob08], steht nicht im Fokus dieser Arbeit.
Auch soll nur der Einfluss der Entwurfsmuster auf die Wartbarkeit untersucht werden. Einfluss der Entwurfsmuster auf andere Aspekte als die Wartbarkeit, wie beispielsweise Performance, Ressourcenverbrauch oder das generelle Systemdesign, werden nicht betrachtet. Somit wird die Wartbarkeit von unabhängigen oder gar in Konkurrenz stehenden Aspekten isoliert.

Um den zeitlichen Rahmen der Arbeit nicht zu sprengen, werden im Vorfeld einige methodische Rahmenbedingungen festgelegt:

- Eine Automatisierung soll nicht für alle Entwurfsmuster untersucht werden, sondern nur für Entwurfsmuster, für die ein großer Einfluss auf die Wartbarkeit vermutet wird.
- Ist in einer Situation die Anwendung mehrerer Entwurfsmuster möglich, so wird nicht automatisiert entschieden, welches das geeignetere Entwurfsmuster ist. Es wird eine vorher manuell festgelegte bzw. zufällige Automatisierungsreihenfolge verwendet.
- Die Veränderung der Wartbarkeit soll anhand von Software-Metriken überprüft werden.
- Aus der Einschränkung auf Software-Metriken folgt, dass nur Wartbarkeitskriterien betrachtet werden, die auch über Metriken messbar sind.

Abschließend werden folgende technische Einschränkungen getroffen:

- Entwurfsmuster sind zwar ein programmiersprachenunabhängiges Konzept, eine sprachenunabhängige Untersuchung von Auswirkungen der Entwurfsmusterautomatisierung ist jedoch aus zeitlichen Gründen nicht vorgesehen. Da sich einige der verwendeten Quellen explizit auf die Programmiersprache Java beziehen und für Java eine breite Auswahl an Werkzeugen, Entwicklungsumgebungen und zugänglichen Testprojekten bereit steht, werden die Auswirkungen der Entwurfsmusterautomatisierung nur für Java untersucht. Da Java eine der am weitest verbreiteten Programmiersprachen ist[1], besteht trotz der Beschränkung auf Java eine hinreichende Relevanz.

[1] Laut Tiobe-Index (http://www.tiobe.com/index.php/tiobe_index) ist Java die zweit-populärste Programmiersprache (Stand 2014)

- ➢ Das automatisierte Einfügen von Entwurfsmustern setzt gewisse Strukturen im Quellcode voraus. Quellcode wird nicht vorher angepasst, um diesen Strukturen zu entsprechen (z.B. Ersetzen von öffentlichen Feldern durch Getter- und Setter-Methoden) – Test-Quellcode wird so genutzt, wie er vorgefunden wird. Andernfalls könnten diese entwurfsmusterfremden Umstrukturierungen die Messungen bzgl. der Wartbarkeit verfälschen.
- ➢ In Objekt-orientierten Software-Projekten werden oftmals kompilierte Klassen- und Funktionssammlungen (sogenannte Bibliotheken) verwendet, um so Software-Module wiederzuverwenden. Da der Quellcode der Bibliotheken oftmals nicht verfügbar ist, bzw. nicht geändert werden kann, werden Änderungen an Bibliotheken bzw. an in ihnen enthaltenen Klassen generell ausgeschlossen, um Quellcodefehler durch die Umstrukturierungen zu vermeiden.

Durch die hier getroffenen Abgrenzungen soll sichergestellt werden, dass keine fremden Einflüsse die Messergebnisse für die Wartbarkeit verfälschen. Die genannten Rahmenbedingungen sollen sichern, dass die Arbeit in der vorgesehen Zeit geschafft werden kann.

1.5 Aufbau der Arbeit

Die Arbeit ist wie folgt gegliedert: Zunächst erläutert Kapitel 2 einige für das weitere Verständnis nötige Grundlagen zur Wartbarkeit und verwendeten Techniken. Dabei wird erklärt, was Wartbarkeit bedeutet, von welchen Kriterien sie abhängt, wie sie verbessert und wie sie mittels Metriken gemessen werden kann. Des Weiteren werden Entwurfsmuster im Allgemeinen vorgestellt und welchen Einfluss sie auf die Wartbarkeit von Quellcode haben. Anschließend werden Refactorings im Allgemeinen erklärt und verwendete Refactorings aufgeführt. Abschließend werden bisherige Arbeiten im Bereich der Entwurfsmusterbewertung und Entwurfsmusterautomatisierung vorgestellt.

Kapitel 3 erläutert die Umsetzung der Entwurfsmusterautomatisierung. Dabei wird zunächst darauf eingegangen wie und warum die Entwurfsmuster für die Automatisierung ausgewählt wurden. Außerdem wird erläutert wie Stellen im Quellcode gefunden werden, die mit bestimmten Entwurfsmustern sinnvoll erweitert werden können und wie die Refactorings durchgeführt werden, um die Anwendung ohne Änderung des externen Verhaltens um Entwurfsmuster zu erweitern. Zudem wird auf Probleme bei der Umsetzung der Entwurfsmusterautomatisierung eingegangen.

Kapitel 4 beschreibt die experimentelle Validierung des Automatisierungsansatzes. Die Experimente betrachten die Auswirkungen der Entwurfsmusterautomatisierung auf die

Wartbarkeit bzw. auf einzelne Wartbarkeitskriterien verschiedener Quellcodebeispiele. Dies erlaubt eine generische Aussage über das Verbesserungspotential des Automatisierungsansatzes.

Kapitel 5 erläutert welche Schlussfolgerungen für die Wartbarkeit aus den Ergebnissen der Entwurfsmusterautomatisierung gezogen werden können. Dabei werden die Auswirkungen der Entwurfsmusterautomatisierung für die Wartbarkeit bzw. einzelne Kriterien der Wartbarkeit bewertet und Vor- und Nachteile der Entwurfsmusterautomatisierung aufgezeigt. Abschließend werden die Ergebnisse mit Thesen anderer Studien verglichen und weitere Forschungsmöglichkeiten aufgezeigt.

2 Grundlagen der Wartbarkeitsmessung und Entwurfsmusterautomatisierung

Übersicht

In den Grundlagen werden verschiedene Aspekte der Arbeit vorgestellt, die für das spätere Verständnis nötig sind. Abschnitt 2.1 erläutert was unter der Wartbarkeit von Software zu verstehen ist, welche Attribute die Wartbarkeit beeinflussen und wie die Wartbarkeit verbessert werden kann. In Abschnitt 2.2 wird eine Einführung in Software-Metriken gegeben. Es werden Metriken vorgestellt und aufgezeigt, wie sie die Wartbarkeit messen können. Abschnitt 2.3 erläutert das Konzept der Entwurfsmuster, stellt die betrachteten Entwurfsmuster vor und zeigt welche Einflüsse sie auf die Wartbarkeit haben. In Abschnitt 2.4 wird der Begriff und die Bedeutung von Refactorings erklärt. Bisherige Untersuchungen des Einflusses von Entwurfsmustern auf die Wartbarkeit von Quellcode, und Studien, bei denen Entwurfsmuster mithilfe von Refactorings (semi-)automatisiert in Software-Design oder -Quellcode eingefügt wurden, werden in Abschnitt 2.5 erläutert.

2.1 Wartbarkeit

2.1.1 Definition

Da die Wartbarkeit vom Begriff der Wartung abgeleitet ist, ist es sinnvoll zunächst die Wartung selbst zu definieren. Die Wartung ist „der Prozess der Änderung eines Software-Systems oder einer -Komponente nach der Auslieferung zur Fehlerbehebung, Verbesserung der Leistung oder anderer Attribute oder zur Anpassung an eine veränderte Umgebung“ [IEE90]. Die Wartbarkeit definiert sich als „der Aufwand mit dem die Wartung durchgeführt werden kann“ [GT03]. Das Ziel der Verbesserung der Wartbarkeit ist somit den Aufwand für die Wartung zu verringern. Die Definition zeigt bereits verschiedene Arten von Änderungen auf, die an einem System durchgeführt werden können. Insgesamt werden vier Arten von Änderungen unterschieden [IEE90; GT03]:

- Korrigierende Änderungen (Änderung um Fehler in der Software zu beheben)
- Adaptive Änderungen (Änderungen um die Software an neue Umgebungen bzw. Bedingungen anzupassen)

- Perfektive Änderungen (Änderungen zur Verbesserung bzw. Erweiterung der Software)
- Präventive Änderungen (Änderungen um Fehlfunktionen in der Software zu verhindern oder die Wartbarkeit der Software zu verbessern)

Der Aufwand für die Durchführung korrigierender, adaptiver und perfektiver Änderungen ist stark von der internen Struktur des Quellcodes abhängig, welche durch präventive Änderungen verbessert werden kann. Dies ist die Aufgabe der Entwurfsmuster, auf die in dieser Arbeit Bezug genommen wird.

2.1.2 Kriterien für die Wartbarkeit

Die Wartbarkeit hängt von verschiedenen Faktoren ab. Die Norm ISO/IEC 9126[1] beschreibt die Wartbarkeit als eines von sechs Qualitätsmerkmalen (siehe Abbildung 2.1) und als Kombination aus fünf verschiedenen Faktoren [ISO01]:

- Analysierbarkeit
- Änderbarkeit
- Stabilität
- Testbarkeit
- Konformität der Wartbarkeit

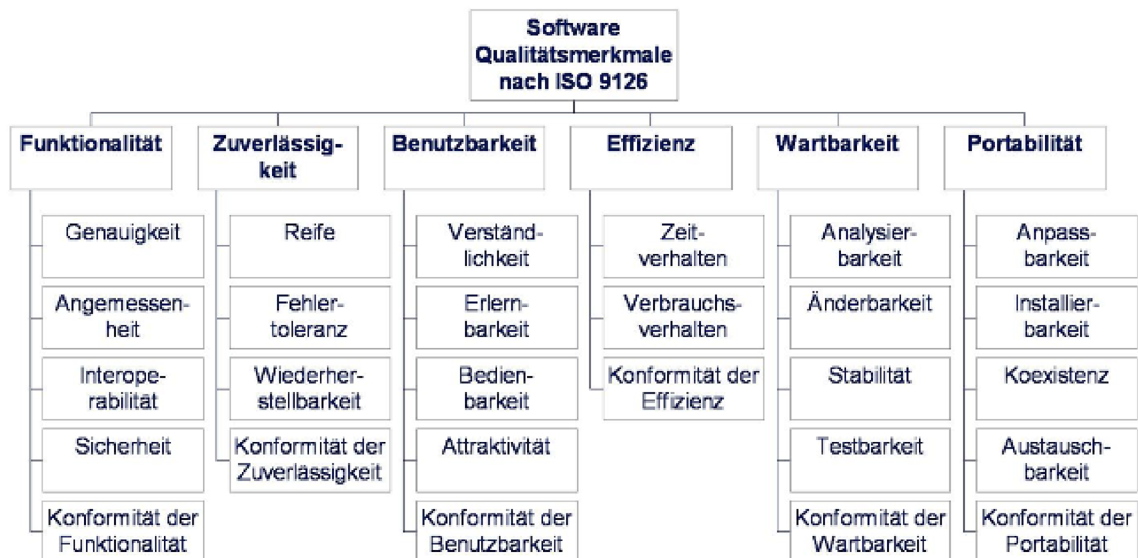


Abbildung 2.1.: Qualitätsmerkmale nach ISO 9216[2]

[1] Die Norm ISO/IEC 25000 wurde inzwischen durch die Norm ISO/IEC 25000 ersetzt, aufgrund nicht verfügbarer Quellen wird sich jedoch weiterhin auf die ISO/IEC 9126 bezogen

[2] Bildquelle: http://www.enzyklopaedie-der-wirtschaftsinformatik.de/wi-enzyklopaedie/Members/herzwurm/QMvS-abb2.png/image_large

Um zu messen, ob ein Entwurfsmuster die Wartbarkeit von Quellcode verbessern kann, müssen die genannten Faktoren aus dem Quellcode abgeleitet werden. Diese Faktoren lassen sich jedoch nicht direkt aus Quellcode lesen, da sie zu abstrakt sind. Stattdessen müssen die Faktoren auf Eigenschaften des Quellcodes abgebildet werden, die tatsächlich anhand von Software-Metriken auslesbar sind und die Faktoren implizieren [LLL08].

Nachfolgend werden die verschiedenen Wartbarkeitsfaktoren erklärt und entsprechende Quellcodeeigenschaften genannt, die diese Faktoren umschreiben. Diese Quellcodeeigenschaften werden später im Abschnitt 2.1.3 genauer beschrieben.

Analysierbarkeit

Die ISO 9126 beschreibt die Analysierbarkeit als den „Aufwand, um Mängel oder Ursachen von Versagen zu diagnostizieren oder um änderungsbedürftige Teile zu bestimmen“ [ISO01].
Hier spielen mehrere Faktoren eine Rolle. Um „Mängel oder Ursachen von Versagen“ zu bestimmen, muss ein Entwickler in der Lage sein den Quellcode zu verstehen[3], andernfalls kann er nicht erkennen, welche Codefragmente den Mangel verursachen [GT03]. Eine Quellcodeeigenschaft, die die Verständlichkeit von Quellcode impliziert, ist die Komplexität.
Um änderungsbedürftige Teile zu bestimmen, muss der Entwickler in der Lage sein zu erkennen, welche Komponenten an der Erledigung einer Aufgabe beteiligt sind. Dies ist umso einfacher, je klarer die Aufgaben der verschiedenen Komponenten voneinander getrennt sind [GT03]. Eine Quellcodeeigenschaft, die die Aufgabentrennung von Software-Komponenten beschreibt, ist die Kohäsion.

Änderbarkeit

Die ISO 9126 beschreibt die Änderbarkeit als „Aufwand zur Ausführung von Verbesserungen, zur Fehlerbeseitigung oder Anpassung an Umgebungsänderungen“ [ISO01].
Kommt es zu Änderungen an einer Software müssen oft mehrere Komponenten geändert werden. Zum Einen die Komponenten, welche die anzupassende Aufgabe bearbeiten. Je weniger Komponenten an der Erledigung der konkreten Aufgabe beteiligt sind, umso

[3]Um Quellcode zu verstehen muss ein Entwickler zum Einen Kenntnis der verwendeten Programmiersprache und -paradigmen besitzen. Dies wird als Grundvoraussetzung angesehen. Zum Anderen muss er erkennen können, was die Intention des Autors war bzw. was der Zweck des Codes ist. Auf diese Art des Verstehens wird sich in dieser Arbeit bezogen.

einfacher ist es die Änderung durchzuführen. Eine Quellcodeeigenschaft, die die Aufgabentrennung von Software-Komponenten beschreibt, ist die Kohäsion.
Zum Anderen müssen ggf. Komponenten angepasst werden, die in irgendeiner Art und Weise von den zuvor geänderten Komponenten abhängig sind. Bezieht sich eine Komponente direkt auf die zu ändernde Komponente, so ist die Wahrscheinlichkeit groß, dass beide verändert werden müssen [GT03]. Eine Quellcodeeigenschaft, welche die Abhängigkeit von Software-Komponenten beschreibt, ist die Kopplung.
Zugleich lassen sich Änderungen schneller durchführen, wenn der für die Bearbeitung der Aufgabe zuständige Code nicht komplett neu geschrieben werden muss, sondern bereits (in Teilen) vorhanden ist [GT03]. Können bestehende Komponenten für die zu bearbeitende Aufgabe verwendet werden, so spricht man von Wiederverwendung. Dies ist ebenfalls eine direkte Eigenschaft von Quellcode. Zugleich wirkt sich Wiederverwendung positiv auf die Änderbarkeit aus, da zu ändernde Funktionalitäten nur an einer Stelle angepasst werden müssen und diese Änderung von allen sie verwendenden Komponenten sofort übernommen werden. Ohne Wiederverwendung müssten dieselben Änderungen ggf. an mehreren Stellen durchgeführt werden.

Stabilität

Die ISO 9126 beschreibt die Stabilität als „Wahrscheinlichkeit des Auftretens unerwarteter Wirkungen von Änderungen“ [ISO01].
Unerwartete Wirkungen von Änderungen treten unter anderem auf, wenn Komponenten geändert werden, die mehrere Aufgaben bearbeiten. Soll die Bearbeitung einer Aufgabe innerhalb der Komponente verändert werden, so kann dadurch ungewollt die Bearbeitung der anderen Aufgaben der Komponente gestört werden. Daher ist die Aufgabentrennung – die Kohäsion – ein Aspekt, der sich auf die Stabilität einer Komponente auswirkt.
Zudem kann bei Änderungen unerwartetes Verhalten auftreten, wenn eine Komponente von außen den internen Zustand einer anderen Komponente ändert, ohne dass diese darauf reagieren kann. Eine Manipulation von außen kann die Komponente in einen undefinierten Zustand versetzen, den sie abgefangen hätte, wenn sie ihren Zustand selbst verändert hätte. Solche unerwarteten Änderungen können verhindert werden, indem der interne Zustand in der Komponente versteckt wird, so dass er nicht unkontrolliert von außen manipuliert werden kann. Das Verbergen von Daten und Verhalten einer Komponente nennt sich Kapselung und kann anhand des Quellcodes geprüft werden.
Zudem kommen ungewollte Auswirkungen von Änderungen zustande, wenn ein Entwickler aufgrund von schwer verständlichem Code das Ausmaß der Änderungen nicht über-

blicken kann. Somit hat auch die Komplexität des Codes einen Einfluss auf die Stabilität von Anwendungen bei Änderungen.

Testbarkeit

Die ISO 9126 beschreibt die Testbarkeit als „Aufwand, der zur Prüfung der geänderten Software notwendig ist“ [ISO01].
Eine geänderte Komponente muss umso gründlicher geprüft werden, je mehr Aufgaben die Komponente bearbeitet. Eine Komponente, die viele Aufgaben bearbeitet, muss daraufhin geprüft werden, ob sie alle Aufgaben nach der Änderung noch korrekt ausführt, selbst wenn nur eine Aufgabe von der Änderung betroffen war. Eine Komponente, die nur eine Aufgabe bearbeitet, ist nach der Verifikation der einen Aufgabe fertig geprüft. Somit spielt die Aufgabentrennung – die Kohäsion – eine große Rolle bei der Testbarkeit. Überdies ist auch von Bedeutung wie komplex die Aufgabe der geänderten Komponente ist, bzw. wie komplex die dafür verwendeten Methoden sind. Eine Methode, die bei der Erledigung viele verschiedene Aspekte betrachten und auf verschiedene Aspekte unterschiedlich reagieren muss, ist bei einer Änderung aufwändiger zu testen als eine Methode, die nur einen Aspekt bearbeitet. Somit spielt die Komplexität der Komponente eine Rolle bei der Testbarkeit.
Außerdem hängt die Testbarkeit zum Teil auch davon ab, wie erprobt die verwendeten Komponenten sind. Wird für die Änderung Quellcode neu geschrieben, so muss dieser intensiv geprüft werden. Kann man stattdessen auf Komponenten zurückgreifen, die bereits mehrfach geprüft wurden und sich im Laufe der Zeit bewährt haben, so ist lediglich zu prüfen, ob die Komponente richtig angewendet wurde – ihre Funktionalität selbst ist nicht mehr zwangsweise Gegenstand der Prüfung [GT03].

Konformität

Die ISO 9126 beschreibt die Konformität als den „Grad, in dem die Software Normen oder Vereinbarungen zur Änderbarkeit erfüllt“ [ISO01].
Zu erfüllende Vereinbarungen und Normen sind keine Eigenschaften des Quellcodes selbst, sondern Anforderungen, die die Anwender (oder der Gesetzgeber) an die Software haben. Diese können hochgradig variabel sein. Dieselbe Software kann manche Vereinbarungen erfüllen, manch andere hingegen nicht. Diese Anforderungen sind extern – es besteht keine Möglichkeit ihren Erfüllungsgrad aus Quellcode zu extrahieren.

2.1.3 Quellcodeeigenschaften als Wartbarkeitskriterien

Die genannten Wartbarkeitskriterien lassen sich – wie beschrieben – auf verschiedene Quellcodeeigenschaften abbilden. Tabelle 2.1 gibt einen Überblick darüber, welche Wartbarkeitskriterien durch welche Quellcodeeigenschaften repräsentiert werden können. Diese Quellcodeeigenschaften werden nachfolgend näher beschrieben.

Wartbarkeitskriterium	Stellvertretende Quellcodeeigenschaften
Analysierbarkeit	Komplexität, Kohäsion
Änderbarkeit	Kohäsion, Kopplung, Wiederverwendung
Stabilität	Kapselung, Kohäsion, Komplexität
Testbarkeit	Kohäsion, Komplexität, Wiederverwendung
Konformität	–

Tabelle 2.1.: Abbildung von Wartbarkeitskriterien nach ISO9126 auf Quellcodeeigenschaften

Kopplung

Die Kopplung gibt an wie stark Klassen voneinander abhängen [IEE90], wobei eine Klasse von einer anderen Klasse abhängig ist, wenn die Klasse auf Methoden oder Felder der anderen Klasse zugreift. Eine starke Kopplung zeigt starke Abhängigkeiten zwischen Klassen – bei Änderung einer Klasse müssen abhängige Klassen ggf. mitgeändert werden, wodurch die Wartbarkeit sinkt. Eine schwache Kopplung hingegen macht Klassen unabhängiger voneinander – Klassen können leichter ausgetauscht oder verändert werden, ohne dass andere Klassen davon betroffen sind. Will man die Wartbarkeit von Quellcode verbessern, sollte die Kopplung zwischen Klassen daher verringert werden. Dies ist möglich, indem anstelle von konkreten Klassen Abstraktionen wie Schnittstellen oder abstrakte Klassen verwendet werden.

Kapselung

Die Kapselung bestimmt wie gut Daten und Verhalten innerhalb einer Klasse von der Außenwelt abgeschirmt werden [IEE90]. Eine hohe Kapselung zeigt, dass Daten und Funktionen (zusammengefasst als "Member") einer Klasse nach außen hin verborgen werden und nur über bestimmte Schnittstellen manipuliert werden können. Eine niedrige Kapselung ermöglicht unerwünschte bzw. unkontrollierte Interaktion zwischen Programmteilen, wodurch das Fehlerrisiko steigt. Um die Wartbarkeit von Quellcode zu verbessern sollte die Kapselung von Klassen durch das Verbergen interner Strukturen erhöht werden.

Kohäsion

Die Kohäsion bestimmt „die Art und den Grad der Zusammengehörigkeit von Aufgaben von einzelnen Software-Modulen“ [IEE90]. Eine hohe Kohäsion zeigt eine hohe Zusammengehörigkeit der Member – sie bearbeiten dieselbe Aufgabe. Eine niedrige Kohäsion hingegen zeigt, dass eine Klasse scheinbar mehrere Aufgaben bearbeitet. Um die Wartbarkeit zu verbessern sollte die Kohäsion erhöht werden, indem Klassen mit niedriger Kohäsion in mehrere Klassen aufgeteilt werden, die jeweils nur eine Aufgabe bearbeiten.

Komplexität

Die Komplexität bestimmt „den Grad zu dem ein System oder eine Komponente ein schwer verständliches und zu verifizierendes Design oder Implementierung hat“ [IEE90]. Eine hohe Komplexität macht eine Programmiereinheit (Programm/Modul/Klasse/Methode) schwer verständlich – Änderungen sind schwer durchzuführen und auftretende Fehler nur schwer lokalisierbar. Eine geringe Komplexität hingegen macht eine Programmiereinheit leicht verständlich, Konsequenzen von Änderungen sind überschau- und daher gut einschätzbar, womit sich Änderungen mit weniger Risiko für ungewolltes Verhalten durchführen lassen. Um die Wartbarkeit zu verbessern sollte die Komplexität daher verringert werden.

Wiederverwendung

Wiederverwendung meint die „erneute Anwendung diverser Arten von Wissen über ein System bei einem anderen, ähnlichen System, um so den Aufwand für die Entwicklung oder Wartung für dieses andere System zu senken“ [GT03].
Diese Definition beinhaltet viele mögliche wiederverwendbare Artefakte, von Wissen über Dokumente bis hin zu Quellcode-Komponenten. Entwurfsmuster selbst sind ebenfalls eine Art der Wiederverwendung, da das Wissen um ein Problem und eine mögliche Lösung durch ein Entwurfsmuster auf verschiedene Kontexte angewandt werden kann. Diese Arbeit bezieht sich auf Wiederverwendung von Quellcode. Dabei gibt es zwei Arten von Quellcode-Wiederverwendung [AC94]:

- Wiederverwendung durch Vererbung: Eine Klasse erbt Methoden und Felder von ihrer Basisklasse.
- Wiederverwendung von Bibliothekskomponenten: Bereits entwickelte Module werden in ein neues Modul eingebunden um so im neuen Modul auf Funktionalitäten des bestehenden Moduls zuzugreifen.

Eine hohe Wiederverwendung reduziert Entwicklungsaufwände, da nicht alles mehrfach geschrieben werden muss. Außerdem verhindert die Wiederverwendung, dass bei Änderungen an vielen Stellen identische Änderungen durchgeführt werden müssen – wurde dieselbe Komponente an mehreren Stellen verwendet, so lassen sich alle diese Stellen durch die Änderung der einen genutzten Komponente anpassen. Eine hohe Wiederverwendung mittels Vererbung sorgt jedoch auch für eine Abhängigkeit zwischen den beteiligten Klassen – Änderungen an Basisklassen können unerwünschte Nebeneffekte in ihren Kindklassen hervorrufen. Eine sehr niedrige Wiederverwendung hingegen weist auf hohe Entwicklungsaufwände und ggf. auf vielen redundanten Code hin. Um die Wartbarkeit zu verbessern gilt es eine gute Balance aus Wiederverwendung und nicht übermäßiger Vererbung zu finden.

2.2 Metriken

Tom DeMarco stellte einst fest, dass man nicht kontrollieren kann, was man nicht messen kann [DeM86]. Um den Einfluss von Entwurfsmustern auf die genannten Wartbarkeitskriterien bestimmen zu können, müssen diese daher anhand des Quellcodes gemessen werden. Dies lässt sich mittels Metriken bewerkstelligen.
Eine Software-Metrik ist „eine Funktion, deren Eingabewerte Software-Daten sind und deren Ausgabe ein einzelner numerischer Wert, der als Grad interpretiert werden kann, zu dem die Software ein gegebenes Qualitätsmerkmal erfüllt“ [IEE90].
Verschiedene Objekt-orientierte Metriken messen dabei unterschiedliche Eigenschaften und arbeiten auf verschiedenen Abstraktionsstufen [AS95]. Die Abstraktionsstufen stellen eine Beziehungshierarchie dar (siehe Abbildung 2.2). Die höchste Abstraktionsstufe umfasst das gesamte System (oder ggf. ein ausgewähltes Subsystem) mit all seinen Klassen. Die zweite Abstraktionsstufe beschreibt Abhängigkeiten zwischen einzelnen Klassen. Die nachfolgende Stufe umfasst nur noch Vererbungsbeziehungen zwischen einzelnen Klassen. Auf Klassenebene werden lediglich Eigenschaften einer einzelnen Klasse gemessen. Die niedrigste Abstraktionsstufe misst nur noch die Eigenschaften einer einzelnen Methode einer Klasse. Anhand dieser Einteilung ist erkennbar, wie detailliert eine Metrik misst. Metriken auf Systemebene geben einen Überblick über das Gesamtsystem, lassen aber im Allgemeinen keine Rückschlüsse auf einzelne Klassen zu. Einzelne Klassen wiederum lassen sich mit Metriken auf Klassenebene gut untersuchen, aber Hinweise auf das System um die Klasse herum geben diese Metriken nicht[4].

[4]Es ist jedoch Möglich aus Klassenmetriken eine Systemmetrik zu erzeugen, indem die einzelnen Werte aufsummiert (und ggf. ein Durchschnitt gebildet) wird.

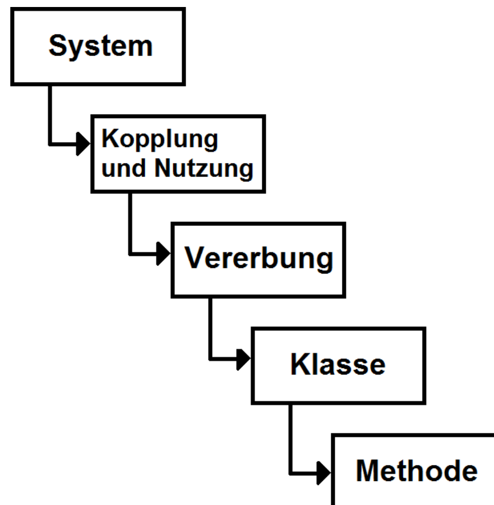


Abbildung 2.2.: Taxonomie für Objekt-orientierte Software-Metriken nach [AS95]

Nachfolgend werden die Metriken vorgestellt, die zur Messung der Wartbarkeit (bzw. einzelner Wartbarkeitskriterien) genutzt werden. Hier wird nur kurz erläutert, was und wie die Metriken messen und welche Metrikwerte ein wartbares System implizieren. Für die konkreten Funktionsweisen der einzelnen Metriken wird auf die jeweiligen Quellen verwiesen.

2.2.1 MOOD-Metrik-Suite

Die "Metrics for Object-Oriented Design" (kurz MOOD) ist eine auf Systemebene (also über alle Klassen) messende Metrik-Sammlung, eingeführt von Abreu [AC94]. Alle Metriken wurden erstellt um Auftreten oder Ausbleiben verschiedener Aspekte zu quantifizieren. Somit können die Metriken als Auftrittswahrscheinlichkeiten angesehen werden, wobei sich die Werte der Metriken dimensionslos zwischen 0 (totales Ausbleiben) und 1 (maximal mögliches Auftreten) bewegen. Die Metrik-Sammlung beinhaltet folgende sechs Einzelmetriken:

Attribute Hiding Factor

Der Attribute Hiding Factor (AHF) misst das Verhältnis von verborgenen Feldern zur Gesamtanzahl der Felder. Damit misst die Metrik wie hoch die Kapselung bzgl. der Felder ist.

Method Hiding Factor

Der Method Hiding Factor (MHF) misst – analog zum AHF – das Verhältnis der verborgenen Methoden zur Gesamtanzahl der Methoden. Somit misst die Metrik die Kapselung bzgl. der Methoden des gesamten Systems.

Attribute Inheritance Factor

Der Attribute Inheritance Factor (AIF) misst den Anteil vererbter Felder an der Gesamtheit der Felder. Somit misst diese Metrik die Wiederverwendung von Feldern.

Method Inheritance Factor

Der Method Inheritance Factor (MIF) misst den Anteil vererbter Methoden an der Gesamtheit der Methoden. Damit misst diese Metrik die Wiederverwendung von Methoden.

Coupling Factor

Der Coupling Factor (CF) misst das Verhältnis gekoppelter Klassen zu nicht gekoppelten Klassen. Eine Klasse ist zu einer anderen Klasse gekoppelt, wenn sie mindestens eine Referenz auf einen Member (Methode oder Attribut) der Klasse nutzt. Der Fokus dieser Metrik liegt also auf der Kopplung.

Polymorphism Factor

Der Polymorphism Factor (PF) misst die Anzahl tatsächlicher polymorpher Situationen (eine Klasse überschreibt ein Member einer Basisklasse) im Verhältnis zur Anzahl maximal möglicher polymorpher Situationen (eine Klasse könnte ein Member einer Basisklasse überschreiben). Diese Metrik misst sowohl die Wiederverwendung (nicht überschriebene Member werden wiederverwendet) als auch die Komplexität (ein System, in dem viele vererbte Member überschrieben werden, ist komplexer als ein System, in dem wenig Member überschrieben werden).

Empfohlene Werte für die MOOD-Metriken

Die Werte der MOOD-Metriken sind unterschiedlich zu interpretieren. Bei manche Metriken implizieren höhere Werte ein wartbareres System, bei anderen niedrigere Werte, bei wieder anderen ist ein bestimmter Wertebereich ein Indikator für ein wartbares System. In verschiedenen Studien wurden für die verschiedenen Metriken Richtwerte ermittelt [AGE95; AM96], welche in Tabelle 2.2 aufgeführt sind. Diese Werte stellen ein 90%-Konfidenzintervall dar, d.h. 90% der (in den Studien untersuchten) Systeme liegen innerhalb dieser Werte. Die Ersteller der Studien gehen für die gemessenen Eigenschaften von einer Gleichverteilung über die Gesamtheit aller Systeme aus und verwenden die Richtwerte aus den Studien somit für alle Systeme. Werte innerhalb der angegebenen Bereiche lassen nicht eindeutig auf ein wartbares System schließen, jedoch liegen sie im üblichen Rahmen. Werte außerhalb dieser Intervalle sollten jedoch die Aufmerksamkeit des Entwicklers erregen, der Entwurf sollte näher untersucht werden. Al-Ja'Afer hat eine klare Korrelation zwischen der Wartbarkeit eines Systems und der Einhaltung der Konfidenzintervalle der MOOD-Metriken festgestellt [AJS04].

Metrik	Untere Grenze	Obere Grenze
AHF	70,2%	89,3%
MHF	10,4%	28,7%
AIF	52,4%	60,0%
MIF	66,2%	80,8%
CF	3,9%	17,7%
PF	3,5%	9,6%

Tabelle 2.2.: 90%-Konfidenzintervall für Werte der MOOD-Metriksammlung

Die Werte für die sechs Metriken sind wie folgt zu interpretieren [AGE95]:

- AHF: Generell sollten alle Felder einer Klasse immer verborgen und nur über entsprechende Methoden zugänglich sein. Eine Ausnahme bilden statische Konstanten. Dennoch gilt, je höher der Wert ist, umso wartbarer ist das System[5].
- MHF: Es ist wünschenswert, dass viele Methoden gekapselt sind, da mit steigender Methodenkapselung die Fehlerdichte und der Fehlerbehebungsaufwand sinkt [AM96]. Dennoch müssen Methoden öffentlich zugänglich sein, da Objekte über die Methoden miteinander kommunizieren.
- AIF: Eine Wiederverwendung ist zwar zur Reduzierung des Entwicklungsaufwands wünschenswert, jedoch führt zu viel Vererbung zu starken Abhängigkeiten. Daher gilt es eine gute Balance zu finden.

[5] Ein AHF-Wert von 100% wird angestrebt, auch wenn dieser Wert nicht innerhalb des Konfidenzintervalls liegt.

- MIF: Analog zum AIF. Die höheren Werte für den MIF gegenüber dem AIF lassen sich darauf zurückführen, dass die Anzahl der vererbten Member exponentiell steigt, jedoch gerade früh in der Vererbungshierarchie mehr Methoden und weniger Felder vererbt werden.
- CF: Eine hohe Kopplung bedeutet eine starke Abhängigkeit zwischen Klassen, was nicht erwünscht ist. Ein Mindestmaß an Interaktion zwischen den Klassen ist jedoch nötig, denn ein System ohne jegliche Interaktion wäre eine zusammenhangslose Funktionssammlung.
- PF: Das Überschreiben von Membern kann ein System in bestimmten Fällen vereinfachen. Werden aber zu viele vererbte Member überschrieben, so weist dies auf eine fehlerhafte Verwendung von Vererbung hin.

2.2.2 Weitere Einzel-Metriken

Lack of Cohesion in Methods

Die Lack of Cohesion in Methods-Metrik (kurz LCOM) misst die fehlende Zusammengehörigkeit von Methoden durch Betrachtung der verwendeten Felder. Ursprünglich eingeführt von Chidamber & Kemerer, als Teil der "C&K-Metrik-Suite" [CK94], erfuhr sie aufgrund ungünstiger Eigenschaften eine Überarbeitung von Henderson-Sellers [HS96]. Die verwendete LCOM-Variante von Henderson-Sellers misst wie viele Methoden einer Klasse wie viele Attribute derselben Klasse verwenden. Benutzen alle Methoden jeweils nur ein Attribut von vielen, so errechnet die Metrik den Wert 1, was der höchst möglichen fehlenden Kohäsion – also keiner Kohäsion – entspricht. Nutzen alle Methoden jeweils alle Attribute der Klasse, so errechnet die Metrik den Wert 0 und damit die höchst mögliche Kohäsion. Je niedriger der Wert der Metrik ist, umso wahrscheinlicher ist es, dass die Klasse nur eine Aufgabe bearbeitet und damit besser zu warten ist.

Lines Of Code

Die Lines of Code (kurz LOC) Metrik misst die Anzahl der Code-Zeilen und damit die Größe einer Programmiereinheit. Dabei gibt es über 10 verschiedene Zählweisen [HS96], die sich unter anderem darin unterscheiden, ob Leerzeilen oder Quellcode-Kommentare mitgezählt oder ausschließlich Anweisungen über das Auftreten den Semikolons ";" gezählt werden. Die Metrik ist auf jede Programmiereinheit (Methoden, Klassen, gesamte Systeme) anwendbar. Die in dieser Arbeit verwendete Version zählt auf Klassenebene alle

nichtleeren Zeilen, die keine reinen Quellcode-Kommentare darstellen.
Die Größe eines Programms korreliert stark mit seiner Wartbarkeit – je größer ein Programm ist, desto schwerer ist es zu warten [SAM12]. Daher implizieren geringere LOC-Werte eine bessere Wartbarkeit.

Zyklomatische Komplexität

Die zyklomatische Komplexität – 1976 von McCabe eingeführt [McC76] und dementsprechend oft als McCabe-Metrik referenziert – ist eine der verbreitetsten Software-Metriken [HS96]. Die Metrik misst die Komplexität einer Programmiereinheit (üblicherweise einer Methode) anhand der Anzahl der verschiedenen möglichen Pfade, die innerhalb der Programmiereinheit durchlaufen werden können.
Generell ist eine Programmiereinheit umso einfacher zu verstehen, je geringer ihre Komplexität ist. McCabe gab als Empfehlung für die zyklomatische Komplexität einer Methode einen Höchstwert von 10 an, da Methoden, die eine zu hohe Komplexität aufweisen, für den Menschen schwer zu begreifen sind.

Halstead

Maurice Halstead führte 1977 eine Reihe von zusammenhängenden Software-Metriken ein, die die Komplexität einer Programmiereinheit anhand der Anzahl der (verschiedenen) Operanden (beispielsweise Variablen und Konstanten) und Operatoren (z.B. Schlüsselwörter, Methodenaufrufe, logische und Vergleichsoperatoren) messen [Hal77]. Zu diesen Metriken gehört das Halstead-Volumen, das anhand der Vokabulargröße der Programmiereinheit (Anzahl der verwendeten unterschiedlichen Operatoren und Operanden) und der Implementierungslänge (Anzahl der insgesamt verwendeten Operatoren und Operanden) errechnet wird. Diese Metrik gibt an, wie Umfangreich die Programmiereinheit ist, wobei die Programmiereinheit als umso komplexer gilt, je höher ihr Halstead-Volumen ist. Eine weitere Metrik ist die Halstead-Schwierigkeit, die aus dem Produkt der unterschiedlichen Operatoren und dem Verhältnis der insgesamt verwendeten Operanden zur Anzahl der unterschiedlichen Operanden errechnet wird. Sie gibt an, wie schwer die Programmiereinheit zu verstehen ist. Die Programmiereinheit gilt als umso verständlicher, je geringer der Wert dieser Metrik ist.

Wartbarkeitsindex

Der Wartbarkeitsindex (englisch „Maintainability Index“, kurz MI) ist eine Metrik, die die Wartbarkeit einer Software-Komponente direkt quantifizieren soll [WO95]. Die Berechnung des Wartbarkeitsindex erfolgt auf Basis der vorher genannten Metriken für die Zeilenanzahl, die zyklomatische Komplexität und der Halstead-Metriken und verrechnet sie anhand von empirisch ermittelten Verhältnissen.
Je höher der Metrikwert ist, umso besser wartbar wird die Komponente eingestuft, wobei ein Wert von mindestens 85 eine gute Wartbarkeit anzeigt, ein Wert von unter 65 weist auf eine schwer zu wartende Komponente hin [WO95].

2.2.3 Eignung von Metriken zum Messen der Wartbarkeitskriterien

Die nachfolgende Aufstellung gibt eine Übersicht über die vorgestellten Wartbarkeitskriterien und durch welche Metriken sie gemessen werden können.

Quellcodeeigenschaft	Metriken
Kopplung	CP
Kapselung	AHF MHF
Kohäsion	LCOM
Komplexität	PF LOC McCabe Halstead-Volumen und Halstead-Schwierigkeit
Wiederverwendung	PF AIF MIF
Wartbarkeit	MI

Tabelle 2.3.: Übersicht über Metriken und ihren Fokus

2.3 Entwurfsmuster und ihr Einfluss auf die Wartbarkeit

2.3.1 Entwurfsmuster im Allgemeinen

Ein Entwurfsmuster (engl.: "Design Pattern") ist Lösungsvorschlag für ein wiederkehrendes Problem beim Software-Design bzw. bei der Software-Erstellung [GHJV95]. Allgemein gesagt ist ein Entwurfsmuster eine positive Entwurfserfahrung, die dokumentiert

wurde, damit sie für ein ähnliches Problem wiederverwendet werden kann, so dass die Lösung nicht erneut "gefunden" werden muss. Damit grenzen sich Entwurfsmuster von sogenannten "Anti-Pattern" ab, die eine negative Erfahrung dokumentieren, mit dem Ziel sie nicht erneut zu machen [Koe98]. Jedes Entwurfsmuster definiert u.a. einen Namen, ein zu lösendes Problem, eine Lösung für das Problem und Konsequenzen der Nutzung. Dabei sind die Muster so allgemein gehalten, dass sie in verschiedenen Situationen angewandt werden können.

2.3.2 Erläuterung der GoF-Entwurfsmuster

Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides – oft referenziert als "die Viererbande" (engl. "Gang of Four", kurz GoF) haben einen der bekanntesten Entwurfsmusterkataloge vorgestellt [GHJV95]. Er umfasst 23 Entwurfsmuster, die sowohl klassenbasierte (Vererbungshierarchien zur Erstellungszeit) als auch objektbasierte (Komposition von Objekten zur Laufzeit) Lösungsvorschläge für Probleme bei der Objekterzeugung, bei der Struktur und beim Verhalten von Objekten bzw. Klassen geben. Eine Übersicht über die Muster, ihre Aufgabe und ihren Gültigkeitsbereich ist in Tabelle 2.4 zu finden.
Nachfolgend werden drei der 23 Entwurfsmuster in aller Kürze vorgestellt[6]. Außerdem wird – für die Beurteilung der Eignung der Muster für die Automatisierung – erläutert, wie sie sich (laut der Angaben des Kataloges selbst) auf die Wartbarkeit von Quellcode auswirken. Dabei wird jedes Muster nur kurz angerissen, für ausführliche Informationen wird auf [GHJV95] verwiesen.

Abstrakte Fabrik

Die Abstrakte Fabrik bietet „eine Schnittstelle zum Erzeugen vom Familien verwandter oder voneinander abhängiger Objekte, ohne ihre konkreten Klassen zu benennen" [GHJV95].
Es wird also die Objekterzeugung für eine Familie von verwandten Objekten gekapselt. Ein Klient der Abstrakten Fabrik kennt die konkrete Klasse des erzeugten Objekts nicht und ist somit vom Objekt entkoppelt. Gleichzeitig wird sichergestellt, dass die erzeugten Objekte zur selben Produktfamilie gehören (beispielsweise dass erzeugte Steuerelemente einer GUI alle dasselbe Look&Feel besitzen). Somit sind weitere Konsistenzprüfungen

[6] Diese Auswahl zeigt bereits die letzten Endes zur Automatisierung ausgewählten Entwurfsmuster. Die Beschreibung der restlichen 20 Entwurfsmuster wurde aus Gründen der Übersichtlichkeit in den Anhang A.1 verschoben.

		Aufgabe		
		Erzeugungsmuster	Strukturmuster	Verhaltensmuster
Gültigkeitsbereich	klassenbasiert	Fabrikmethode	Adapter (klassenbasiert)	Interpreter Schablonenmethode
	objektbasiert	Abstrakte Fabrik Erbauer Prototyp Singleton	Adapter (objektbasiert) Brücke Dekorierer Fassade Fliegengewicht Kompositum Proxy	Befehl Beobachter Besucher Iterator Memento Strategie Vermittler Zustand Zuständigkeitskette

Tabelle 2.4.: Übersicht über Entwurfsmuster

bzgl. der Produktfamilie nicht nötig, wodurch sich die Komplexität der Klienten verringert.

Kompositum

Ein Kompositum „fügt Objekte zu Baumstrukturen zusammen, um Teil-Ganzes-Hierarchien zu repräsentieren. Das Kompositionsmuster ermöglicht es Klienten, sowohl einzelne Objekte als auch Kompositionen von Objekten einheitlich zu behandeln.“ [GHJV95]
Da Klienten auf diese Weise nicht mehr zwischen einfachen und komplexen Objekten unterscheiden müssen, sondern dies vom Kompositum übernommen wird, wird die Komplexität des Klienten verringert.

Zustand

Das *Zustand*-Muster „ermöglicht es einem Objekt, sein Verhalten zu ändern, wenn sein interner Zustand sich ändert. Es wird so aussehen, als ob das Objekt seine Klasse gewechselt hat.“ [GHJV95]
Das *Zustand*-Muster kapselt zustandsabhängiges – und damit zusammengehöriges – Verhalten in eigene Objekte, was der Kohäsion zugute kommt. Weiterhin kann das *Zustand*-Muster Bedingungsanweisungen im Code entfernen, die sonst in potentiell mehreren Methoden vorhanden wären. So müsste eine Klasse, die zwischen mehreren internen Zuständen unterscheidet, je nach aktuellem Zustand aus verschiedenen Tätigkeiten wählen

(wenn Zustand A, dann Tätigkeit A1, sonst ... – und das in allen vom aktuellen Zustand beeinflussten Methoden). Sind alle vom Zustand beeinflussten Tätigkeiten in einem eigenen Zustandsobjekt gekapselt, so kann der Klient die Tätigkeit polymorph über die Zustandsklasse aufrufen (`Zustand.Tätigkeit1();` – was "Tätigkeit1" ist hängt vom Zustand ab, der Klient muss nicht mehr selbst wählen). Somit wird die Zustände-unterscheidende Klasse deutlich weniger komplex.

2.3.3 Übersicht der Auswirkungen der Muster auf die Wartbarkeit

Aus den Beschreibungen des vorherigen Abschnitts (und des Anhangs A.1) geht bereits hervor, dass Entwurfsmuster verschiedene Auswirkungen auf die Wartbarkeit haben. Diverse Muster können die Kopplung lockern, indem Abstraktionen (Interfaces oder abstrakte Klassen) anstelle konkreter Klassen genutzt werden. Manche Muster können die Kapselung erhöhen, indem interne Strukturen in einzelnen Klassen verborgen werden. Wieder andere Muster können die Kohäsion erhöhen, indem zusammengehörige Funktionalitäten in eigene Klassen gekapselt werden. Auch können die Muster die Komplexität verringern, indem sie Strukturen optimieren und große Klassen in kleinere Klassen mit entsprechenden Zuständigkeiten aufteilen. Und wieder andere Muster können die Wiederverwendung erhöhen, indem mehrfach genutzte Algorithmen und Strukturen in eigene Klassen ausgelagert und somit erneut verwendet werden können. Tabelle 2.5 führt für alle 23 GoF-Entwurfsmuster auf, welche Wartbarkeitskriterien wie beeinflusst werden. Die Wertungen ergeben sich direkt aus den in Abschnitt 2.3.2 und Anhang A.1 genannten Auswirkungen auf die Wartbarkeitskriterien.

Entwurfsmuster	**Einfluss auf Wartbarkeitskriterien**				
	Kopplung	Kapselung	Kohäsion	Komplexität	Wieder-verwendung
Erzeugendenmuster					
Abstrakte Fabrik	positiv	-	-	positiv	-
Erbauer	positiv	-	-	positiv	-
Fabrikmethode	positiv	-	-	positiv	-
Prototyp	positiv	-	-	positiv	-
Singleton	-	-	-	positiv	-
Strukturmuster					
Adapter	-	-	-	-	positiv
Brücke	positiv	-	-	-	-
Dekorierer	-	-	positiv	positiv	-

Entwurfsmuster	**Einfluss auf Wartbarkeitskriterien**				
	Kopplung	Kapselung	Kohäsion	Komplexität	Wieder-verwendung
Fassade	positiv	-	-	-	-
Fliegengewicht	-	-	-	-	-
Kompositum	-	-	-	positiv	-
Proxy	-	-	-	-	-
		Verhaltensmuster			
Befehl	positiv	-	-	-	-
Beobachter	positiv	-	-	-	-
Besucher	-	negativ	positiv	-	-
Interpreter	-	-	-	-	-
Iterator	-	-	-	positiv	-
Memento	positiv	positiv	-	positiv	-
Schablonenmethode	-	-	-	-	positiv
Strategie	-	-	positiv	positiv	positiv
Vermittler	positiv	-	-	positiv und negativ	-
Zustand	-	-	positiv	positiv	-
Zuständigkeitskette	positiv	-	-	-	-

Tabelle 2.5.: Einfluss von Entwurfsmustern auf Wartbarkeitskriterien

Die Entwurfsmuster beeinflussen mitunter auch andere Kriterien als die genannten, wie beispielsweise den Speicherbedarf, der durch das *Fliegengewicht*-Muster optimiert werden soll. Das *Fliegengewicht*-, *Proxy*- und *Interpreter*-Muster haben beispielsweise keinen Einfluss auf die gewählten Wartbarkeitskriterien.
Entwurfsmuster können neben den positiven jedoch auch negative Auswirkungen haben. So brechen beispielsweise einige Muster die Kapselung von Klassen auf und nahezu alle Muster erhöhen die Anzahl der Klassen des Systems. Daher ist es nicht sinnvoll Entwurfsmuster bei jeder Gelegenheit anzuwenden, sondern nur da, wo ihre Vorteile ihre Nachteile überwiegen bzw. nur da, wo die von ihnen erzeugten Vorteile auch genutzt werden. So bringt es beispielsweise keinen Vorteil Klassen nur über Abstraktionen miteinander kommunizieren zu lassen und so austauschbar zu machen, wenn niemals geplant ist konkrete Klassen tatsächlich auszutauschen.
Zudem weisen Lewerentz et. al [LRS00] darauf hin, dass es Gegensätze bei Qualitätskriterien gibt. So können Maßnahmen, die sich positiv auf ein Qualitätskriterium auswirken

sich automatisch negativ auf ein anderes auswirken. Diese Gegensätze können oft nicht vermieden werden, sondern müssen gegeneinander und gegen die Ziele des Entwicklers bzw. die Anforderungen an das System abgewogen werden, um zu entscheiden, ob die Entwurfsmuster sinnvoll eingesetzt werden können.

2.4 Refactorings

Ein Quellcode-Refactoring meint „eine Änderung an der internen Struktur einer Software um sie einfacher verständlich oder günstiger anpassbar zu machen, ohne das beobachtbare Verhalten zu verändern“ [Fow99]. Dies umfasst sowohl lokale Änderungen, beispielsweise das Umbenennen einer Variable in einer Methode, als auch programmweite Änderungen, wie beispielsweise das Verschieben einer Klasse.

2.4.1 Verwendete Refactorings

Für das automatisierte Einfügen von Entwurfsmustern werden verschiedene Refactorings genutzt. Die nachfolgende Auswahl an genutzten Refactorings ist [Fow99] entnommen. Für jedes Refactoring ist kurz angegeben was es tut, welche Bedingungen für die Anwendung des Refactorings erfüllt sein müssen und warum es das Systemverhalten nicht verändert.

Extract Interface

Aus einer Klasse wird eine Schnittstelle extrahiert, indem alle benötigten öffentliche Methoden in eine neue Schnittstelle aufgenommen werden, welche dann von der Klasse implementiert wird (siehe Abbildung 2.3). Die Schnittstelle lässt sich von Klienten (außerhalb der Vererbungshierarchie) anschließend genauso benutzen, wie die zugrunde liegende Klasse, um die Klienten somit von der konkreten Klasse zu entkoppeln.
Damit das Refactoring durchgeführt werden kann, darf der Name der neu zu erstellenden Schnittstelle von keiner anderen Schnittstelle oder Klasse im selben Gültigkeitsbereich verwendet werden, da es sonst zu Namenskonflikten kommt.
Da die Klasse, aus der die Schnittstelle extrahiert wird, nicht verändert wird – bis auf die Angabe dass sie die aus ihr erstellte Schnittstelle implementiert – verändert sich das Verhalten des Systems nicht. Da die Schnittstelle neu erstellt wurde und somit vorher nicht referenziert werden konnte, kann auch hier keine Veränderung im Systemverhalten entstehen.

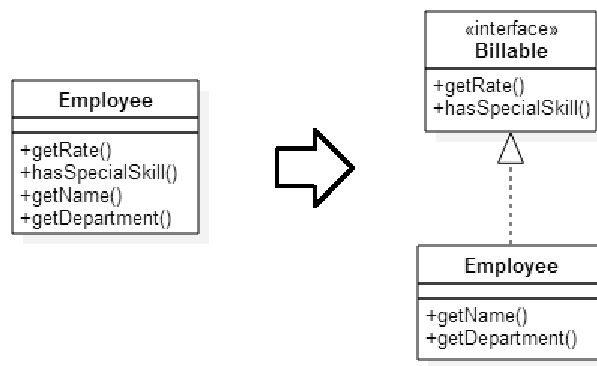


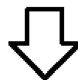
Abbildung 2.3.: Extract Interface Refactoring

Extract Method

Codefragmente werden aus einer Methode in eine neu erstellte Methode in derselben Klasse verschoben, wobei vom verschobenen Code verwendete lokale Variablen und Referenzen als Eingabeparameter für die neue Methode, und das Ergebnis des Fragments als Rückgabewert verwendet werden. Dies ist beispielhaft in Abbildung 2.4 abgebildet.
Damit das Refactoring durchgeführt werden kann, darf in der Klasse keine Methode mit derselben Signatur wie die zu erstellende Methode existieren, da es sonst zu Namenskonflikten kommt.
Die neu erstellte Methode wird an der Stelle aufgerufen, an der die verlagerten Codefragmente vorher zu finden waren. Da der Code an derselben Stelle mit denselben Werten wie vorher ausgeführt wird, verändert sich das Verhalten des Systems nicht. Es wurde lediglich eine Indirektion hinzugefügt.

```
void printOwing(double amount) {
    printBanner();

    // print details
    System.out.println("name:" + _name);
    System.out.println("amount:" + amount);
}
```



```
void printOwing(double amount) {
    printBanner();
    printDetails(amount);
}

void printDetails(double amount) {
    System.out.println("name:" + _name);
    System.out.println("amount:" + amount);
}
```

Abbildung 2.4.: Extract Method Refactoring

Move Method

Eine Methode wird in eine andere Klasse verschoben (siehe Abbildung 2.5), wobei die Methode in der ursprünglichen Klasse per Delegation referenziert oder komplett entfernt wird.
Bedingung für dieses Refactoring ist, dass in der Zielklasse keine Methoden mit derselben Signatur wie die zu verschiebende Methode existiert, da es sonst zu Namenskonflikten kommen kann.
Da derselbe Code an derselben Stelle aufgerufen wird – lediglich anhand einer anderen Referenz – kann sich das Systemverhalten nicht verändern.

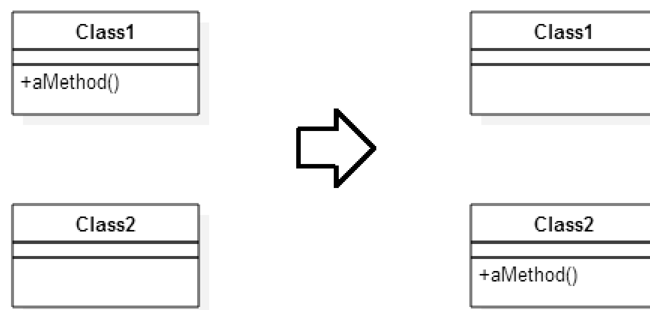


Abbildung 2.5.: Move Method Refactoring

Replace Conditional with Polymorphism

Bedingungsanweisungen werden durch polymorphe Methodenaufrufe von Objekten ersetzt (siehe Abbildung 2.6). Dazu werden Klassen erstellt, welche die jeweiligen Bedingungen repräsentieren. Die die jeweiligen Bedingungen betreffenden Blöcke der Bedingungsanweisung werden in Methoden in den zugehörigen Klassen ausgelagert, welche dann anstelle der Bedingungsanweisung ausgeführt werden.
Damit dieses Refactoring durchgeführt werden kann, dürfen die einzelnen Blöcke der Bedingungsanweisungen jeweils maximal eine lokale Variable setzen, da diese als Rückgabewert fungiert und Methoden maximal einen Rückgabewert haben können.

Replace Constructor with Factory Method

Die Instanziierung eines Objekts wird in eine eigene Methode ausgelagert. Klienten rufen diese Methode auf, anstatt das Objekt direkt zu erstellen (siehe Abbildung 2.7). Dadurch wird die Objekterstellung in der Methode gekapselt und kann durch Subklassen einfach

```
double getSpeed() {
    switch(_type) {
    case EUROPEAN:
        return getBaseSpeed();
    case AFRICAN:
        return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
    case NORWEGIAN_BLUE:
        return (_isNailed) ? 0 : getBaseSpeed(_voltage);
    default:
        break;
    }
    throw new RuntimeException("should be unreachable");
}
```

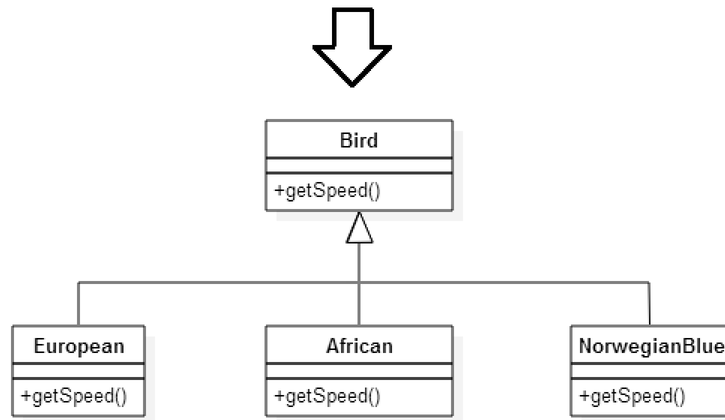
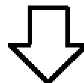


Abbildung 2.6.: Replace Conditional with Polymorphism Refactoring

und zentral überschrieben werden.
Damit das Refactoring durchgeführt werden kann, darf in der Zielklasse keine Methode mit derselben Signatur wie den neuen Methoden existieren.
Die Klasse wird durch denselben Konstruktor mit denselben Parametern instanziiert, lediglich in einer Methode gekapselt. Daher kann auch hier das Systemverhalten nicht geändert werden.

```
public Employee(int type) {
    _type = type;
}
```



```
static Employee createEmployee(int type) {
    return new Employee(type);
}
```

Abbildung 2.7.: Replace Constructor with Factory Method Refactoring

Replace Type Code with State/Strategy

(Über mehrere Methoden verteilte) Codefragmente, die abhängig vom Wert einer Instanzvariablen das Verhalten der Klasse bestimmen, werden in den Variablenwert repräsentie-

rende Klassen ausgelagert und polymorph verwendet, anstatt das Verhalten über Bedingungsanweisungen zu ermitteln. Dies ist beispielhaft in Abbildung 2.8 abgebildet.

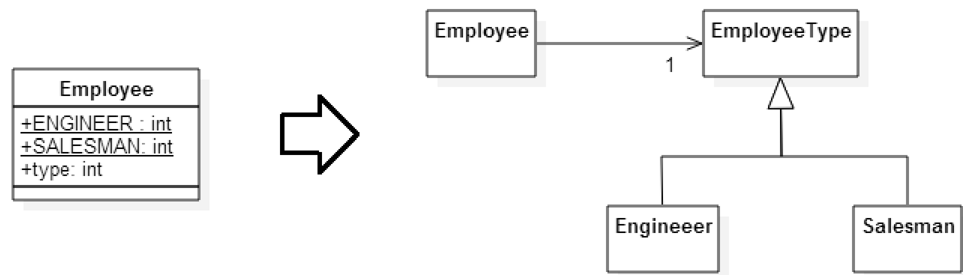


Abbildung 2.8.: Replace Type Code with State/Strategy Refactoring

2.5 Bisherige Arbeiten

Die zwei folgenden Abschnitte 2.5.1 und 2.5.2 fassen bisherige Arbeiten zur Bewertung des Einflusses von Entwurfsmustern auf die Wartbarkeit und zur Entwurfsmusterautomatisierung zusammen und führen Vor- und Nachteile der einzelnen Ansätze auf.

2.5.1 Bewertung von Entwurfsmustern bezüglich Wartbarkeit

Prechelt et al. haben die Auswirkungen von Entwurfsmustern auf die Wartbarkeit von Quellcode untersucht [PUT$^+$01]. In der Studie sollten verschiedene Systeme, die jeweils einmal mit und einmal ohne Entwurfsmuster umgesetzt wurden, von Entwickler-Teams manuell erweitert werden. Überprüft wurde dann bei welchen Systemen die Anpassungen schneller durchgeführt werden konnten. Die Messungen zeigten, dass manche Änderungen in den Systemen mit Entwurfsmustern schneller durchgeführt werden konnten, und manche Änderungen in den Systemen ohne Entwurfsmuster. Somit wurde keine generelle Aussage darüber getroffen, ob die Entwurfsmuster die Wartbarkeit verbessern konnten oder nicht.

2.5.2 Entwurfsmusterautomatisierung

Bisher wurden verschiedenste Arten untersucht, um Software-Entwürfe (sowohl in Form von Modellen wie UML als auch implementierter Quelltext) um Entwurfsmuster zu erweitern. Dabei wurden sowohl semi- als auch vollautomatisierte Ansätze verfolgt.

Eden et al. entwickelten eine Sprache, mit der Entwurfsmuster, die original nur in natürlicher Sprache definiert wurden, formal spezifiziert werden können [EGY97]. Dabei werden einzelne Schritte beschrieben, die nötig sind um Entwurfsmuster zu implementieren. In einem prototypischen Tool wurden die Einzelschritte automatisiert umgesetzt und Entwurfsmuster als Sequenz dieser Schritte implementiert.

Darauf aufbauend entwickelte Ó Cinnéide eine Methode um Entwurfsmuster als eine Sequenz von Minipattern zu beschreiben [ÓCN99; ÓC01; ÓCN01]. Ein Minipattern beschreibt eine Teilaufgabe eines Entwurfsmusters, wie beispielsweise das Kapseln der Objekterstellung, das Extrahieren einer Abstraktion aus einer konkreten Klasse oder den Austausch von Referenzen auf eine konkreten Klasse durch eine Referenz auf ihre Abstraktion. Für die Umsetzung jedes Minipatterns wurde eine wiederverwendbare Minitransformation erstellt. Für jede Minitransformation wurden Vor- und Nachbedingungen definiert und begründet, weshalb sie das ursprüngliche Verhalten des Systems nicht ändern. Da ein Entwurfsmuster durch eine Reihe verhaltensbewahrender Minitransformationen realisiert wird, ist auch die gesamte Transformation verhaltensbewahrend. Der Fokus der Arbeit liegt auf dem Einfügen von vorgegebenen Entwurfsmustern an vom Entwickler definierten Stellen. Das Einfügen von Entwurfsmustern ohne den Einfluss eines Entwicklers will die Methode nicht bewerkstelligen.

Jeon et al. fokussierten das Erkennen von Stellen im Quellcode, an denen Entwurfsmuster sinnvoll und ohne Zutun eines Entwicklers eingefügt werden können [JLB02]. Dazu ermittelten sie Regeln, die mithilfe einfacher, aus Quellcode ausles- bzw. ableitbarer Eigenschaften geprüft werden. Die einfachen Regeln wurden zu komplexeren Regeln zusammengeschaltet, die auf Code-Stellen und für sie geeignete Entwurfsmuster hinweisen. Um Entwurfsmuster nur an Stellen einzufügen, an denen ihre Vorteile auch voll ausgenutzt werden können, prüften sie basierend auf der Versionshistorie der Software, ob an den gefunden Stellen bisher Änderungen durchgeführt wurden, denen die Entwurfsmuster zugute gekommen wären. Gefundene Entwurfsmusterkandidaten wurden mittels der Transformationen von Ó Cinnéide umgesetzt.

Gomes et al. erstellten einen erfahrungsbasierten Ansatz zum Einfügen von Entwurfsmustern in ein Systemdesign, das in Unified Modeling Language (UML) vorliegt [GPP+02]. Dabei wurde eine Erfahrungsdatenbank angelegt, die einerseits UML-Klassendiagramme von Systemteilen enthält, welche um Entwurfsmuster erweitert wurden, als auch das Design desselben Systemteils nach dem Einfügen der Entwurfsmuster. Mittels dieser Erfahrungsdatenbank werden Systeme auf Stellen hin untersucht, die einem bekannten Design ähneln, dass später um Entwurfsmuster erweitert wurde. Das Design der identifizierten

Stellen wurde dann auf die beteiligten Akteure des bekannten Designs übertragen, um die Rollen der einzelnen Klassen beim Entwurfsmuster zu erkennen. Daraufhin wird das Design so umgestellt, dass es dem Design des bereits bekannten und um ein Muster erweiterten Designs aus der Erfahrungsdatenbank entspricht. Nachteilig an dieser Methode ist, dass erst eine Erfahrungsdatenbank angelegt und gefüllt werden muss. Die Qualität der eingefügten Entwurfsmuster hängt entscheidend von der Qualität, der in der Datenbank vorliegenden Entwürfe, ab. Des Weiteren behandelt der Ansatz lediglich UML-Modelle, welche wiederum nicht alle Informationen betrachten, die im Quellcode vorliegen.

Amoui [AMAL06] und Jensen [Jen10] entwickelten Ansätze, welche Entwurfsmuster über evolutionärer Algorithmen in Systementwürfe einfügen. Dabei wird das Einfügen von Entwurfsmustern auf ein Optimierungsproblem reduziert. UML-Diagramme von Systemen werden über kleine Mutationen verändert und mittels einer Fitness-Funktion bewertet. Die Fitness-Funktionen sind so ausgewählt, dass Entwürfe bevorzugt werden, in denen durch die Mutationen Entwurfsmuster entstanden sind. Die veränderten Systemdesigns werden über viele Iterationen hinweg erneut verändert und bewertet. Am Ende werden die von der Fitness-Funktion am höchsten bewerteten Designs verwendet. In diesen Designs sind oftmals Entwurfsmuster entstanden. Nachteilig an dieser Methode ist erneut die Beschränkung auf UML-Diagramme als Informationsträger, die Zufälligkeit der entstandenen Entwurfsmuster und die sehr lange Berechnungszeit (z.T. mehrere Tage).

Shlezinger et al. haben ein Framework erstellt, welches verschiedene entwurfsmusterbezogene Modelle auf UML-Basis enthält [SRBD10]. Zum einen sind Problemmodelle enthalten, die beschreiben welche Probleme es in Designentwürfen zu lösen gilt. Des Weiteren sind Modelle enthalten, die beschreiben wie Probleme in verschiedenen Kontexten über Entwurfsmuster gelöst werden können. Als Brücke zwischen Problem- und Lösungsmodellen sind Modelle beinhaltet, die beschreiben wie ein Problemmodell in ein zugehöriges Lösungsmodel überführt werden kann. Der Ansatz ist semi-automatisch, es ist ein Entwickler notwendig, der die Stellen identifiziert, an denen Entwurfsmuster eingefügt werden sollen.

Christopoulou et al. beschreiben ein Verfahren zum automatisierten Einfügen des *Strategie*-Entwurfsmusters in Quellcode [CGZS12]. Dabei werden Bedingungsanweisungen, die den Wert von Instanzvariablen prüfen, auf ihre Eignung als *Strategie*-Muster-Ausgangspunkt hin untersucht. In den Bedingungsanweisungen enthaltene Code-Blöcke werden ggf. in eigene Methoden einer generierten Strategie-Klasse ausgelagert und die Methoden anstelle der Bedingungsanweisungen polymorph aufgerufen. Dieses Verfahren wurde in einem Plug-In für die Entwicklungsumgebung Eclipse umgesetzt, welches jedoch

aufgrund von Programmierfehlern nicht nutzbar ist. Das Plug-In funktioniert prinzipiell vollautomatisch. Ein Entwickler legt lediglich den Suchbereich fest, das Plug-In sucht in diesem Bereich selbstständig nach Möglichkeiten zum Einfügen des *Strategie*-Musters und der Entwickler muss das Einfügen des Musters an den Stellen nur noch bestätigen.

Mit Ausnahme des Ansatzes von Christopoulou wird in keiner der Arbeiten überprüft, welchen Einfluss das Einfügen der Entwurfsmuster auf die Wartbarkeit hat. Christopoulou überprüft in seiner Arbeit zwar die Auswirkungen auf den Quellcode, beschränkt sich dabei jedoch ausschließlich auf die Größe und die Komplexität von Methoden, in denen das *Strategie*-Muster angewendet wurde. Es liegt in der Natur der vorgenommenen Änderungen, dass die Größe und die Komplexität abnehmen, da ausschließlich Code von diesen Methoden in neue Methoden verschoben wird. Da neu erstellte Methoden und Klassen nicht betrachtet werden, liefern die Daten in der Studie keine objektive Aussage über die Wartbarkeit des Gesamtsystems.

Zusammenfassung

In diesem Kapitel wurde der Begriff der Wartbarkeit erläutert, verschiedene Kriterien vorgestellt, welche maßgeblichen Einfluss auf die Wartbarkeit haben und Maßnahmen zur Verbesserung der Wartbarkeit aufgeführt. Anschließend wurden verschiedene Software-Metriken vorgestellt, die in der Lage sind besagte Kriterien zu messen. Mittels dieser Metriken soll später nachvollzogen werden können, ob die Wartbarkeit verbessert durch Entwurfsmuster wurde. Nachfolgend wurde eine Einführung in Entwurfsmuster gegeben und aufgezeigt, wie diese die Wartbarkeit von Quellcode beeinflussen. Des Weiteren wurden verwendete Refactorings aufgeführt und abschließend bekannte Verfahren zum semi-automatisierten Einfügen von Entwurfsmustern in Software-Entwürfe bzw. Quellcode vorgestellt.

3 Umsetzung der Entwurfsmusterautomatisierung

Übersicht

Das folgende Kapitel geht darauf ein, wie die Entwurfsmuster automatisiert in den Quellcode eingefügt werden können, um so die Wartbarkeit des Codes zu verbessern. Abschnitt 3.1 erklärt zunächst welche Entwurfsmuster zur Verbesserung der Wartbarkeit automatisiert werden sollen und warum diese ausgewählt wurden. In Abschnitt 3.2 wird das verwendete Verfahren zum Erkennen geeigneter Stellen für Entwurfsmuster (im Folgenden Spots genannt) erklärt. Abschnitt 3.3 erläutert anschließend die Umsetzung der automatischen Refactorings, mit denen an den zuvor ermittelten Spots Quellcode so umstrukturiert wird, dass das passende Entwurfsmuster eingefügt wird. Abschließend werden in Abschnitt 3.4 verschiedene Implementierungsentscheidungen erläutert.

3.1 Auswahl verwendeter Entwurfsmuster

Zunächst wird ermittelt, welche Entwurfsmuster für die Automatisierung in dieser Arbeit ausgewählt werden. Dafür wird zuerst betrachtet, welche Entwurfsmuster überhaupt einen Einfluss auf die vorgestellten Wartbarkeitskriterien haben. Entwurfsmuster ohne einen entsprechenden Einfluss werden herausgefiltert. Anschließend wird geprüft, welche Entwurfsmuster sich vollständig automatisieren lassen. Dies schließt sowohl das automatisierte Detektieren von geeigneten Spots als auch das Refactoring des Codes zu entsprechenden Entwurfsmusterstrukturen ein, so dass ein Entwickler vor, während und nach dem Refactoring keine weiteren Aktionen durchführen muss. Anhand dieser Betrachtungen werden Entwurfsmuster verworfen, die ohne weitere Interaktion mit einem Entwickler nicht vollständig automatisierbar sind. Aus den verbleibenden Entwurfsmustern werden jene ausgewählt, für die ein höherer Einfluss auf die Wartbarkeit vermutet wird. Dabei soll nach Möglichkeit aus jeder Gruppe (Erzeugungs-, Struktur- und Verhaltensmuster) eines automatisiert werden, um eine Auswahl zu bieten, die unterschiedliche Ansätze widerspiegelt und somit ein möglichst breites Spektrum an Lösungsideen abdeckt.

3.1.1 Vorauswahl anhand von erwartetem Einfluss auf die Wartbarkeit von Quellcode

Jedes Entwurfsmuster löst ein anderes Problem. Da lediglich der Einfluss von Entwurfsmustern auf die Wartbarkeit untersucht wird, sollen keine Entwurfsmuster automatisiert werden, die solche Probleme lösen, die nicht mit den in 2.1.2 genannten Wartbarkeitskriterien in Verbindung stehen, da keine messbare Verbesserung eines betrachteten Kriteriums zu erwarten ist. Die Entwurfsmuster *Fliegengewicht*, *Proxy* und *Interpreter* lassen sich ausschließen, da sie keine der genannten Kriterien verändern, wie Tabelle 2.5 bereits gezeigt hat.

3.1.2 Vorauswahl anhand bereits ermittelter Automatisierbarkeit

Ó Cinnéide hat die Automatisierbarkeit der verschiedenen GoF-Entwurfsmuster untersucht [ÓCN01]. Tabelle 3.1 zeigt seine Einschätzungen dazu, wie gut das Einfügen von Entwurfsmustern unterstützt werden kann. Dabei bezieht er sich darauf, ob für die Muster eine konkrete Vorbedingung gefunden und ein funktionierendes Refactoring entwickelt werden konnte. Vorausgesetzt wird dabei immer, dass der Entwickler die Code-Stelle und die Rollen der beteiligten Klassen selbst festlegt. Außerdem bezieht Ó Cinnéide sich in den meisten Fällen darauf, dass nicht schlechte Wartbarkeit der Grund für eine Überarbeitung ist, sondern dass neue Anforderungen an funktionierenden Code gestellt wurden, die ein vorher nicht benötigtes Maß an Flexibilität erfordern (ein Beispiel dafür wird in Abschnitt 3.3.1 näher erläutert). Damit verfolgt er bei den Refactorings hauptsächlich das Ziel, die durch ein Entwurfsmuster vorgegebenen Strukturen zu schaffen, um die neuen Anforderungen anschließend manuell als Entwurfsmuster umsetzen zu können. Die vollständige Umstellung von fertigem Code ohne Entwurfsmuster zu fertigem Code mit Entwurfsmustern steht bei Ó Cinnéide nicht im Fokus.

Anhand der Tabelle 3.1 lassen sich die Muster *Fassade*, *Fliegengewicht*, *Beobachter*, *Besucher*, *Interpreter* und *Vermittler* ausschließen, da sie selbst mit Wissen des Entwicklers schwer umsetzbar sind. Abzüglich des *Proxy*-Musters, das anhand der Tabelle 2.5 bereits ausgeschlossen werden konnte, bleiben von ursprünglich 23 Entwurfsmustern noch 16 übrig: *Abstrakte Fabrik*, *Erbauer*, *Fabrikmethode*, *Prototyp*, *Singleton*, *Adapter*, *Brücke*, *Dekorierer*, *Kompositum*, *Befehl*, *Iterator*, *Memento*, *Schablonenmethode*, *Strategie*, *Zustand* und *Zuständigkeitskette*.

Zweck	Entwurfsmuster	Umsetzbarkeit
Erzeugungsmuster	Abstrakte Fabrik	Exzellent
	Erbauer	Exzellent
	Fabrikmethode	Exzellent
	Prototyp	Exzellent
	Singleton	Exzellent
Strukturmuster	Adapter	Exzellent
	Brücke	Exzellent
	Dekorierer	Teilweise
	Fassade	Unpraktikabel
	Fliegengewicht	Unpraktikabel
	Kompositum	Exzellent
	Proxy	Teilweise
Verhaltensmuster	Befehl	Teilweise
	Beobachter	Unpraktikabel
	Besucher	Unpraktikabel
	Interpreter	Unpraktikabel
	Iterator	Teilweise
	Memento	Teilweise
	Schablonenmethode	Exzellent
	Strategie	Exzellent
	Vermittler	Unpraktikabel
	Zustand	Teilweise
	Zuständigkeitskette	Exzellent

Exzellent: Nachvollziehbarer Precursor und funktionierende Transformation vorhanden
Teilweise: Nachvollziehbarer Precursor und funktionierende Transformation weitestgehend vorhanden, ggf. wenige Nacharbeiten des Entwicklers nötig
Unpraktikabel: Precursor und/oder Transformation konnten nicht gefunden werden

Tabelle 3.1.: Automatisierbarkeit von Entwurfsmustern laut [ÓCN01]

Die übrigen Muster werden nachfolgend auf ihre Eignung und ihre Umsetzbarkeit hin untersucht. Dabei ist darauf zu achten, dass die Muster ohne weiteres Zutun eines Entwicklers umgestellt werden sollen.

3.1.3 Vorauswahl anhand von vollständiger Automatisierbarkeit

Nicht alle der verbliebenen Entwurfsmuster sind vollständig automatisierbar. Die Einschätzungen Ó Cinnéides, aus dem vorherigen Abschnitt, beruhen teilweise darauf lediglich die leeren Strukturen für die Implementierung eines Entwurfsmusters zu schaffen. Nachfolgend werden exemplarisch die drei letztendlich gewählten Entwurfsmuster bzgl. ihrer vollständig autonomen Automatisierbarkeit hin betrachtet. Die Betrachtung der übrigen Muster lässt sich in Anhang A.2 nachlesen.

Abstrakte Fabrik

Die *Abstrakte Fabrik* verschiebt lediglich die Erzeugung verschiedener Objekte in eine einzelne Klasse und lässt Klienten nur über Interfaces mit den erzeugten Objekten kommunizieren. Die Interfaces können ggf. mit dem *Extract Interface*-Refactoring erzeugt werden. Das Verschieben der Objekterstellung in eine Methode bewerkstelligt das *Extract Method*-Refactoring, das Verschieben aller neu erstellten Erzeugungsmethoden in die neue Abstrakte Fabrik ist mittels *Move Method*-Refactoring möglich. Abschließend müssen nur noch alle Verweise auf konkrete Klassen durch Referenzen auf das erzeugte Interface ersetzt werden. Alle Teilschritte der Transformation sind ohne weiteres Entwicklerwissen möglich. Als mögliche Spots beschreibt [JLB02] solche Codestellen, an denen eine Klasse Instanzen von verschiedenen Klassen erstellt und zwischen ihnen eine Assoziationsbeziehung herstellt, indem ein Objekt als Attribut des anderen gesetzt wird. Dies lässt sich automatisiert ermitteln, daher kann die Abstrakte Fabrik vollständig autonom automatisiert werden.

Kompositum

Das *Komposium*-Muster hat einen positiven Einfluss auf die Komplexität eines Objekts, das vorher zwischen einem zusammengesetzten Objekt und seinen einzelnen Bestandteilen unterscheiden musste. Mit dem *Komposium*-Muster kann ein Klient sowohl das zusammengesetzte Objekt (Kompositum) als auch seine Bestandteile (Komponenten) identisch behandeln. Die Anwendung des *Komposium*-Musters würde beinhalten, Methoden von Komponenten auch auf das Kompositum anwenden zu können. Dies setzt jedoch die Interpretation von Rückgabewerten der Methoden der Komponenten voraus. Gibt eine Methode einer Komponente einen Wert zurück, so ist ohne Wissen über die Bedeutung des Wertes nicht ermittelbar, was die Methode für das Kompositum zurückgeben soll. Es kann nicht ermittelt werden, wie die Werte der Komponenten verknüpft werden müssen (beispielsweise ob numerische Werte addiert, multipliziert o.ä. werden müssen). Besitzt jedoch keine Methode einen Rückgabewert, so ist die Automatisierung des *Komposium*-Musters möglich. Somit würde das Kompositum bei Aufruf einer Komponenten-Methode diese Methode einfach auf alle seine Komponenten anwenden, ohne Rückgabewerte interpretieren zu müssen.
Ó Cinnéide geht für die Automatisierung des *Komposium*-Musters von einer iterierbaren Container-Klasse (z.B. einer Liste) als zusammengesetztem Objekt aus [ÓC01]. Zunächst muss für die Transformation eine Abstraktion der Komponentenklasse erstellt werden (*Extract Interface*-Refactoring). Diese wird um die Methoden der Container-Klasse er-

weitert (beispielsweise Methoden zum Hinzufügen und Entfernen von Elementen). Die Komponenten implementieren die so entstandene Schnittstelle, wobei die Komponenten-Methoden bereits implementiert sind und Container-Methoden leer implementiert werden. Anschließend wird eine neue Kompositumklasse erstellt, welche die vorher erzeugte Kompositum-Schnittstelle implementiert und eine Referenz auf die ursprünglich genutzte Container-Klasse in einer privaten Instanzvariable speichert. Die Container-Methoden werden an das gespeicherte Container-Objekt delegiert, die Komponenten-Methoden hingegen delegieren den Aufruf rekursiv an alle im Container enthaltenen Objekte. Nach diesen Transformationen ist die Struktur des *Komposium*-Musters fertig erstellt. Abschließend müssen auf das Kompositum bezogene Codefragmente aus dem Klienten in die Komponenten-Methoden verschoben bzw. durch den Aufruf der Methode des Kompositums ersetzt werden. Dies ist in Abbildung 3.1 beispielhaft dargestellt. Wie dort ersichtlich ist, wurde vorher zwischen einzelnen und zusammengesetzten Objekten unterschieden. Nach der Umstellung kann der Klient dieselben Operationen auf einzelnen und zusammengesetzten Objekten durchführen.

Zustand

Im Fokus des Refactorings zum *Zustand*-Muster steht das Ersetzen einer den Zustand der Klasse per Enumeration repräsentierenden Instanzvariable und anhand dieser Variable durchgeführte Bedingungsanweisungen. Diese werden durch eine den Zustand repräsentierenden Klasse und polymorphe Methodenaufrufe dieser Klasse ersetzt. Für diese Transformation wird zunächst das bereits erklärte *Replace Type Code with State/Strategy*-Refactoring angewendet, um zustandsrepräsentierende Codefragmente in eigene Klassen zu verschieben. Anschließend werden Bedingungsanweisungen per *Replace Conditional with Polymorphism*-Refactoring durch Methodenaufrufe der vorher erstellten Klassen ersetzt. Die Spots lassen sich durch die Suche nach Instanzvariablen mit einem Enumerationsdatentyp finden, die an Bedingungsanweisungen beteiligt sind. Somit lassen sich alle nötigen Informationen aus dem Quellcode ermitteln – das *Zustand*-Muster ist automatisierbar [CGZS12].

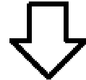
Übersicht über automatisierbare Entwurfsmuster

Anhand der Betrachtungen bzgl. der Automatisierbarkeit der einzelnen Muster, deren Ergebnisse in Tabelle 3.2 zusammengefasst sind, lassen sich die Entwurfsmuster *Fabrikmethode*, *Adapter*, *Brücke*, *Dekorierer*, *Iterator*, *Memento*, *Schablonenmethode* und *Zuständigkeitskette* ausschließen. Für die einzelnen Ausschlussgründe wird auf Anhang A.2

```
private Collection<FooBar> fooCollection = new ArrayList<IFooCompositum>();
private FooBar singleFoo = new FooBar();

// ...

public void printCollection() {
    for (FooBar fooBar : fooCollection) {
        fooBar.print();
    }
}
public void printSingle() {
    singleFoo.print();
}
```



```
private IFooCompositum fooCompositum = new FooCompositum();
private IFooCompositum singleFoo = new FooBar();

// ...

public void printCollection() {
    fooCompositum.print();
}
public void printSingle() {
    singleFoo.print();
}


class FooCompositum implements IFooCompositum {
    private Collection<IFooCompositum> fooCollection = new ArrayList<IFooCompositum>();

    @Override
    public void print() {
        // invoke method for all components
        for (IFooCompositum foo : fooCollection) {
            foo.print();
        }
    }

    @Override
    public void add(IFooCompositum foo) {
        // delegate to collection
        fooCollection.add(foo);

    }
    @Override
    public void remove(IFooCompositum foo) {
        // delegate to collection
        fooCollection.remove(foo);

    }
}

interface IFooCompositum {
    void print();
    void add(IFooCompositum foo);
    void remove(IFooCompositum foo);
}
```

Vorher: Der Klient unterscheiden zwischen einem iterierbaren, zusammengesetzten Objekt und dessen einzelnen Elementen, indem durch das zusammengesetzte Objekt iteriert wird.
Nachher: der Klient behandelt das zusammengesetzte und das Einzelobjekt identisch – die Unterscheidung wurde hinter dem Interface in der Kompositum- bzw. Komponentenklasse versteckt.

Abbildung 3.1.: Refactoring zum Kompositum-Entwurfsmuster

verwiesen. Somit verbleiben die Muster*Abstrakte Fabrik*, *Erbauer*, *Singleton*, *Kompositum*, *Befehl*, *Strategie* und *Zustand* als Kandidaten für die Automatisierung.

Zweck	Entwurfsmuster	Autonom automatisierbar
Erzeugungsmuster	Abstrakte Fabrik	Ja
	Erbauer	Ja
	Fabrikmethode	Nein
	Prototyp	Nein
	Singleton	Ja
Strukturmuster	Adapter	Nein
	Brücke	Nein
	Dekorierer	Nein
	Kompositum	Ja
Verhaltensmuster	Befehl	Ja
	Iterator	Nein
	Memento	Nein
	Schablonenmethode	Nein
	Strategie	Ja
	Zustand	Ja
	Zuständigkeitskette	Nein

Tabelle 3.2.: Übersicht über autonom automatisierbare Entwurfsmuster

3.1.4 Vorauswahl anhand von erwartetem Einfluss auf die Wartbarkeit

Um eine Auswahl möglichst unterschiedlicher Entwurfsmuster zu automatisieren, soll aus den verschiedenen Gruppen der Entwurfsmuster (Erzeugenden-, Struktur- und Verhaltensmuster) jeweils eines automatisiert werden.

Erzeugungsmuster

Von den ursprünglich fünf Erzeugungsmustern wurden die drei Muster *Abstrakte Fabrik*, *Erbauer* und *Singleton* als vollständig autonom automatisierbar erachtet. Für sie werden folgende Einschätzungen getroffen:

- Abstrakte Fabrik: Dieses Muster entkoppelt ggf. mehrere Klassen, je nachdem wie viele Objekte in die Fabrik aufgenommen werden. Mit einer Verringerung der Komplexität ist hingegen nicht zu rechnen, da die durch das Muster überflüssigen Konsistenzprüfungen nicht automatisiert erkannt und durch ein Refactoring entfernt werden.

- Erbauer: Dieses Muster verringert die Komplexität eines Klienten vermutlich am meisten, da ein komplexer Zusammenbau von Objekten ggf. komplett aus dem Klienten entfernt werden kann. Es entkoppelt den Klienten dabei ggf. von mehreren Klassen. Jedoch kann das Hauptanliegen des Musters – die Erbauer verschiedene Repräsentationen erstellen zu lassen – nicht ohne weiteres erfüllt werden, da automatisiert erkannt werden müsste, welche verschiedenen Repräsentationen behandelt werden sollen.
- Singleton: Das Singleton-Muster entkoppelt ggf. eine Klasse von einer anderen. Eine Verringerung der Komplexität eines Klienten wird nicht erwartet, da das Erkennen und Entfernen von Zugriffskontrollmechanismen in Klienten nicht Teil von Refactorings ist. Daher wird ein geringer Einfluss auf die Wartbarkeit vermutet.

Anhand dieser Einschätzungen wird die Abstrakte Fabrik zur Automatisierung gewählt, da durch die Entkopplung vieler Klassen der größte positive Einfluss auf die Wartbarkeit vermutet wird.

Strukturmuster

Von den ursprünglich sieben Strukturmustern ist lediglich das *Kompositum*-Muster als automatisierbar eingeschätzt worden. Trotz der Einschränkung, dass die Automatisierung nur für Klassen mit rückgabewertlosen Methoden vollzogen werden kann, und der daher vermuteten geringen Anzahl potentieller Spots für das Refactoring, wird es mangels Alternativen in dieser Gruppe für die Automatisierung ausgewählt.

Verhaltensmuster

Von den ursprünglich elf Verhaltensmustern wurden die drei Muster *Befehl*, *Strategie* und *Zustand* als vollständig autonom automatisierbar erachtet. Für sie werden folgende Einschätzungen getroffen:

- Befehl: Das Befehl-Muster entkoppelt jeweils nur eine Klasse von einer anderen. Weitere Wartbarkeitsverbesserungen sind durch ein Refactoring nicht zu erwarten. Daher wird ein geringer Einfluss auf die Wartbarkeit angenommen.
- Strategie: Das Strategie-Muster verringert die Komplexität eines Klienten stark, da komplette Algorithmen in eigene Klassen gekapselt werden. Jedoch sind Familien von Algorithmen schwer zu identifizieren, so dass lediglich einzelne Algorithmen gekapselt werden würden, was nicht der Fokus des Musters ist.

- Zustand: Das Muster verringert die Komplexität des Klienten ebenfalls stark. Im Gegensatz zum Strategie-Muster kann der Fokus des *Zustand*-Musters jedoch besser vollständig automatisiert behandelt werden – die Zustände der Klasse sind einfacher zu identifizieren, als Familien von Algorithmen.

Anhand dieser Einschätzungen wird das *Zustand*-Muster zur Automatisierung gewählt, da ein höherer Einfluss auf die Wartbarkeit als beim *Befehl*-Muster und eine einfachere Umsetzung gegenüber dem *Strategie*-Muster erwartet wird.

Um die nun ausgewählten Entwurfsmuster automatisiert in Quellcode einzufügen müssen zwei Schritte unternommen werden:

1. Detektieren geeigneter Spots für ein Entwurfsmuster
2. Transformation des Codes mittels Refactorings, bis der Code die Struktur des Entwurfsmusters aufweist.

Diese Schritte werden in den nachfolgenden Abschnitten erläutert.

3.2 Verfahren zur Ermittlung geeigneter Spots für Entwurfsmuster

Für die Detektierung der Spots wurde der Ansatz von Jeon et al ([JLB02]) übernommen, da dieser auf Quellcode anwendbar ist, während andere Ansätze auf der Auswertung von UML-Diagrammen basieren. Die Funktionsweise wird nachfolgend näher erläutert.

3.2.1 Spot-Detektierungsansatz von Jeon

Der Ansatz repräsentiert ein Programm $\mathcal{P}$ als ein 8-Tupel $(\mathcal{C}, \mathcal{I}, \mathcal{M}, \mathcal{R}, \mathcal{L}, \mathcal{V}, \phi, \eta)$, wobei die einzelnen Elemente wie folgt beschrieben sind:

- $\mathcal{C}$ ist eine Menge von Klassen
- $\mathcal{I}$ eine Menge von Schnittstellen
- $\mathcal{M} \subseteq \mathcal{C}_o \times TEXT \times \mathcal{C}_r \times \mathcal{C}_{p1} \times .. \times \mathcal{C}_{pi} \times .. \times \mathcal{C}_{pn}$ ist eine Relation, die eine beinhaltende Klasse $\mathcal{C}_o$, einen Methodennamen $TEXT$, einen Rückgabetypen $\mathcal{C}_r$ und eine Parameterliste $\mathcal{C}_{p1}..\mathcal{C}_{pn}$ in Beziehung setzt und somit eine Methode beschreibt.

- ➢ $\mathcal{R} \subseteq (\mathcal{C}\times\mathcal{C})\cup(\mathcal{C}\times\mathcal{M})\cup(\mathcal{M}\times\mathcal{C})\cup(\mathcal{M}\times\mathcal{M})\cup(\mathcal{I}\times\mathcal{I})\cup(\mathcal{C}\times\mathcal{I})\cup(\mathcal{M}\times\mathcal{C}\times\mathcal{C})\cup(\mathcal{M}\times\mathcal{C}\times\mathcal{C}\times TEXT)$ beschreibt Beziehungen zwischen Klassen, Schnittstellen und Methoden, wie Assoziation, Aggregation, Vererbung und Implementierung, und Relationen zwischen Klassen und Methoden, wie Aufrufe, Objekterzeugung, Rückgabewerte oder Parameterübergabe-Abhängigkeiten.
- ➢ $\mathcal{L} : \mathcal{C}\cup\mathcal{I} \to TEXT$ ist eine Funktion, die eine Klasse $\mathcal{C}$ oder eine Schnittstelle $\mathcal{I}$ auf ihren Namen $TEXT$ abbildet, wobei $TEXT$ eine Menge von Zeichenketten darstellt.
- ➢ Die Funktion $\mathcal{V} : \mathcal{R} \to VISIBILITY = \{private, protected, public\}$ bildet eine Relation $\mathcal{R}$ auf ein Element der Sichbarkeitsmenge $VISIBILITY$ ab.
- ➢ $\phi : \mathcal{M} \to \mathcal{C}_o \times TEXT \times \mathcal{C}_r \times \mathcal{C}_{p1} \times ..\mathcal{C}_{pn}$ ist eine Funktion, die eine Methode $\mathcal{M}$ auf ihre beinhaltende Klasse $\mathcal{C}_o$, den Methodennamen $TEXT$, den Rückgabewert $\mathcal{C}_r$ und eine Parameterliste $\mathcal{C}_{p1}..\mathcal{C}_{pn}$ abbildet. $\phi(m)_o$ benennt die beinhaltende Klasse der Methode m, $\phi(m)_n$ stellt den Namen von m dar und $\phi(m)_i$ den i-ten Parameter der Methode m.
- ➢ Die Funktion $\eta : \mathcal{C}\times\mathcal{C}\times TEXT \to MULTIPLICITY = \{zero_or_one, one, one_or_more\}$ bildet eine Relation zwischen Klasse $c1$ und $c2$ mit dem Namen $TEXT$ auf ein Element der Menge $MULTIPLICITY$ ab.

Anhand dieser Tupel beschreiben Jeon et al. niedere Prädikate dreier verschiedener Typen:

- ➢ Prädikate, die direkt aus Java-Quellcode ermittelt werden können, z.B. "Klasse A erstellt eine Instanz der Klasse B" oder "Methode M gibt ein Objekt der Klasse C zurück".
- ➢ Prädikate, die anhand von Analysen und Heuristiken geschlussfolgert werden können, z.B. "wenn Klasse A eine Assoziationsbeziehung mit Klasse B hat und Klasse A eine Instanz von Klasse B erstellt, dann besteht eine Aggregationsbeziehung zwischen Klasse A und B".
- ➢ Prädikate, die anhand von versionshistorischen Daten des Programms ermittelt werden können, z.B. "Klasse A aus Version V wurde in einer späteren Version V’ verändert".

Die versionshistorischen Prädikate werden genutzt, um nur solche Spots einem Refactoring zu unterziehen, die in der Vergangenheit bereits (isomorph) geändert wurden. Damit soll sichergestellt werden, dass Entwurfsmuster nur dort eingefügt werden, wo weitere Änderungen wahrscheinlich sind. Dies soll verhindern, dass Programme durch Entwurfsmuster verkompliziert werden, ohne dass die Flexibilität der Entwurfsmuster auch genutzt wird. Obwohl dieser Gedanke als richtig anzunehmen ist, werden die Prädikate des dritten

Typs in dieser Arbeit nicht verwendet, da lediglich der messbare Einfluss auf die Wartbarkeit untersucht werden soll. Die Wahrscheinlichkeit zukünftiger Änderungen ist bei der Messung von Momentaufnahmen, wie es bei Metriken der Fall ist, nicht von Bedeutung. Die Prädikate des zweiten Typs wurden in dieser Arbeit ebenfalls nicht verwendet, da sie Eigenschaften bzw. Relationen beschreiben, die bei der Detektierung der gewählten Entwurfsmuster nicht benötigt wurden. Auf diese Prädikate wird daher nicht weiter eingegangen.
Die Prädikate des ersten Typs wurden hingegen verwendet. Jeon et al. beschrieben damit verschiedenste Relationen. Die nachfolgenden Prädikate wurden für die Spot-Detektierung der gewählten Entwurfsmuster verwendet:

- $\boldsymbol{creates(m : \mathcal{M}, c : \mathcal{C})}$: Methoden und Klassen mit "$(m, c) \in \mathcal{R}$", wobei die Methode m ein Objekt der Klasse c mittels des new-Schlüsselworts erstellt.
- $\boldsymbol{creates(c1 : \mathcal{C}, c2 : \mathcal{C})}$: Klassen, die die Bedingung "$(c1, c2) \in \mathcal{R} \wedge \exists m \in \mathcal{M}(\phi(m)_o = c1 \wedge creates(m, c2))$" erfüllen.
- $\boldsymbol{calls(m1 : \mathcal{M}, m2 : \mathcal{M})}$: Methoden, welche die Bedingung "$(m1, m2) \in \mathcal{R}$" erfüllen und in Methode $m1$ ein Ausdruck existiert, der die Methode $m2$ aufruft.
- $\boldsymbol{make_aggre(m : \mathcal{M}, c1 : \mathcal{C}, c2 : \mathcal{C})}$: Methoden und Klassen, die die Bedingung "$(m, c1, c2) \in \mathcal{R} \wedge \exists m1 \in \mathcal{M}(calls(m, m1) \wedge \phi(m1)_o = c1 \wedge \exists i \leq n(\phi(m1)_i = c2))$" erfüllen.
- $\boldsymbol{make_assoc(m : \mathcal{M}, c1 : \mathcal{C}, c2 : \mathcal{C}, t : TEXT)}$: wenn die Bedingungen "$(m, c1, c2, t) \in \mathcal{R}$" erfüllt wird und ein Ausdruck in Methode m existiert, der das Attribut t von $c1$ auf ein Objekt der Klasse $c2$ setzt.

Durch Verkettungen der niederen Prädikate können höhere Prädikate erstellt werden, die Hinweise darauf geben, dass Klassen bzw. Klassenverbünde Eigenschaften aufweisen, die als Voraussetzung für das Einfügen von Entwurfsmustern gelten. Ein Spot für die *Abstrakte Fabrik* lässt sich durch das folgende höhere Prädikat implizieren [JLB02]:
$\exists c \in \mathcal{C}, PC \subseteq \mathcal{C}(x \in PC(creates(c, x) \wedge (\exists t \in TEXT, m \in \mathcal{M}, y \in PC(x \neq y \wedge \phi(m)_o = c \wedge ((make_assoc(m, x, y, t) \vee make_assoc(m, y, x, t)) \vee (make_aggre(m, x, y) \vee make_aggre(m, y, x))))))) \rightarrow CandidateConcreteFactory(c, PC)$

Dieses Prädikat beschreibt Spots, bei denen eine Klasse Instanzen von mindestens zwei verschiedenen Klassen erzeugt und eine Assoziations- oder Aggregationsbeziehung zwischen ihnen herstellt, indem ein Objekt als Attribut eines anderen Objekts gesetzt oder als Parameter einer Methoden eines anderen Objekts verwendet wird. Die Assoziations- bzw. Aggregationsbeziehung impliziert eine Zusammengehörigkeit zwischen den Objek-

ten. Sie werden als Mitglieder eine Produktfamilie angesehen.

Komposium- und *Zustand*-Entwurfsmuster wurden von Jeon et al. nicht definiert. Für diese Entwurfsmuster wurden eigene Prädikate erstellt, die im nachfolgenden Abschnitt vorgestellt werden.

3.2.2 Erweiterung des Spot-Detektierungsansatzes

In Abschnitt 3.1.3 wurden für das *Komposium*- und *Zustand*-Muster bereits informell vorgestellt, wie Spots auffindbar sind. Diese werden nachfolgend formell durch selbsterstellte Prädikate definiert. Zunächst werden folgende niedere Prädikate ergänzt:

- $\boldsymbol{isAttribute(t : TEXT, c : \mathcal{C})}$: t ist der Name eines Attributs der Klasse c.
- $\boldsymbol{isEnum(c : \mathcal{C})}$: Die Klasse c erweitert die Klasse „Enum“.
- $\boldsymbol{isEnum(t : TEXT, c : \mathcal{C})}$: Die Bedingung "$isAttribute(t, c)$" ist erfüllt, die Klasse des von t repräsentierten Attributs ist $c1 \in \mathcal{C}$ und die Bedingung $isEnum(c1)$ ist erfüllt.
- $\boldsymbol{genericCollection(t : TEXT, c1 : \mathcal{C}, c2 : \mathcal{C})}$: die Bedingung "$isAttribute(t, c1)$" wird erfüllt, die Klasse des von t repräsentierten Attributs implementiert das „Collection“-Interface[1] und die Typvariable der Collection ist $c2$.
- $\boldsymbol{switches(m : \mathcal{M}, t : TEXT)}$: die Bedingung "$isAttribute(t, \phi(m)_o)$" wird erfüllt und t ist innerhalb von m der überprüfte Ausdruck einer Switch-Anweisung
- $\boldsymbol{switches(c : \mathcal{C}, t : TEXT)}$: die Bedingung "$\exists m \in \mathcal{M}(\phi(m)_o = c \wedge switches(m, t)))$" wird erfüllt
- $\boldsymbol{conditional(m : \mathcal{M}, t : TEXT)}$: die Bedingung "$isAttribute(t, \phi(m)_o)$" wird erfüllt und t ist innerhalb von m der überprüfte Ausdruck einer *if*-Anweisung
- $\boldsymbol{conditional(c : \mathcal{C}, t : TEXT)}$: die Bedingung "$\exists m \in \mathcal{M}(\phi(m)_o = c \wedge conditional(m, t)))$" wird erfüllt

Mittels dieser neuen Prädikate lassen sich Spots für das *Komposium*-Muster wie folgt detektieren:

$$\exists c1, c2 \in C, \exists t \in TEXT(genericCollection(t, c1, c2) \wedge (\forall m \in M((\phi(m)_o = c2) \rightarrow (\phi(m)_r = void)))) \rightarrow CandidateComposite(c1, t, c2)$$

[1] Das Interface „java.util.Collection“ ist ein in Java vordefinierter Datentyp.

Dieses höhere Prädikat beschreibt Spots, bei denen eine Klasse $c1$ eine Instanzvariable t besitzt, die als Datentyp eine Collection einer anderen Klasse $c2$ besitzt. Des Weiteren darf keine Methode der Klasse $c2$ einen Rückgabewert haben. Für solche Spots kann eine einheitliche Schnittstelle aus der Klasse $c2$ und der verwendeten Collection erstellt werden, welche die Klasse $c1$ anschließend nutzt, um so nicht mehr zwischen der Collection und der von ihr verwalteten Elemente unterscheiden zu müssen.

Als höheres Prädikat zur Detektion von Spots für das *Zustand*-Muster wird folgender Ausdruck definiert:
$\exists c \in \mathcal{C}, \exists t \in TEXT(isEnum(t,c) \wedge (switches(c,t) \vee conditional(c,t))) \rightarrow CandidateState(c,t)$

Das Prädikat beschreibt Spots, bei denen eine Klasse c eine Instanzvariable t besitzt, welche eine Enumeration als Datentyp besitzt und die innerhalb einer Methode der Klasse in einer Switch- oder *if*-Anweisung geprüft wird. Solch eine Instanzvariable impliziert eine Zustandsabfrage innerhalb der Klasse.

Die genannten Spots beschreiben stets die Mindestanforderung für das Einfügen eines Entwurfsmusters. Spots, die anhand dieser Prädikate gefunden werden, müssen ggf. auf weitere ausschließende Faktoren geprüft werden. Jeon et al. nutzen für die Refactorings der detektierten Spots den Minipattern-Ansatz von Ó Cinnéide. Die Methode von Ó Cinnéide wird auch in dieser Arbeit verwendet und im folgenden Abschnitt beschrieben.

3.3 Verfahren für entwurfsmusterbildende Refactorings

Für die Refactorings von Quellcode zu Entwurfsmustern wurde der Ansatz von Ó Cinnéide ([ÓC01]) übernommen. Seine Methode ist ebenfalls auf existierenden Quellcode anwendbar. Er definiert für die verschiedenen Entwurfsmuster Ausgangspunkte („Precursor“), die eine Code-Struktur beschreiben, welche Entwurfsmuster sinnvoll erscheinen lassen. Ein Entwurfsmuster wird in eine Menge von „Minipatterns“ – wiederkehrende Teilaufgaben eines Entwurfsmusters – zerlegt. Für jedes erkannte Minipattern wurde eine Minitransformation erzeugt, die ein Refactoring des Codes für den Teilbereich des Minipatterns darstellt. Ein Refactoring von der entwurfsmusterfreien Ausgangssituation hin zum Entwurfsmuster wird dann als Sequenz der Minitransformationen realisiert.

3.3.1 Ausgangssituationen

Die von Ó Cinnéide beschriebenen Precursor der Entwurfsmuster versuchen Situationen zu beschreiben, die weder eine lose Sammlung von Klassen ohne Beziehungen zwischen einander darstellen, noch als schlechter Code bzw. Anti-Pattern angesehen werden können [ÓC01]. Vielmehr werden Situationen beschrieben, bei der Klassen über Beziehungen zueinander verfügen, die dem aktuellen Zweck des Codes angemessen sind, jedoch zukünftige Änderungen erschweren. Beispielsweise ist eine Ausgangssituation für das *Fabrikmethode*-Muster, dass eine Klientenklasse ein Objekt einer anderen konkreten Produktklasse erstellt und referenziert. Dies ist oftmals angemessen und stellt keinen schlechten Code dar. Ergibt sich jedoch eine Anforderung, dass die Klientenklasse mit anderen Produkt-Typen arbeiten muss, so sind die Referenzen auf die konkrete Produktklasse zu unflexibel.

3.3.2 Minipatterns

Die meisten Entwurfsmuster enthalten Teilaufgaben, die auch in anderen Entwurfsmustern wiederzufinden sind. Dazu gehören beispielsweise die Erstellung bzw. Verwendung einer Abstraktion anstelle einer konkreten Klasse, oder das Kapseln einer Objekterstellung. Ó Cinnéide nennt solche identifizierten Teilaufgaben „Minipattern“ [ÓC01]. Folgende Minipattern wurden definiert und in dieser Arbeit verwendet:

- **Abstraction**: Eine Produktklasse muss eine Schnittstelle besitzen, die sich von Klienten genauso verwenden können, wie die Produktklasse selbst.
- **EncapsulateConstruction**: Eine Klasse kapselt die Erzeugung einer Produktklasse in einer (überschreibbaren) Konstruktionsmethode[2] – die Produktklasse wird nur noch über diese Methode instanziiert.
- **AbstractAccess**: Eine Klasse referenziert eine Produktklasse ausschließlich über die Schnittstelle der Produktklasse und entkoppelt sich somit von der Produktklasse.
- **PartialAbstraction**: Eine Erzeugerklasse erbt von einer abstrakten Klasse, welche die Erzeugung von Produktklassen über abstrakte Methoden vorschreibt
- **Delegation**: Ein Teil einer existierenden Klasse wird in eine neue Komponentenklasse verschoben. Aufrufe der verschobenen Teile werden von der existierenden Klasse an die neuen Komponentenklasse delegiert.

[2]Die Konstruktionsmethoden ersetzen die Konstruktoren der Produktklassen nicht. Sie weisen aber dieselben Signaturen auf und können somit exakt wie die Konstruktoren genutzt werden.

Zu jedem Minipattern wurde eine „Minitransformation“ erstellt, welche die Minipatterns in eine entsprechende Programmeinheit einfügt. Ó Cinnéide führt für jede Minitransformation Vor- und Nachbedingungen ein. Zu den Vorbedingungen zählen beispielsweise, dass keine Klassen oder Interfaces mit denselben Namen wie neu zu erstellende Typen selben im Gültigkeitsbereich vorhanden sein dürfen, oder dass Klassen, aus denen Abstraktionen erstellt werden, keine statischen Methoden oder öffentliche Felder besitzen dürfen. Zudem zeigt Ó Cinnéide auf, dass das Verhalten der Software bei Einhaltung der Vorbedingungen nicht verändert wird. Da die Entwurfsmuster-Refactorings eine Verkettung der verhaltensbewahrenden Minitransformationen sind, sind auch die komplexen Entwurfsmuster-Refactorings verhaltensbewahrend. Die Zusammensetzung der Refactorings aus den Minipatterns wird im nächsten Abschnitt beschrieben.

3.3.3 Komplexe Refactorings

Die komplexen Refactorings sind als Sequenz von Minitransformationen und ggf. Hilfsfunktionen wie beispielsweise „createEmptyClass“, „addClass“ oder „replaceObjCreationWithMethodInvocation“ umgesetzt. Für die Hilfsfunktion zeigte Ó Cinnéide ebenfalls ihre verhaltenswahrenden Eigenschaften.
Alle Refactorings sind als Funktionen mit Eingabeparametern beschrieben. Diese Parameter werden im Ansatz von Ó Cinnéide vom Entwickler selbst ermittelt und übergeben. In dieser Arbeit hingegen werden die Resultate der Spot-Detektierung aus Abschnitt 3.2 als Eingabeparameter für die Funktion genutzt. Somit laufen die Spot-Detektierung und das zugehörige Refactoring vollständig automatisiert ab. Nachfolgend werden die einzelnen Refactorings erläutert.

Abstrakte Fabrik

Das *Abstrakte Fabrik*-Entwurfsmuster kann mit der folgenden Sequenz von Minitransformationen automatisiert eingefügt werden [ÓC01].

```
applyAbstractFactory(SetOfClasses products,
                     String newFactoryName,
                     String newAbsFactoryName){
  addClass(createEmptyClass(newFactoryName));            // 1
  ForAll c:Class, c in products{
    Abstraction(nameOf(c));
    AbstractAccess(allClasses, nameOf(c));
    EncapsulateConstruction(newFactoryName, nameOf(c));
```

```
  }                                                     // 2
  applySingleton(newFactoryName, newAbsFactoryName);    // 3
  ForAll e:ObjCreationExprn, classCreated(e) in products{
    replaceObjCreationWithMethodInvocation(e,
         newAbsFactoryName + "getInstance().create" + classCreated(e));
  }                                                     // 4
}
```

Zuerst wird eine leere Fabrik-Klasse mit dem über die Variable `newAbsFactoryName` spezifizierten Name erstellt (1). Anschließend wird für alle angegebenen Produktklassen eine Abstraktion erstellt, Referenzen auf die konkrete Produktklasse durch eine Referenz auf die zuvor erstellte Abstraktion ersetzt und die Erzeugung der Produktklasse in eigene Konstruktionsmethoden in der am Anfang generierten Fabrik-Klasse gekapselt (2). Die Methode `applySingleton` erzeugt eine Singleton-Struktur, indem über die PartialAbstraction-Minitransformation aus der Fabrik-Klasse eine abstrakte Basisklasse generiert und vererbt wird (3). Zuletzt werden programmweit alle Erzeugungen der Produktklassen durch einen Aufruf der Konstruktionsmethoden der Abstrakten Fabrik-Klasse ersetzt (4).
Nach dem Refactoring werden die Produktklassen ausschließlich über die Abstrakte Fabrik erzeugt und über eine entsprechende Schnittstelle referenziert. Die konkrete Fabrik, mit der die Abstrakte Fabrik parametrisiert ist, lässt sich leicht durch eine andere konkrete Fabrik austauschen. Somit würden die Produktklassen programmweit, durch eine zentrale Änderung, durch Produktklassen der neuen konkreten Fabrik ersetzt, was einen Flexibiliätsgewinn darstellt.

Kompositum

Das *Kompositum*-Entwurfsmuster kann mit der folgenden Sequenz von Minitransformationen automatisiert eingefügt werden [ÓC01].

```
applyComposite(Class client, String compositeAttribute){
  Class collection = getCollectionTypeOf(client, compositeAttribute);
  Class component = getComponentTypeOf(client, compositeAttribute);// 1
  moveCollectionIterationContentToComponent(client,
      compositeAttribute, component);                             // 2
  Interface compositeInterface = Abstraction(component);
  addCollectionMethods(compositeInterface, collection);           // 3
  addCollectionMethods(component, collection);                    // 4
  Class composite = addClass(createEmptyClass(
      nameOf(component) + "Composite"));
```

```
  addImplementationLink(composite, compositeInterface);            // 5
  addNewField(composite, collection,
      compositeInterface, "container");                            // 6
  implementMethods(composite, collection, "container");
  implementMethodsByDelegation(composite, component, "container"); // 7
  AbstractAccess(component, compositeInterface);
  AbstractAccess(collection, compositeInterface);                  // 8
  replaceCollectionWithCompositum(client, collection,
      compositeInterface);                                         // 9
}
```

Zunächst wird der Typ der Collection und der Komponentenklasse selbst anhand des Klienten-Attributes ermittelt, das in ein Kompositum transformiert werden soll[3] (1). Anschließend werden in der Klientenklasse sämtliche Inhalte von Schleifen, die durch die Collection-Instanzvariable iterieren, mittels *ExtractMethod*-Refactoring in eigene Methoden der Komponentenklasse verschoben (2). Aus der Komponentenklasse wird dann die Kompositum-Schnittstelle erzeugt und dieser Schnittstelle die Methoden des Collection-Typs hinzugefügt (3). Die Komponente implementiert die Collection-Methoden leer (4). Anschließend wird eine Kompositumklasse erstellt, welche die erzeugte Schnittstelle implementiert (5). Der Kompositumklasse wird ein Attribut vom Typ der Collection hinzugefügt, welche jedoch nicht Elemente der ursprünglich verwendete Komponentenklasse verwaltet, sondern der neu erstellten Schnittstelle. Diese neue Collection kann sowohl Elemente der Komponentenklasse als auch der Kompositumklasse aufnehmen, da beide die Schnittstelle aus Schritt 3 implementieren (6). Anschließend werden die Methoden der Schnittstelle in der Kompositumklasse implementiert – die Collection-Methoden werden direkt an die Collection selbst delegiert, die Komponenten-Methoden hingegen an die Elemente der Collection (7). Als vorletzter Schritt werden im Klienten alle Referenzen auf die Komponentenklasse und die ursprüngliche Collection durch Referenzen auf die erzeugte Schnittstelle ersetzt (8). Abschließend werden im Klienten sämtliche Iterationen durch die Collection-Klasse durch die in Schritt 2 erzeugte Methoden ausgewechselt und die Collection selbst durch eine Instanz der in Schritt 5 erzeugten Kompositumklasse ersetzt (9).

Nach dem Refactoring unterscheidet der Klient nicht mehr zwischen der ursprünglich verwendeten Collection und ihren Elementen. Sämtliche Iterationen durch die Collection sind in die Kompositumklasse verschoben und hinter einer Schnittstelle gekapselt, womit der Klient an Komplexität verliert.

[3] Beispielsweise ist bei der generischen Collection `HashMap<ClassABC>` der Typ der Collection „HashMap“ und „ClassABC“ der Datentyp der Komponente

Zustand

Das *Zustand*-Entwurfsmuster kann mit der folgenden Sequenz von Minitransformationen automatisiert eingefügt werden [ÓC01; CGZS12].

```
applyState(Class context, String stateVariable, Class stateEnumeration) {
  SetOfMethods methods =
      extractStateStatements(context, stateVariable);            // 1
  Class state = addClass(createEmptyClass("State");
  Delegation(context, methods, "state");                         // 2
  Interface stateInterface = Abstraction(state);
  AbstractAccess(state, stateInterface);                         // 3
  ForAll s:String in stateEnumeration {
    Class subState = addClass(createEmptyClass(s + "State"));
    addExtendsLink(subState, stateInterface);
    ForAll m:Method in methods {
      addStateContent(context, m, s, subState);                  // 5
    }
  }                                                              // 4
  addSetStateMethod(context);                                    // 6
  replaceStateVariable(context, stateVariable, stateInterface);  // 7
  deleteClass(state);                                            // 8
}
```

Zunächst werden in der Kontextklasse alle Anweisungen, welche die übergebene Zustandsvariable prüfen (if- bzw. switch-Anweisungen) per *Extract Method*-Refactoring in eigene Methoden verschoben (1). Dann wird eine neue Zustandklasse erzeugt und alle Kontext-Methoden aus Schritt 1 werden an die neue Zustandklasse delegiert (2). Anschließend wird aus der Zustandsklasse eine Abstraktion erstellt und in der Kontextklasse alle Referenzen auf die Zustandsklasse durch die Abstraktion ersetzt (3). Dann wird für jede Konstante der zustandsrepräsentierenden Enumeration eine eigene Subklasse der Zustandsklasse erzeugt (4). Jede Zustandssubklasse implementiert die in Schritt 1 extrahierten Methoden, indem die Codefragmente, die den jeweiligen Zustand für die entsprechende Konstante behandeln, in die jeweilige Methode verschoben werden (5). Die Kontextklasse erhält anschließend eine Methode zum Setzen des Zustands, welche den Delegaten aus Schritt 2 auf eine dem Zustand entsprechende Subklasse der Zustandsklasse setzt (6). Diese Methode wird genutzt, um Zustandsübergänge zu initiieren. Die ursprüngliche Zustandsvariable kann dann entfernt und jegliche Zugriffe auf sie durch polymorphe Aufrufe der entsprechenden Zustandsunterklasse ersetzt werden (7). Abschließend kann die zu Beginn erzeugte Zustandklasse gelöscht werden – sie wurde nur benötigt um die zustandsbehandelnden Methoden zu sammeln und eine Abstraktion aus ihr zu erzeugen.

Nach dem Refactoring ist zustandsabhängiges Verhalten in den Zustandssubklassen gekapselt. Die Kontextklasse ruft polymorph die Methoden der aktuell verwendeten Zustandsklasse auf und ersetzt sie bei Zustandsänderungen durch die korrekte Zustandsklasse.

3.4 Implementierungsentscheidungen

Für die Implementierung der Spot-Detektierung und der Refactorings wurde ein Plug-in für die Entwicklungsumgebung Eclipse entwickelt. Für die Analysen des Quellcodes, die für die Spot-Detektierung nötig sind, wurde das „Java Development Tools“ Framework (kurz „JDT“) verwendet, welches einen Quellcode-Parser und diverse Zugriffs- und Analysemethoden bereitstellt. Die Refactorings wurden auf unterschiedliche Weise bewerkstelligt. Manche Refactorings sind bereits Teil von Eclipse und über dessen API programmatisch steuerbar. In diesen Fällen wurden die vorhandenen Eclipse-Refactorings verwendet. In Fällen, in denen keine nutzbaren Refactorings vorhanden waren, wurden die Refactorings mittels Modifikation der abstrakten Syntaxbäume (englisch: "Abstract Syntax Tree", kurz "AST") des Codes bewerkstelligt, welche ebenfalls vom JDT-Framework unterstützt wird. Gefundene Spots mussten zunächst auf Ausschlusskriterien hin untersucht werden, um Compilerfehler nach den Refactorings zu vermeiden. Die Eclipse-Refactorings prüfen bereits selbstständig verschiedene Vorbedingungen. Diese Prüfungen sind oftmals aber nicht ausreichend, daher wurde die Prüfung der Vorbedingungen aller Refactorings erweitert bzw. komplett selbst implementiert.

Zusammenfassung des Kapitels

In diesem Kapitel wurde die Umsetzung der Entwurfsmusterautomatisierung beschrieben. Zunächst wurden Entwurfsmuster identifiziert, bei denen eine Verbesserung der Wartbarkeit vermutet wird und die sich voraussichtlich gut automatisieren lassen. Anschließend wurde das Verfahren vorgestellt, mit dem geeignete Code-Stellen für die ausgewählten Entwurfsmuster ermittelt werden können. Darauf folgend wurde eine Methode aufgezeigt, wie die ermittelten Stellen um die zugehörigen Entwurfsmuster erweitert werden, um somit die Wartbarkeit zu erhöhen.

4 Ergebnisse der Entwurfsmusterautomatisierung

Übersicht

Nachfolgend wird erläutert wie der Einfluss der Entwurfsmusterautomatisierung auf die Quellcodewartbarkeit experimentell überprüft wird und welche Ergebnisse dabei gesammelt wurden. Abschnitt 4.1 erläutert genutzte Metrik-Software, das verwendete Qualitätsmodell, die Testprojekte, anhand denen der Ansatz der Entwurfsmusterautomatisierung überprüft wurde, und den Ablauf der Experimente. In Abschnitt 4.2 wird eine Übersicht über die detektierten und transformierbaren Spots in Testprojekten und daraus resultierende Möglichkeiten für durchführbare Experimente gegeben. Die durchgeführten Experimente und ihre Ergebnisse sind in Abschnitt 4.3 aufgeführt.

4.1 Vorbemerkungen zu den Experimenten

4.1.1 Verwendete Metrik-Messwerkzeuge

Um die genutzten Metriken zu messen wurden verschiedene Anwendungen verwendet. Für die Metriken „Maintainability Index“ und „Halstead Volume“ wurde das Programm „Java Source Code Metrics“[1] der Firma „Semantic Designs“ in einer mit Java 1.5 kompatiblen Version verwendet. Die LCOM-Metrik wurde mittels des Eclipse-Plug-ins „Metrics 1.3.6“[2] gemessen. Für die Mood-Metriken wurde kein geeignetes Tool gefunden, welches Java-Anwendungen auf Quellcodebasis analysieren kann. Daher wurde ein Eclipse Plug-in auf Basis von JDT (Java Development Tools) entwickelt, welche die Metriken wie in [AC94] beschrieben umsetzt.

4.1.2 Verwendetes Qualitätsmodell

Die Mood-Metriken messen verschiedene Aspekte der Wartbarkeit. Da die Entwurfsmuster – wie in Abschnitt 2.3.3 bereits erwähnt – sowohl positive als auch negative Einflüsse

[1] http://www.semanticdesigns.com/Products/Metrics/JavaMetrics.html
[2] http://metrics.sourceforge.net

auf die verschiedenen Wartbarkeitskriterien haben können, ist es möglich, dass sich Werte für manche Metriken verbessern, die Werte für andere hingegen verschlechtern. Für solche Fälle gilt es die verschiedenen Metrikwerte gegeneinander zu rechnen, um die Gesamtwartbarkeit des Systems zu messen. Dazu wird ein Qualitätsmodell von Al-Ja'Afer verwendet, das die Werte der Einzelmetriken, nach ihrer Priorität gewichtet, zu einem einzelnen Gesamtwert zusammenfasst, anhand dessen die Wartbarkeit des Systems abgelesen werden kann [AJS04].
Dazu werden die gemessenen Einzelmetrikwerte gegen ihre zugehörigen Konfidenzintervalle (siehe Abschnitt 2.2.1) verglichen. Liegen sie innerhalb des Intervalls, so erhalten sie den Qualitätsgrad 100%. Liegen sie außerhalb des Konfidenzintervalls, aber innerhalb eines, aus anderen Studien [AM96; HCN98] extrahierten, Toleranzbereichs, so wird ihr Qualitätsgrad zwischen 100% und 60% interpoliert. Liegen sie auch außerhalb des Toleranzbereichs, so wird ihr Qualitätsgrad zwischen 60% und 0% interpoliert. Die Einzelgrade werden anschließend mit den in Tabelle 4.1 genannten Wichtungen multipliziert, welche die die Bedeutung der Faktoren für die Wartbarkeit repräsentieren [AM96; HCN98]. Die resultierenden gewichteten Einzelgrade werden zusammenaddiert und der erhaltene Wert durch die Summe der Wichtungen dividiert. Daraus ergibt sich wieder ein Wert zwischen 0 und 100, der die Wartbarkeit des Systems repräsentiert, wobei ein höherer Wert ein besser wartbares System impliziert.

Faktor	Priorität	Wichtung
Attribute Hiding Factor	Gering	1
Method Hiding Factor	Mittel	2
Attribute Inheritance Factor	Gering	1
Method Inheritance Factor	Mittel	2
Coupling Factor	Hoch	3
Polymorphism Factor	Mittel	2

Tabelle 4.1.: Wichtung der einzelnen Mood-Metriken für das Qualitätsmodell

4.1.3 Testprojekte

Für die Experimente wurde eine breite Auswahl verschiedener Java-Testprojekte unterschiedlicher Art und Größe zusammengetragen. Dazu zählen verschiedene OpenSource-Projekte sowie Forschungs- und studentische Projekte von der Otto-von-Guericke-Universität Magdeburg.

Zu den OpenSource-Projekten gehören:

- „Maven“[3] von der „Apache Software Foundation“[4]
- verschiedene OpenSource-Software, die zufällig von der Plattform „GitHub“[5] oder „SourceForge“[6] ausgewählt wurde:
  - „Eclipse Checkstyle Plug-in“[7] (Eclipse Plug-in zur Prüfung der Einhaltung von Design-Richtlinien)
  - „h2o“[8] (Analyse-Framework für große Datenmengen)
  - „MapDB“[9] (eingebettete Datenbank-Engine)
  - „mcMMO“[10] (Erweiterung für das Onlinespiel Minecraft)
  - „Neo4j“[11] (Graph-Datenbank-Anwendung)
  - „Musicbrainz-Data“[12] (Zugriffsprotokoll für Musik-Metadaten-Datenbank)
  - „Ninja“[13] (Web-Framework)
  - „OrientDB“[14] (NoSQL Datenbank-Management-System)
  - „Oryx“[15] (echtzeitfähiges und skalierbares Machine-Learning-Framework)
  - „Squirrel-SQL“[16] (graphischer SQL-Client)
  - „Titan“[17] (verteilte Graphen-Datenbank)

Zu den Forschungs- und studentischen Projekten gehören:

- „Vecs“ (IDE und Framework zur Transformation und Verifikation von Systemmodellen)
- mehrere Schachprojekte

Die verschiedenen Projekte sind ihrerseits teilweise in Unterprojekte aufgeteilt. Die somit insgesamt 50 Projekte beinhalten insgesamt 7288 Klassen. Jedes Projekt enthält zwischen 1 und 1246, im Durchschnitt 146 Klassen.

[3] http://maven.apache.org/
[4] http://www.apache.org/
[5] https://github.com/
[6] http://sourceforge.net/
[7] http://sourceforge.net/projects/eclipse-cs/
[8] https://github.com/0xdata/h2o
[9] https://github.com/jankotek/MapDB
[10] https://github.com/mcMMO-Dev/mcMMO
[11] https://github.com/neo4j/neo4j
[12] https://github.com/lastfm/musicbrainz-data
[13] https://github.com/ninjaframework/ninja
[14] https://github.com/orientechnologies/orientdb
[15] https://github.com/cloudera/oryx
[16] http://sourceforge.net/projects/squirrel-sql/
[17] https://github.com/thinkaurelius/titan

4.1.4 Allgemeiner experimenteller Ablauf

Der allgemeine Ablauf der Experimente ist wie folgt:

1. Die Wartbarkeit eines Testprojekts wird initial mittels der Metriken gemessen.
2. Entwurfsmuster werden automatisiert detektiert und eingefügt.
3. Die Wartbarkeit des Testprojekts wird erneut gemessen.
4. Es wird ausgewertet wie die eingefügten Entwurfsmuster sich auf die einzelnen Wartbarkeitskriterien und die Gesamtwartbarkeit ausgewirkt haben.

Generell werden nach jedem Refactoring erneut die Metrik-Werte erfasst. Somit lässt sich prüfen, ob und wie die Wartbarkeit sich über die Iterationen hin verändert. Untersucht wird ebenfalls wie viele Klassen eine Beispielanwendung beinhaltet und welche Refactorings wie oft angewendet werden können. Somit lassen sich Aussagen über die Anwendbarkeit des Verfahrens treffen.

Der variierende Aspekt der Experimente ist welche Entwurfsmuster in welcher Reihenfolge automatisiert implementiert werden. Um mögliche Reihenfolgen zu ermitteln, muss anfänglich überprüft werden, welche Spots gefunden und wie diese sinnvoll miteinander kombiniert werden können.

4.2 Übersicht über detektierte und transformierbare Spots in Testprojekten

Zunächst wurde untersucht wie viele Spots, mit dem in Abschnitt 3.2 erläuterten Ansatz, für die verschiedenen Entwurfsmuster detektiert werden konnten. Innerhalb der Testprojekte wurden zahlreiche potentielle Spots für die Entwurfsmuster gefunden. Diese Spots erfüllen die Mindestvoraussetzungen für die Entwurfsmuster. Jedoch zeigten sich nahezu alle Spots bei weiteren Überprüfungen als nicht automatisiert transformierbar. Tabelle A.1 des Anhangs A.3 zeigt eine Übersicht über alle Projekte und in ihnen gefundene bzw. transformierbare Spots für die automatisierten Entwurfsmuster. Nachfolgend werden die Ergebnisse der Spot-Findung gezeigt und die Gründe für die Nicht-Transformierbarkeit der Spots erläutert.

4.2.1 Abstrakte Fabrik

Für das *Abstrakte Fabrik*-Refactoring wurden in 19 der 50 Testprojekt insgesamt 72 Spots gefunden. Davon ließen sich jedoch lediglich zwei Spots im Testprojekt „JChessThreesome“ – eines der studentischen Schachprojekte – umsetzen. Alle anderen Spots wurden bei der Vorbedingungsprüfung aus den folgenden Gründen als nicht transformierbar ausgeschlossen:

- Zu abstrahierende Klassen enthalten öffentliche Felder, die in Java nicht in eine Schnittstelle übertragbar sind. Daher kann für diese Klassen keine Schnittstelle erstellt werden, die von anderen Klassen exakt wie die konkrete Klasse verwendet werden kann.
- Zu abstrahierende Klassen enthalten statische Methoden, die in Java nicht in eine Schnittstelle übertragbar sind. Daher kann für diese Klassen keine Schnittstelle erstellt werden, die von anderen Klassen exakt wie die konkrete Klasse verwendet werden kann.
- Zu abstrahierende Klassen sind private verschachtelte statische Unterklassen ihrer Container-Klassen. Diese können nicht außerhalb der Container-Klasse erzeugt werden, daher kann ihre Erzeugung nicht in eine Fabrik-Klasse verlagert werden.
- Zu abstrahierende Objekte greifen in Methoden auf private Felder anderer Instanzen derselben Klasse zu (z.B. vergleichen *equals*-Methoden den Wert einer eigenen Instanzvariablen mit der entsprechenden Instanzvariable eines Vergleichsobjekts). Da diese Felder nicht in der erstellten Abstraktion vorhanden sind, das Vergleichsobjekt aber über die Abstraktion referenziert wird, würde es nach einem Refactoring zu Compilerfehlern kommen.
- Zu abstrahierende Klassen erben von Klassen aus einer nicht anpassbaren Bibliothek. Daher können Methoden mit Referenzen auf die eigene Klasse nicht angepasst werden.
- Die zu erstellende Abstraktion existiert bereits, jedoch mit anderen/unvollständigen Methoden-Deklarationen. Daher kann die Abstraktion nicht erstellt werden.

Die beiden übrig gebliebenen Spots, die sich beide im selben Testprojekt befinden, überschneiden sich – sie beziehen sich auf dieselbe Klasse. Somit fällt bei Umstellung eines Spots der andere Spot weg, da beteiligte Klassen bereits transformiert sind und der Spot durch den Detektierungsalgorithmus nicht mehr gefunden wird.

4.2.2 Kompositum

Für das *Kompositum*-Refactoring wurden in 37 der 50 Testprojekt insgesamt 465 Spots gefunden. Davon war jedoch lediglich ein Spot im Projekt „de.ovgu.cse.vecs.saml“ – ein Unterprojekt von „Vecs“ – transformierbar. Alle anderen Spots wurden bei der Vorbedingungsprüfung aus den folgenden Gründen als nicht transformierbar ausgeschlossen:

- Zu abstrahierende Klassen enthalten öffentliche Felder, die in Java nicht in eine Schnittstelle übertragbar sind. Daher kann für diese Klassen keine Schnittstelle erstellt werden, die von anderen Klassen exakt wie die konkrete Klasse verwendet werden kann.
- Zu abstrahierende Klassen enthalten statische Methoden, die in Java nicht in eine Schnittstelle übertragbar sind. Daher kann für diese Klassen keine Schnittstelle erstellt werden, die von anderen Klassen exakt wie die konkrete Klasse verwendet werden kann.
- Das zusammengesetzte Objekt besteht aus Klassen aus einer nicht anpassbaren Bibliothek. Daher können Methoden mit Referenzen auf die eigene Klasse nicht angepasst werden.
- Das zusammengesetzte Objekt referenziert bereits Schnittstellen.
- Das zusammengesetzte Objekt referenziert Wildcards bzw. generischen Klassen, für die keine Abstraktion erzeugt werden kann.
- Die zu erstellende Abstraktion existiert bereits, jedoch mit anderen/unvollständigen Methoden-Deklarationen. Daher kann die Abstraktion nicht erstellt werden.

4.2.3 Zustand

Für das *Zustand*-Refactoring wurden in 12 der 50 Testprojekt insgesamt 40 Spots gefunden. Davon ließen sich jedoch lediglich ein Spot im Testprojekt „h2o“ umsetzen. Alle anderen Spots wurden bei der Vorbedingungsprüfung aus den folgenden Gründen als nicht transformierbar ausgeschlossen:

- Die Enumeration, die den Zustand einer Klasse repräsentiert, beinhaltet selbst Methodendeklarationen. Daher ist das zustandsrepräsentierende Feld nicht einfach durch eine Zustandsklasse ersetzbar, da dadurch die Aufrufe der Methoden der Enumeration nach dem Refactoring Compilerfehler verursachen würden.
- Das zustandsrepräsentierende Feld ist öffentlich (wird von anderen Objekten referenziert) und kann daher nicht durch eine Zustandsklasse ersetzt werden.

- ➢ Das zustandsrepräsentierende Feld wird innerhalb der Klasse nur gegen *null* geprüft, nicht jedoch gegen eine Enumerationskonstante. Daher handelt es sich nicht wirklich um einen Zustand.
- ➢ Die Klasse, in der der Spot gefunden wurde, ist eine private statische verschachtelte Klasse. Zustandsklassen müssten ebenfalls in die entsprechende Containerklasse hinzugefügt werden, wodurch die Klasse stark aufgebläht werden würde.

4.2.4 Konsequenzen der Spotdetektierungsergebnisse

Die Ergebnisse der Spotdetektierung und -untersuchung bestimmen die durchführbaren Experimente. Da alle transformierbaren Spots verschiedener Entwurfsmuster in jeweils verschiedenen Testprojekten liegen, ist es nicht möglich Experimente durchzuführen, bei denen mehrere verschiedene Entwurfsmuster-Refactorings im selben Projekt vorgenommen werden. Dieses Experiment hätte zeigen können, ob sich die verschiedenen Refactorings gegenseitig ausschließen oder ergänzen können.
Ebenso ist es nicht möglich in einem Projekt dasselbe Entwurfsmuster-Refactoring mehrfach durchzuführen, da für das *Kompositum* und den *Zustand* jeweils nur ein transformierbarer Spot gefunden wurde und die Spots für die *Abstrakte Fabrik* sich überschneiden. Dieses Experiment hätte beispielsweise prüfen können, ob eine massenhafte Anwendung eines als wartbarkeitsverbessernd eingeschätzten Musters sich negativ auswirken kann, da das Projekt mit Klassen überladen wird, ober ob ihre Wirkung sich vervielfacht.
Somit blieb lediglich die Anwendung einzelner Entwurfsmuster-Refactorings als Experiment übrig. Diese Experimente und ihre Ergebnisse werden im folgenden Abschnitt behandelt.

4.3 Durchführung und Ergebnisse der Entwurfsmusterautomatisierung

Laut Zielsetzung sollen sowohl die Auswirkungen auf bestehende Klassen als auch auf das neue Gesamtsystem überprüft werden. Darum wird für jedes Experiment für jedes betrachtete Kriterium eine Hyphothese für das bestehende System (bestehende Klassen), für neu erstellte Klassen und für das neue Gesamtsystem (angepasste und neu erstellte Klassen) aufgestellt. Daraus wird eine Gesamthypothese für jedes Refactoring abgeleitet, welche anschließend über die gemessenen Metrikwerte überprüft wird. Diese Experimente werden im Folgenden behandelt.

4.3.1 Abstrakte Fabrik

Es gilt zu prüfen, ob das automatisierte Einfügen des *Abstrake Fabrik*-Musters die Wartbarkeitskriterien (Kohäsion, Kopplung, Kapselung, Komplexität und Wiederverwendung) positiv beeinflussen kann.

Hypothesen

Folgende Auswirkungen auf die Wartbarkeitskriterien werden anhand der in Abschnitt 2.3.2 erläuterten Eigenschaften der Entwurfsmuster vermutet.
Kopplung:

- Bestehendes System: Verringert die Kopplung in Klienten, da Klienten nur noch über Schnittstellen mit Klassen kommunizieren.
- Neue Klassen: Die erstellende Fabrik selbst kennt die Produkte zwar, nutzt sie aber nicht, daher keine hohe Kopplung.
- Neues Gesamtsystem: Die Kopplung wird verringert.

Kapselung:

- Bestehendes System: Keine Auswirkungen, da lediglich die Objekterstellung verlagert wird.
- Neue Klassen: Neue Klassen haben nur private Felder (hohe Kapselung) und wenige Methoden, die aber alle öffentlich sind (geringe Kapselung). insgesamt vermutlich keine signifikanten Auswirkungen.
- Neues Gesamtsystem: Keine signifikanten Auswirkungen.

Kohäsion:

- Bestehendes System: Keine Auswirkungen, da lediglich die Objekterstellung verlagert wird.
- Neue Klassen: Neu erstellte Fabriken haben 1 Feld, das von nur einer Methode genutzt wird, daher vermutlich starke Kohäsion.
- Neues Gesamtsystem: Keine signifikanten Auswirkungen.

Komplexität:

- Bestehendes System: Keine Auswirkungen auf bestehende Klassen, da (außer des abstrakten Zugriffes auf Produkte) nichts geändert wird.
- Neue Klassen: Sind einfach gehalten, daher geringe Komplexität.

- Neues Gesamtsystem: Eine erhöhte Klassenanzahl durch die Klassen, aber keine signifikanten Auswirkungen.

Wiederverwendung:

- Bestehendes System: Keine messbaren Auswirkungen auf bestehende Klassen, da keine Vererbungsstrukturen an ihnen geändert werden, sondern lediglich Interfaces implementiert werden (Wiederverwendung wird aber dennoch erhöht, da derselbe Code nun durch einfaches Austauschen der Fabrik auch mit anderen Produktfamilien genutzt werden kann, was sich aber nicht messen lässt).
- Neue Klassen: benutzen keinen bestehenden Code, daher keine veränderte Wiederverwendung messbar.
- Neues Gesamtsystem: Keine signifikanten Auswirkungen.

Gesamtauswirkungen für das *Abstrakte Fabrik*-Entwurfsmuster:
Die Wartbarkeit wird erhöht, da die Kopplung verringert wird, andere Kriterien sich aber nicht bemerkenswert ändern.

Ergebnis

Beide Spots wurden einzeln umgestellt und zeigen ähnliche Ergebnisse. Der erste Spot für die *Abstrakte Fabrik* im „JChessThreesome“-Projekt beinhaltet eine Klientenklasse, die Objekte zweier anderer Klassen erstellt und zwischen ihnen eine Assoziationsbeziehung herstellt. Durch das Refactoring wurden die Erzeugung der beiden instanziierten Klassen in die Abstrakte Fabrik-Klasse verschoben. Somit wurden der Klient, der beide Klassen verwendete, und einige weitere Klienten, die jeweils nur eine der Klassen verwendeten, von den Klassen entkoppelt. Das Refactoring des zweiten Spots verhielt sich analog, jedoch wurden drei Klassen in die Abstrakte Fabrik-Klasse verschoben und somit mehr Klassen entkoppelt.
Tabelle 4.2 zeigt die Wartbarkeit (gemäß des in Abschnitt 4.1.2 erklärten Qualitätsmodells) des Projekts vor den Refactorings, die Tabellen 4.3 und 4.4 nach den Refactorings des ersten bzw. des zweiten Spots. Tabelle 4.5 zeigt die Auswirkungen auf die einzelnen am Refactoring des ersten Spots beteiligen Klassen, anhand der Metriken LOC, Zyklomatische Komplexität, Halstead Volumen und Schwierigkeit, dem Wartbarkeitsindex und LCOM[18].

[18] Da die Ergebnisse der Refactorings des ersten und zweiten Spots übereinstimmend sind – wie die Tabellen 4.3 und 4.4 zeigen – wurde auf die detaillierte Präsentation der Ergebnisse des Refactorings des zweiten Spots verzichtet.

Metrik	Wichtung	Min	Max	Wert	Grad
AHF	1	75,2	100	97,67	100%
MHF	2	12,7	21,8	17.50	100%
AIF	1	52,7	66,3	27,93	45%
MIF	2	66,4	78,5	39,67	39%
CF	3	0	11,2	3,32	100%
PF	2	2,7	9,6	4,59	100%
Wartbarkeit: 83,9%					

Tabelle 4.2.: Wartbarkeit des „JChessThreesome“-Projekts vor den Refactorings

Metrik	Wichtung	Min	Max	Wert	Grad
AHF	1	75,2	100	97,69	100%
MHF	2	12,7	21,8	16,28	100%
AIF	1	52,7	66,3	27,78	45%
MIF	2	66,4	78,5	38,07	38%
CF	3	0	11,2	3,21	100%
PF	2	2,7	9,6	4,86	100%
Wartbarkeit: 83,7%					

Rot: Werte haben sich verschlechtert (vom Konfidenzintervall weg verändert)
Gelb: Werte haben sich nicht oder innerhalb des Konfidenzintervalls verändert

Tabelle 4.3.: Wartbarkeit des „JChessThreesome“-Projekts nach dem Refactoring des ersten Spots

Metrik	Wichtung	Min	Max	Wert	Grad
AHF	1	75,2	100	97,69	100%
MHF	2	12,7	21,8	16,71	100%
AIF	1	52,7	66,3	27,78	45%
MIF	2	66,4	78,5	38,67	38%
CF	3	0	11,2	3,08	100%
PF	2	2,7	9,6	4,97	100%
Wartbarkeit: 83,7%					

Rot: Werte haben sich verschlechtert (vom Konfidenzintervall weg verändert)
Gelb: Werte haben sich nicht oder innerhalb des Konfidenzintervalls verändert

Tabelle 4.4.: Wartbarkeit des „JChessThreesome“-Projekts nach dem Refactoring des zweiten Spots

Tabelle 4.2 zeigt zunächst, dass die Werte für AHF, MHF, CF und PF im Konvidenzintervall lagen und das Gesamtsystem somit vor dem Refactoring bereits einen hohen Wartbarkeitswert von 83,9% aufwies. Sowohl AIF als auch MIF wiesen Werte auf, die deutlich unterhalb des Konfidenzintervalls liegen.

Typname	LOC			CyclomaticComplexity			HalsteadVolume			HalsteadDifficulty			MaintainabilityIndex			LCOM		
	Vorher	Hinterher	Änderung	Vorher	Hinterher	Änderung	Vorher	Hinterher	Änderung	Vorher	Hinterher	Änderung	Vorher	Hinterher	Änderung	Vorher	Hinterher	Änderung
ChessBoardTest ***	94	94	0	22	22	0	3390,86	3444,52	+53,66	73,84	75,65	+1,81	161,53	161,46	-0,07	0	0	0
GameTest ***	116	117	+1	35	35	0	3773,09	4039,15	+266,06	113,51	121,79	+8,28	164,26	163,71	-0,55	0,353	0,353	0
MoveTest ***	45	45	0	8	8	0	1275,51	1437,71	+162,20	38,50	42,58	+4,08	165,03	164,30	-0,73	0,444	0,444	0
SquareTest ***	52	52	0	10	10	0	1430,60	1566,50	+135,90	45,27	49,17	+3,90	154,04	153,56	-0,48	0,133	0,133	0
SquareUtils ***	117	118	+1	19	19	0	4864,80	4917,35	+52,55	101,34	102,28	+0,94	119,13	118,81	-0,32	0	0	0
AbstractFactory		21	21		11	11		450,03	450,03		10,88	10,88		126,25	126,25		0	0
Factory		2	2		1	1		22,19	22,19		0,08	0,80		0	0		0	0
ISquare		16	16		1	1		289,99	289,99		7,60	7,60		194,74	194,74		0	0
Square **	83	83	0	39	39	0	2075,68	2087,51	+11,83	61,04	59,72	-1,32	166,52	166,52	0	0,819	0,819	0
Game **	164	164	0	56	56	0	5711,69	5733,61	+21,92	162,31	161,76	-0,55	131,37	131,37	0	0,85	0,85	0
IGame		19	19		1	1		516,58	516,58		15,81	15,81		196,48	196,48		0	0
GameFrame ***	182	182	0	21	21	0	6807,55	6858,79	+51,24	115,12	115,84	+0,72	129,84	129,81	-0,03	0,896	0,896	0
ActionListener ***	6	8	+2	2	2	0	39,43	93,45	+54,02	1,50	3,27	+1,77	123,30	113,70	-9,60	0	0	0
Complete System	4273	4359	+86	873	887	+14	271387,75	278003,01	+6615,26	1904,37	1942,49	+38,12	152,02	153,9	+1,88	0,221	0,212	-0,009

* Klientenklasse – Ausgangspunkt des Refactorings
** Klassen, die in die Abstrakte Fabrik-Klasse aufgenommen wurden
*** Weitere Klassen, die von den per Abstrakter Fabrik erzeugten Klassen entkoppelt wurden
Grün: Werte zeigen eine Verbesserung gegenüber einem vorher vorhandenen Wert an
Rot: Werte zeigen eine Verschlechterung gegenüber einem vorher vorhandenen Wert an
Gelb: Werte haben sich gegenüber einem vorher vorhandenen Wert nicht verändert

Tabelle 4.5.: Metrikergebnisse für das Abstrakte Fabrik-Refactoring des ersten Spots des „JChessThreesome“-Testprojekts

Die Tabellen 4.3 und 4.4 zeigen als auffälligste Auswirkung einen Abfall der Methodenvererbung (-1,6% bzw. -1%), wodurch die Bewertung dieses Faktors sinkt und die Wartbarkeit des Systems als schlechter bewertet wurde. Dies ist damit zu erklären, dass die neuen Schnittstellen zahlreiche Methoden deklarieren, die aber nicht weitervererbt, sondern von Klassen implementiert werden. Die geringfügige Verringerung bzw. Verschlechterung des *Attribute Inheritance Factor* (-0,16%) ergibt sich daraus, dass die neuen Klassen keine Attribute vererben und somit den Durchschnitt senken. Dies führte nicht zu weiteren Abstufungen der Wartbarkeit. Der *Method Hiding Factor* wurde verringert (-1,22% bzw. 0,79%), bedingt durch die Abwesenheit jeglicher privater Methoden in den Fabrik-Klassen. Genauso wurde der *Coupling Factor* verringert (-0,11% bzw. -0,24%), als Folge der Nutzung der Abstraktionen anstelle der konkreten Klassen. Der *Polymorphism Factor* wurde erhöht (+0,27% bzw. +0,38%). Jedoch verblieben alle drei Werte innerhalb des Konfidenzintervalls, womit sich keine Änderungen in der Wartbarkeitsbewertung ergaben. Die Erhöhung des *Attribute Hiding Factor* (+0,02%) ist, im Vergleich zu den anderen Änderungen, vernachlässigbar.

Insgesamt verschlechterte sich die Wartbarkeit des Projekts laut dem Qualitätsmodell um 0,2%.

Aus Tabelle 4.5 lässt sich ablesen, dass die einzelnen, am Refactoring beteiligen Klassen nicht wartbarer geworden sind. Für alle instanziierenden Klassen haben sich alle Metrikwerte verschlechtert oder sind konstant geblieben. Die in die Abstrakte Fabrik aufgenommenen Klassen weisen eine leichte Verbesserung der Halstead Schwierigkeitsmetrik und eine leichte Verschlechterung des Halstead Volumen auf, alle anderen Werte blieben konstant. Das Gesamtsystem weist dennoch eine minimal höhere Kohäsion und Wartbarkeit auf, was darauf zurückgeführt werden kann, dass die neu erstellten Typen (die Abstraktionen und die Fabrik-Klassen) sehr hohe Wartbarkeitswerte haben und diese den Schnitt heben.

Auswertung der Hypothesen

Die Hypothesen zur verringerten Kopplung lassen sich durch die Werte der Metriken bestätigen. Die CF-Werte sind gesunken, was eine verringerte Kopplung zeigt.

Die Hypothesen zur konstant-bleibenden Kapselung lassen sich durch die Werte der Metriken nicht vollständig bestätigen. Zwar wurde – wie vermutet – die Attribut-Kapselung minimal erhöht und die Methoden-Kapselung verringert, jedoch fiel die Verringerung der

Methodenkapselung unerwartet stark aus. Dies ist damit zu erklären, dass die Methoden der generierten Schnittstellen, die nicht vererbt werden, nicht bedacht wurden, aber entsprechend in die Messung einfließen.

Die Hypothesen zur konstant-bleibenden Kohäsion lassen sich durch die Werte der Metriken bestätigen. Die bestehenden Klassen zeigen keine Veränderung, die neu erstellten Klassen hingegen sind stark kohäsiv. Die Kohäsion des Gesamtsystems wird durch die neuen Klassen nur minimal verbessert.

Die Hypothesen zur konstant-bleibenden Komplexität lassen sich durch die Werte der Metriken bestätigen. Die bestehenden Klassen zeigen eine nur leicht verschlechterte Wartbarkeit, die neu erstellten Klassen hingegen sind sehr gut wartbar.

Die Hypothesen zur Wiederverwendung lassen sich durch die Werte der Metriken bestätigen. Die Werte der AIF, MIF und PF zeigen eine geringere tatsächliche Wiederverwendung an, die messbar verringerte Kopplung macht die zukünftige Wiederverwendung jedoch einfacher.

Die Hypothesen zur Auswirkung auf die Wartbarkeit lassen sich durch die Werte der Metriken im Groben bestätigen. Definitiv sind die entkoppelten Klassen laut der Metriken leichter austauschbar, bei gleichzeitigem minimalen Abfall der Wartbarkeitsbewertung. Die Ergebnisse werden jedoch durch den Aspekt verzerrt, dass das System vorher bereits eine gute Wartbarkeit aufwies. Eine Verringerung der Kopplung konnte keine Verbesserung im Qualitätsmodell bewirken, da die Kopplung vorher bereits im optimalen Bereich lag. Die geringfügige Zunahme der Komplexität konnte rechnerisch nicht durch den Vorteil der Entkopplung ausgeglichen werden.

4.3.2 Kompositum

Es gilt zu prüfen, ob das automatisierte Einfügen des *Kompositum*-Musters die Wartbarkeitskriterien (Kohäsion, Kopplung, Kapselung, Komplexität und Wiederverwendung) positiv beeinflussen kann.

Hypothesen

Folgende Auswirkungen auf die Wartbarkeitskriterien werden anhand der in Abschnitt 2.3.2 erläuterten Eigenschaften der Entwurfsmuster vermutet.

Kopplung:

- Bestehendes System: Es wird lediglich die iterierbare Container-Klasse durch die Kompositumklasse ersetzt, daher keine signifikante Änderung der Kopplung.
- Neue Klassen: Neue Klassen verwenden, je nach Klienten-Code, unterschiedlich viele andere Klassen. Es ist jedoch von einer geringen Kopplung auszugehen.
- Neues Gesamtsystem: Keine signifikanten Änderungen.

Kapselung:

- Bestehendes System: Vermutlich leicht verminderte Kapselung, da der Komponente neue, öffentliche Methoden hinzugefügt werden.
- Neue Klassen: Wenige, dafür private Attribute und wenige, dafür ausschließlich öffentliche Methoden gleichen sich vermutlich gegenseitig aus.
- Neues Gesamtsystem: Die neuen Methoden sorgen vermutlich für leicht reduzierte Kapselungswerte.

Kohäsion:

- Bestehendes System: Der Komponente werden neue Methoden hinzugefügt, welche ggf. keine Attribute nutzen, weswegen die Kohäsion leicht sinken sollte.
- Neue Klassen: Die Kompositumklasse implementiert mehrere Methoden, die alle dasselbe Attribut nutzen, weswegen eine hohe Kohäsion vermutet wird.
- Neues Gesamtsystem: Die hohe Kohäsion der Kompositumklasse und die verminderte Kohäsion der Komponente gleichen sich vermutlich aus.

Komplexität:

- Bestehendes System: Der Klient wird vereinfacht, die Komponentenklasse durch die (leeren) Kompositum-Methoden leicht komplexer. Die Werte gleichen sich vermutlich gegenseitig aus.
- Neue Klassen: Die Kompositumklasse enthält lediglich Delegationen an die gespeicherte Container-Klasse und an die darin enthaltenen Komponentenklassen. Sie ist, je nach Anzahl der Komponenten- und Container-Methoden, vermutlich unterdurchschnittlich komplex.
- Neues Gesamtsystem: Die Komplexität ist abhängig von der Anzahl der auf das Kompositum bezogenen und ersetzbaren Codefragmente des Klienten. Prinzipiell wird der Klient einfacher, das System hingegen Komplexer, so dass vermutlich eine höhere Komplexität gemessen wird.

Wiederverwendung:

- ➢ Bestehendes System: Keine signifikanten Auswirkungen.
- ➢ Neue Klassen: Die neuen Klassen erben und vererben nichts, daher keine erhöhte Wiederverwendung.
- ➢ Neues Gesamtsystem: Keine signifikanten Auswirkungen.

Gesamtauswirkungen für das *Kompositum*-Entwurfsmuster:
Die messbare Wartbarkeit bleibt in etwa gleich, da die verbesserten Werte des Klienten und die leicht verschlechterten Werte des Gesamtsystems sich gegenseitig ausgleichen.

Ergebnis

Der einzig verbliebene Spot für das *Kompositum*-Muster umfasst eine Klasse (Klient), die ein Set von Klassen (Komponenten) für die Validierung von Quellcode referenziert. Das Set wird im Konstruktor des Klienten mit den Komponenten befüllt und in allen vier Methoden des Klienten per Schleife durchlaufen, um diverse Aktionen auf den einzelnen Komponenten durchzuführen. Nach dem Refactoring, das die Inhalte der Schleifenblöcke in die Komponentenklassen verschoben und in der Kompositumklasse Aufrufe dieser verschobenen Methoden für alle Komponenten eingefügt hat, konnten alle Schleifen aus dem Klienten entfernt und durch einfache Aufrufe der entsprechenden Kompositum-Methoden ersetzt werden.
Tabelle 4.6 zeigt die Wartbarkeit (gemäß des in Abschnitt 4.1.2 erklärten Qualitätsmodells) des Projekts vor dem Refactoring, Tabellen 4.7 danach.

Tabelle 4.6 zeigt, dass das Ausgangssystem als mäßig wartbar einzustufen ist. Während die Feld-Kapselung sich im Toleranzintervall bewegt, weisen alle anderen Werte starke Abweichungen auf. Zwar ist die Kopplung im optimalen Bereich des Qualitätsmodells, jedoch mit 0,4% dennoch sehr gering. Auch die Vererbungs- bzw. Wiederverwendungswerte AIF und MIF liegen deutlich unterhalb des Konfidenzintervalls, der Polymorphie-Wert PF deutlich darüber. Erklärbar sind diese Werte durch die Art des Testprojektes. Bei „Vecs“ handelt es sich um ein Framework zum Transformieren und Verifizieren von Systemmodellen, und beinhaltet einen Parser für selbst definierbare „Domain Specific Languages“[19] (DSLs). Dieser Parser wird – auf Basis von „Xtext“[20] – durch abstrakte Klassen beschrieben, die der Framework-Nutzer über Polymorphie an die DSL anpassen

[19] Domain Specific Languages sind (Programmier)Sprachen, die speziell an ein bestimmtes Fachgebiet (Domäne) angepasst sind.
[20] http://www.eclipse.org/Xtext/ - ein Eclipse-Framework für die Entwicklung von DSLs

Metrik	Wichtung	Min	Max	Wert	Grad
AHF	1	75,2	100	69,24	68%
MHF	2	12,7	21,8	6,12	39%
AIF	1	52,7	66,3	6,28	10%
MIF	2	66,4	78,5	28,22	28%
CF	3	0	11,2	0,39	100%
PF	2	2,7	9,6	52,39	34%
Wartbarkeit: 52,7%					

Tabelle 4.6.: Wartbarkeit des „de.ovgu.cse.vecs.saml“-Projekts vor dem Refactoring

Metrik	Wichtung	Min	Max	Wert	Grad
AHF	1	75,2	100	69,24	68%
MHF	2	12,7	21,8	6,07	38%
AIF	1	52,7	66,3	6,28	10%
MIF	2	66,4	78,5	29,55	29%
CF	3	0	11,2	0,39	100%
PF	2	2,7	9,6	49,70	36%
Wartbarkeit: 53,1%					

Grün: Werte haben sich verbessert (zum Konfidenzintervall hin verändert)
Rot: Werte haben sich verschlechtert (vom Konfidenzintervall weg verändert)
Gelb: Werte haben sich nicht oder innerhalb des Konfidenzintervalls verändert

Tabelle 4.7.: Wartbarkeit des „de.ovgu.cse.vecs.saml“-Projekts nach dem Refactoring

Name	Vorher	Hinterher	Änderung
ISamlChecker		0	0
SamlChecker	0,714	0,714	0
SamlCheckerComposite		0	0
SamlJavaValidator		0	0
Complete System	0,151	0,151	0

Tabelle 4.8.: Metrikergebnisse für das Kompositum-Refactoring des „de.ovgu.cse.vecs.saml“-Testprojekts

muss. Somit liegen hohe Polymorphie, niedrige Wiederverwendung und Kopplung in der Natur des Projektes – bedingt durch exzessive Nutzung von Abstraktionen.

Tabelle 4.7 zeigt für das transformierte System eine Verbesserung der Wartbarkeit (+0,4%), bedingt durch eine deutliche Verringerung der Polymorphie (-2,7%) und Erhöhung der Methodenvererbung (+1,33%). Diese überwiegen auch einen geringen Abfall der Bewertung für die Methodenkapselung (-0,05%). Die anderen Werte blieben konstant oder änderten sich so geringfügig, dass keine Änderung der Wartbarkeitsbewertung dadurch bewirkt wurde.

Da innerhalb des Testprojektes diverse Programmierkonstrukte aus der Java-Version 1.7 verwendet wurden, die verwendete Metrik-Software jedoch nur bis Java 1.5 kompatibel ist, konnten die weiteren Metriken mit Ausnahme der LCOM-Metrik nicht gemessen und somit Auswirkungen auf einzelne Klassen nicht analysiert werden. Es ließ sich durch eine manuelle Messung beobachten, dass die LOC des Klienten sich um 10 – von 88 auf 78 – verringert haben. Die LCOM-Metrik zeigt keinerlei Änderung für bestehende Klassen und eine starke Kohäsion für die neu erstellten Klassen.

Auswertung der Hypothesen

Die Hypothesen zur Kopplung lassen sich durch die Werte der Metriken bestätigen. Die CF-Werte sind konstant geblieben, das Refactoring hatte keinen messbaren Einfluss auf die Kopplung.

Die Hypothesen zur Kapselung lassen sich durch die Werte der Metriken bestätigen. AHF bliebt konstant und MHF ist leicht gesunken, das Refactoring hatte aber insgesamt keinen starken Einfluss auf die Kapselung.

Die Hypothesen zur konstant-bleibenden Kohäsion lassen sich durch die Werte der Metriken bestätigen. Die neu erstellten Klassen zeigen eine starke Kohäsion, insgesamt blieb die Kohäsion aber konstant.

Über die Hypothesen zur Komplexität lässt sich keine Aussage treffen. Dies liegt zum einen daran, dass die Metriken für die einzelnen Klassen nicht gemessen werden konnten. Und zum anderen daran, dass die gemessenen Werte der Systemmetriken, durch den Framework-Kontext, Wartbarkeitsverbesserungen suggerieren, die so nicht stattgefunden haben. Der Polymorphie-Wert des Ausgangssystems ist so hoch, dass jede neue Klasse ohne Polymorphie bereits einen starken Einfluss auf die Metrikergebnisse hat, selbst wenn die Vererbung nicht im Fokus des Entwurfsmusters liegt. Dieser Effekt lässt keine Aussage über tatsächliche Veränderungen der Komplexität anhand von Polymorphie zu.

Für die Hypothesen zur Wiederverwendung trifft dasselbe zu, wie bei den Komplexitätshypothesen. Der bei der Komplexität erläuterte Effekt zeigt sich hier erneut bei der Polymorphie, aber auch (in abgeschwächter Form) anhand der MIF-Metrik bei der Vererbung. Diese war vorher sehr gering, so dass jede neue Klasse mit "normalen" Werten

einen starken Einfluss auf die Systemmetrik hat, selbst wenn dies eigentlich nicht durch das Entwurfsmuster selbst bewerkstelligt wird.

Durch die fehlenden bzw. nicht aussagekräftigen Metrikwerte lässt sich in dieser Arbeit keine objektive Aussage zu den tatsächlichen Auswirkungen auf die Wartbarkeit treffen. Die getroffenen Hypothesen können weder belegt noch widerlegt werden. Nur die Verringerung der LOC des Klienten lässt vermuten, dass die Klientenklasse tatsächlich vereinfacht werden konnte.

4.3.3 Zustand

Es gilt zu prüfen, ob das automatisierte Einfügen des *Zustand*-Musters die Wartbarkeitskriterien (Kohäsion, Kopplung, Kapselung, Komplexität und Wiederverwendung) positiv beeinflussen kann.

Hypothesen

Folgende Auswirkungen auf die Wartbarkeitskriterien werden anhand der in Abschnitt 2.3.2 erläuterten Eigenschaften der Entwurfsmuster vermutet.
Kopplung:

- Bestehendes System: Internes Verhalten und damit ggf. auch Referenzen auf andere Klassen werden verschoben, daher wird die Kopplung vermutlich verringert.
- Neue Klassen: Neue Klassen übernehmen ggf. Referenzen von bestehenden Klassen, referenzieren die Zustand-beinhaltende Klasse, haben aber insgesamt eine unterdurchschnittliche Kopplung zu anderen Klassen.
- Neues Gesamtsystem: Leichte verringerte Kopplung.

Kapselung:

- Bestehendes System: Zustandsabhängiges Verhalten wird zwar gekapselt, aber es werden kaum Daten versteckt. Gegebenenfalls müssen sogar vorher private bzw. geschützte Methoden öffentlich gemacht werden, damit die Zustandklassen sie aufrufen können. Die messbare Kapselung wird daher vermutlich leicht verringert.
- Neue Klassen: Zustandsklassen haben fast nur öffentliche Methoden (außer es können private Methoden mit verschoben werden), es werden keine Methoden in ihrer Sichtbarkeit verringert. Daher verringert sich die Kapselung.
- Neues Gesamtsystem: Leicht verringerte Kapselung.

Kohäsion:

- Bestehendes System: Eine zustandsrepräsentierende Variable wird durch eine Referenz auf eine Zustandsklasse ersetzt, daher keine Änderung der Kohäsion. Eine Ausnahme bestünde, falls die Zustände Instanzvariablen nutzen, die nirgendwo anders genutzt werden und daher in die Zustandsklassen verschoben werden können. In diesem Fall ist eine Erhöhung der Kohäsion zu erwarten.
- Neue Klassen: Neue Klassen sammeln zusammengehörige Funktionalität und können eine starke Kohäsion aufweisen, falls Felder aus der zustandsbeinhaltenden Klasse in die Zustandsklassen verschoben werden können
- Neues Gesamtsystem: Die Kohäsion erhöht sich wenn Felder verschoben werden können, ansonsten bleibt sie konstant.

Komplexität:

- Bestehendes System: Viele Verzweigungen werden aus vorhandenen Klassen entfernt, die Komplexität sinkt.
- Neue Klassen: Neue Klassen sind simpel gehalten und zeigen eine unterdurchschnittliche Komplexität.
- Neues Gesamtsystem: Es wird eine Verringerung der Komplexität vermutet.

Wiederverwendung:

- Bestehendes System: Keine signifikanten Auswirkungen.
- Neue Klassen: Die Zustandklassen bilden zwar eine neue Vererbungshierarchie, dennoch werden kaum Methoden wiederverwendet, sondern abstrakte Methoden von den abgeleiteten Klassen selbst, weswegen die Werte insgesamt vermutlich sinken.
- Neues Gesamtsystem: Keine signifikante Änderung.

Gesamtauswirkungen für das *Zustand*-Entwurfsmuster:
Durch die Verbesserungen für den Klienten und die simplen Zustandsklassen verbessert sich die Wartbarkeit des Systems.

Ergebnis

Der einzig verbliebene Spot für das *Zustand*-Muster umfasst eine Klasse (Klient), die per Instanzvariable eine Enumeration mit drei Konstanten referenziert. Der Klient prüft in nur einer Methode den Wert der Instanzvariablen über eine *if*-Anweisung und führt je nach Wert verschiedene Aktionen aus. Durch das Refactoring wurde der Inhalt der 13

Zeilen umfassenden *if*-Anweisung in die entsprechenden, neu erstellten Zustandklassen verschoben und die *if*-Anweisung komplett durch einen einzigen Methodenaufruf ersetzt. Tabelle 4.9 zeigt die Wartbarkeit (gemäß des in Abschnitt 4.1.1 erklärten Qualitätsmodells) des Projekts vor dem Refactoring, Tabelle 4.10 danach. Tabelle 4.11 zeigt die Auswirkungen auf die einzelnen am Refactoring des ersten Spots beteiligen Klassen, anhand der Metriken LOC, Zyklomatische Komplexität, Halstead Volumen und Schwierigkeit, dem Wartbarkeitsindex und LCOM.

Metrik	Wichtung	Min	Max	Wert	Grad
AHF	1	75,2	100	67,596	60%
MHF	2	12,7	21,8	37,486	59%
AIF	1	52,7	66,3	89,714	25%
MIF	2	66,4	78,5	85,040	58%
CF	3	0	11,2	0,420	100%
PF	2	2,7	9,6	19,040	57%
Wartbarkeit: 66,6%					

Tabelle 4.9.: Wartbarkeit des „h2o“-Projekts vor dem Refactoring

Metrik	Wichtung	Min	Max	Wert	Grad
AHF	1	75,2	100	67,625	60%
MHF	2	12,7	21,8	37,426	59%
AIF	1	52,7	66,3	89,706	25%
MIF	2	66,4	78,5	83,008	58%
CF	3	0	11,2	0,418	100%
PF	2	2,7	9,6	19,055	57%
Wartbarkeit: 66,6%					

Grün: Werte haben sich verbessert (zum Konfidenzintervall hin verändert)
Rot: Werte haben sich verschlechtert (vom Konfidenzintervall weg verändert)
Gelb: Werte haben sich nicht oder innerhalb des Konfidenzintervalls verändert

Tabelle 4.10.: Wartbarkeit des „h2o“-Projekts nach dem Refactoring

Tabelle 4.9 zeigt für das Ausgangsprojekt eine eher mäßig eingestufte Wartbarkeit. Wieder ist die Kapselung zwar im optimalen Bereich des Qualitätsmodells, aber mit 0,42% dennoch sehr gering. Alle anderen Werte liegen deutlich außerhalb des Konfidenzintervalls, wobei die Vererbung, die Polymorphie und die Methodenkapselung zu hoch und die Attribut-Kapselung zu niedrig ist.

Tabelle 4.10 zeigt für das transformierte Projekt nahezu dieselben Werte an. Sämtliche Änderungen liegen im Bereich von 0,06% und darunter, sodass das Qualitätsmodell exakt dieselbe Wartbarkeit wie vorher ermittelt. Dieses Ergebnis ist durch die hohe Anzahl von Typen innerhalb des Projektes zu erklären, denn eine veränderte und fünf neu erstellte

Typname	LOC			CyclomaticComplexity			HalsteadVolume			HalsteadDifficulty			MaintainabilityIndex			LCOM		
	Vorher	Hinterher	Änderung	Vorher	Hinterher	Änderung	Vorher	Hinterher	Änderung	Vorher	Hinterher	Änderung	Vorher	Hinterher	Änderung	Vorher	Hinterher	Änderung
DefaultStatTypeStateTree		15	15		8	8		372,23	372,23		10,36	10,36		127,43	127,43		0	0
ENTROPYStatTypeStateTree		15	15		8	8		385,50	385,50		10,23	10,23		127,25	127,25		0	0
GINIStatTypeStateTree		15	15		8	8		385,50	385,50		10,23	10,23		127,25	127,25		0	0
MSEStatTypeStateTree		23	23		11	11		693,08	693,08		18,62	18,62		116,60	116,60		0	0
StatTypeStateTree		19	19		10	10		442,17	442,17		9,23	9,23		122,12	122,12		0	0
Tree *	717	704	-13	128	124	-4	45842,20	45344,40	-497,80	594,03	597,21	+3,18	113,06	113,81	+0,75	0,268	0,268	0
Complete System	86538	86623	+85	35412	35453	+41	9998065,11	10006863,99	+8798,88	62870,77	62963,68	+92,91	141,39	141,40	+0,01	0,187	0,187	0

* Zustand-enthaltende Klasse
Grün: Werte zeigen eine Verbesserung gegenüber einem vorher vorhandenen Wert an
Rot: Werte zeigen eine Verschlechterung gegenüber einem vorher vorhandenen Wert an
Gelb: Werte haben sich nicht verändert

Tabelle 4.11.: Metrikergebnisse für das Zustand-Refactoring des „h2o“-Testprojekts

Klassen stehen 1245 bestehenden und unveränderten Typen gegenüber (vergleiche Tabelle A.1), sodass neue Werte nur wenig Einfluss auf den Durchschnitt haben. Dennoch haben sich die Kapselungswerte leicht verbessert (AHF +0,03%, MHF -0,06%), genauso wie die Vererbungswerte (AIF -0,01%, MIF -0,03%). Der Grad der Polymorphie hat sich minimal geringfügig verschlechtert (PF +0,015%).

Tabelle 4.11 zeigt die Auswirkungen detaillierter an. Hier ist deutlich ablesbar, dass die transformierte Klasse wartbarer geworden ist, da sich die Werte für LOC, zyklomatische Komplexität, Halstead Volumen und den Wartbarkeitsindex verbessert haben. Die neuen Klassen haben durchweg gute Werte für die Wartbarkeit, da sie nur wenig Code enthalten. Das Gesamtsystem zeigt trotz Zunahme von Klassen und den damit verbundenen erhöhten Werten für die Anzahl der Code-Zeilen, zyklomatischen Komplexität usw. dennoch eine insgesamt leicht erhöhte Wartbarkeit anhand des Wartbarkeitsindex.

Auswertung der Hypothesen

Die Hypothesen der verringerten Kopplung lassen sich durch die Werte der Metriken bestätigen. Die CF-Werte sind minimal gesunken, was eine verringerte Kopplung zeigt. Ob dies jedoch darauf zurückzuführen ist, dass koppelnde Codefragmente aus der zustandsbeinhaltenden Klasse in die zustandsrepräsentierenden Klassen ausgelagert wurden, oder die neu erstellten Klassen eine unterdurchschnittliche Kopplung haben, lässt sich anhand der gemessenen Metriken nicht ermitteln. Eine manuelle Prüfung ergab, dass der zweite Fall zutrifft, da keine Abhängigkeiten aus dem Klienten entfernt wurden.

Die Hypothesen der verringerten Kapselung lassen sich durch die Werte der Metriken ebenfalls bestätigen. Der MHF-Wert wies mit -0,06% die stärkste Änderung auf. Anhand der gemessenen Metriken lässt sich nicht sagen, ob die Änderung durch das Aufbrechen der Kapselung innerhalb der zustandsbeinhaltenden Klasse zustande kam, oder weil die zustandsrepräsentierenden Klassen unterdurchschnittlich wenig private Methoden besitzen. Eine manuelle Prüfung zeigte, dass der letztere Fall zutrifft, da keine geschützten Methoden des Klienten öffentlich gemacht wurden.

Die Hypothesen zur konstant-bleibende Kohäsion lassen sich durch die Werte der Metriken bestätigen. Die transformierte Klasse zeigt dieselbe Kohäsion wie vor dem Refactoring und die neu erstellten Klassen sind stark kohäsiv. Die Gesamtsystemwerte sind im Schnitt trotz konstant geblieben.

Die Hypothesen zur verringerten Komplexität lassen sich durch die Werte der Metriken klar bestätigen. Die transformierte Klasse ist weniger komplex als vor dem Refactoring und auch die neu erstellten Klassen zeigen eine geringe Komplexität.

Die Hypothesen zur konstant-bleibenden Wiederverwendung lassen sich durch die Werte der Metriken exakt bestätigen. Der Wert für AIF bleibt nahezu konstant, MIF sinkt geringfügig und gleichzeitig steigt PF geringfügig an, was auf das Überschreiben der abstrakten Methoden zurückzuführen ist.

Insgesamt bestätigen die Werte der Metriken die Hypothese, dass das *Zustand*-Muster die Wartbarkeit des Testprojektes erhöht hat.

Zusammenfassung

In diesem Kapitel wurden die Auswirkungen der Entwurfsmusterautomatisierung experimentell untersucht. Dazu wurde zunächst automatisiert nach Spots für Entwurfsmuster gesucht und gefundenen Spots daraufhin geprüft, ob sie transformierbar sind. Für die gefundenen, transformierbaren Spots wurden zunächst Hypothesen aufgestellt, wie sich die Refactorings auf die Wartbarkeitskriterien auswirken. Anschließend wurden für die Spots die Metrikwerte erfasst, dann die Refactorings durchgeführt und anschließend erneut die Metriken gemessen. Anhand der erfassten Werte wurden die Hypothesen geprüft und daraus abgeleitet, ob die Refactorings die Wartbarkeit verbessern konnten.

5 Schlussfolgerungen und Ausblick für die Wartbarkeit durch Entwurfsmusterautomatisierung

Übersicht

Dieses Kapitel fasst die Ergebnisse der Entwurfsmusterautomatisierung zusammen und zieht daraus Schlüsse, was sie für die Wartbarkeit von Software bedeuten. Die Ergebnisse werden außerdem mit denen einer anderen Studie zum Einfluss der Wartbarkeit verglichen. Abschließend wird betrachtet, welche Möglichkeiten zur weiteren Forschung die Entwurfsmusterautomatisierung bietet.

5.1 Ergebnisse der Entwurfsmusterautomatisierung

5.1.1 Anwendbarkeit der Entwurfsmusterautomatisierung

Die Experimente zur Entwurfsmusterautomatisierung haben vor allem gezeigt, dass die automatisierte Implementierung von Entwurfsmustern mit dem gewählten Ansatz, bei nicht vorverarbeitetem Quellcode, kaum anwendbar ist. Nicht einmal 1% der Code-Stellen, welche die Mindestanforderung für ein Entwurfsmuster erfüllten, lies sich auch transformieren. Die Gründe dafür sind vielschichtig. Zum Teil beruhen sie auf schlechtem Programmierstil in den Testprojekten, wie beispielsweise der Verwendung von öffentlichen Attributen (von Fowler als ein „Code Smell“ – „stinkender Code“ – beschrieben [Fow99]), die das Generieren einer Abstraktion verhindern, die für das jeweilige Entwurfsmuster nötig ist. Zum Teil verhinderten auch dem Zweck angemessene Code-Konstrukte die Automatisierung. Beispiele dafür sind statische Methoden, die dem Generieren einer Abstraktion ebenfalls im Wege stehen, und die Wiederverwendung von Bibliothekskomponenten, welche von den betrachteten Projekten aus nicht transformierbar sind. Auch waren die gewählten Refactorings teilweise nicht mächtig genug, um die detektierten Spots vollständig transformieren zu können. Dies betraf beispielsweise Enumerationen, die selbst Methoden definierten – das *Zustand*-Refactoring war nicht dafür ausgelegt diese Methoden der Enumerationen ebenfalls zu extrahieren.

An den wenigen Stellen, an denen der Code doch transformiert werden konnte, haben sich verschiedene Ergebnisse gezeigt, welche im nächsten Abschnitt erläutert werden.

5.1.2 Einfluss auf die Wartbarkeit

Zunächst sei festgehalten, dass die Entwurfsmuster die Aspekte der Wartbarkeit, die in ihrem Fokus stehen, stets verbessern konnten. So verringerte die *Abstrakte Fabrik* messbar die Kopplung innerhalb des Projektes, das *Kompositum*-Muster verringerte die Anzahl der Zeilen – und damit vermutlich auch die Komplexität – der transformierten Klienten-Klasse, und auch das *Zustand*-Muster reduzierte die Komplexität.
Bei vielen Entwurfsmustern wird eine Verbesserung des fokussierten Aspektes durch Mechanismen erreicht, die sich negativ auf andere Aspekte auswirken. Daher müssen zur Beurteilung der Wartbarkeit auch die anderen Aspekte bewertet werden. Hier zeigten sich für die verschiedenen Entwurfsmuster unterschiedliche Ergebnisse. Während das automatisierte Implementieren des *Zustand*-Entwurfsmusters die Wartbarkeit des Systems, auch mit Betrachtung der anderen Aspekte als der Komplexität, messbar verbessern konnte, hat die *Abstrakte Fabrik* die Wartbarkeit laut der Messergebnisse eher negativ beeinflusst. Die Messergebnisse hier wurden jedoch dadurch verzerrt, dass die im Fokus der *Abstrakten Fabrik* stehende Kopplung bereits vor dem Refactoring sehr guten Messwerte aufzeigte. Diese konnten im verwendeten Qualitätsmodell nicht weiter verbessert werden, während die anderen Aspekte sich verschlechtern konnten. So war es aufgrund des verwendeten Qualitätsmodells nicht möglich, dass sich die Verbesserung der Kopplung gegen die Verschlechterung der anderen Aspekte rechnerisch wieder kompensierte. Für das *Kompositum*-Entwurfsmuster konnten keine verwertbaren Daten zum Einfluss auf die Wartbarkeit ermittelt werden. Die gemessenen Metriken wiesen in vielen Bereichen vor dem Refactoring zwar realistische, aber dennoch so extreme Werte auf, dass neue Klassen, die Werte "im üblichen Bereich" aufzeigen, einem Extrem in die entgegengesetzte Richtung gleich kamen. Somit wurden die Metrikwerte durch das Refactoring stark verändert, ohne dass die genaue Ursache für die Wertveränderung extrahierbar ist. Somit kann auch keine Aussage darüber getroffen werden, ob das Entwurfsmuster die Wartbarkeit an dieser Stelle überhaupt beeinflusst.

5.1.3 Aussagekraft der Ergebnisse

Generell können aufgrund der sehr geringen Datenmenge – jeweils ein Spot für *Zustand* und *Kompositum* und zwei Spots für die *Abstrakte Fabrik* in 50 Testprojekten – keine allgemeinen Aussagen getroffen werden, da gemessene Effekte nicht anhand weiterer Stichproben verifiziert werden konnten.
Für das *Zustand*-Muster kann die generelle Eignung zur Wartbarkeitsverbesserung – auch durch eine automatisierte Implementierung – vermutet werden, da die Messergebnisse

exakt die zuvor getroffenen Hypothesen bestätigten. Eine wirklich qualifizierte Aussage kann jedoch erst dann getroffen werden, wenn die Ergebnisse anhand weiterer Refactorings reproduziert werden können.

Für das *Abstrakte Fabrik*- und *Kompositum*-Muster haben die in dieser Arbeit erzielten Ergebnisse wenig bzw. gar keine Aussagekraft.

5.2 Vor- und Nachteile der Entwurfsmusterautomatisierung

Trotz der uneindeutigen Untersuchungsergebnisse der Entwurfsmusterautomatisierung, können dennoch Vor- und Nachteile des Ansatzes erkannt und benannt werden.

5.2.1 Vorteile und Chancen

Ein großer Vorteil des Entwurfsmusterautomatisierungsansatzes ist, dass Entwurfsmuster wesentlich schneller eingefügt werden können, als es manuell der Fall ist. Die Detektierung der zahlreichen Spots erfolgte mit dem verwendeten Ansatz selbst in den größten Testprojekten (mit über 1000 Klassen) innerhalb weniger Minuten. Auch die Überprüfung der Spots und die Refactorings selbst waren automatisiert in wenigen Sekunden abgeschlossen. Eine manuelle Suche nach Anwendungsmöglichkeiten für Entwurfsmuster in den Projekte benötigt ein Vielfaches der Zeit.
Eine Chance, die der Ansatz der Automatisierung bietet, ist der Lerneffekt für unerfahrene Entwickler. Der verwendete Ansatz hat lediglich die Strukturen des Quellcodes durch Refactorings verändert. Erweitert man diesen Ansatz beispielsweise um das Einfügen von erklärenden Quellcode-Kommentaren oder um anderen Informationsquellen, so kann unerfahrenen Entwicklern an eigenen, konkreten Beispielen aufgezeigt werden, wo und wie die Entwurfsmuster wirken.

5.2.2 Nachteile und Risiken

Ein großer Nachteil des Ansatzes ist die Abhängigkeit von vorhandenen Code-Strukturen. Entwurfsmuster können in verschiedenen Kontexten angewendet werden, wodurch sich unzählige Möglichkeiten zur Umsetzung ergeben. Die Algorithmen zur Detektierung der Spots und Implementierung der Muster erwarten jedoch bestimmte Strukturen im Quellcode. Sind diese Strukturen nicht vorhanden, so können die Algorithmen nicht angewandt werden. Dies zeigte sich in dieser Arbeit sehr stark. Die Algorithmen können zwar stetig

weiterentwickelt und an andere Strukturen angepasst werden, jedoch ist es schwer das Wissen einen Entwicklers algorithmisch so nachzubilden, dass es flexibel genug ist, um verschiedenste Situationen autonom bewältigen zu können.
Weiterhin besteht die Gefahr, dass automatisierte Entwurfsmuster nicht alle ihre Akteure mit einbeziehen. So ist es beispielsweise bei der *Abstrakte Fabrik* möglich, dass nicht alle Mitglieder einer Produktfamilie erkannt werden. Gehören mehr Klassen zur Produktfamilie, als automatisiert erkannt wurden, so kann es passieren, dass durch den Austausch einer konkreten Fabrik Mitglieder der Produktfamilie nicht mit ausgetauscht werden. Es besteht das Risiko von unerwartetem Verhalten, dessen Ursache anschließend nur schwer zu identifizieren ist.
Zusätzlich kann durch Automatisierung ein Aufwand-Nutzen-Gefälle entstehen, indem große Umstrukturierungen nur minimale Verbesserungen mit sich bringen. Beispielsweise wurden beim Refactoring zum *Kompositum*-Muster u.a. eine Kompositumklasse mit 27 Methoden und rund 200 Zeilen Code erzeugt, um die Klientenklasse um 10 Zeilen Code zu vereinfachen. Daher ist die Entwurfsmusterautomatisierung ohne hinreichende (automatisierte) Vorkontrollen mit Vorsicht zu genießen.

5.3 Vergleich mit den Ergebnissen anderer Studien

Die Ergebnisse aus dieser Arbeit bzgl. des *Zustand*-Musters decken sich mit den Erkenntnissen von Christopoulou zur Automatisierung des *Strategie*-Musters, welches dem *Zustand*-Muster von der Implementierung her sehr ähnlich ist [Fow99; CGZS12]. Christopoulou stellte fest, dass durch die automatisierte Transformation von Quellcode hin zum Strategie-Muster die Komplexität der transformierten Klassen gesenkt werden kann [CGZS12]. Diese Aussage kann anhand der Messungen dieser Arbeit verifiziert werden. Die Ergebnisse dieser Arbeit zeigen darüber hinaus, dass nicht nur die transformierte Klasse an Wartbarkeit gewinnt, sondern auch die beim Refactoring erstellten Klassen eine hohe Wartbarkeit besitzen und somit trotz erhöhter Klassenanzahl die Wartbarkeit des gesamten Systems steigt.

Für das *Abstrakte Fabrik*- und *Kompositum*-Muster decken sich die Ergebnisse ebenfalls mit bekannten Studien. In ihren Studien konnten Prechelt et al. nicht eindeutig feststellen, ob die Anwendung der beiden Muster dabei hilft Software einfacher (bzw. schneller) zu warten [PUT+01]. Auch in dieser Arbeit konnte keine allgemeine Aussage darüber getroffen werden, wobei sich die Gründe für die Unklarheiten unterscheiden. Während in dieser Arbeit kaum und nur verzerrte Daten vorliegen, die eine objektive Interpretation

unmöglich machen, zeigten die vorhanden Daten in Prechelts Studien zu wenig Unterschiede zwischen der Wartbarkeit von Systemen mit und ohne Entwurfsmustern, um eine objektive Aussage treffen zu können.

5.4 Möglichkeiten zur weiteren Forschung

In dieser Arbeit wurden von Beginn an Einschränkungen gemacht, um sowohl die Auswirkungen der Entwurfsmuster isoliert zu betrachten, als auch den zeitlichen Rahmen der Arbeit einzuhalten. Daher wurde beispielsweise bewusst auf eine Vorverarbeitung von Quellcode der Testprojekte verzichtet. Da durch eine Vorverarbeitung aber vermutlich eine Vielzahl an Spots umstellbar wäre, können die Auswirkungen einer automatisierten Vorverarbeitung zusammen mit dem Einfügen der Entwurfsmuster untersucht werden.
Auch wurde nur eine geringe Zahl an Entwurfsmustern untersucht. Verschiedene Entwurfsmuster wurden als automatisierbar erachtet (beispielsweise das *Erbauer*- und *Befehl*-Muster), aber aus zeitlichen Gründen nicht überprüft. Die Auswirkungen dieser Muster auf die Wartbarkeit kann Gegenstand weiterer Untersuchungen werden, genauso wie die Verwendung anderer Software-Metriken zur Messung der Wartbarkeit.
Die Entwicklung intelligenterer Spotdetektierungsalgorithmen ist ebenfalls weitere Untersuchungen wert. Im verwendeten Ansatz wurde beispielsweise die Existenz von Abstraktionen, für zu transformierende Klassen, vor einem Refactoring teilweise als Ausschlusskriterium angesehen. Ein interessanter Ansatz könnte sein sich beispielsweise für das *Fabrikmethode*-Muster speziell auf vorhandene Abstraktionen zu konzentrieren und zu ermitteln, ob und wie eine Fabrik die Nutzung vorhandener Abstraktionen unterstützen kann.
Eine weitere mögliche Untersuchung wäre es Entwurfsmuster automatisiert nach Bedarf zu implementieren. In dieser Arbeit wurden Entwurfsmuster implementiert und daraufhin untersucht, welche Wartbarkeitskriterien sie beeinflussen. Andersherum ist es auch möglich Projekte zuerst auf Schwächen bzgl. mancher Wartbarkeitskriterien zu untersuchen und dementsprechend automatisiert solche Entwurfsmuster einzufügen, die genau diese Wartbarkeitskriterien verbessern können.

Literaturverzeichnis

[AC94] ABREU, Fernando B. ; CARAPUÇA, Rogério: Object-oriented software engineering: Measuring and controlling the development process. In: *proceedings of the 4th International Conference on Software Quality* Bd. 186, 1994

[AC11] AJOULI, Akram ; COHEN, Julien: Refactoring Composite to Visitor and Inverse Transformation in Java. In: *arXiv preprint arXiv:1112.4271* (2011)

[AGE95] ABREU, Fernando B. ; GOULÃO, Miguel ; ESTEVES, Rita: Toward the design quality evaluation of object-oriented software systems. In: *Proceedings of the 5th International Conference on Software Quality, Austin, Texas, USA*, 1995, S. 44–57

[AJS04] AL-JA'AFER, J ; SABRI, K: Metrics for object oriented design (MOOD) to assess Java programs. In: *King Abdullah II school for information technology, University of Jordan, Jordan* (2004)

[AM96] ABREU, Fernando B. ; MELO, Walcelio: Evaluating the impact of object-oriented design on software quality. In: *Software Metrics Symposium, 1996., Proceedings of the 3rd International* IEEE, 1996, S. 90–99

[AMAL06] AMOUI, Mehdi ; MIRARAB, Siavash ; ANSARI, Sepand ; LUCAS, Caro: A genetic algorithm approach to design evolution using design pattern transformation. In: *International Journal of Information Technology and Intelligent Computing* 1 (2006), Nr. 2, S. 235–244

[AS95] ARCHER, Clark ; STINSON, Michael: Object-Oriented Software Measures / DTIC Document. 1995. – Forschungsbericht

[CGZS12] CHRISTOPOULOU, Aikaterini ; GIAKOUMAKIS, Emmanouel A. ; ZAFEIRIS, Vassilis E. ; SOUKARA, Vasiliki: Automated refactoring to the Strategy design pattern. In: *Information and Software Technology* 54 (2012), Nr. 11, S. 1202–1214

[CK94] CHIDAMBER, Shyam R. ; KEMERER, Chris F.: A metrics suite for object oriented design. In: *Software Engineering, IEEE Transactions on* 20 (1994), Nr. 6, S. 476–493

[DeM86] DEMARCO, Tom: *Controlling software projects: management, measurement, and estimates*. Prentice Hall PTR, 1986

[EGY97] EDEN, Amnon H. ; GIL, Joseph ; YEHUDAI, Amiram: Automating the application of design patterns. In: *JOOP* 10 (1997), Nr. 2, S. 44–46

[EYG97] EDEN, Amnon H. ; YEHUDAI, Amiram ; GIL, Joseph: Precise specification and automatic application of design patterns. In: *Automated Software Engineering, 1997. Proceedings., 12th IEEE International Conference* IEEE, 1997, S. 143–152

[Fow99] FOWLER, Martin: *Improving the Design of Existing Code*. Addison-Wesley, 1999. – ISBN 0–201–48567–2

[GHJV95] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

[GPP+02] GOMES, Paulo ; PEREIRA, Francisco C. ; PAIVA, Paulo ; SECO, Nuno ; CARREIRO, Paulo ; FERREIRA, José L ; BENTO, Carlos: Using CBR for automation of software design patterns. In: *Advances in Case-Based Reasoning*. Springer, 2002, S. 534–548

[GT03] GRUBB, Penny ; TAKANG, Armstrong A.: *Software Maintenance - Concepts and Practice*. Second. World Scientific, 2003. – ISBN 981–238–425–1

[Hal77] HALSTEAD, Maurice H.: Elements of software science. (1977)

[HCN98] HARRISON, Rachel ; COUNSELL, Steve J. ; NITHI, Reuben V.: An evaluation of the MOOD set of object-oriented software metrics. In: *Software Engineering, IEEE Transactions on* 24 (1998), Nr. 6, S. 491–496

[HS96] HENDERSON-SELLERS, Brian: *Object-Oriented Metrics. Measures of Complexity*. Prentice Hall, 1996

[IEE90] IEEE Standard Glossary of Software Engineering Terminology. In: *IEEE Std. 610.12-1990* (1990)

[IEE98] IEEE Standard for Software Maintenance. In: *IEEE Std. 1219-1998* (1998)

[ISO01] ISO/IEC: *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001

[Jam06] JAMALI, Seyyed M.: Object oriented metrics. In: *A survey approach Technical report, Department of Computer Engineering, Sharif University of Technology, Tehran, Iran* (2006)

[Jen10] JENSEN, Adam C.: *Using Evolutionary Computation to Automatically Refactor Software Designs to Include Design Patterns*, Michigan State University, Diss., 2010

[JH07] JUILLERAT, Nicolas ; HIRSBRUNNER, Béat: Toward an implementation of the" form template method" refactoring. In: *Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on* IEEE, 2007, S. 81–90

[JLB02] JEON, Sang-Uk ; LEE, Joon-Sang ; BAE, Doo-Hwan: An automated refactoring approach to design pattern-based program transformations in java programs. In: *Software Engineering Conference, 2002. Ninth Asia-Pacific* IEEE, 2002, S. 337–345

[Koe98] KOENIG, Andrew: Patterns and antipatterns. In: *The patterns handbook: techniques, strategies, and applications* (1998), S. 383–390

[LH93] LI, Wei ; HENRY, Sallie: Object-oriented metrics that predict maintainability. In: *Journal of systems and software* 23 (1993), Nr. 2, S. 111–122

[LK94] LORENZ, Mark ; KIDD, Jeff: *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., 1994

[LLL08] LINCKE, Rüdiger ; LUNDBERG, Jonas ; LÖWE, Welf: Comparing software metrics tools. In: *Proceedings of the 2008 international symposium on Software testing and analysis* ACM, 2008, S. 131–142

[LRS00] LEWERENTZ, Claus ; RUST, Heinrich ; SIMON, Frank: Quality—Metrics—Numbers—Consequences. In: *Software-Metriken*. Springer, 2000, S. 51–70

[McC76] MCCABE, Thomas J.: A complexity measure. In: *Software Engineering, IEEE Transactions on* (1976), Nr. 4, S. 308–320

[ÓC01] Ó CINNÉIDE, Mel: *Automated application of design patterns: a refactoring approach*, Trinity College Dublin., Diss., 2001

[ÓCN99] Ó CINNÉIDE, Mel ; NIXON, Paddy: A methodology for the automated introduction of design patterns. In: *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on* IEEE, 1999, S. 463–472

[ÓCN01] Ó CINNÉIDE, Mel ; NIXON, Paddy: Automated software evolution towards design patterns. In: *Proceedings of the 4th International Workshop on Principles of Software Evolution* ACM, 2001, S. 162–165

[PUT+01] PRECHELT, Lutz ; UNGER, Barbara ; TICHY, Walter F. ; BROSSLER, Peter ; VOTTA, Lawrence G.: A controlled experiment in maintenance: comparing design patterns to simpler solutions. In: *Software Engineering, IEEE Transactions on* 27 (2001), Nr. 12, S. 1134–1144

[RMT09] RIAZ, Mehwish ; MENDES, Emilia ; TEMPERO, Ewan: A systematic review of software maintainability prediction and metrics. In: *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement* IEEE Computer Society, 2009, S. 367–377

[Rob08] ROBERT, Martin C.: *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice-Hall, Inc., 2008

[SAM12] SJØBERG, Dag I. ; ANDA, Bente ; MOCKUS, Audris: Questioning software maintenance metrics: a comparative case study. In: *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement* ACM, 2012, S. 107–110

[SRBD10] SHLEZINGER, Galia ; REINHARTZ-BERGER, Iris ; DORI, Dov: Modeling design patterns for semi-automatic reuse in system design. In: *Journal of Database Management (JDM)* 21 (2010), Nr. 1, S. 29–57

[WO95] WELKER, Kurt D. ; OMAN, Paul W.: Software maintainability metrics models in practice. In: *Crosstalk, Journal of Defense Software Engineering* 8 (1995), Nr. 11, S. 19–23

A Anhang

A.1 Beschreibung der einzelnen Entwurfsmuster

Abstrakte Fabrik

Die Abstrakte Fabrik bietet „eine Schnittstelle zum Erzeugen vom Familien verwandter oder voneinander abhängiger Objekte, ohne ihre konkreten Klassen zu benennen“ [GHJV95].
Es wird also die Objekterzeugung für eine Familie von verwandten Objekten gekapselt. Ein Klient der Abstrakten Fabrik kennt die konkrete Klasse des erzeugten Objekts nicht und ist somit vom Objekt entkoppelt. Gleichzeitig wird sichergestellt, dass die erzeugen Objekte zur selben Produktfamilie gehören (beispielsweise dass erzeugte Steuerelemente einer GUI alle dasselbe Look&Feel besitzen). Somit sind weitere Konsistenzprüfungen bzgl. der Produktfamilie nicht nötig, wodurch sich die Komplexität der Klienten verringert.

Erbauer

Der Erbauer trennt „die Konstruktion eines komplexen Objekts von seiner Repräsentation, so dass derselbe Konstruktionsprozess unterschiedliche Repräsentationen erzeugen kann“ [GHJV95].
Ein Klient erzeugt ein komplexes Objekt also nicht selbst, sondern lässt es von einem anderen Objekt erzeugen. Dadurch weiß der Klient auch hier nicht, von welcher konkreten Klasse das erzeugte Objekt ist – wieder sind sie entkoppelt. Da der Klient das komplexe Objekt nicht selbst erzeugen und zusammensetzen muss, verringert sich somit auch seine Komplexität.

Fabrikmethode

Die Fabrikmethode „definiert eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts, aber lässt Unterklassen entscheiden, von welcher Klasse das zu erzeugende Objekt ist. Fabrikmethoden ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu delegieren.“ [GHJV95]

Der Klient ist somit ebenfalls vom zu erzeugenden Objekt entkoppelt, da er nicht weiß welche konkrete Klasse das Objekt haben wird. Des Weiteren ermöglichen Fabrikmethoden die Verbindung paralleler Vererbungshierarchien. So können beispielsweise Manipulationen von Objekten in Manipulator-Objekte gekapselt werden, die genauso hierarchisch gegliedert sind, wie die zu manipulierenden Objekte selbst. Über eine Fabrikmethode kann ein Objekt einen auf es spezialisierten Manipulator erstellen. Beispielsweise gibt `A.ErzeugeManipulator()` eine Instanz der Klasse "ManipulatorA" zurück, während `B.ErzeugeManipulator()` ein "ManipulatorB"-Objekt erzeugt. Somit ist ein Klient von der Aufgabe befreit für ein Objekt ein zugehöriges Objekt aus einer anderen Klassenhierarchie ausfindig zu machen, wodurch er weniger komplex wird.

Prototyp

Ein Prototyp „bestimmt die Arten zu erzeugender Objekte durch die Verwendung eines prototypischen Exemplars und erzeugt neue Objekte durch Kopieren dieses Prototyps" [GHJV95].
Ein Klient muss die konkrete Klassen eines Prototypen nicht kennen und kann dennoch eine neue Instanz des Prototypen erstellen. Wieder sind Klient und zu erzeugendes Objekt voneinander entkoppelt. Auf diese weise können eine Vielzahl von Objekten erzeugt werden, ohne dass die Klassen dem System bekannt sind – beispielsweise wenn dynamisch geladene Objekte verwendet werden. Somit müssen diese Klassen nicht für das System adaptiert werden, womit die Komplexität des Systems sinkt.

Singleton

Das Singleton-Entwurfsmuster sichert ab, „dass eine Klasse genau ein Exemplar besitzt, und stellt einen globalen Zugriffspunkt darauf bereit" [GHJV95].
Über das Singleton-Muster wird also eine Zugriffskontrolle auf die einzige Instanz einer Klasse realisiert – Klienten müssen ggf. nötige Zugriffskontrollen nicht selbst realisieren, was der Komplexität zugute kommt.

Adapter

Das Adapter-Muster „passt die Schnittstelle einer Klasse an eine andere, von ihren Klienten erwartete Schnittstelle an. Das Adaptermuster lässt Klassen zusammenarbeiten, die wegen inkompatibler Schnittstellen ansonsten dazu nicht in der Lage wären." [GHJV95]

Der Adapter kann sowohl klassen- als auch objektbasiert realisiert werden, indem Aufrufe zur Erstellungszeit bereits festgelegt werden (klassenbasiert), oder der Adapter zur Laufzeit mit einem Objekt konfiguriert wird, an das Aufrufe dann delegiert werden (objektbasiert). Durch das Adapter-Muster kann die Wiederverwendung erhöht werden, da Klassen wiederverwendet werden, die ohne den Adapter wegen falscher Schnittstellen nicht erneut genutzt werden könnten.

Brücke

Das Brücken-Muster „entkoppelt eine Abstraktion von ihrer Implementierung, so dass beide unabhängig voneinander variiert werden können“ [GHJV95].
Die Abstraktion gibt abstrakte Algorithmus vor, eine konkrete Implementierung setzt diese um, ist jedoch zur Laufzeit konfigurierbar. Da die Abstraktion nicht um die konkrete Klasse der Implementierung weiß, wird die Kopplung verringert. Da der Klient einer Brücke nur mit der Abstraktion arbeitet, jedoch von der Implementierung abgeschirmt ist, sind auch diese voneinander entkoppelt.

Dekorierer

Ein Dekorierer „erweitert ein Objekt dynamisch um Zuständigkeiten. Dekorierer bieten eine flexible Alternative zur Unterklassenbildung, um die Funktionalität einer Klasse zu erweitern.“ [GHJV95]
Das Dekorierer-Muster arbeitet nach dem Prinzip "Komposition statt Vererbung" und verhindert somit einen starken Anstieg von Unterklassen, wenn es mehrere Zuständigkeiten gibt, um die Objekte erweitert werden können. Somit sinkt die Komplexität des gesamten Systems. Des Weiteren wird die Kohäsion der Objekte verbessert, da Eigenschaften der Erweiterungen nicht in das eigentliche Objekt aufgenommen werden müssen, sondern im Dekorierer gekapselt sind. Wird beispielsweise ein graphisches Steuerelement dynamisch um einen Rahmen erweitert, so müssen die Eigenschaften des Rahmens (Breite, Farbe, etc.) nicht in das Steuerelement selbst aufgenommen werden, sondern verbleiben im Rahmen-Dekorierer. Dieser wiederum kann um verschiedene Steuerelement gelegt werden, so dass nicht für jedes zu umrahmende Steuerelement eine Unterklasse "UmrahmtesSteuerelement" gebildet werden muss.

Fassade

Eine Fassade „bietet eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems. Die Fassadenklasse definiert eine abstrakte Schnittstelle, welche die Benutzung des Subsystems vereinfacht.“ [GHJV95]
Die Fasse fasst viele Schnittstellen zu einer Schnittstelle zusammen, wodurch Klienten mit weniger Objekten arbeiten müssen und die Komplexität verringert wird. Außerdem können Klient und die hinter der Fassade verwendeten Subsystemklassen entkoppelt werden. Dies wird jedoch nicht automatisch von der Fassade gewährleistet, Klienten können die Subsystemklassen nach wie vor direkt verwenden.

Fliegengewicht

Das Fliegengewicht-Muster „nutzt Objekte kleinster Granularität gemeinsam, um große Mengen von ihnen effizient verwenden zu können“ [GHJV95].
So wird von Objekten jeweils nur eine Instanz erstellt und diese an den Stellen, an denen sie gebraucht werden, nur referenziert. Dadurch werden Speicher und Rechenzeit gespart, da ein Objekt nicht mehrfach erstellt werden muss.

Kompositum

Ein Kompositum „fügt Objekte zu Baumstrukturen zusammen, um Teil-Ganzes-Hierarchien zu repräsentieren. Das Kompositionsmuster ermöglicht es Klienten, sowohl einzelne Objekte als auch Kompositionen von Objekten einheitlich zu behandeln.“ [GHJV95]
Da Klienten auf diese Weise nicht mehr zwischen einfachen und komplexen Objekten unterscheiden muss, sondern dies vom Kompositum übernommen wird, wird die Komplexität des Klienten verringert.

Proxy

Der Proxy „kontrolliert den Zugriff auf ein Objekt mithilfe eines vorgelagerten Stellvertreterobjekts“. [GHJV95]
Die Verwendungsmöglichkeiten des Proxy-Musters sind vielfältig. So kann ein Proxy beispielsweise genutzt werden, um rechen- oder speicherintensive Objekte erst dann anzulegen, wenn sie wirklich gebraucht werden. Vorher werden sie vom Stellvertreter-Objekt günstiger repräsentiert. Ein weitere mögliche Anwendung ist der Schutz des vertretenen

Objekts – unterschiedliche Objekte mit unterschiedlichen Berechtigungen werden vom Proxy erst auf ihre Zugriffsrechte geprüft, bevor Operationen an das eigentliche Objekt weitergeleitet werden. Auch wäre es möglich mithilfe des Proxy Berechnungen des eigentlichen Objektes zu speichern, damit diese nicht erneut teuer berechnet werden müssen.

Befehl

Das Befehl-Entwurfsmuster „kapselt einen Befehl als ein Objekt. Dies ermöglicht es Klienten mit verschiedenen Anfragen zu parametrisieren, Operationen in eine Queue zu stellen, ein Logbuch zu führen und Operationen rückgängig zu machen." [GHJV95]
Ein Vorteil des Befehls ist die Entkopplung zwischen dem Sender und dem Empfänger des Befehls. Da benötigte Informationen zur Durchführung des Befehls im Befehl-Objekt gekapselt sind, kann er von verschiedenen Empfängern bearbeitet werden, ohne dass der Sender weiß wer den Befehl bearbeitet.

Beobachter

Das Beobachter-Muster „definiert eine 1-zu-n-Abhängigkeit zwischen Objekten, so dass die Änderung des Zustands eines Objekts dazu führt, dass alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden." [GHJV95]
Interessierte Objekte können sich an einem beobachteten Objekt registrieren, um über Änderungen unterrichtet zu werden. Ein bekanntes Beispiel hierzu ist das Abonnieren eines Newsletters. Das beobachtete Objekt weiß nicht welche Objekte sich bei ihm registrieren, denn es kommuniziert mit beobachtenden Klassen lediglich über eine Abstraktion. Somit sind das Objekt und seine Beobachter voneinander entkoppelt.

Besucher

Der Besucher „kapselt eine auf den Elementen einer Objektstruktur auszuführende Operation als ein Objekt. Das Besuchermuster ermöglicht es, eine neue Operation zu definieren, ohne die Klassen der von ihr bearbeiteten Elemente zu verändern." [GHJV95]
Eine neue Operation, die viele/alle Objekte einer Struktur betreffen würden, wird nicht in den Objekten selbst definiert, sondern in einem Objekt, das die Operationen von Außen über die Objekte durchführt. Dies erhöht die Kohäsion der Objekte, denn die Operationen werden von anderen Operationen getrennt, die nichts mit der Aufgabe des Besuchers

zu tun haben. Des Weiteren wird die Komplexität der bearbeiteten Objekte nicht durch neue Operationen erhöht, denn sie selbst bleiben unverändert. Jedoch muss ggf. die Kapselung der besuchten Objekte aufgebrochen werden, damit der Besucher alle benötigten Informationen von außen erreichen kann.

Interpreter

Das Interpreter-Muster „definiert für eine gegebene Sprache eine Repräsentation der Grammatik sowie einen Interpreter, der die Repräsentation nutzt, um Sätze in der Sprache zu interpretieren" [GHJV95].
Dieses Muster beschreibt alles rundherum um den speziellen Themenkomplex der Auswertung interpretierbarer Ausdrücke. Konkrete Auswirkungen auf die Wartbarkeit werden nicht beschrieben.

Iterator

Das Iterator-Muster „ermöglicht den sequentiellen Zugriff auf die Elemente eines zusammengesetzten Objekts, ohne seine zugrundeliegende Repräsentation offenzulegen" [GHJV95].
Da der Iterator alle Operationen bereitstellt, die zum Iterieren durch die Elemente eines zusammengesetzten Objekts benötigt werden, muss dieses Objekt die Operationen nicht selbst bereitstellen, wodurch die Komplexität des Objekts abnimmt. Die Einsparungen bezüglich der Komplexität wird noch weiter erhöht, je mehr verschiedene Arten der Traversierung bereitgestellt werden sollen, da jede Traversierung durch einen eigenen Iterator repräsentiert wird, anstatt sie im zusammengesetzten Objekt bereit zu stellen.

Memento

Das Memento-Entwurfsmuster „erfasst und externalisiert den internen Zustand eines Objekts, ohne seine Kapselung zu verletzten, so dass das Objekt später in diesen Zustand zurückversetzt werden kann" [GHJV95].
Das Memento ist somit ein Datenspeicher, der sich beispielsweise dazu nutzen lässt, Zustände eines Objekts für eine Rückgängig-/Undo-Funktion zu speichern. Da alle benötigten Daten im Memento gespeichert sind, werden diese Zustandsinformationen vor einem Klienten versteckt und der Klient und das Objekt, dessen Zustand im Memento gekapselt ist, können voneinander entkoppelt werden. Außerdem wird der Klient um die Aufgabe

erleichtert die Daten für eine Historie selbst zu verwalten, wodurch sich seine Komplexität reduziert. Auch wirkt sich das Memento positiv auf die Kapselung aus, da das Objekt, dessen Zustand im Memento gespeichert ist, seinen Zustand nicht offenlegen muss.

Schablonenmethode

Eine Schablonenmethode „definiert das Skelett eines Algorithmus in einer Operation und delegiert einzelne Schritte an Unterklassen. Die Verwendung einer Schablonenmethode ermöglicht es Unterklassen, bestimmte Schritte eines Algorithmus zu überschrieben, ohne seine Struktur zu verändern.“ [GHJV95]
Da die Grundstruktur des gekapselten Algorithmus erhalten bleibt, auch wenn einzelne seiner Schritte in Unterklassen verändert werden, kann der Algorithmus in mehreren Situationen wiederverwendet und nach Belieben durch Unterklassen angepasst werden.

Strategie

Die Strategie „definiert eine Familie von Algorithmen, kapselt jeden einzelnen und macht sie austauschbar. Das Strategiemuster ermöglicht es, den Algorithmus unabhängig von ihn nutzenden Klienten zu variieren.“ [GHJV95]
Ein Vorteil der Strategie ist, dass sie Bedingungsanweisungen im Code entfernt, die sonst in potentiell mehreren Methoden vorhanden wären. So müsste ein Klient, der aus mehreren Algorithmen wählt, je nach gewähltem Algorithmus aus verschiedenen Tätigkeiten wählen (wenn Algorithmus A, dann Tätigkeit A1, sonst ... – und das in allen vom Algorithmus beeinflussten Methoden). Sind alle vom Algorithmus beeinflussten Tätigkeiten in einem Strategie-Objekt gekapselt, so kann der Klient die Tätigkeit polymorph über die Strategie aufrufen (Strategie.Tätigkeit1() -> was Tätigkeit1 ist, hängt von der Strategie ab, der Klient muss nicht mehr selbst wählen). Somit wird der Klient deutlich weniger komplex. Des Weiteren wirkt sich das Strategie-Muster positiv auf die Kohäsion aus, da alle den Algorithmus betreffenden Methoden in der Strategie-Klasse gekapselt und nicht mit den anderen Methoden des Klienten vermengt werden. Zudem können gekapselte Algorithmen in anderen Kontexten wiederverwendet werden.

Vermittler

Das Vermittler-Muster „definiert ein Objekte, welches das Zusammenspiel einer Menge von Objekten in sich kapselt. Vermittler fördern lose Kopplung, indem sie Objekte davon

abhalten, aufeinander explizit Bezug zu nehmen. Sie ermöglichen es, das Zusammenspiel der Objekte von ihnen unabhängig zu variieren." [GHJV95]
Abgesehen von der losen Kopplung vereinfacht der Vermittler das Protokoll zwischen den verschiedenen Objekten, da die Objekte nicht mehr selbst miteinander kommunizieren – das Kommunikationsverhalten ist nicht mehr über mehrere Klassen verteilt, sondern zentralisiert. Die Vermittler-Klasse selbst kann dadurch – je nach Anzahl beteiligter Objekte und dem Umfang der Interaktionen – jedoch selbst sehr komplex werden.

Zustand

Das Zustand-Muster „ermöglicht es einem Objekt, sein Verhalten zu ändern, wenn sein interner Zustand sich ändert. Es wird so aussehen, als ob das Objekt seine Klasse gewechselt hat." [GHJV95]
Analog zum Strategie-Muster ist auch beim Zustand-Muster zustandsabhängiges – und damit zusammengehöriges – Verhalten in eigene Objekte gekapselt, was sowohl der Kohäsion zugute kommt, als auch durch die Reduktion von Bedingungsanweisungen die Komplexität einer Klasse stark reduzieren kann.

Zuständigkeitskette

Das Zuständigkeitskette-Muster „vermeidet die Kopplung eines Auslösers einer Anfrage mit seinem Empfänger, indem mehr als ein Objekt die Möglichkeit erhält, die Aufgabe zu erledigen. Es verkettet die empfangenden Objekte und leitet die Anfrage an der Kette entlang, bis ein Objekt sie erledigt." [GHJV95]
Da der Sender somit nicht mehr um den Bearbeiter seiner Anfrage weiß, wird die Kopplung zwischen den Objekten verringert.

A.2 Automatisierbarkeit der einzelnen Entwurfsmuster

Hier werden für einzelne Entwurfsmuster Betrachtungen zu ihrer Automatisierbarkeit aufgezeigt. Entwurfsmuster, die nicht vollständig autonom – also anhand von aus dem Quellcode ermittelbare Informationen – automatisierbar sind, kommen für die Automatisierung nicht in Frage. Dies beinhaltet sowohl, dass sich passende Spots detektieren lassen, als auch dass sich eine vollständige Transformation erstellen lässt.

A.2.1 Erzeugungsmuster

Erbauer

Für das Erbauer-Muster muss die Erzeugung eines komplexes Objekts in eine eigene Klasse samt Abstraktion ausgelagert werden. Dazu müssen alle Code-Fragmente, die das Objekt erzeugen, mittels *Extract Method*-Refactoring in eine neue Klasse verschoben werden. Aus dieser Klasse wird eine Abstraktion erstellt (mittels *Extract Interface*-Refactoring). Dies ist nötig, um verschiedene Erbauer-Objekte erstellen und Klienten mit ihnen parametrisieren zu können. Auch für das zu erstellende Objekt muss eine Abstraktion erstellt und von den Klienten verwendet werden, damit über den Konstruktionsprozess verschiedene Repräsentationen erzeugt werden können. Weiteren Aktionen des Entwicklers sind nicht nötig, das Muster kann automatisiert werden. Die Spotdetektierung beinhaltet das Auffinden von Objekterzeugungen, an deren Anschluss weitere Attribute des Erstellten Objekts verändert bzw. dem Objekt weitere Objekte hinzugefügt werden. Dies ist automatisiert problemlos erkennbar.

Fabrikmethode

Die Fabrikmethode ist Teil der Abstrakten Fabrik. Es muss lediglich die Objekterzeugung mittels *Extract Method*-Refactoring in eine Methode verschoben, alle Objekterzeugungen durch den Aufruf dieser Methode ersetzt und Referenzen auf die konkrete Klasse durch eine ggf. per *Extract Interface*-Refactoring erzeugte Schnittstelle ersetzt werden. Die Fabrikmethode kann daher ohne weiteres Wissen des Entwicklers automatisiert werden. Als simpelster Ausgangspunkt für die Spotfindung dient die Erzeugung eines Objekts in einer anderen Klasse. Jede solcher Objekterzeugungen über eine Fabrikmethode in einem Klienten zu kapseln ist jedoch nicht sinnig, da andere Klassen, die Instanzen derselben Klasse erzeugen wollen, dann entweder über den genannten Klienten gehen müssten und

somit eine neue Abhängigkeit geschaffen werden würde, oder jede Klasse ihre eigene Fabrikmethode erstellen muss. Das Fabrikmethode-Muster ist somit zu feinkörnig, um sinnvoll automatisiert Spots identifizieren zu können. Daher wird die Fabrikmethode von der Automatisierung ausgeschlossen, obwohl sie sehr einfach transformierbar ist.

Prototyp

Das Prototyp-Muster setzt voraus, dass ein Objekt eine Kopie bzw. einen Klon von sich erstellen kann. Die Erstellung einer solche *Clone*-Methode ist nicht in jedem Falle automatisierbar [ÓC01], beispielsweise wenn das zu kopierende Objekt komplexe Objekte beinhaltet. Es ist nicht ausreichend Referenzen innerhalb des Prototyps auf den Klon zu übertragen, sondern alle Objekte müssen rekursiv kopiert werden. Für einfache Datentypen ist das Erstellen einer *Clone*-Methode möglich. Kann ein Objekt geklont werden und ist im aktuellen Kontext eine passende Instanz des zu erstellenden Objektes vorhanden, so ist das Muster automatisierbar. Dazu muss vom zu kopierenden Objekt eine Abstraktion erzeugt (*Extract Interface*-Refactoring), Referenzen auf die konkrete Klasse durch Referenzen auf die Abstraktion und Aufrufe des Konstruktors durch Aufrufe der *Clone*-Methode des zu kopierenden Objekts ersetzt werden. Es ist jedoch schwer automatisiert zu entscheiden, ob ein vorhandenes Objekt ein wirklich passender Prototyp für ein zu erstellendes Objekt ist. Aufgrund der genannten Schwierigkeiten und Einschränkungen wird das Prototyp-Muster daher von der Automatisierung ausgeschlossen.

Singleton

Das Singleton-Muster ist automatisierbar, indem einer Klasse eine statische Methode zur Erzeugung einer Instanz von sich selbst hinzugefügt wird. Diese Methode speichert eine erzeugte Instanz in einer statischen Klassenvariable und gibt immer nur diese erzeugte Instanz zurück. Zusätzlich muss der Konstruktor vor der Außenwelt versteckt (Sichtbarkeit *protected*) und Konstruktoraufrufe außerhalb der vorher erzeugten Methode durch den Aufruf dieser Methode ersetzt werden. Somit ist ein Objekt der Klasse nur noch über die erzeugte Methode möglich. Das Muster kann somit Zutun eines Entwicklers automatisiert werden. Die Spotfindung ist ebenfalls weitestgehend autonom möglich, indem geprüft wird ob systemweit nur eine Instanz eines Objektes erzeugt wird und diese Instanz keinen eigenen Zustand besitzt.

A.2.2 Strukturmuster

Adapter

Das Adapter-Muster soll dafür sorgen, dass Objekte verwendet werden können, die ohne Anpassung nicht verwendbar sind. Eine Adaption von einer Schnittstelle auf eine andere Schnittstelle ist trivial, solang bekannt ist welche Funktion der einen Schnittstelle auf welche Funktion der anderen abgebildet werden soll. Dies ist Wissen, dass nur der Entwickler besitzt [ÓC01], wodurch das Adapter-Muster nicht ohne dieses Wissen automatisiert werden kann. Des Weiteren wird in dieser Arbeit von kompilierbarem Code ausgegangen. Da der Code bei inkompatiblen Schnittstellen nicht kompilierbar wäre, muss bereits eine funktionierende (oder eine leere und damit keine) Lösung vorliegen, damit überhaupt ein Refactoring vorgenommen werden kann. In beiden Fällen ist ein Refactoring nicht nötig. Damit kann das Adapter-Muster ausgeschlossen werden.

Brücke

[Fow99] benennt dieses Refactoring als "Vererbung auflösen". Es ist anwendbar, wenn eine Vererbungshierarchie mehrere verschiedene Aufgaben gleichzeitig übernimmt – beispielsweise eine Klasse zur Berechnung eines Ergebnisses und eine davon abgeleitete Subklasse für die Darstellung des Ergebnisses – und diese Vererbungsbeziehung aufgetrennt werden soll (siehe Abbildung A.1). Das erstellen einer neuen Vererbungshierarchie ist automatisiert möglich, auch das verschieben entsprechender Funktionalitäten in neue Klassen (*Extract Method*- bzw. *Move Method*-Refactorings) und das Erzeugen der Delegation von der alten Vererbungsstruktur auf die neue ist trivial. Es obliegt jedoch dem Entwickler zu entscheiden, welche Funktionalitäten der verschiedenen Klassen der Vererbungsstruktur zu welcher Aufgabe gehören [Fow99]. Das Ermitteln dieser Zusammenhänge ist automatisiert nicht sicher möglich, daher wird das Brücken-Muster von der Automatisierung ausgeschlossen.

Dekorierer

[ÓC01] zeigt, dass die bloße Struktur des Musters leicht hergestellt werden kann, indem man für verschiedene Objekte mit denselben (bzw. in großen Teilen übereinstimmenden) Schnittstellen eine abstrakte Wrapper-Klasse erstellt. Die Funktionalitäten, um die ein Objekt dynamisch erweitert werden soll, können dann in konkreten Wrapper-Klassen umgesetzt und diese Wrapper-Klassen mit den zu erweiternden Objekten parametrisiert

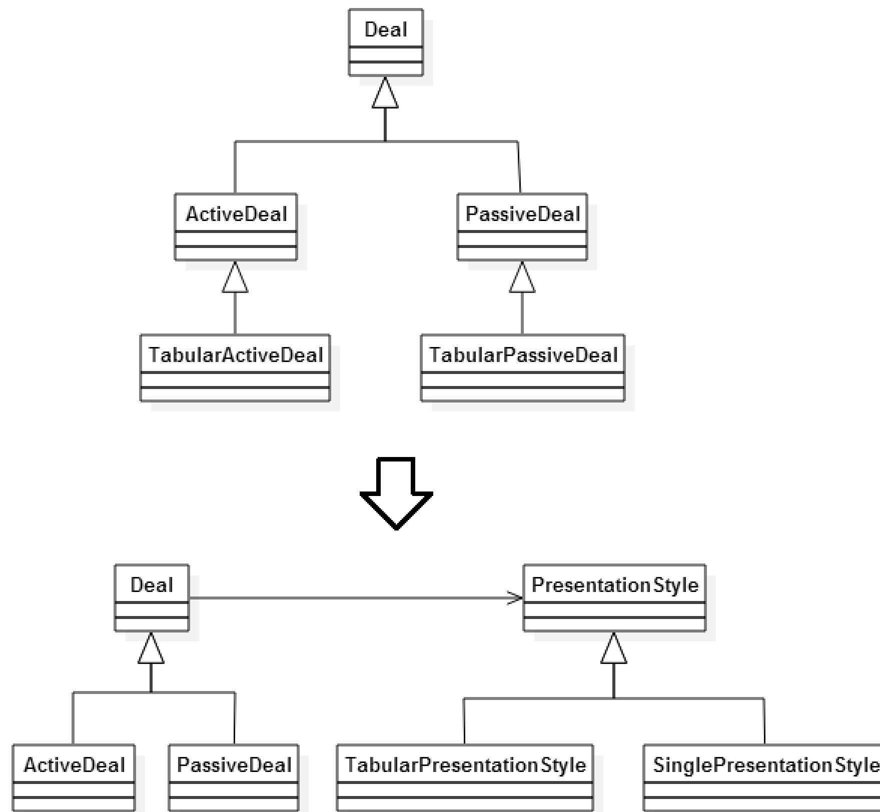


Abbildung A.1.: Refactoring zum Brücken-Entwurfsmuster

werden. Es obliegt jedoch dem Entwickler die jeweiligen Funktionalitäten, um die ein Objekt dynamisch erweitert werden soll, zu ermitteln und aus den Klassen in die entsprechenden Wrapper-Klassen zu extrahieren. Dieses Wissen ist nicht ohne Weiteres aus dem Quelltext extrahierbar, weswegen das Dekorierer-Muster von der Automatisierung ausgeschlossen wird.

A.2.3 Verhaltensmuster

Befehl

Beim Befehl-Muster wird ein Aufruf als ein Objekt gekapselt. Dies lässt sich automatisieren, indem das aufgerufene Objekt von einer Wrapper-Klasse umschlossen wird, welche die aufzurufende Methode aufruft. Diese Wrapper-Klasse stellt das Befehlsobjekt dar, mit dem ein Klient parametrisiert werden kann. A.2

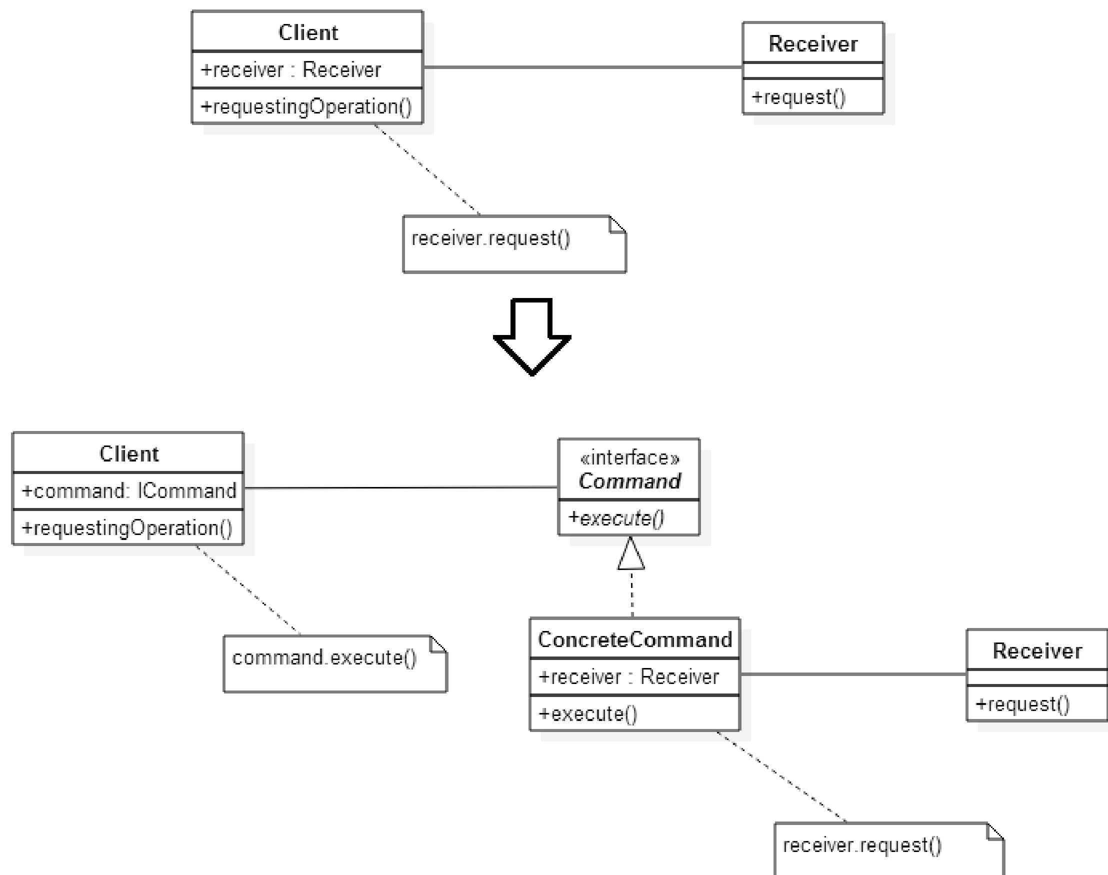


Abbildung A.2.: Refactoring zum Befehl-Entwurfsmuster

Iterator

[ÓC01] beschreibt zwei mögliche Ausgangspunkte für das Iterator-Muster:

- Eine zusammengesetzte Klasse, die keinen Iterator besitzt oder
- Eine zusammengesetzte Klasse, die die Iterator-Methoden selbst über einen Cursor implementiert, anstatt diese in ein separates Iterator-Objekt zu kapseln.

Den ersten Fall schließt Ó Cinnéide selbst aus, da automatisiert nicht eindeutig ermittelbar ist wie über das Aggregat zu iterieren ist [ÓC01]. Den zweiten Fall erscheint nutzbar, jedoch muss der Entwickler angeben welche Methoden und Felder an der Iteration beteiligt sind. Dies ist Entwicklerwissen, denn nicht jede Methode, die auf eine Objektsammlung zugreift, ist auch Teil der Iterations selbst (beispielsweise Methoden zum Sortieren, Leeren oder Ermitteln der beinhalteten Elemente). Daher ist das Iterator-Muster nicht autonom transformierbar.

Memento

Das Memento-Muster versucht den Status eines Objekts (im folgenden Mementee) zu externalisieren, um ihn ohne Aufbrechen der Kapselung außerhalb der Klasse speichern und wieder zurücksetzen zu können. Das Refactoring selbst ist trivial:

- ➢ Es wird eine Memento-Klasse erstellt, die dieselben Felder wie die Mementee-Klasse hat. Diese Klasse implementiert eine leere Schnittstelle, die ein Memento repräsentiert.
- ➢ Die Mementee-Klasse wird um zwei Methoden createMemento und setMemento erweitert, die jeweils eine Instanz des Memento-Schnittstellen-Typs zurückgeben bzw. als Argument annehmen.
- ➢ Die createMemento-Methode erstellt eine Instanz der konkreten Memento-Klasse und überträgt die Werte der Felder der Mementee-Klasse auf das Memento. Komplexe Objekte müssen dabei geklont werden. Die setMemento-Methode tut das umgekehrte – sie überträgt die Werte der Memento-Felder auf die Felder der Mementee-Klasse.
- ➢ Klienten können den Zustand der Mementee-Klasse abfragen und über die leere Memento-Schnittstelle referenzieren – so sind sie von der konkreten Memento-Klasse entkoppelt.

Nicht trivial hingegen ist das Finden entsprechender Spots. Ó Cinnéide geht bereits davon aus, dass entsprechende Methoden bereits existieren und nur angepasst werden müssen. Automatisiert den Zweck solcher Methoden zu erkennen, um sie als Memento-Ausgangspunkt zu identifizieren, ist nicht ohne Weiteres möglich, auf bestimmte Methodennamen hin zu prüfen zu Fehleranfällig. Auch von Seiten des Klienten ist nur schwer zu erkennen, wann versucht wird den Zustand eines Objekts zu ermitteln, zu speichern und zurückzusetzen. Das Einfügen des Memento-Musters ohne den externalisierten Zustand zu verwenden ist nicht sinnig. Daher wird das Memento-Muster mangels detektierbarer Spots nicht automatisiert.

Schablonenmethode

Die Schablonenmethode ist ebenfalls trivial zu automatisieren, wenn ein Entwickler festlegt, welche Methoden über die Schablone repräsentiert werden sollen und welche Aspekte sich im repräsentierten Algorithmus unterscheiden. Um dies automatisiert zu entscheiden müssten die Algorithmen auf Code-Clone hin untersucht und die Unterschiede heraus gefiltert werden. Dies ist nicht trivial, daher wird das Schablonenmethode-Muster von der Automatisierung ausgeschlossen.

Strategie

Die Strategie verhält sich nahezu identisch zum Zustand-Muster, bezieht sich jedoch nicht auf einen veränderlichen Zustand innerhalb einer Klasse, sondern auf einen bei Erstellung einer Klasse ausgewählten Algorithmus. Implementierungstechnisch verhält es sich exakt wie das Zustand-Muster, [Fow99] unterscheidet beispielsweise nicht zwischen den Mustern. Dementsprechend ist es ebenso mittels *Replace Type Code with State/Strategy*- und *Replace Conditional with Polymorphism*-Refactorings ohne Zutun eines Entwicklers identifizier- und transformierbar. Der Unterschied besteht darin, dass nicht zustandsrepräsentierende Code-Fragmente in eigene Klassen verschoben werden, sondern zu verschiedenen Algorithmen gehörende Fragmente [CGZS12].

Zuständigkeitskette

Die Zuständigkeitskette arbeitet mit Delegation. Ein Objekt, das ein anderes Objekt bearbeitet, kann das zu bearbeitende Objekt an ein drittes Objekt zur Bearbeitung weiterleiten. Die Delegation selbst ist einfach zu automatisieren, wenn die Rollen der Objekte bekannt sind. Jedoch automatisiert zu entscheiden wann ein Objekt hinreichend bearbeitet ist und wann es an welches andere Objekt weitergeleitet werden muss, das ist nicht trivial zu entscheiden. Daher wird die Zuständigkeitskette in dieser Arbeit nicht zur Automatisierung nicht umgesetzt.

A.3 Erkannte und transformierbare Spots für Entwurfsmuster-Refactorings

Projektname	Anzahl Typen	Spots für Abstrakte Fabrik		Spots für Kompositum		Spots für Zustand	
		gefunden	umstellbar	gefunden	umstellbar	gefunden	umstellbar
de.ovgu.cse.vecs.saml	926	0	0	10	**1**	0	0
de.ovgu.cse.vecs.saml.mwe2	7	0	0	0	0	0	0
de.ovgu.cse.vecs.saml.tests	70	0	0	1	0	0	0
de.ovgu.cse.vecs.saml.ui	28	0	0	0	0	0	0
de.ovgu.cse.vecs.standalone	1	0	0	0	0	0	0
h2o-master	1246	0	0	32	0	22	**1**
hibernate-search-build-config	10	0	0	1	0	0	0
jchess - bbahl	58	0	0	8	0	0	0
jchess - mbozem	88	13	0	0	0	0	0
JChessThreesome	89	2	**2***	1	0	0	0
MapDB-master	151	0	0	10	0	1	0
maven-aether-provider	29	0	0	5	0	0	0
maven-artifact	34	0	0	2	0	0	0
maven-compat	227	0	0	31	0	0	0
maven-core	415	4	0	113	0	1	0
maven-model	87	3	0	8	0	0	0
maven-model-builder	117	1	0	23	0	0	0
maven-plugin-api	27	1	0	3	0	0	0
maven-repository-metadata	7	3	0	1	0	0	0
maven-settings	17	1	0	2	0	0	0
maven-settings-builder	33	0	0	5	0	0	0
mcMMO-master	317	0	0	70	0	5	0
musicbrainz-data-master	32	0	0	6	0	0	0
neo4j-io	86	0	0	4	0	1	0
neo4j-primitive-collections	151	0	0	6	0	0	0
net.sf.eclipsecs.core	121	1	0	14	0	1	0
net.sf.eclipsecs.sample	3	0	0	0	0	0	0
net.sf.eclipsecs.ui	166	4	0	17	0	0	0
ninja-core	244	3	0	10	0	0	0
orient-commons	157	0	0	0	0	0	0
orientdb-client	15	0	0	1	0	0	0
orientdb-distributed	21	0	0	1	0	0	0
orientdb-enterprise	16	0	0	0	0	0	0
orientdb-graphdb	104	0	0	3	0	0	0
orientdb-object	58	0	0	14	0	3	0
orientdb-server	177	2	0	18	0	1	0
orientdb-tests	305	4	0	16	0	1	0
orientdb-tools	4	0	0	2	0	0	0
oryx-common	99	0	0	7	0	0	0

Projektname	Anzahl Typen	Spots für Abstrakte Fabrik		Spots für Kompositum		Spots für Zustand	
		gefunden	umstellbar	gefunden	umstellbar	gefunden	umstellbar
SerpentChess	81	1	0	0	0	1	0
squirrel-sql	839	4	0	0	0	0	0
titan-berkeleyje	14	0	0	2	0	0	0
titan-cassandra	60	0	0	0	0	0	0
titan-core	385	1	0	13	0	1	0
titan-es	3	0	0	0	0	2	0
titan-hbase	17	0	0	0	0	0	0
titan-lucene	5	0	0	1	0	0	0
titan-persistit	13	0	0	1	0	0	0
titan-rexster	1	0	0	0	0	0	0
titan-test	127	1	0	3	0	0	0
Gesamt	7288	72	2	465	1	40	1

* die gefundenen Spots überschneiden sich - sie sind einzeln, jedoch nicht zusammen umstellbar

Tabelle A.1.: Übersicht über erkannte und transformierbare Spots für Entwurfsmuster-Refactorings

Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich bei der Anfertigung dieser Masterarbeit unterstützt haben.
Ein besonderer Dank gilt meinem Betreuer, Sebastian Nielebock, der mich von Anfang an mit wertvollen Hinweise geradezu überschüttet hat, und auch als die Probleme die Überhand zu gewinnen drohten, immer ein offenes und aufmerksames Ohr hatte. Des Weiteren danke ich natürlich den fleißigen Korrekturlesern Carsten Seidel und Martin Reyher, die offene und aufmerksame Augen beigesteuert haben. Nicht zu vergessen sind auch meine Freunde, die offene und aufschäumende Biere gegen drohende sowie akute schlechte Laune angeboten haben, auch wenn ich viel zu oft ablehnen musste. Außerdem gilt ein besonderer Dank an meine Familie, die mich generell immer unterstützen und mir viel Kleinkram vom Hals halten, so dass ich mich auf das Wesentliche konzentrieren kann. Und Danke an Lara, die schlechtes Timing mit sehr viel Geduld und „Tschaka“ ausgeglichen hat... ;-)

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Magdeburg, 14. November 2014 ....

Masterand