Ein Verfahren zur automatisierten Erkennung von Möglichkeiten zum Einsatz von Entwurfsmustern in objektorientierten Softwaresystemen

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften der
Universität Mannheim

Vorgelegt von
Stefan Burger, Master of Science
aus Rheinzabern

Mannheim, 2017

Dekan: Prof. Dr. Heinz Jürgen Müller, Universität Mannheim
Referent:. Prof. Dr. Oliver Hummel, Hochschule Mannheim
Korreferent: Prof. Dr. Ralf Reussner, Karlsruhe Institute of Technology

Tag der mündlichen Prüfung: 13.09.2017

2017

Ein Verfahren zur automatisierten Erkennung von Möglichkeiten zum Einsatz von Entwurfsmustern in objektorientierten Softwaresystemen

Stefan Burger, MSc
Universität Mannheim
Software-Engineering-Gruppe
B6, 26 68159 Mannheim
2017

Titelbild Quelle: http://www.freedigitalphotos.net/images/view_photog.php?photogid=721

Danksagung

Viele verfolgen hartnäckig den Weg, den sie gewählt haben, aber nur wenige das Ziel.
- Friedrich Nietzsche

Wenn man einen so weiten Weg gegangen ist, hat man viele Menschen getroffen, die einen beeinflusst haben und denen man zu Dank verpflichtet ist. Ich möchte einigen Personen danken, die mich in verschiedenen Abschnitten meines Lebens unterstützt haben und ohne die diese Arbeit nie zustande gekommen wäre. Danke!

Zuallererst möchte ich mich bei meiner Familie, meinen Eltern und meinem Bruder bedanken, die mir über all die Jahre geholfen haben, das Ziel nicht aus den Augen zu verlieren und es nun endlich zu erreichen. Danke!

Dann möchte ich mich bei meinen ältesten Freunden Martin Heid sowie Thomas und Benjamin Kriese bedanken, die mir trotz aller Wirrungen in meinem Leben immer zur Seite gestanden haben. Danke!

Ohne die Diskussion und die Motivation meiner Doktoranden-Kollegen Emanuel Heidinger, Johannes Blankenstein, Thilo Fath, Christian Blümm, Dimitri Tassetto, Christoph Heller und Johannes Schels von der EADS wäre diese Arbeit nie zustande gekommen. Im Zuge dessen möchte ich mich auch bei Helga Bayer und Josef Schalk für ihre Unterstützung bedanken. Danke!

Ebenfalls danke an die Freunde und Freundinnen aus München: Oliver Riske, Sven Konietzko, Daniel Richter, Sebastian Dännert, Marc Söker, Julian Richter, Eckehard Reinhold, Andreas Büning, Malte Buss, Peter Müller, Lucia Thomas Höller, Quirin Gagesch, Andreas Bernhard, Ulriich Lindemann, Max Rott, Julia Loose, Markus Wilczek und Lisa Ockerblohm, die mir mit Rat und Tat zur Seite standen. Danke!

Viele gute Ratschläge und Ideen kamen auch von meinen Studienkollegen und Freunden aus Karlsruhe: Sebastian Docktor, Malisa Ilic, Marjan Samardzic, Michael Pirek und Mark Dannenberg. Danke!

Stephan Rothschuh danke ich für die Bewältigung der Mammutaufgabe, meine Wörter und Sätze in eine lesbare Rechtschreibung zu formen. Danke!

In stürmischen Zeiten braucht man einen ruhigen Hafen. So geht mein Dank an Ulrich Hölscher, Claudia Weiß und Thomas Motsch, die es mir ermöglicht haben, diesen Hafen zu finden und diese Arbeit zu beenden. Danke!

Danke für die Unterstützung der Kolleginnen und Kollegen des Teams D5: Thomas Schmidt, Robert Müller, Stefan Voggenreither, Jürgen Englerth, Martin Peckl, Josef Sperr, Stefan Pfaffel, Udo Raum, Oliver Flasse und Claudia Meinhardt. Danke!

Für die Unterstützung bei der Durchquerung der wissenschaftlichen Untiefen möchte ich mich ganz besonders bei meinem Doktorvater Oliver Hummel und meinen Korreferent Ralf Reussner bedanken. Ohne Ihr Zutun und Ihre Unterstützung hätte ich den Weg nie gefunden. Auch wenn er es nicht wusste, aber im Vergleich zu anderen Betreuern war er wirklich gut erreichbar. Danke!

Zuletzt möchte ich noch einigen Personen aus vergangen Tagen danken, die mich erst auf diesen Weg, bewusst oder unbewusst, gesendet haben. Wolfgang Klein, Marianne Ochsenreither, Peter Weinrich, Werner Wittmann, Peter Henning, Lothar Gmeiner und Bill Sverdlik. Danke!

Bleibt noch, den vielen Ungenannten zu danken, die meinen Weg gekreuzt haben in den letzten Jahren und dabei geholfen haben dieses Projekt ins Ziel zu bringen. Danke!

Eidesstattliche Versicherung

Eidesstattliche Versicherung gemäß § 7 Absatz 2 Buchstabe c) der Promotionsordnung der Universität Mannheim zur Erlangung des Doktorgrades der Naturwissenschaften:

1. Bei der eingereichten Dissertation zum Thema

   „Beiträge zur automatischen Identifikation und Ausbesserung von Design Smells in objektorientierten Softwaresystemen“

   handelt es sich um mein eigenständig erstelltes eigenes Werk.

2. Ich habe nur die angegebenen Quellen und Hilfsmittel benutzt und mich keiner unzulässigen Hilfe Dritter bedient. Insbesondere habe ich wörtliche Zitate aus anderen Werken als solche kenntlich gemacht.

3. Die Arbeit oder Teile davon habe ich bislang nicht an einer Hochschule des In- oder Auslands als Bestandteil einer Prüfungs- oder Qualifikationsleistung vorgelegt.

4. Die Richtigkeit der vorstehenden Erklärung bestätige ich.

5. Die Bedeutung der eidesstattlichen Versicherung und die strafrechtlichen Folgen einer unrichtigen oder unvollständigen eidesstattlichen Versicherung sind mir bekannt.

Ich versichere an Eides statt, dass ich nach bestem Wissen die reine Wahrheit erklärt und nichts verschwiegen habe.

_____

Stefan Burger

Zusammenfassung

Entwurfsmuster (engl. Design Pattern) gelten nach wie vor als eine Möglichkeit, die Struktur (objektorientierten) Quellcodes zu verbessern. Richtig eingesetzt fördern sie die Lesbarkeit und die Flexibilität beim Erweitern der Programmfunktionalitäten. Um die richtige Entscheidung für ein Entwurfsmuster zu treffen, benötigen Entwickler aber ein hohes Detailwissen und viel Erfahrung zu den einzelnen Patterns, um bereits beim Entwurf passende Patterns einsetzen zu können, so dass dies üblicherweise nur erfahrenen Entwicklern und Architekten gelingt.

So entsteht die Notwendigkeit, auch bestehende Systeme auf Schwachpunkte und Einsatzmöglichkeiten von Patterns untersuchen zu müssen. Das kann gerade bei großen Projekten viel Zeit und Aufwand benötigen. Hinzu kommt noch, dass sich Quellcode im Laufe der Zeit verändert, so dass die nachträgliche Erkennung von Einsatzmöglichkeiten für Entwurfsmuster ein sehr wichtiger Forschungsbereich ist.

Diese Arbeit stellt daher eine Lösung vor, wie Entwickler bei der Auswahlentscheidung für geeignete Entwurfsmuster in bestehenden Systemen unterstützt werden können. Es wird ein sogenanntes Empfehlungssystem entwickelt, beschrieben und evaluiert, das direkt auf Basis des Quellcodes arbeitet. Nach der Analyse liefert es sowohl die zu verbessernde Quellcodezeilen, mit einer Bewertung des Verbesserungspotenzials, als auch einen Vorschlag für ein geeignetes Pattern. Durch die automatische Analyse des Quellcodes müssen Entwickler nicht mehr jede Quellcodezeile manuell analysieren, sondern können punktgenaue Vorschläge auf ihre Tauglichkeit überprüfen. Zusätzlich liefert das Empfehlungssystem eine Einschätzung über das mögliche Verbesserungspotenzial zurück und unterstützt somit bei der Entscheidung, wo akuter Handlungsbedarf besteht.

Die Entwicklung der Kandidaten-Erkennungs-Systematik und die Implementierung eines Prototypen (*Design Pattern Candidate Detection Tools* oder DPCDT genannt) konzentriert sich auf sechs repräsentative Entwurfsmuster der Gang-of-Four. Diese Arbeit beschreibt für diese sechs Patterns detaillierte Analysen über ihr Verhalten sowie ihren Aufbau und als Resultat für jedes der Patterns eine automatisierte Kandidaten-Erkennungsregel.

Zum praktischen Einsatz kommen diese aufgestellten Regeln in einem neuartigen Recommedation System, das in der Lage ist, Design-Smells und damit Kandidaten für einen Pattern-Einsatz in Java-Quellcode besser zu identifizieren, als das mit bisherigen Werkzeugen der Fall ist. Eine Evaluation der Systematik und des Systems fand durch eine Gruppe unabhängiger Software-Entwickler auf Basis der Ergebnisse einer Analyse von 25 Open-Source-Projekten durch das DPCDT statt. In Zusammenarbeit mit einem Industriepartner wurde der Quellcode einer weiteren Anwendung, welche vom Partner zur Verfügung gestellt wurde, analysiert und die Ergebnisse mit Entwicklern und Projektleiter der Anwendung diskutiert. Dabei wurden positive Rückmeldungen erzielt, die belegen, dass diese Arbeit einen großen Schritt nach vorne bei der Auswahl und Anwendung von Entwurfsmuster darstellt.

Abstract

Design Patterns are both a blessing and a curse – a blessing because patterns are able to improve source code quality in several ways like flexibility, readability etc., and a curse because an incorrect use has negative effects on the source code quality.

In order to use the right Design Pattern at the right position, developers need to have a thorough understanding of the application structure and behaviour. Only a manual search through a source code is able to provide opportunities for using Design Patterns. This takes a lot of time in big projects with several thousand lines of code. Time, which in most projects could be put to better use in other tasks. Furthermore, source code evolves, which means it changes over the years through adding new features or fixing bugs. These changes open up new opportunities for Design Patterns. However, checking the whole project for Design Pattern opportunities after every major release is impossible.

This thesis presents a novel approach that aims to help developers in choosing Design Patterns and identifying source code lines in which to use them. The approach implements a Design Pattern opportunity recommendation system, which uses the available source code as a foundation its analysis. As a result, it delivers a list of source code lines that can be optimized by means of a Design Pattern and recommends a Design Pattern to use for each of those lines. An advantage of this new approach is a fully automatic source code analysis. This saves time, allowing developers to focus on their core tasks and later improve their work by implementing the recommended changes.

For this thesis, I have designed a recommendation system based on detection rules and thresholds for six Gang-of-Four patterns (builder, decorator, façade, mediator, state and strategy). The Design Pattern Candidate Detection Tool is able to analyse every kind of Java source code. For the purpose of evaluation, 25 open source projects were analysed with this tool. All results were evaluated by a group of software experts. Additionally, a software used by an insurance company was analysed, and the results were evaluated in collaboration with that company's software experts

.

Inhaltsverzeichnis

1 Einleitung

„Wer ein Problem anpackt, hat es schon halb gelöst."
– Horaz (65-8 v.Chr.), röm. Dichter

1.1 Motivation

Zu Beginn des 21. Jahrhunderts hat sich die Verfügbarkeit von Informationen auf der Welt grundlegend verändert. Informationen, die früher ganze Bibliotheken gefüllt haben, sind nun überall und zu jeder Zeit an jedem Ort verfügbar. Mit der großflächigen Verbreitung von Computern hat sich die Arbeitsweise, die Art der Kommunikation, ja sogar das Konsumverhalten von Unternehmen und Menschen gewandelt. Für die Menschheit hat das „Information Age" (Castells, 1996) begonnen. Computer sind längst keine großen schwarzen Kisten mehr, welche riesige Hallen füllen, sondern sie passen mittlerweile ans Handgelenk oder begleiten uns in Form eines Smartphones in der Hosentasche.

2010 veröffentlichte das IEEE Annals of the History of Computing den Artikel „Understanding How Computing Has Changed the World" von Thomas J. Misa (Misa, 2007), in welchem der Autor die weitreichenden Änderungen durch die Erfindung und Verbreitung von Computern beschreibt. Neben der hohen Verbreitung gab es noch einen weiteren wichtigen Aspekt, der die Veränderung unseres täglichen Lebens durch Computer erst ermöglichte: Der Ausbau von Kommunikationsnetzen. Erst durch deren Kabel und Funkwellen wurden Computer Teil des Alltages.

Computer und Netzwerke stellen nur das Grundgerüst für den Teil des Information Age dar, den viele Menschen tagtäglich verwenden. Eine Anwendung wäre ohne Software, die als Abstraktionsschicht zwischen Hardware und Benutzer fungiert, undenkbar. Software kommt hierbei eine Vermittlerrolle zu. Zum einen muss sie komplexe logische Schritte verschiedener Prozesse einfach und verständlich darstellen. Zum anderen kapselt sie die Komplexität und das Verhalten, wie das Behandeln von Fehlern, aus den unteren Schichten ab. Fehler in der Software haben Auswirkungen auf Hardware und Benutzer. Je höher die Qualität einer Software, desto höher die Akzeptanz beim Benutzer, da sich dieser schnell mit der Software zurecht findet und auch seine Aufgaben besser erledigen kann.

Durch ihre wichtige Stellung können qualitative Mängel innerhalb einer Software verheerende Auswirkungen auf das Vertrauen des Nutzers in die Software und somit auch auf das Vertrauen des Nutzers in den Hersteller haben. Als bspw. 2013 die Neuauflage des Erfolgsspiels Sim City veröffentlicht wurde, schlug die Freude vieler Fans schnell in Wut und Fassungslosigkeit um: Das Produkt war so mit Fehlern übersät, dass der Spielstart bis zu 30 Minuten dauerte und selbst mehrere Updates nur wenig Besserung brachten. Hinzu kam noch, dass es nur mit einer bestehenden Internet-Verbindung spielbar und die Server-Infrastruktur in den ersten Wochen mehrfach nicht erreichbar war. Die Kunden gingen auf die Barrikaden. Der Publisher Electronic Arts (EA) (Stöcker, 2013) ruderte zurück und nahm das Spiel sogar für kurze Zeit aus dem Verkauf. Anfang 2015 wurde das Entwicklerstudio von EA geschlossen, was auch auf die schlechte Qualität der Software zurückzuführen war (Nestler, 2015).

Sim City ist nur ein Beispiel für schlechte Softwarequalität mit beträchtlichen finanziellen Auswirkungen. Die Zahl qualitativ mangelhafter Softwaresysteme wächst jedoch nach wie vor (Hoffmann, 2013). Die Antwort auf die Frage, was unter schlechter bzw. guter Qualität von Software verstanden wird, ist sehr komplex.

Für den Anwender definiert sie sich über die für ihn wahrnehmbaren Merkmale wie Benutzbarkeit, Funktionalität und Korrektheit, also über die nach außen hin sichtbaren, sogenannten externen Qualitätsmerkmale (Stavrinoudis & Xenos, 2008). Für den Anwendungsentwickler gehören neben diesen externen Merkmalen auch interne Merkmale wie Wartbarkeit, Portierbarkeit und Verständlichkeit (vgl. z. B. (McConnell, 2004)) zur Softwarequalität.

Externe Qualitätsmerkmale sind relativ leicht zu definieren, da sie über Antwortzeiten, die Anzahl der auftretenden Fehler oder Rückmeldungen durch die Benutzer ermittelt werden können. Die interne Qualität hingegen ist deutlich schwieriger zu bestimmen. Beispielsweise können Wartbarkeit oder Verständlichkeit von Quellcodes nur schwer in Worte, geschweige denn in Zahlen gefasst werden. Es liegt zwar nahe, dass diese vor allem durch die Qualität des Quelltextes selbst beeinflusst werden, aber auch das heutige Verständnis von gutem Code basiert eher auf der Anwendung verschiedener Coding Guidelines und Best Practices (vgl. (McConnell, 2004) und (Martin, 2008)) als auf einer klaren Definition und konkret messbaren Kriterien.

Ansätze, die versuchen, Codequalität in Zahlen oder Worte zu fassen, sind dennoch keineswegs neue Ideen, sondern werden bereits seit den 1970er Jahren intensiv erforscht. Insbesondere die bereits über 35 Jahre alte Methode von Halstead (vgl. (Halstead, 1977)), den Quellcode zu vermessen, um beispielsweise sein „Volumen“ über die Anzahl der enthaltenen Operanden (also z. B. der Variablen) und Operatoren (wie Zuweisungen, Rechenoperationen etc.) zu bestimmen, ist intuitiv einleuchtend und wird entsprechend von zahlreichen heute auf dem Markt verfügbaren Messwerkzeugen unterstützt. Gleiches gilt für die Zählung von Codezeilen oder -statements sowie für Komplexitätsmetriken wie etwa McCabes zyklomatische Komplexität (McCabe, 1976) welche die Anzahl und Verschachtelungstiefe von Verzweigungen erfassen. Ferner wurden schon in den 1990er Jahren die ersten Metriken für die damals in der Praxis ankommenden objektorientierten Programmiersprachen (kurz: OO-Sprachen) entwickelt. Sowohl bei traditionellen Metriken als auch bei ihren moderneren Nachfolgern wurde allerdings bislang nicht zweifelsfrei geklärt, wie sich erhobene Messwerte auf die Codequalität und letztlich auf die interne oder gar die externe Softwarequalität auswirken.

In den späten 1980er Jahren begannen Beck, Cunningham und Gamma mit der Arbeit an einer anderen Strategie, um die Qualität von Quellcodes zu erhöhen. Ihr Ansatz baut auf einer Idee aus der Architektur zur Gestaltung von Gebäuden und geht auf C. Alexander zurück.

Die Idee von Alexander basierte auf der Sammlung von anerkannten Lösungen für die immer wiederkehrenden Probleme beim Bau von Gebäuden. Sein Ziel war es, Muster und eine Sprache dafür zu beschreiben, wie erfolgreich Städte und Gebäude erschaffen werden können. In seiner Arbeit Pattern Language (Alexander, Ishikawa, & Silverstein, 1977) beschrieb er 253 solcher Muster. Während sein Ansatz in der Architektur auf Kritik stieß, wurde er später in der Software-Entwicklung aufgegriffen. Seine Definition von Muster lautete:

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice: (Alexander, Ishikawa, & Silverstein, 1977)

Dieser Ansatz ist heute in der Software-Entwicklung bekannt als Design Pattern oder Entwurfsmuster. Durch die Verwendung von Entwurfsmustern werden wiederkehrende Probleme effizient bewältigt.

Ein Beispiel für wiederkehrende Probleme in der Entwicklung von Software liegt in der Abkapselung von Wissen über die Verwendung verschiedener spezifischer Funktionen, die alle das gleiche Ergebnis erzeugen, z. B. bei der Berechnung des kürzesten Weges, aber ein unterschiedliches Vorgehen verwenden. Häufig wird für eine solche Aufgabe der Dijkstra-Algorithmus (Dijkstra, 1959) verwendet. In einigen Anwendungsfällen kann aber auch Bellman-Ford-Algorithmus (Bellman, 1956) oder Floyd-Warshall-Algorithmus (Floyd, 1962) nötig sein. Die drei genannten Algorithmen berechnen den kürzesten Weg in einem Gebilde von Knoten. Sie unterscheiden sich jedoch in der Art der Berechnung. Benötigt ein Programm alle drei Algorithmen, so wäre eine mögliche Lösung die Kapselung hinter einer generischen Schnittstelle. Die genannte Schnittstelle verlangt den Graphen sowie einen Start- und Endpunkt. Je nach Aufgabe kann der Algorithmus gewählt werden.
Der Aufrufer benötigt kein Wissen über die verwendeten Algorithmen, da das Ergebnis immer im gleichen Format zurück geliefert wird. Veränderungen am Algorithmus, z. B. neue Heuristiken, können unabhängig vom Aufrufer durchgeführt werden. Bekannt ist dieser Lösungsansatz als Strategy Pattern. Eine Strategy kapselt die einzelnen Algorithmen voneinander ab, so dass diese unabhängig voneinander verändert werden können.

Gamma et al. veröffentlichten 1994 den heute noch bekanntesten Katalog an Entwurfsmustern. Die sogenannte „Gang of Four“ (GoF) beschreibt in ihrem Buch 23 Entwurfsmuster, unterteilt in drei Kategorien (Erzeugung, Struktur und Verhalten).

In den darauffolgenden Jahren wurden weitere Patterns entwickelt. So stieg die Anzahl der verfügbaren Patterns stetig an. Einige Pattern werden von Rising (Rising, 2000) oder Schmidt et at. (Schmidt, Stal, Rohnert, & Buschmann, 2013) beschrieben.

Trotz einer hohen Anzahl an verfügbaren Patterns, stellt sich ihr Einsatz heute immer noch als schwierig dar. Zum einen ist eine Konkretisierung von Patterns komplex und erfordert viel Know-how (Sommerlad & Noble, 2007). Zum anderen stellt aber bereits die Auswahl eines geeigneten Entwurfsmusters eine Herausforderung für den Entwickler dar. So schreibt Peter Sommerlad in einem Artikel für das IEEE Software-Magazin über den Einsatz von Entwurfsmuster durch Entwickler mit wenig Erfahrung:

„… the relative lack of expertise or brilliance can easily result in bigger software design disasters than would occur without them.“ (Sommerlad & Noble, 2007)

Im Gegensatz zu Software-Metriken sind die positiven Auswirkungen von Entwurfsmuster, wenn richtig eingesetzt, unumstritten. Doch die Wissenslücke um ihren Einsatz konnte bisher noch nicht geschlossen werden.

1.2 Problemstellung

Das größte Problem beim Einsatz von Entwurfsmuster bleibt das Erkennen des richtigen Stelle für den Einsatz eines Entwurfsmusters und die Auswahl des richtigen Patterns. In der bisherigen Literatur wurden, basierend auf Templates und Expertensystemen, Methoden entwickelt, die Entwickler bei der Auswahl eines passenden Entwurfsmusters unterstützen sollen. Eine ausführliche Liste entsprechender Ansätze wird in Kapitel 2.4 diskutiert. Zusammenfassend kann gesagt werden, dass einige gute Grundlagen erarbeitet wurden, jedoch keiner der Ansätze überzeugen konnte. Dies mag daran liegen, dass die meisten Verfahren nur bei der Auswahl des

Entwurfsmusters unterstützen. Es bleibt dem Entwickler überlassen, zuerst die richtige Stelle im Quellcode oder im Design zu finden, an der ein Pattern überhaupt möglich und sinnvoll wäre.

Ein weiteres Problem stellt die nachträgliche Implementierung von Entwurfsmustern im Zuge von Wartungsarbeiten oder neuen Anforderungen dar. Von ihrer Grundidee her sind Entwurfsmuster dafür entwickelt worden, in der Design-Phase eingesetzt zu werden. Viele aktuelle Verfahren unterstützen die Entwickler deshalb nur bei der Auswahl von Patterns während dieser Phase. Doch Veränderungen einer Software durch Anpassungen im Laufe ihres Lebenszyklus können große Auswirkungen auf ihr ursprüngliches Design haben. Diese Veränderungen eröffnen neue Möglichkeiten für den sinnvollen Einsatz von Entwurfsmustern. Der Einbau von neuen Funktionen oder anderen Algorithmen kann ein Design soweit verändern, dass der Einsatz von Entwurfsmustern sinnvoll wäre, um die Wartbarkeit oder Flexibilität der Software weiter zu erhöhen oder wenigstens auf dem gleichen Niveau zu halten, denn späte Veränderungen an Software können zu großem Aufwand führen (Slaughter, Harter, & Krishnan, 1998).

Ein weiterer Punkt, warum die nachträgliche Betrachtung eines Quellcodes hinsichtlich Entwurfsmustern sinnvoll ist, liegt in den Annahmen die zur ursprünglichen Auswahl der Entwurfsmuster geführt hat. Nicht jedes Pattern wird für eine Aufgabe richtig gewählt oder die ursprüngliche Aufgabe hat sich mit neuen Funktionalitäten soweit abgewandelt, dass ein anderes Pattern geeigneter wird. Daraus entsteht der Bedarf, ein bereits implementiertes Entwurfsmuster nachträglich durch ein anderes zu ersetzen. So kann aus einem Bridge-Pattern beispielsweise eine Facade oder aus einem Builder ein Decorator werden. Des Weiteren besteht die Möglichkeit, dass aus einem gewählten Entwurfsmuster durch ständige Veränderung ein Anti-Pattern (Smith & Williams, 2000) wird. So kann ein Mediator schnell zum Anti-Pattern God-Klasse mutieren.

Im Zuge dieser Arbeit wurde untersucht wie Entwurfsmuster effektiv eingesetzt und wie Entwickler bei der Suche nach Verbesserungspotenzial unterstützt werden können. Im Fokus standen folgende Forschungsfragen:

- *Wann und wo sollte ein Entwurfsmuster angewandt werden?*
  Bei der Verwendung von Entwurfsmustern ist es schwierig, überhaupt einen Punkt (z. B. Klasse oder Quellcodezeile) zu finden, an dem man sie sinnvoll einsetzen kann.

- *Wie könnte eine Auswahl eines adäquaten Entwurfsmusters erfolgen?*
  Wird eine Stelle im Quellcode identifiziert, muss eine Auswahl getroffen werden, welches Entwurfsmuster hier eine Verbesserung bringen könnte. Dabei kollidiert diese Frage mit der Frage zuvor. Es bleibt zu klären, ob erst die Verortung oder erst die Auswahl des Entwurfsmusters beantwortet werden sollte. Auf der einen Seite müsste, wenn eine Codestelle identifiziert wurde, das notwendige Pattern schon bekannt sein. Auf der anderen Seite sollte erst das Entwurfsmuster gewählt werden, um zu wissen, nach welchem Problem im Code gesucht werden soll.

- *Ist es möglich, die Auswahl und die Suche nach der entsprechenden Stelle zu automatisieren?*
  Um Entwickler zu entlasten, sollte die Analyse automatisch durchgeführt werden, denn gerade bei großen Projekten würde es viel Zeit und Aufwand bedeuten, immer wieder den kompletten Quellcode zu analysieren. Wie bei automatisierten Softwaretests würde ein automatisiertes Vorschlagssystem nach jedem Build-Prozess oder Repository Commit eine Analyse durchführen. Der Entwickler würde ohne weiteres Zutun die Vorschläge geliefert bekommen. Ein solches Vorgehen spart Zeit.

1.3 Ziel und Vorgehen

Ausgehend von den vier geschilderten Problemstellungen, kann eine klare wissenschaftliche Zielsetzung für diese Arbeit festgelegt werden.

Sie lautet wie folgt:

Ziel der Arbeit

Das Ziel dieser Arbeit liegt in der Entwicklung einer Methode zur automatischen Erkennung von schlecht entworfenen Quellcode-Strukturen bzw. Design Smells und der Unterstützung bei Auswahl eines passenden Entwurfsmusters, das die interne Software-Qualität an der gefundenen Stelle verbessern könnte.

Um dieses Ziel zu erreichen, wurde ein Vorgehen aus fünf Schritten festgelegt.

1. Festlegung der notwendigen Anforderungen zur Erkennung von Quellcodestellen
   Im ersten Schritt ist es notwendig, festzulegen, welche Anforderungen erfüllt sein müssen, damit eine Quellcodestelle identifiziert werden kann, welche mit einem Entwurfsmuster verbessert werden soll. Dazu zählen die Auswahl entsprechender Technologien wie Syntaxanalyse und das Festlegen eines Vorgehensmodells. Bei der Definition des Vorgehensmodells sind folgende Punkte zu beachten:
   - Wie kann eine Sammlung von Daten basierend auf dem Quellcode erfolgen, so dass diese später bei der Detektion von Verbesserungspotenzial in anderen Anwendungen weiterverwendet werden können?
   - Wie sollte ein Identifikationsmechanismus für Entwurfsmusterkandidaten definiert werden (z. B. Regelsystem), der mit den zuvor gesammelten Daten arbeitet?
   - Welche Schritte sind notwendig, um aus den gesammelten Daten Informationen über die Struktur des Quellcodes zu erhalten?
   - Welche Möglichkeiten gibt es, aus den gewonnen Informationen Wissen abzuleiten, welches es erlaubt, eine bestimmte Stelle als verbesserbar einzustufen?
2. Bestimmung allgemeiner Schritte zur Definition von Erkennungsregeln
   Es ist zu klären, wie für ein spezifisches Pattern eine entsprechende Stelle gefunden werden kann. Ausgehend von den oben festgelegten Methoden und gewählten Technologien, wird für die GoF-Patterns die Möglichkeit untersucht, eine Regel zur Erkennung eines Entwurfsmuster-Kandidaten aufzustellen. Dabei ist auch festzustellen, welche Entwurfsmuster sich für eine Erkennung in diesem Sinne eignen und welche ausgeschlossen werden müssen.
3. Festlegung von Erkennungsmerkmalen zur Bestimmung des Verbesserungsbedarfes
   Es gilt, ein Verfahren zu definieren, das erkennt, wann ein Entwurfsmuster einen

vorhandenen Quellcode verbessern kann und ob eine Anpassung zwingend notwendig ist. Dazu müssen zuerst Merkmale (z.B. definiert werden, die es erlauben potenziell verbesserungswürdige Stellen zu beschreiben). Die Merkmale müssen so festgelegt sein, dass sie eine zu verbessernde Quellcode-Stelle allgemein beschreiben können, um universell einsetzbar zu sein. Es muss untersucht werden, wie weit Merkmale pro Entwurfsmuster definiert werden müssen oder ob sie für mehrere Patterns genutzt werden können. Damit das Verbesserungspotenzial bestimmt werden kann, bedarf es eines Verfahrens, das erkennt, wann die Ausprägung eines Merkmals stark genug ist um ein Pattern einzusetzen.

4. Implementierung des Design Pattern Candidate Detection Tools und Analyse von Projekten
   Um die Tauglichkeit der Ergebnisse der Aufgaben 1 bis 3 nachzuweisen, ist es erforderlich, ein Programm zu implementieren, das in der Lage ist, Quellcodes zu analysieren und die Regeln auszuwerten. Zudem wird eine Auswahl an Projekten getroffen, die ein möglichst genaues Abbild der Realität darstellt. Außerdem muss die Stichprobe groß genug sein, um eine Evaluierung zu ermöglichen.

5. Evaluierung der Ergebnisse
   Die vom Tool gewonnenen Daten müssen evaluiert werden, um eine Tauglichkeit des Konzeptes nachzuweisen. Dazu ist ein Verfahren festzulegen und dieses auf die zuvor ermittelten Ergebnisse anzuwenden. Die Ergebnisse der Evaluierung werden im letzten Schritt diskutiert und ein abschließendes Fazit über das gesamte Vorgehen gezogen.

1.4 Abgrenzung

Das hier behandelte Themengebiet umfasst große Bereiche des Software-Engineerings. Um den Arbeitsumfang einzuschränken, gibt es eine folgende Abgrenzung der behandelten Themen.

Die Arbeit fokussiert sich auf die interne Software-Qualität eines Quellcodes und dessen Wartbarkeit. Externe Qualitätsmerkmale wie Verfügbarkeit oder Laufzeit werden nicht berücksichtigt.

Ein zentraler Punkt der Arbeit liegt im Nachweis der Möglichkeit, Entwicklern Entwurfsmuster-Vorschläge zu liefern, um bestehenden Quellcode zu verbessern. Genutzte Technologien, Bibliotheken oder Programme, z. B. PMD, werden nur kurz erläutert, um ihre Verwendung innerhalb der Arbeit zu erklären.

Des Weiteren wird nicht in Detail auf Themen eingegangen, die zur Unterstützung von Methoden oder Prozessen innerhalb der Arbeit verwendet werden und damit im Detail nicht von Wichtigkeit für das eigentliche Ziel sind. Hierzu gehören z. B. *Abstract Syntax Tree* (AST) oder Graphentheorie.

Um den Umfang des *Design Pattern Candidate Detection Tool* einzugrenzen und den Fokus auf den Nachweis der Machbarkeit und Definition der Erkennungsmethoden zu setzen, werden nur Programme und Quellcodes in der Programmiersprache Java verwendet.

Die untersuchten Entwurfsmuster werden auf die Liste der von der Gang of Four veröffentlichten 23 begrenzt. Diese Entwurfsmuster sind weit verbreitet und waren schon Gegenstand von anderen Untersuchungen und Analysen in ähnlichen Bereichen. Der Verbreitungsgrad anderer Patterns ist zum Teil sehr gering und einige dieser Patterns finden nur in bestimmten Umgebungen ihren Einsatz, z. B. JEE-Anwendungen. Es wird davon ausgegangen, dass diese Ansätze verallgemeinert werden können.

1.5 Gliederung der Arbeit

In den folgenden Kapiteln wird auf die verschiedenen Themen und Tätigkeiten eingegangen, die notwendig waren, um Kandidaten für Entwurfsmuster im Quellcode zu identifizieren. Diese Schritte gehen deutlich über bisherige Arbeiten hinaus.

Kapitel 2 liefert zunächst einen vertieften Einblick in den aktuellen Stand der Forschung zu Themen, die für diese Arbeit wichtig sind. Zu diesen Themen gehören die Definition von Softwarequalität, Softwaremetriken, die Definition und Erkennung von Code Smells und Refactorings, die Erkennung von Entwurfsmuster und eine kurze Beschreibung von Recommendation Systems. Es werden auch die aktuellen Herausforderungen und Lücken der einzelnen Themen kurz beleuchtet.

Eine Einführung in die hier näher untersuchten Entwurfsmuster und ihren Charakteristiken sind in Kapitel 3 zu finden.

In Kapitel 4 wird das allgemeine Vorgehen zur Festlegung von Erkennungsregeln und die Bestimmung von Grenzwerten mit Hilfe der Software-Indizes der Merobase-Suchmaschine erläutert.

Die verwendeten Technologien und einige Grundlagen, die für die Festlegung der Erkennungsregeln sowie der Implementierung der Analysesoftware notwendig sind, werden ebenfalls in Kapitel 4 ausgefüllt.

Kapitel 5 legt die Details der einzelnen Erkennungsregeln dar und beschreibt Erkennungsideen für Entwurfsmuster, die noch nicht genauer analysiert wurden. Weiterer Bestandteil des Kapitels sind Erläuterungen dazu, warum für bestimmte Entwurfsmuster keine Regeln möglich sind.

Für weitere Evaluierungen wurde ein Prototyp implementiert, der in der Lage ist, Java-Quellcode zu analysieren. Die implementieren Erkennungsmethoden und die Architektur werden in 6 skizziert.

Die durch das DPCDT gewonnenen Ergebnisse aus der Analyse von Open Source- und ausgewählten Industrie-Projekten sind in Kapitel 7 zusammengefasst. Dieses Kapitel beschreibt auch die einzelnen Schritte zur Evaluierung der gesammelten Ergebnisse durch eine Expertenbefragung.

Abschließend werden in Kapitel 8 und 9 die Ergebnisse zusammengefasst sowie ein weiterer Ausblick auf mögliche zukünftige Tätigkeiten gegeben.

2 Aktueller Stand der Forschung und Potenzial

„Das schönste Glück des denkenden Menschen ist, das Erforschliche erforscht zu haben und das Unerforschliche zu verehren."

- Johann Wolfgang von Goethe (1749–1832), dt. Schriftsteller

Die statische Analyse von Quellcode und die dazu gehörende Bestimmung der Softwarequalität gehören zu den Bereichen des Software-Engineerings, die seit Jahrzehnten Ziel von verschiedenen Forschungsansätzen waren bzw. sind. Aus diesen Forschungen stammen Ansätze wie Refactoring, Softwaremetriken, Patterns u. v. m. Die folgenden Abschnitte fassen den aktuellen Stand der unterschiedlichen Forschungsrichtungen zusammen, welche im Zusammenhang mit dieser Arbeit stehen. Dabei werden ebenfalls die Unterschiede zwischen Zielen dieser Arbeit und der bisherigen Forschung herausgearbeitet. Die Ausarbeitung der Unterschiede unterstreicht den neuartigen Anspruch dieser Forschungsarbeit. Zum Zeitpunkt des Entstehens der Arbeit sind dem Autor keine ähnlichen Ansätze bekannt.

2.1 Statische Softwareanalyse mit Softwaremetriken

Ein früher Ansatz, die Qualität einer Software zu bestimmen, liegt in der Vermessung ihrer statischen Quellcode-Merkmale wie LoC, Anzahl an Methoden u. v. m. Dieses Vorgehen wird in der Literatur unter dem Begriff statische Code-Analyse (Louridas, 2006) behandelt. Innerhalb der statischen Code-Analyse gibt es den Begriff der Softwaremetriken, welcher die verschiedenen Berechnungsmethoden und Messgrößen zusammenfasst, die zur Vermessung verwendet werden, um z. B. mögliche Schwachstellen in der Softwarequalität zu identifizieren.

Der Ansatz des Vermessens von Quellcode hat seinen Ursprung in den 1970er Jahren und wird seitdem intensiv erforscht. Ziel war damals wie heute, die Codequalität in Zahlen erfassbar zu machen. Die nachfolgende Tabelle 1 zeigt eine Übersicht über die bekanntesten Softwaremetriken, die auch von vielen kommerziell verfügbaren Werkzeugen unterstützt werden.

Tabelle 1: Übersicht bekannter Metriken

Halstead-Metriken (Halstead, 1977)	**Objektorientierte Metriken** (Chidamber & Kemerer, 1994)
• Operators • Operands • Vocabulary • Volume • Difficulty • Effort • Length	• Lack of Cohesion of Methods • Depth of Inheritance Tree • Count of Base Classes • Count of Coupled Classes • Count of Derived Classes • Count of All Methods • Count of Instance Methods • Count of Instance Variables • Count of Methods
LoC-basierte Metriken	**Komplexitätsmetriken** (McCabe, 1976)
• Lines of Code • Commented Lines of Code • Number of Statements • Number of Declarations	• Maximale zyklomatische Komplexität • Durchschnittliche zyklomatische Komplexität • Maximale Verschachtelung

Eine der ersten und heute immer noch häufig verwendeten Softwaremetriken wurde von McCabe (McCabe, 1976) entworfen. Die sogenannte zyklomatische Komplexität (CC) basiert auf der Grundlage der Graphentheorie und gehört zu den Komplexitätsmetriken, d.h. die Metrik soll die Komplexität des Quellcodes erfassen und somit die Verständlichkeit für den Leser in einer Zahl zusammenfassen. Grundannahme von McCabe war: Je komplexer der zugrundeliegende Programmflussgraph, desto schwieriger wird es, den Quellcode und seine Funktion zu verstehen. Die Metrik definiert sich über die Anzahl der Verzweigungen innerhalb einer Funktion, wobei jede Verzweigung (z. B. IF-ELSE oder FOR Statement) die Metrik erhöht. McCabe selbst definiert eine erste Grenze bei 10; ab diesem Wert wird es laut seiner Definition schwer, die Funktion zu verstehen.

Eine andere Gruppe von Metriken wurde von Halstead (Halstead, 1977) unter dem Namen Halsteads *Software Science Metrics* vor über 35 Jahren entwickelt. Halsteads Arbeiten waren die ersten Ansätze, die Vermessung von Quellcode wissenschaftlich zu beschreiben und Metriken zu entwickeln, die mit anderen Maßeinheiten wie Meter und Liter vergleichbar sind. Grundlage jeder seiner Metriken ist die Anzahl der enthaltenen Operanden (z. B. die enthaltenen Variablen) und Operatoren (z. B. Zuweisungen, Rechenoperationen etc.). Diese zu bestimmen, sollte laut Halstead intuitiv einleuchtend sein. In der Praxis zeigt sich die Anwendung als komplex, da es keine klare Definition gibt, was ein Operand und was ein Operator ist. Bei seiner Methodik unterscheidet Halstead zwischen der Gesamtanzahl von Operatoren bzw. Operanden und der Anzahl einzigartiger Operatoren und Operanden. So gibt N die Gesamtmenge von Operatoren (N1) und Operanden (N2) inklusive Mehrfachnennungen im System an. Bei der Zählung verschiedener Operatoren (n1) und Operanden (n2) zählt jeder Operator und Operand im Gegensatz zur Gesamtmenge nur exakt einmal. Aufbauend auf diesen Grundlagen erstellte Halstead mehrere

Berechnungswege zur Bestimmung verschiedener, von ihm festgelegter Qualitätsmerkmale. Eine seiner Metriken ist der Schwierigkeitsgrad (difficulty level, D) welcher sich wie folgt berechnet:

$$D = \frac{n1}{2} * \frac{N2}{n2}$$

Halstead ging davon aus, dass sich der Schwierigkeitsgrad, einen Quellcode zu verstehen, und somit auch die Neigung des Programms zu Fehlern erhöht, wenn z. B. die Gesamtanzahl an Operanden (N2) oder die Anzahl neuer Operatoren (n1) steigt. Die Erhöhung der von n1 oder N2 führt dazu, dass mehr Informationen (z. B. mehr genutzte Variablen) vom Leser verarbeitet werden müssen. Eine andere Metrik von Halstead, das Volumen (V), gibt die Größe des Quellcodes wieder. Die Berechnung erfolgt aus der Gesamtanzahl Operatoren (N1) und Operanden (N2) sowie des Binär-Logarithmus aus der Summe der verschiedenen Operatoren (n1) und Operanden (n2). Mathematisch wird dies wie folgt dargestellt:

$$V = N1 + N2 * \log_2(n1 + n2)$$

Steigt die Anzahl an Operatoren und Operanten im System, so wird auch das Volumen größer. Es gibt noch eine Anzahl weiterer Halstead-Metriken wie Programm Level, Implementierzeit oder die geschätzte Anzahl der ausgelieferten Bugs; für weiterführende Informationen wird auf die bekannte Literatur verwiesen (Halstead, 1977).

Eines der ersten komplexeren Modelle zur Bestimmung der Wartbarkeit von Software stellt der sogenannte *Maintainability Index* (MI) dar, welcher von Coleman et al. (Coleman, Ash, Lowther, & Oman, 1994) in den 1990er Jahren bei HP entwickelt wurde. Die Berechnung dieses Modells basiert auf einer Kombination verschiedener Softwaremetriken (z.b. zyklomatische Komplexität) und konstanter Werte. Die konstanten Werte werden vorher über ein empirisches Verfahren ermittelt. Die genaue Formel lautet:

$$\begin{aligned} &Maintainability\ Index \\ &\quad = 172 * 5{,}2 \\ &\quad * ln(Volumen) - 0{,}23 * Cyclomatic\ Complexity - 16{,}2 * ln(Lines\ of\ Code) \end{aligned}$$

Das Ergebnis stellt die Wartbarkeit eines Systems dar, wobei ein Wert von unter 65 als schwer, ein Wert zwischen 65 und 85 als durchschnittlich und ein Wert ab 86 als gut zu warten definiert wurde. Genauere Untersuchungen des Modells und ein Vergleich der Ergebnisse zwischen verschiedenen Systemen und die dünne empirische Grundlage der fixen Werte zeigten jedoch massive Schwächen dieses Verfahrens (Welker, 2001).

Der Vollständigkeit halber sollte auch die Gruppe der objektorientierten Metriken (OO-Metriken) kurz erwähnt werden. Aufbauend auf den zuvor genannten Metriken entwickelten Chidamber und Kemerer (Chidamber & Kemerer, 1994) eine Reihe von Metriken, die speziell für den Einsatz in objektorientierten Programmiersprachen geeignet sind. Sie reagierten auf die Kritik, Non-OO-Metriken seien für den Einsatz in modernen Sprachen nicht geeignet. Ihre theoretischen Grundlagen haben OO-Metriken in den Arbeiten von Roberts (Roberts, 1979). Der Autor vergleicht ein objektorientiertes Design mit einem relationalen System, das auf Elementen, Beziehungen und Operationen basiert. Chidamber und Kemerer verwenden dieses Modell, um ihre Metriken zu erstellen. In dieser Gruppe von OO-Metriken gehören *Weighted Methods per Class* (WMC) oder *Depth of Inheritance Tree* (DIT).

Über die Jahrzehnte wurde eine Vielzahl an Softwaremetriken zur Quellcode-Analyse veröffentlicht. Im Anhang an diese Arbeit (siehe Kapitel 10) findet sich eine große (wenn auch unvollständige) Liste bekannter Metriken. Die verschiedenen Metriken sind nach dem untersuchten statistischen Merkmal zusammengefasst. So gibt es eine ganze Reihe von Metriken, die Lines of Code als Grundlage verwenden, oder eine andere Gruppe, die Verbindungen zwischen Klassen nutzt. Eine detaillierte Beschreibung der verschiedener Softwaremetriken und ihrer Grundlagen findet sich im Buch von Fenton et al. (Fenton, Pfleeger, & Lawrence, 1998)

2.1.1 Lückenhafte Metriken

Die große Anzahl verschiedener Metrikvarianten, die auf verschiedenen Konzepten beruhen, sind ein Indikator dafür, dass noch kein allgemeingültiges Messverfahren im Software-Engineering bekannt und akzeptiert ist. Jede der zuvor genannten Metriken hat Schwachpunkte, die die Aussagen der Ergebnisse einiger Metriken sogar widerlegen. Unglücklicherweise, gibt es bis heute keine Metrik die klar definiert ist und deren Resultate wissenschaftlich anerkannt sind.

Burger et al. (Burger & Hummel, 2013) untermauern in ihrer Arbeit die bestehenden Kritiken an Softwaremetriken, indem sie zunächst beispielhaft zeigen, wie Messergebnisse gängiger Metriken mit schlechter lesbarem, aber funktional identischem Code künstlich „verbessert“ werden können. Darauf aufbauend präsentierten sie erste Analyseergebnisse (von Fowlers „Video-Store“ (Fowler, 2002) und dem Lucene-Open-Source-Projekt[1]), die darlegen, dass bekannte Softwaremetriken Code-Verbesserungen durch Refactoring nicht erkennen können, bzw. umgekehrt betrachtet, Refactoring die Codequalität verschlechtert. Drei aufeinander aufbauende Messreihen mit insgesamt 1900 Vergleichen der bekanntesten Softwaremetriken aus der Literatur wurden vorgestellt. Deren Resultate ähneln sich bei allen drei vermessenen Beispielen – trotz ihrer Verschiedenheit – sehr stark: zumeist steigt die Mehrzahl (insgesamt 47%) der Metrik Ergebnisse nach einem Refactoring deutlich an, nur bei 15% aller durchgeführten Refactorings verringert sie sich, was auf Grund des großen Umfangs von möglichen Kombinationen stichprobenartig auch analytisch an Hand des Aufbaus der Refactorings und Metriken nachvollzogen werden konnte.

Kaur et al. (Kaur, Minhas, Mehan, & Kakkar, 2009) untersuchten in ihrer Analyse sowohl McCabes CC als auch diverse Halstead-Metriken und kamen zu dem Schluss, dass die jeweiligen Definitionen einige schwerwiegende Lücken aufzeigen. Beispielhaft genannt seien hier die lückenhafte Definition von Halsteads Operanden und Operatoren oder die Abhängigkeit von McCabes zyklomatischer Komplexität von der Programmlänge.

Des Weiteren sollten an dieser Stelle Jones et al. (Jones, 1994) erwähnt werden. Ihre Arbeit verdeutlicht die Stärken und Schwächen verschiedener Softwaremetriken. Sie kommen ebenfalls zu dem Schluss, dass die grundlegenden Definitionen gravierende Lücken beinhalten. Ein Beispiel, das genannt wird, ist die ungenaue Definition der Lines of Code (LoC), wenn eine Zeile als Quellcode oder als Statement gezählt werden soll und wie mit Zeilenumbrüchen umzugehen ist. Auch die besonderen Einflüsse der genutzten Programmiersprache und Programmstile auf die Ergebnisse gängiger Metriken mit LoC-Bezug werden in den Definitionen nicht berücksichtigt.

Stamelos et al. (Stamelos, Angelis, Oikonomou, & Bleris, 2002) erarbeiteten eine ausführliche Fallstudie zur Anwendung von Softwaremetriken, basierend auf über 100 in C geschriebenen Applikationen (z. B. mail) aus der Linux-Distribution SUSE 6.0. Doch auch der Erkenntnisgewinn aus den Ergebnissen blieb ernüchternd: Der Vergleich der Ergebnisse verschiedenster Metriken

[1] http://lucene.apache.org/

mit ihren teils bekannten Grenzwerten (z. B. aus McCabe (McCabe, 1976) oder Halstead (Halstead, 1977)) erlaubt keine eindeutige Aussage über die Qualität der verschiedenen Applikationen. Ferner fanden sich bei dieser Untersuchung in den Ergebnissen verschiedene Ausreißer, die weit über den Grenzwerten lagen, ohne dass dafür konkrete Ursachen in Form von „schlechtem" Code festzustellen waren.

Ebenfalls in der Diskussion stehen OO-Metriken. Alshayeb und Li (Alshayeb & Li, An Empirical Validation of Object-Oriented Metrics in Two Different Iterative Software Processes, 2003) analysierten die Ergebnisse verschiedener Messungen mit OO-Metriken. Die Ergebnisse basieren auf der Auswertung von drei Datensets. Jedes dieser Sets zeigte die Entwicklung bzw. Veränderungen einer Software im Laufe der Zeit über verschiedene Versionen hinweg. Auf jede Version wurde ein Satz von OO-Metriken angewandt. Aus den so gewonnenen Ergebnissen folgerten die Autoren, dass der Einsatz von OO-Metriken die Veränderungen von einem System zwischen zwei Versionen abbilden kann, solange das Design nicht grundlegend verändert wird. Einen Einsatz für Wartungsarbeiten oder zur Überprüfung der Software-Evolution über mehrere Versionen hinweg schlossen sie jedoch aus.

Auch die Ergebnisse von Nagappan et al. (Nagappan, Ball, & Zeller, 2006) und Schröter et al. (Schröter, Zimmermann, & Zeller, 2006) sind in Zusammenhang mit Softwaremetriken zu erwähnen. Diese konnten jüngst einen Zusammenhang zwischen Komplexitätsmetriken und der Anzahl an Fehlern in Softwaresystemen empirisch nachweisen. Dazu sammelten sie Daten aus der Fehlerdatenbank, der Versionsverwaltung und dem Quellcode verschiedener Microsoft-Produkte. In einem zweiten Schritt wurden gefundene Fehler direkt mit den dazugehörigen Modulen verknüpft. Für diese Module wurde die Komplexität berechnet. Durch diese historischen Komplexitätsdaten waren sie in der Lage, Fehler in neuen Modulen vorauszusagen.

Ein weiterer Nachweis der Abhängigkeit zwischen Coupling-Metriken und Softwarefehlern wurde von Binkley et al. (Binkley & Schach, 1998) ausgeführt. Es bleibt allerdings nach wie vor unklar, wie sich hohe Werte bei Komplexitätsmetriken auf andere Qualitätsfaktoren wie z. B. die Verständlichkeit des Quellcodes im Detail auswirken, auch wenn als erste Vermutung natürlich ein Zusammenhang zwischen hohen Komplexitätsmesswerten und schlechter Verständlichkeit naheliegend scheint.

Huston (Huston, 2001) untersuchte den Effekt von Entwurfsmuster auf den Quellcode und deren Metrik-Ergebnisse. Im Zuge dieser Untersuchung hat er ein mathematisches Modell, basierend auf Metriken wie *Coupling between Objects* (CBO), entwickelt. Es erlaubt den Vergleich von Quellcodestellen mit und ohne Pattern. Ausgehend von den gesammelten Daten kam der Autor zu dem Schluss, dass der Einsatz von Metriken fragwürdig ist, wenn Entwurfsmuster eingesetzt werden, denn nach dem Einsatz von Entwurfsmuster zeigten einige Messungen höhere und somit eigentlich schlechtere Metrik-Ergebnisse.

2.2 Softwarequalität

Die Verwendung von Metriken soll helfen, die Qualität einer Software zu bestimmen und bei Bedarf zu verbessern. Jedes Jahr werden qualitativ mangelhafte Softwaresysteme auf dem Markt veröffentlicht, darunter einige namhafte Produkte wie die Computerspiele Sim City und Battlefield 4 der Firma Electronic Arts (vgl. Kapitel 1.1). Mangelhafte Software führt oft zu hohen finanziellen Verlusten, wie z. B. beim Start der Ariane 5 (Lions & others, 1996), die wegen eines Softwarefehlers abstürzte, was zu einem Millionenschaden führte. Um die Qualität eines Systems zu bewerten, sollten die verschiedenen Sichtweisen derjenigen Gruppen einbezogen werden, die mit dem System in Verbindung stehen (z. B. Entwickler, Benutzer etc.). Aus der Sicht eines Benutzers ergibt sich

die Qualität eines Systems aus Merkmalen wie der Reaktionszeit oder der Fehlerhäufigkeit. Wird nun die Sicht des Entwicklers in Betracht gezogen, so stehen für ihn zusätzlich noch Aspekte wie Wart- und Lesbarkeit des Codes im Fokus.

2.2.1 Definitionen von Softwarequalität

Die verfügbaren Definitionen von Softwarequalität sind genauso vielschichtig wie die Anzahl der Gruppen, die mit einem System in Beziehung gebracht werden können. Bis heute gibt es keine allgemein anerkannte Definition für Softwarequalität. Kitchenham et al. (Kitchenham & Pfleeger, 1996) untersuchten, wie der Begriff der Qualität in verschiedenen Bereichen der Wirtschaft und Wissenschaft wahrgenommen wird. Ihre Schlussfolgerung fällt sehr allgemein aus und kann wie folgt zusammengefasst werden:

„Quality is a complex and multifaceted concept. “ (Kitchenham & Pfleeger, 1996)

Neben dieser Aussage gelang es den Autoren, diese Komplexität der Qualität in fünf Gesichtspunkten abzubilden, wie in Abbildung 1 dargestellt.

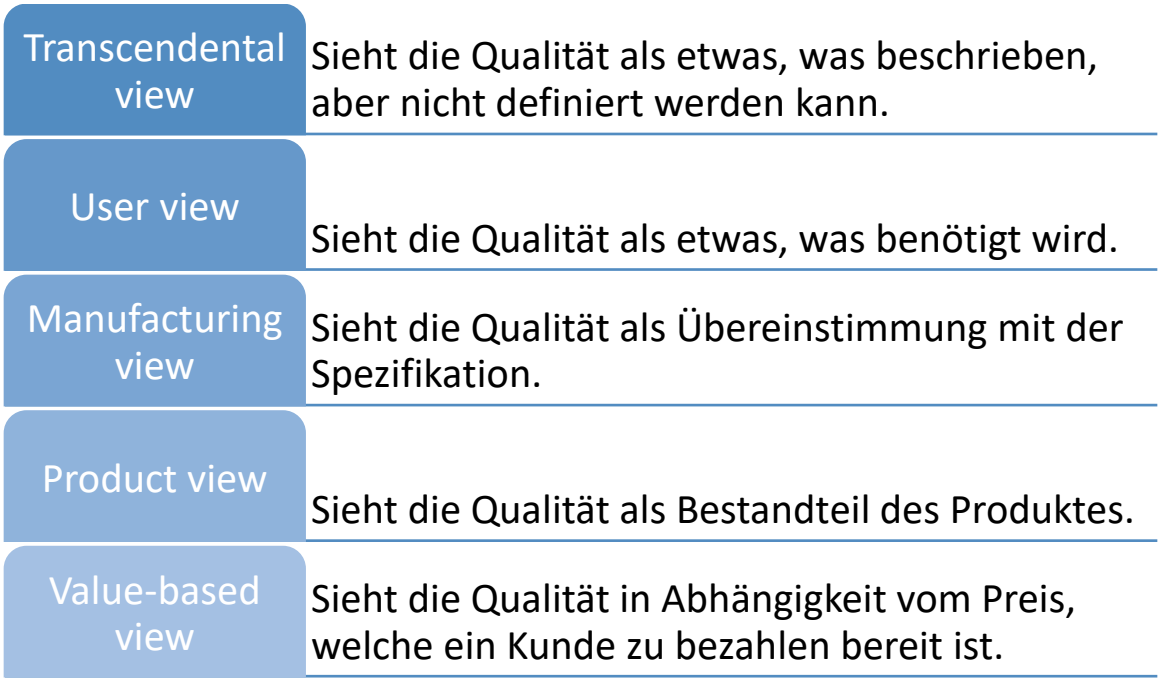


Abbildung 1: Sichtenmodell der Qualität (Kitchenham & Pfleeger, 1996)

Diese Sichten von Kitchenham et al. können nun mit den Benutzergruppen verbunden werden, die in Verbindung zur Software stehen, vom Entwickler über den Administrator bis zum Kunden. Aus dieser Kombination entsteht ein Modell der Aspekte und Sichten, unter denen die Beschreibung der Qualität einer Software in Betracht gezogen werden muss.
Das US-Verteidigungsministerium erarbeitete 1985 eine weitere Definition des Begriffes Softwarequalität (Gillies, 1992).

„The degree to which the attributes of the software enable it to perform its intended end use.“

Diese Definition zielt auf das Erfüllen der Anforderungen eines Projektes. Damit ähnelt sie der Definition aus der Norm ISO 9126 (ISO I. , 2001), welche lautet:

„... die Gesamtheit der Merkmale und Merkmalswerte eines Software-Produkts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen“ (Hoffmann, 2013).

Schon die Vielzahl inhaltlich unterschiedlicher Definitionen zeigt die Herausforderungen, die Softwarequalität mit sich bringt. Die verschiedenen Definitionen stellen unterschiedliche Aspekte in den Vordergrund. In einer Definition wird Softwarequalität rein auf das Erfüllen von Anforderungen reduziert. Eine Festlegung auf solch eine Definition hätte zur Folge, dass alle Qualitätskriterien als Anforderungen definiert sind, was die Anzahl nicht-funktionaler Anforderungen massiv erhöhen werden. Andere Definitionen sehen die verschiedenen Charakteristiken und Abschnitte des Software-Lebenszyklus. Hier liegt die Lücke in der Anzahl der verschiedenen Lebenszyklen und möglichen Charakteristiken. Beides wird nicht genauer definiert.

2.2.2 Arten der Qualität

Eine inhaltliche Auseinandersetzung über die möglichen Charakteristiken von Softwarequalität findet schon seit Längerem in der Wirtschaft und Wissenschaft statt. Um die neusten Eigenschaften und Veränderungen in der Softwareentwicklung zu berücksichtigen, wurde die Normenreihe DIN ISO/IEC 25000 Software-Engineering-Qualitätskriterien und Bewertung von Softwareprodukten (SQuaRE) – Leitfaden für SQuaRE (ISO, 2005) veröffentlicht. Diese beinhaltet Konzepte und Methoden zur Bewertung von Softwaresystemen und deren Qualität. Unter anderem stellt sie eine Weiterentwicklung der nun ersetzten ISO 9126 (ISO I. , 2001) dar.

Mit der neuen Version der Norm wurden für die Qualität von Software sechs Aspekte und deren Ausprägungen definiert. Abbildung 2 zeigt die verschiedenen, in ISO 25010, einem Teil der Reihe ISO 25000, definierten 27 Aspekte (z.B. Reife) und sechs Charakteristiken von Funktionalität bis Portabilität. Diese können noch in interne und externe Qualitätsmerkmale unterschieden werden.

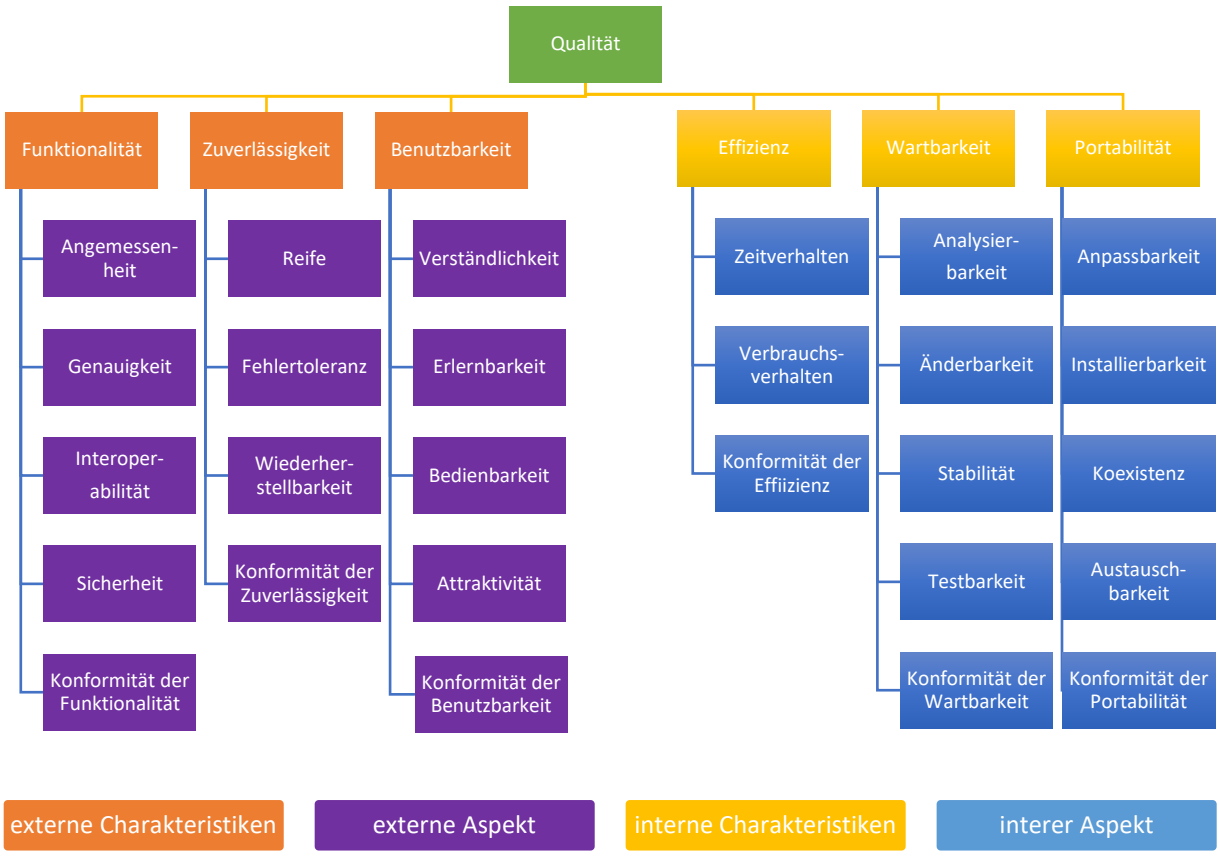


Abbildung 2: Softwarequalitäts-Charakteristiken nach ISO 25010

Für den Benutzer einer Software definiert sich Qualität über die für ihn wahrnehmbaren Charakteristiken wie Benutzbarkeit, Funktionalität und Zuverlässigkeit, also über die „nach außen" sichtbaren, sogenannten externen Qualitätscharakteristiken. Externe Charakteristiken sind relativ leicht zu erheben, da sie über Antwortzeiten, die Anzahl der auftretenden Fehler oder Rückmeldungen durch Benutzer ermittelt werden können.

Für den Anwendungsentwickler wird ein weiteres Spektrum an Qualitätscharakteristiken benötigt, welche neben den externen auch die internen Merkmale der Software wie Wartbarkeit, Portierbarkeit und Effizienz berücksichtigt (vgl. bspw. (McConnell, 2004)). Schon die reine Definition von Wartbarkeit oder Lesbarkeit stellt sich als komplexe Aufgabe heraus. In der ISO 9126 (ISO I. , 2001) wird Wartbarkeit wie folgt definiert:

"Maintainability is the capability of software to be modified." (ISO I. , 2001)

Diese Definition in der Praxis anzuwenden, stellt sich als schwierig heraus, da selbst sehr komplexe Software mit entsprechend hohem Aufwand modifiziert werden kann. Somit fehlen der Definition noch weitere Angaben – z. B., ab wann die Fähigkeit zur Veränderung nicht mehr in einem wirtschaftlich vertretbaren Rahmen gegeben ist. Änderbarkeit und Analysierbarkeit sind interne Merkmale eines Quellcodes. Diese für den Entwickler maßgebenden Merkmale beeinflussen die Wartbarkeit eines Codes und basieren auf dem Quellcode, sind aber schwerer in Zahlen zu fassen als die offensichtlichen externen Charakteristiken. Wie zuvor gezeigt, gibt es im Software-Engineering verschiedene Ansätze, die Attribute eines Quellcodes in Zahlen zusammenzufassen, angefangen von LoC über zyklomatische Komplexität (McCabe, 1976) bis hin zu objektorientierten Metriken (OO-Metriken) (Chidamber & Kemerer, 1994). Zusammengefasst liegt bis heute keine allgemein akzeptierte und wissenschaftlich fundierte Methode zur Bestimmung der internen Merkmale vor. Das heutige Verständnis von gutem Code basiert eher auf der Anwendung von Coding Guidelines und Best Practices (vgl. (Martin, 2008) und (McConnell, 2004)), als auf einer klaren Definition und messbaren Kriterien.

2.2.3 Fehlende Möglichkeiten, Qualität zu bestimmen

Mit der von der ISO verbesserten Normenreihe 25000 wurde die Definition der Softwarequalität geschärft. Die Erweiterung der Charakteristiken erlaubt ein besseres Einordnen. Alles in allem steht nun eine feste Basis zur Verfügung, um Qualität zu beschreiben. Nichtsdestoweniger bleibt es für Entwickler und Verantwortliche schwierig, eine Verbindung zwischen Definition und eigentlicher Anwendung herzustellen. Es gab einige Ansätze, ein Modell zu entwickeln, das diese Lücke schließen sollte.

Ein bekanntes Model im Bereich der Software-Entwicklung stammt von Basili et al. (Van Solingen, Basili, Caldiera, & Rombach, 2002). Die sogenannte Goal-Question-Metrik-Vorgehensweise (GQM) basiert auf einer Top-Down-Vorgehensweise und wird auch für die Bestimmung von Softwarequalität eingesetzt. An oberster Stelle steht das Ziel der Analyse (Goal). Mit einer Reihe von Fragen wird dieses Ziel näher definiert. Jeder definierten Frage wird eine Auswahl an Metriken zugeordnet. Die Vorgehensweise hat das Ziel, das Sammeln von Informationen über das Programm von Anfang an zu organisieren. Dieses Modell wird in einigen Bereichen gerne angewandt und oft in modifizierter Version weiterverwendet. Das Modell enthält keine Metrik-Vorschläge, was dazu führt, dass Entwicklern die Entscheidung auferlegt wird, welche Metriken wann zu verwenden sind.

Ein praktischerer Ansatz wurde von Heitlager et al. (Heitlager, Kuipers, & Visser, 2007) beschrieben. Sie definieren ein Modell, das die Wartbarkeit von Software bestimmt. Diese Model basiert auf einer Menge von Softwaremetriken, welche bei über 100 Einzelprogrammen gemessen

wurden. Des Weiteren nutzen sie als Basis ihres Vorgehens die Charakteristiken, die in der ISO 9126 beschrieben wurden, dem Vorgänger der Reihe ISO 25000. Für die Bestimmung der Grenzwerte wurden die Messungen statistisch analysiert und vier Grenzen bestimmt (low, moderate, high und very high). Alves et al. (Alves, Ypma, & Visser, 2010) knüpfen an diese Arbeit an und veröffentlichten einige Grenzwerte für Metriken. Diese Arbeiten wurden von Burger et al. (Burger & Hummel, 2012) untersucht, wobei versucht wurde, sie auf die Kabinen-Software von Airbus-Zivilflugzeugen anzuwenden. Es stellt sich heraus, dass die Grenzwerte nicht für Software aus jeder Domäne geeignet sind. Außerdem erlaubt das Verfahren nur eine Aussage über die gesamte Software und liefert keine Informationen darüber, wo die Probleme im Quellcode verborgen sind.

Neben vielen theoretischen Ansätzen aus den Bereichen der Wissenschaft gibt es auch kommerzielle Modelle, die von der Industrie entwickelt wurden. Hierzu zählt das von *IBM* im *Rational Logiscope*® (IBM Software United States, 2009) verwendete Qualitätsmodell. Es wurde, nach heutigem Stand des Autors, noch keiner genaueren Studie unterzogen. Einzelheiten darüber, wie das Modell aufgebaut ist, sind nicht bekannt.

Das ISIS-Modell (Rauch, Kuhn, & Friedrich, 2008) der Karlsruher Firma *andrena objects* (Andrena Objects AG, 2015) nutzt beispielsweise den Ansatz, die verschiedenen in Tabelle 2 beschriebenen Codemetriken bzw. -merkmale in einem Index zu kombinieren, der Aussagen zur Codequalität von Komponenten oder ganzen Applikationen erlaubt.

Tabelle 2: Qualitätsmerkmale von ISIS

Codemerkmale des ISIS-Modells
Testabdeckung
Zahl der Paket-Zyklen
Zyklomatische Komplexität
Klassengrößen
Methodenlängen
Compiler-Warnungen
Komponenten-Abhängigkeiten

ISIS erlaubt somit zwar eine „mechanische“ Verbesserung der Codestruktur in den genannten Aspekten, eine konkrete Benennung von Code-Smells ist damit aber nicht möglich.

2.2.4 Fehlende Zuordnung von Metriken zu Qualitätsmerkmalen

Wie zuvor gezeigt, bauen Modelle wie die zuvor erwähnten von Heitlager et al. (Heitlager, Kuipers, & Visser, 2007) oder das ISIS-Modell (Rauch, Kuhn, & Friedrich, 2008) auf verschiedenen Softwaremetriken auf. Wie beim *Maintainability Index* verwendet auch ISIS die zyklomatische Komplexität von McCabe. Diese steht in der Kritik, ungenau definiert zu sein und nur wenig Aussagekraft zu besitzen (Jones, 1994). Dies kann Auswirkungen auf die Aussagekraft des Ergebnisses von ISIS haben.

Das Grundgerüst, auf dem die Metriken basieren, ist sehr instabil definiert. So schrieben Fenton et al. (Fenton, Pfleeger, & Lawrence, 1998), dass die Lücke zwischen dem „Wie“ und dem „Was“

hinsichtlich der Messung größer ist, als sie sein dürfte. Einen der Hauptgründe für diese Lücke sieht Fenton im Fehlen eines klaren Metrik-Frameworks, das bei Vermessung und Interpretation hilft. Sie erarbeiteten einen Vorschlag für ein solches Framework, welches aber einige Grundprobleme, wie genaue Definition oder Aussagekräfte Grenzwerte, der Software-Metriken nicht lösen könne.

Bis heute fehlt eine klare Zuordnung zwischen Metriken und Softwarequalitäts-Charakteristiken in der Art, welche Metrik auf welche Charakteristik Einfluss hat und umgekehrt. Samoladas et al. (Samoladas, Gousios, Spinellis, & Stamelos, 2008) unternahmen in ihrem Projekt Software Quality Observatory for Open Source Software (SQO-OSS) den Versuch, eine Zuordnung zwischen Metriken und Software-Charakteristiken des ISO 9126 (ISO I. , 2001) herzustellen. Tabelle 3 zeigt einen Auszug aus der so entstandenen Zuordnung.

Tabelle 3 Auszug aus der Zuordnung Charakteristiken zu Software Metriken nach Samoladas et al. (Samoladas, Gousios, Spinellis, & Stamelos, 2008)

Charakteristik	Metrik
Analyzability	Cyclomatic Number
Changeability	Lack of Cohesion (LCOM)
Stability	Coupling between objects (CBO)
Testability	Average cyclomatic complexity per method

Die erfolgte Zuordnung ist erstens unvollständig und zweitens ist unklar, warum welche Metrik welcher Charakteristik zugeordnet wurde. So kann als frag würdig erachtet werden, warum die zyklomatische Komplexität nur beim Testen eine Rolle spielt, aber nicht bei der Veränderbarkeit. McCabe selbst sah seine Metrik gerade als Indikator, wie gut ein Programmcode zu lesen ist. Je höher die Schwierigkeit, ein Programm zu verstehen, desto höher auch die Hürde, es zu verändern.

Das Problem der nicht eindeutigen Grenzwerte wurde bspw. in einer Studie über die Verwendung und Aussagekraft von Softwaremetriken in Bezug auf die Wartbarkeit von Programmen, welche von Burger et al. (Burger & Hummel, 2012) durchgeführt wurde, deutlich. Ihre Studie nutzt als Grundlage den Quellcode der Kabinenkontrollsoftware der Firma Airbus. Nach der Untersuchung des Quellcodes mit verschiedenen Softwaremetriken und der Analyse der Ergebnisse kamen sie zu dem Schluss, dass auf Grundlage der genutzten Metriken keine Aussage über die Wartbarkeit getroffen werden kann. Wie ein Ausschnitt der Ergebnisse, welcher in Tabelle 4 dargestellt ist, zeigt, sind viele höchste Werte weit über dem Grenzwert und nur wenige der Werte bewegen sich überhaupt innerhalb der gezogenen Grenzen aus der Literatur. Eigentlich müsste die Software wenig zuverlässig sein, doch sie ist für kritische Systeme in einem Flugzeug zuständig und gilt als sehr stabil. Die gesammelten Metrik-Ergebnisse zeigten also ein widersprüchliches Bild für eine Anwendung, die für ein kritisches System zuständig ist.

Tabelle 4 Auszug der Ergebnisse der Softwareuntersuchung. Es wird der in der Literatur genannte Grenzwert, der Höchste Wert, Mittelwert, Standard Abweichung, Median und Werte innhalber des Grenzwertes dargestellt

Metrik	Grenzwert	Höchster Wert	Mittel-wert	Standard-Abwei-chung	Median	Innerhalb des Grenzwerts
Amount of comments	>= 0.5	0.25	0.82	0,37	0,72	**81%**
Functions (per File)	7 +/- 2	242	7	5,34	6	**71%**
Halst. Diffficulty	< 30	2.324	186,5	143	152	**2,8%**
Halstead Volume	100 .. 8000	1.673 M	20.697	78.183	9.053	**46%**
Halstead Size of Statements	<= 7	46	2,99	7,92	0,90	**95%**
Fan-In	<= 7	1.476	196	215	136	**0%**
Fan-Out	<= 7	3.701	195	443	105	**0%**
Cyclomatic Complexity	0..10	383	45.7	51	30	**45%**
Nesting Level	0..5	12	7,2	2,4	7	**59%**

Jones (Jones, 1994) diskutiert in seiner Arbeit einen weiteren problematischen Punkt in Bezug auf Metriken: Die eigentliche Anwendbarkeit der einzelnen Metriken ist auf ein Programmierparadigma angepasst. So wurden die Halstead-Metriken für prozedurale Sprachen entwickelt, was ihren Einsatz und ihre Ergebnisse bei der Anwendung auf Programmiersprachen wie Java oder C# diskussionswürdig erscheinen lässt. Des Weiteren existiert kein allgemein akzeptiertes Modell zur Bewertung der Softwarequalität, wie beispielsweise der Wartbarkeit eines Programmes.

2.3 Code Smells und Refactoring

Schwachstellen und Auffälligkeiten in Softwaresystemen werden unter dem Begriff Code Smells zusammengefasst werden. Unter Code Smells wird eine Menge von Codemustern verstanden, die häufige Fehler in der Programmierung darstellen daher und immer wieder auftreten. Eine der wichtigsten Arbeiten zum Thema Code Smells stammt von Fowler (Fowler, 2002). In seinem Buch publizierte er eine Liste von bekannten Smells, wie Middle Man oder Long Method u. v. m.

2.3.1 Erkennung von Code Smells

Es gibt verschiedene Ansätze, Code Smells zu identifizieren. Einige dieser werden im Folgenden beschrieben und diskutiert.

In seiner Dissertation verfolgt Seng et al. (Seng, 2008) einen Ansatz zur Erkennung von Code Smells, der auf dem Einsatz von Softwaremetriken und einem evolutionären Algorithmus basiert. Als Ergebnis liefert sein Tool Codestellen und Refactoring-Vorschläge mit Hinweisen darauf, wie und wo die Qualität des Codes verbessert werden könnte.

An der Erkennung von Code Smells arbeiteten auch Tourwé et al. (Tourwé & Mens, 2003). Ihr Ansatz zur Identifikation von Schwachstellen im Code nutzt ein *Logic Meta Programming Framework*, welches auf objektorientierter Programmierung und einer deklarativen Meta-Sprache basiert. Ein anderer Ansatz der automatischen Erkennung von Code Smells und ihrer Visualisierung wurde von van Emden et al. (Van Emden & Moonen, 2002) entwickelt. Zur Erkennung definieren sie mehrere Smell-Aspekte, eine Art einfacher Filter, die dabei helfen, auffällige Codestellen zu identifizieren. Munro (Munro, 2005) verwendet ein Set von Softwaremetriken wie Lines of Code

(LoC) oder Weighted Methods per Class (WMC) zur automatischen Erkennung von Code Smells. Als Schwellwerte verwendet der Autor den Median einiger Metriken oder feste Werte. Eine genauere Herleitung fehlt.

2.3.2 Bereinigung von Code Smells durch Refactoring

Neben verschiedenen Code Smells beschreibt Fowler noch Verbesserungsvorschläge, die genutzt werden können, um die Code Smells zu beseitigen. Diese werden unter dem Namen Refactoring zusammengefasst. Eine der ersten Arbeiten im Bereich Refactoring wurde 1992 von Opdyke (Opdyke, 1992) veröffentlicht. In seiner Dissertation werden u. a. erste Refactoring-Patterns zum Erzeugen von Super- und Unterklassen beschrieben. Die Grundidee von Refactoring ist es, bestehenden Code ohne Veränderung der Funktionalität neu zu strukturieren, um dadurch die Codekomplexität zu reduzieren und spätere Anpassungen und Erweiterungen einfacher durchführen zu können sowie Zeit und Kosten für Weiterentwicklungen zu reduzieren (Alshayeb & Li, An Empirical Validation of Object-Oriented Metrics in Two Different Iterative Software Processes, 2003).

Eine Übersicht über die jüngere Forschung im Bereich Refactoring liefern beispielsweise Mens et al. (Mens & Tourwé, 2004), die auch mögliche positive Auswirkungen von Refactoring auf einzelne Qualitätsindikatoren wie Wiederverwendbarkeit oder Komplexität beschreiben.

Bisher sind nur drei Arbeiten geläufig, die objektorientierte Softwaremetriken vor und nach einem Refactoring vergleichen. Alshayeb (Alshayeb, Empirical investigation of refactoring effect on software quality, 2009) untersuchte die Auswirkungen von Refactoring auf fünf Software-Qualitätscharakteristiken (adaptability, maintainability, understandability, reusability und testability). Dazu wurden die Auswirkungen von Refactoring auf einzelne Methoden dreier ausgewählter Programme mit Hilfe von neun Metriken (DIT, NOC, CBO, RFC, FOUT, WMC, NOM, LOC und LCOM [(Chidamber & Kemerer, 1994))) analysiert und auf die Indikatoren übertragen. Die Untersuchung kam zu dem wenig greifbaren Ergebnis, dass das Zusammenspiel zwischen Refactorings und Qualitätscharakteristiken sehr komplex zu sein scheint und künftig genauer untersucht werden sollte.

In Du Bois' (Du Bois & Mens, 2003) Untersuchungen eines einfach LAN-Simulator-Systems erhöhten (+) sich Messwerte der fünf genutzten objektorientierten Metriken nach einem Refactoring bei 50% der Ergebnisse (siehe Tabelle 5). Bei 30% blieb das Ergebnis unverändert, (o) und bei nur rund 20% sanken (-) die Messwerte.

Tabelle 5: Überblick über die Ergebnisse von Du Bois et al. (Du Bois & Mens, 2003)

Refactoring	Number of Methods	Number of Children	Coupling betw. Obj.	Response for a Class	Lack of Cohesion
Encapsulate Field	+	O	o	+	+
Pull Up Method subclass	-	O	-	-	-
Pull Up Method superclass	+	O	+	+	+
Extract Method	+	O	o	+	+

Auch Stroggylos und Spinellis (Stroggylos & Spinellis, 2007) kommen in ihren Stichproben auf teilweise deutliche Erhöhungen der Metrik-Ergebnisse von bis zu 50 Prozent. Ihre Arbeit untersuchte ebenfalls nur objektorientierte Metriken (wie z. B. die in Tabelle 5 ebenfalls gezeigten) und kleinere, unabhängige Stichproben von Programmen wie Apache Log4j, MySQL Connector/J und JBoss und bleibt somit abermals von begrenzter Aussagekraft.

Eine Analyse der Auswirkungen von Refactorings auf Softwarewartungen durch Wilking et al. (Wilking, Kahn, & Kowalewski, 2007) zeigt eine leichte Verbesserung der Wartbarkeit restrukturierter Software, die allerdings einen zusätzlichen Aufwand für das Refactoring mit sich bringt. Die Ergebnisse wurden durch Studentengruppen gewonnen, die bei der Implementierung einmal auf Refactoring setzten und einmal nicht. Dennoch ist eine Hoffnung der Refactoring-Befürworter, dass eine regelmäßige Überarbeitung der Codestruktur zumindest einer Komplexitätserhöhung des Quelltexts entgegenwirkt.

Eine vorangegangene Arbeit von Burger et al. (Burger & Hummel, 2013) legt den Verdacht nahe, dass heute verfügbare Softwaremetriken und Qualitätsmodelle Best Practices guter Softwaredesigns (d. h. Refactorings und Entwurfsmuster) nicht erfassen können. Sie zeigen, wie Messergebnisse gängiger Metriken mit schlechter lesbarem, aber funktional identischem Code künstlich „verbessert“ werden können. (vgl. Kapitel 2.1.1)

Zusammenfassend bleibt festzuhalten: Bisherige Untersuchungen basierten zumeist auf wenigen Refactoring-Patterns und analysierten nur eine geringe Anzahl von Metriken. Des Weiteren existiert noch keine detaillierte Analyse über den Einfluss aller von Fowler gelisteten Refactorings auf gängige Softwaremetriken, sodass nach wie vor unklar bleibt, inwieweit den beiden ein ähnliches Verständnis der Codekomplexität zugrunde liegt.

Die Erkennung von Code Smells und der Einsatz von Refactorings sind unverzichtbare Bestandteile der Entwicklung von Software geworden. Durch den richtigen Einsatz von Refactoring erhöht sich die Qualität der Software. Das rechtzeitige Erkennen von Code Smells beugt späteren komplexen Änderungen am Quellcode vor. Eine Bereinigung der identifizierten Stellen kann durch Refactorings erfolgen. Refactorings sind strukturelle Anpassungen am Quellcode, die keinen Einfluss auf die Funktionalität haben. Durch den richtigen Einsatz von Refactorings können Smells nicht nur gelöst, sondern an der entsprechenden Stelle ggf. auch ein Entwurfsmuster implementiert werden. Das ist nicht bei allen Code Smells und schon gar nicht bei jeder Anwendung von Refactoring möglich. Aus diesem Grund ist es schwierig, auf der Grundlage von Smells Möglichkeiten für neue Entwurfsmuster zu identifizieren.

2.4 Entwurfsmuster – Erkennung und Auswirkungen

Entwurfsmuster dienen als Muster zur Lösung von wiederkehrenden Design-Problemen. Neben immer wiederkehrenden Herausforderungen in der Entwicklung sind Entwurfsmuster auch in der Lage Code- und Design Smells auszubessern, welche auftreten wenn Probleme ineffektiv gelöst werden. Während Code Smells direkt im Quellcode auftreten, sind Design Smells in einer höheren Ebene zu finden.

Als ebenso komplex zeigt sich die Identifikation von bereits implementierten Entwurfsmustern. Eine solche Detektion ist notwendig, um mehr über die Struktur und Funktion eines Programmes herauszufinden. So beschreiben Tsantalis et al. (Tsantalis, Chatzigeorgiou, Stephanides, & Halkidis, 2006) die Aufgabe der Detektion von Patterns wie folgt:

„The identification of Design Patterns as part of the reengineering process can convey important information to the designer.“ (Tsantalis, Chatzigeorgiou, Stephanides, & Halkidis, 2006)

Dong et at. (Dong, Zhao, & Peng, 2009) untersuchten eine Reihe von 13 Pattern Mining Tools für C++- und Java-Quellcode. In der Evaluierung dieser Tools wurden Erkenntnisse über die Verteilung bzw. das Auftreten von verschiedenen Design Patterns gesammelt. Teil ihrer Experimente waren Ergebnisvergleiche zwischen den Tools. Hierzu wurden die Ergebnisse von zwei unabhängigen Analysetools, basierend auf den Veröffentlichungen von Tsantalis et al. (

(Tsantalis, Chatzigeorgiou, Stephanides, & Halkidis, 2006)) und Guéhéneuc et al. (Guéhéneuc & Antoniol, DeMIMA: A Multilayered Approach for Design Pattern Identification, 2008), miteinander verglichen. Guéhéneuc et al. verwenden einen kombinierten Ansatz, bestehend aus numerischen Eigenschaften (z. B. Größe, Komplexität, Anzahl der Methoden etc.) und einer Strukturanalyse des Quellcodes. Abschließend sollte der Ansatz von Tsantalis et al. zur Identifikation von veränderten Design Patterns angeführt werden. Dieser basiert auf einem Graphen-Algorithmus. Tabelle 6 zeigt den Vergleich der Ergebnisse. Die beiden Tools fanden eine sehr unterschiedliche Anzahl an Design Patterns – trotz der gleichen Codebasis.

Tabelle 6: Vergleich der Ergebnisse der beiden Design Pattern-Analysetools aus (Dong, Zhao, & Peng, 2009)]

System	JHotDraw 5.1		JRefactory 2.6.24		JUnit 3.7	
Autor	(Tsantalis, Chatzigeorgiou, Stephanides, & Halkidis, 2006)	(Guéhéneuc & Antoniol, DeMIMA: A Multilayered Approach for Design Pattern Identification, 2008)	(Tsantalis, Chatzigeorgiou, Stephanides, & Halkidis, 2006)	(Guéhéneuc & Antoniol, DeMIMA: A Multilayered Approach for Design Pattern Identification, 2008)	(Tsantalis, Chatzigeorgiou, Stephanides, & Halkidis, 2006)	(Guéhéneuc & Antoniol, DeMIMA: A Multilayered Approach for Design Pattern Identification, 2008)
Adapter	18	1	7	7	1	0
Composite	1	1	0	0	1	1
Decorator	3	1	1	0	1	1
Factory Method	3	3	4	1	0	0
Observer	5	2	0	0	4	3
Prototype	1	2	0	0	0	0
Singleton	2	2	12	2	0	2
State	23	2	12	2	3	0
Template Method	5	2	17	0	1	0
Visitor	1	0	2	2	0	0

Ein Verfahren zur Entdeckung von Design Pattern-Implementierungen in vorhandenem Quellcode, unabhängig von der verwendeten OO-Programmiersprache, wurde von Fabry et al. (Fabry & Mens, 2004) entwickelt. In dem genannten Verfahren werden Metainformationen wie Methodenaufrufe oder Variablenreferenzen aus dem Parserbaum extrahiert. Dazu wird ein Logic Meta Programming Framework verwendet, um Schlussfolgerungen aus dem Parserbaum zu ziehen. Dieses Framework nutzt eine deklarative Meta-Sprache und OO-Techniken. Dadurch, dass in jeder OO-Sprache gleiche Metainformationen im Parserbaum, etwa Klassen, Vererbungen etc., verwendet werden, ist das Verfahren in der Lage, sprachunabhängig zu sein. Es ist nur ein Transfer des Parserbaums in ein unabhängiges Format nötig. Eine solche Transformation wurde beispielhaft mit Java und Smalltalk-Programmcode durchgeführt. Der Prototyp war in der Lage, Template, Method und Visitor Patterns in fünf Anwendungen zu erkennen.

Heuzeroth et al. (Heuzeroth, Holl, Hogstrom, & Löwe, 2003) entwickelten ein Verfahren, um bereits implementierte Design Patterns zu identifizieren, das auf einer Kombination von dynamischer und statistischer Analyse von Quellcode basiert. Dazu beschrieben sie für einzelne Design Patterns Regeln und Rollen. Regeln wurden als Tupel von Elementen wie Klassen, Methoden etc. definiert. Rollen leiteten sich aus Elementen wie der Klasse eines Patterns ab. Der statische Analyseteil wurde auf dem AST durchgeführt. Dieser erlaubte eine Iteration über alle Elemente des Quellcodes. Die gefundenen Elemente wurden im dynamischen Teil weiter untersucht. Die Methode unterstützt vier GoF-Patterns. Durch dieses Vorgehen aus statischen und dynamischen Teilen war es ihnen möglich, bereits implementierte Design Patterns zu identifizieren. Die Anzahl an *false Positive* war relativ klein bei dieser Methodik. Jedoch besteht die Möglichkeit einer hohen Anzahl an *false-negative*-Ergebnissen.

2.4.1 Auswirkungen von Design Patterns auf die Software-Qualität

Während Refactorings die Struktur eines Quellcodes transformiert, ohne seine Funktion zu verändern, wird es bei der nachträglichen Implementierung von Entwurfsmuster erforderlich, Struktur eines Programmes anzupassen. Je nach Entwurfsmuster kann es nötig sein, neue Schnittstellen einzufügen oder neue Klassen zu erschaffen. Die nachträgliche Implementierung einer Facade verlangt Anpassungen von der aufrufenden als auch auf der aufgerufenen Seite, da nun nur noch die Facade aufgerufen werden kann und keine Klasse dahinter mehr direkt. Dieses Beispiel zeigt, wie Entwurfsmuster Einfluss auf die Struktur von Quellcode haben können und somit auch auf dessen interne Qualität aber nicht auf seine Funktionalität.

Eine interessante Untersuchung dazu, welche Auswirkungen Design Patterns auf die interne Softwarequalität haben können, kommt von Hegedűs et al. (Hegedűs, Bán, Ferenc, & Gyimóthy, 2012). Diese haben mehrere hundert Revisionen des Open Source-Projektes JHotDraw (Gamma & Eggenschwiler, JHotDraw as Open-Source Project, 2015) untersucht. Hierbei fanden sich Hinweise darauf, dass Design Patterns die Qualität des Codes verbessern. Ihre Untersuchung beschränkte sich auf die Version 7 und dort auf die Revisionen bei denen der Einsatz von Entwurfsmuster dokumentiert wurde. Beim Einsatz von Design Patterns muss der Entwickler eine gewisse Vorsicht walten lassen, denn nicht überall macht der Einsatz Sinn. Im schlimmsten Fall hat er negative Auswirkungen auf die Qualität des Programmes. So kann ein falsch eingesetztes Pattern die Lesbarkeit des Quellcodes verringern.

Prechelt et al. (Prechelt, Unger-Lamprecht, Philippsen, & Tichy, 2002) haben die Auswirkungen von Entwurfsmustern auf die Wartbarkeit anhand von zwei Experimenten beobachtet. In ihren Versuchen wurden zwei Aufgaben ausgegeben bei denen Wartungsarbeiten an Java- und C++-Programmen durchgeführt werden sollten. Beide Programme beinhalteten Entwurfsmuster, jedoch waren diese nicht bei jedem Teilnehmer dokumentiert. Ziel war es herauszufinden, ob Entwurfsmuster bei der Wartung von Quellcode hilfreich sind. Die Ergebnisse zeigen dass Entwurfsmuster sowohl bei der Wartung hilfreich sind, als auch die Fehleranzahl bei Änderungen minimieren. So reduziert sich die Wartungszeit bei Programmen in denen Design Patterns kommentiert sind um bis zu 43%. Wobei die Experimente wenige Teilnehmer hatten um statistisch aussagekräftig zu sein.

Khomh et al. (Khomh & Guéhéneuc, 2008)) untersuchten drei verschiedene Design Patterns auf ihre Einsatzgebiete und kamen zu dem Schluss, dass der Einsatz auch fragwürdig sein kann. Diese Begründung haben sie an einigen Patterns ausgeführt. Beispielsweise verletzt das Flightweight Pattern das Open-Close-Prinzip, wenn es nicht richtig angewendet wird. So führen sie weiter aus dass einige Pattern zu komplex sind und es zu viele Implementierungsarten gibt, was die Qualität das Quellcodes nicht verbessert.

Einen ähnlichen Schluss zogen Wydaeghe et al. (Wydaeghe, et al., 1998) bei der Analyse von sechs Design Patterns und ihren Auswirkungen auf Wiederverwendbarkeit, Modularität und Verständlichkeit. Ebenfalls zum gleichen Schluss kam auch Wendorff (Wendorff, 2001) bei seiner Evaluierung der Verwendung von Design Patterns in kommerziellen Systemen.

2.4.2 Anwendungshäufigkeit von Design Patterns

Die Erkennung von Design Patterns in vorhandenem Quellcode hat das Ziel, ein System besser zu verstehen und die Qualität des Systems oder des Designs nachzuweisen. Es gibt (Stand heute) keine Studie, die die Verteilung von allen GoF-Design-Patterns betrachtet. Die nachfolgenden Studien analysieren immer nur Teile der GoF-Patterns.

Hahsler (Hahsler, 2003) untersuchte die Verteilung von 22 der 23 bekannten Design Patterns in mehreren Open Source-Programmen der Plattform SourceForge (Dice Holdings, Inc, 2015). Damit ist dies die einzige Studie, die nahezu alle Design Patterns von Gamma et al. in ihrer Studie betrachtet hat. Ziel der Arbeit war es, die Unterschiede zwischen den Programmen mit und denen ohne Design Patterns herauszuarbeiten. Zur Identifikation der verschiedenen Design Patterns wurde der *CVS Commit Log* herangezogen. Die Log-Nachrichten wurden extrahiert und mit regulären Ausdrücken analysiert, um das Auftreten eines Design Patterns anhand des Namens zu finden. Abbildung 3 zeigt die Verteilung der auf diese Weise gefundenen Design Patterns.

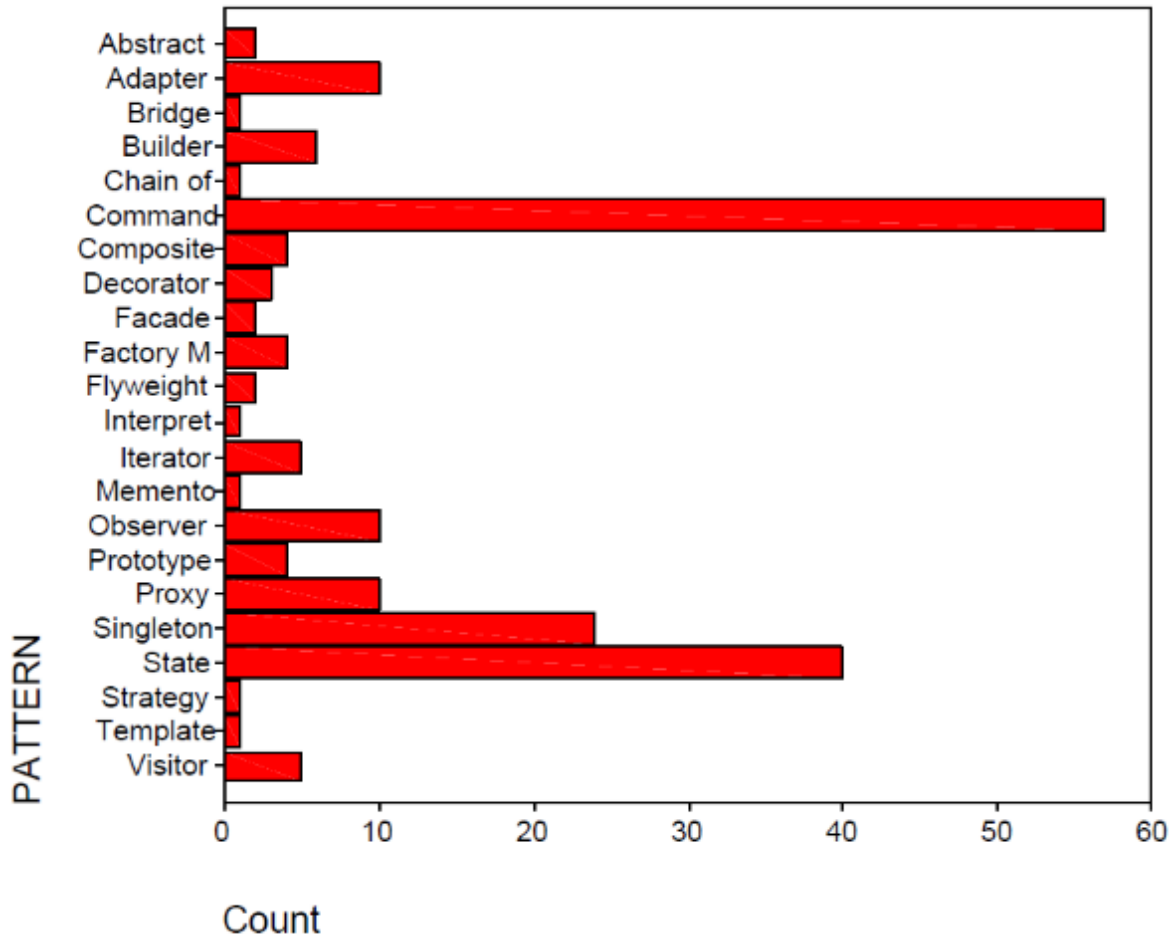


Abbildung 3: Auftreten von Design Patterns (Hahsler, 2003)

Die Autoren stellen fest, dass einige Design Patterns, wie State, wesentlich häufiger auftraten als andere, obwohl das nicht erwartet wurde. Es stellte sich heraus, dass Entwickler dazu neigen, State und Command als übliche Namen für Dateien zu wählen, ohne dass das Pattern eingesetzt wurde. Diese Dateinamen erschienen auch im CVS-Log und führten zu den hohen Ergebnissen. Die Ergebnisse sind somit nur wenig aussagekräftig. Das Mediator-Pattern fehlt gänzlich in der Untersuchung. Hahsler geht nicht auf das Fehlen des Mediator Patterns ein.

In ihrer empirischen Studie untersuchen Aversano et al. (Aversano, Canfora, Cerulo, Del Grosso, & Di Penta, 2007) für zehn der 23 GOF Design Patterns die Häufigkeit ihres Auftretens innerhalb von drei Programmen sowie die Frequenz, in der die Patterns modifiziert werden. Andere Patterns wurden vom Tool nicht erkannt und somit nicht in Betracht gezogen. Die folgende Tabelle 7 zeigt die kleinste und größte Anzahl der gefundenen Design Patterns pro Anwendung, welche in den verschiedenen Releases identifiziert wurden.

Tabelle 7: Übersicht der gefundenen min. und max. Anzahl an Design Patterns in den verschiedenen Releases (Aversano, Canfora, Cerulo, Del Grosso, & Di Penta, 2007)

Erzeuger				
	Factory Method	Prototype	Singleton	
JHotDraw	2-9	4-14	2-7	
ArgoUML	0-7	0-9	76-174	
Eclipse-JDT	47-54	0-6	21-45	
Struktur				
	Adapter-Command	Composite	Decorator	
JHotDraw	18-24	1-2	3-11	
ArgoUML	7-27	1	6-15	
Eclipse-JDT	106-270	1-4	16-25	
Verhalten				
	Observer	State-Strategy	Template Method	Visitor
JHotDraw	6-10	43-78	4-6	1
ArgoUML	5-9	5-48	12-33	0
Eclipse-JDT	16-25	11-33	63-109	0-71

Die drei untersuchten Projekte hatten unterschiedliche Größen, gemessen an den LoC. Je größer das Programm, desto mehr Design Patterns wurden identifiziert. Die Identifikation wurde basierend auf der Arbeit von Tsantalis et al. (Tsantalis, Chatzigeorgiou, Stephanides, & Halkidis, 2006) durchgeführt. Doch je nach Pattern-Typ trifft diese Behauptung nicht ganz zu. So wurden in JHotDraw mehr Strategy und State Patterns (beide Patterns wurden zusammen aufgeführt, weil ihre Ähnlichkeit zu groß ist) gefunden, als im größeren ArgoUML (CollabNet, Inc., 2015).

Eine weitere Studie zum Thema Erkennung von 15 GoF Design Patterns wurde von Guéhéneuc et al. (Guéhéneuc, Sahraoui, & Zaidi, Fingerprinting design patterns, 2004) durchgeführt. Ziel der Studie war es, die Häufigkeit der Verwendung von Design Patterns zu bestimmen. Das Verfahren zur Erkennung dieser Design Patterns wurde auf der Grundlage von *explanation-based constraint programming* (Rich, Alexandron, & Naveh, 2009) entwickelt. Die Ergebnisse sind in Tabelle 8 aufgelistet.

Werden die Ergebnisse beider Studien verglichen, fallen die Unterschiede in der Anzahl gefundner Design Patterns in den gleichen Programmen auf. Wie zu sehen ist, wurden bei JHotDraw nur 2 State und 4 Strategy Patterns gefunden. Im Vergleich zur Studie von Aversano et al. zuvor sind in dieser Stude um den Faktor 7 bis 13 weniger Patterns identifiziert worden. Zwar werden in der Studie von Guéhéneuc et al die Version 5.1 und bei Aversano et al. die Versionen 5.2 bis 5.4 einbezogen, doch ein so hoher Anstieg der verwendeten State und Strategy Pattern ist unwahrscheinlich.

Tabelle 8: Übersicht der Ergebnisse von Guéhéneuc et al. (Guéhéneuc, Sahraoui, & Zaidi, Fingerprinting design patterns, 2004)

	JHotDraw v5.1	JRefactory v2.6.24	JUnit	Lexi v0.0.1	NetBeans V1.0.X	QuickUML 2001	Total	Number of roles (Gamma, Helm, Johnson, & Vlissides, 1994)	Number of classes playing a role per design motif
Number of classes	173	575	157	127	5812	224	7,068		
Design Motifs[5]	Number of micro-architectures similar to design motifs per program								
Abstract Factory					12	1	13	5	217
Adapter	1	17			8		26	4	230
Builder		2		1		1	4	4	24
Command	1				1	1	3	5	67
Composite	1		1			2	4	4	107
Decorator	1		1				2	4	64
Factory Method	3	1					4	4	67
Iterator			1		5		6	5	30
Observer	2		3	2		1	8	4	93
Prototype	2						2	3	32
Singleton	2	2	2	2		1	9	1	9
State	2	2					4	3	32
Strategy	4						4	3	36
Template Method	2						2	2	36
Visitor		2					2	4	138
						Total	93	55	1182

Eine industrienahe Studie über den Zusammenhang von Design Patterns und Softwarefehlern wurde von Vokác (Vokac, 2004) durchgeführt. In seiner Arbeit begleitete der Autor drei Jahre lang die Entwicklungs- und Wartungsarbeiten an einem kommerziellen Software-Produkt mit 500.000 LoC. Ziel der Studie war es, die Defektrate in Klassen, die zu einem Entwurfsmuster gehörten, zu vergleichen. Im Zuge dieser Analyse wurde eine Rangfolge für das Auftreten von 11 Design Patterns aufgestellt. Die Auswahl der untersuchten Design Patterns wurde nach folgenden Kriterien getroffen:

- Das Auftreten des Pattern-Namens in wissenschaftlichen Veröffentlichungen
- Das Auftreten des Pattern-Namens in relevanten Foren
- Das Auftreten des Pattern-Namens auf Webseiten
- Die Struktur des Patterns, die ein effektives Testen erlaubt

Diese Rangfolge basiert auf der Häufigkeit der Verwendung von Design Pattern-Namen in der Literatur, in wissenschaftlichen Publikationen von ISI Web of Knowledge, ACM, IEEE und Anzahl Suchergebnisse bei diversen Suchmaschinen. Tabelle 9 zeigt die Rangfolge für das Vorkommen in wissenschaftlichen Publikationen. Raw zeigt die Anzahl der gefundenen Treffer bzw. Content die relevanten Artikel und Tabelle 10 die Rangliste der Design Patterns, basierend auf den Anzahl Ergebnissen durch Suchmaschinen im Internet. Dazu wurden die Ergebnisse kumuliert und eine Rangliste über den Median der Häufigkeiten ermittelt.

Tabelle 9: Rang von Design Patterns, basierend auf der Häufigkeit in der Literatur (Vokac, 2004)

	ISI		IEEE/ACM	
Pattern	Raw	Content	Raw	Content
Composite	4	4	42	15
Observer	8	6	9	8
Factory	5	4	16	9
Decorator	3	4	4	4
Adapter	1	3	5	4
Bridge	3	0	15	5
Singleton	2	2	4	3
Visitor	1	1	4	4
Proxy	1	0	5	4
Facade	3	1	2	2
Template Method	1	1	1	1
Sum	32	26	107	59

Tabelle 10: Rang von Design Patterns, basierend auf den kumulierten Ergebnissen von Suchanfragen über Suchmaschinen (Vokac, 2004)

Pattern	Median Rang	st.dev	Total hits
Factory	1.0	0.41	24124
Proxy	3.0	1.55	19856
Composite	4.0	1.10	17010
Bridge	5.0	1.72	16105
Observer	5.0	0.75	15677
Singleton	6.0	2.32	20294
Adapter	8.0	0.52	14484
Template Method	9.0	3.67	13424
Visitor	9.0	2.58	13832
Facade	10.0	0.82	9700
Decorator	11.0	0.00	7521

Auch diese Ergebnisse sind mit Vorsicht zu betrachten. Die Auswahlkriterien haben einen sehr starken Bezug zum Namen des Patterns. Wie zuvor schon Hashler et al. feststellten, können bestimmte Namen wie State oder Command bei Suchen zu falschen Ergebnissen führen. Dazu kommt, dass bei Gamma et al. die meisten Patterns mehrere Namen besitzen (im GoF-Buch unter der Kategorie „Also Known as“ zu finden). So kann Decorator auch Wrapper genannt werden. Für Suchen auf Webseiten und in Foren kann dies ausschlaggebend sein.

Eine weitere Studie über den Zusammenhang von Design Patterns und Änderungen an den zugehörigen Programmen wurde von Bieman et al. (Bieman, Straw, Wang, Munger, & Alexander, 2003) durchgeführt. Dazu wurden fünf Programme untersucht, um herauszufinden, ob Änderungen am Programm auch Änderungen am Entwurfsmuster zur Folge haben. Um eine solche Analyse durchführen zu können, bedarf es einer Identifikation aller Design Patterns in den zu untersuchenden Programmen. Tabelle 11 zeigt die Anzahl und den Typ der gefundenen Design Patterns.

Tabelle 11: Lister der identifizierten Patterns pro System (Bieman, Straw, Wang, Munger, & Alexander, 2003)

Pattern	C++Sys	SysA	SysB	Netbeans	jRefactory
Adapter	0	1	0	1	16
Builder	0	0	1	4	2
Command	0	0	0	8	0
Creator	0	0	0	1	0
Factory Method	1	1	4	18	0
Filter	0	0	0	0	2
Iterator	4	0	0	1	0
Proxy	1	0	0	0	0
Singleton	10	1	3	2	1
State	0	2	0	0	3
Strategy	0	1	1	0	0
Visitor	0	0	0	0	2

Alle genannten Studien liefern sehr unterschiedliche Werte, was einen Vergleich erschwert. Außerdem ist die Schnittmenge an verwendeten Patterns klein. Auch fehlt eine klare Aussage, ob die gefundenen Design Patterns alle nach Gammas Definition (Gamma, Helm, Johnson, & Vlissides, 1994) implementiert wurden. So bleibt unklar, welches Entwurfsmuster wie häufig von Entwicklern verwendet wird. Zudem ist bei vielen Studien der Auswahlprozess des genutzten Patterns nicht klar definiert oder nicht akkurat. Das macht die Ergebnisse anzweifelbar. So wurden immer wieder einige GoF-Patterns nicht betrachtet. Bei einigen Studien kommt noch hinzu, dass mit Textsuche gearbeitet wurde. Gerade Begriffe wie Command oder State können noch andere Bedeutungen haben. Auch haben gleiche Patterns unterschiedliche Namen. Zusammenfassend kann gesagt werden: Die Studien zeigen erste Indikationen, aber keine klaren Ergebnisse.

3 Design Patterns – Charakteristika und Grundlagen

„„Wer hohe Türme bauen will, muss lange beim Fundament verweilen.“
- Anton Bruckner, (1824–1896), österreichischer Komponist

Das Fundament dieser Arbeit basiert auf den Arbeiten zu Design Patterns von Gamma et al. (Gamma, Helm, Johnson, & Vlissides, 1994), die in ihrem Buch zusammengetragen wurden. Gerade das Design einer Software ist wichtig für ihre Stabilität, außerdem ermöglicht ein gutes Design eine flexible Veränderbarkeit. Die richtige Verwendung von Design Patterns verbessert das Design. In ihrer Arbeit definieren sie Design Patterns wie folgt:

„A Design Pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context.” (Gamma, Helm, Johnson, & Vlissides, 1994)

Aus dieser Definition wird erkennbar, dass das Entwurfsmuster mehr als nur die Beschreibung eines Lösungsweges ist. Jedes Entwurfsmuster besteht aus vier Bestandteilen:

1. **Pattern Name**: Der Name eines Entwurfsmusters soll so gewählt werden, dass er die nachfolgenden Eigenschaften in ein oder zwei Worten zusammenfasst.

2. **Problemstellung**: Beschreibt, bei welcher Art von Problemen und in welchem Kontext das Entwurfsmuster eingesetzt werden sollte.

3. **Lösungsweg**: Hier wird beschrieben, wie das Pattern das oben genannte Problem löst. In der Beschreibung von Gamma gibt es hierzu nie eine genaue Implementierung, sondern nur ein abstraktes Beschreiben, da Patterns auf jeden Kontext (Programmiersprache, Quellcodestelle etc.) angepasst werden müssen.

4. **Konsequenzen**: Stellt die Konsequenzen dar, die durch den Einsatz des Design Patterns auftreten können. Dazu zählen auch die Einflüsse des Patterns auf die Flexibilität oder Portabilität der Software.

Design Patterns sind Sammlungen von wiederkehrenden Problemen, für die sie eine spezifische Lösung bilden. Es werden die Teilnehmer und ihre Rollen (z. B. Klassen und Interfaces) identifiziert, um ihre Zusammenarbeit oder ihr Verhalten zu verbessern.

Das folgende Beispiel zeigt ein solch bekanntes und immer wiederkehrendes Problem. Der Zugriff auf verschiedene Objekte oder Ressourcen eines Programmes soll reglementiert und besser kontrolliert werden. Mittel der Wahl könnte ein Proxy-Pattern sein. Als Stellvertreter steht es zwischen dem Aufrufer und dem Objekt. Dieses Beispiel wird auch Protection Proxy genannt. Durch den Einsatz des Proxies kann nicht ungefiltert auf Ressourcen zugegriffen werden. Benutzer müssen die notwendigen Rechte oder Rollen besitzen. Die Kontrolle hierfür übernimmt das Pattern. Die Art der Ressourcen kann vielseitig sein.

So kann eine Software den Zugriff auf Dateien oder Netzwerkverbindungen einschränken. Der Proxy implementiert dann einen Autorisierungalgorithmus, z.B. Token oder Password. Jeder Zugriff auf eine Ressource muss durch das Pattern führen. Somit kann der Zugriff, abhängig von der Implementierung, bis ins Detail kontrolliert werden.

Gamma et al. haben 23 Design Patterns, die solche allgemeinen Probleme und Lösungen darstellen, zusammengetragen. Von diesen 23 Design Patterns wurden sechs (ca. 25% alle GoF-Patterns) für eine genauere Analyse im weiteren Verlauf der Arbeit ausgewählt. Im Verlauf der Analyse sollen Erkennungsregeln für Kandidaten dieser Patterns festgelegt werden. Die Auswahl deckt jede der drei bekannten Kategorien von Patterns (Behavioural, Creational und Structural, siehe Abbildung 20) ab und versucht, immer ca. 25% der Pattern-Menge einer Kategorie abzudecken. Vom Typ Creational, definiert die GoF fünf Patterns; von diesen wurde das Builder-Pattern ausgewählt. Decorator und Facade sind die Vertreter der sieben strukturellen Design Patterns (Structural). Aus den elf Patterns vom Typ Behavioural, oder Verhalten, wurden Mediator, State und Strategy ausgewählt. Folgende Abbildung 4 zeigt die bekannten GoF-Patterns bzw. in Fettdruck die zur näheren Analyse ausgewählten. In den nachfolgenden Abschnitten werden die ausgewählten Design Patterns und ihre Charakteristiken kurz vorgestellt. Die Methoden zur Festlegung der Erkennungsregeln werden in Kapitel 4.5 im Detail erklärt.

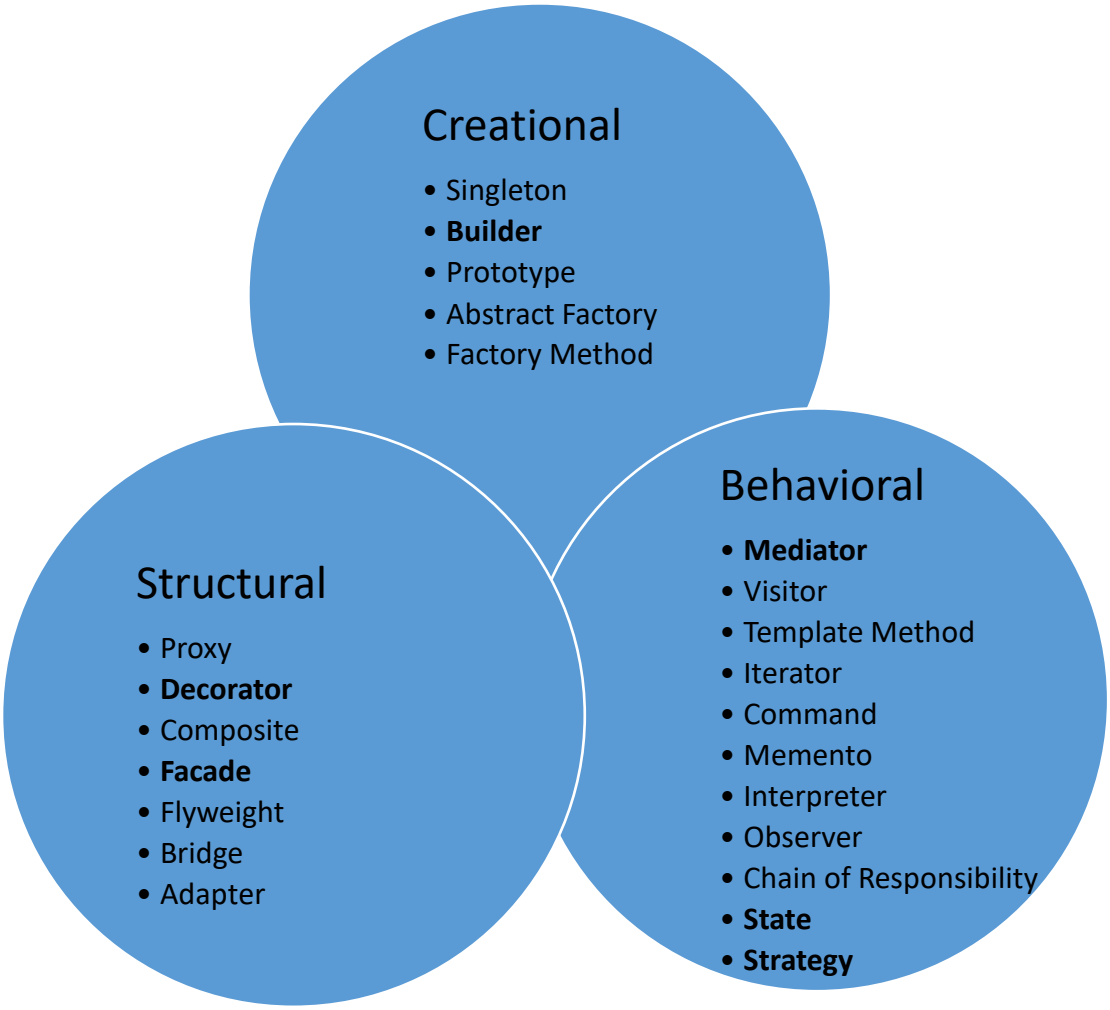


Abbildung 4: Übersicht der nach Gamma beschriebenen Entwurfsmuster (fett = ausgewählt für nähere Analyse)

Um eine Erkennungsregel festzulegen, müssen zuerst die Grundlagen eines Design Patterns analysiert werden. An erster Stelle steht die Definition nach Gamma. Diese liefert erste Einsatzzwecke und Umrisse des Pattern-Verhaltens. Mit Hilfe von Beispielimplementierungen und dem Vergleich von weiteren Definitionen und Einsatzgebieten aus anderen Veröffentlichungen wie Head First (Freeman, Robson, Bates, & Sierra, 2004), Design Pattern for Dummies (Holzner, 2006) usw. entsteht ein spezifischer Analyse-Dokument-Steckbrief jedes Design Pattern. Diese Design Pattern-Analyse enthält auch die generelle Struktur, wie das Pattern in seiner Grundform implementiert wird. Diese Analysen und ihre Ergebnisse werden in den nachfolgenden Abschnitten diskutiert.

3.1 Bedeutung und Anwendung der Entwurfsmuster Erkennung

Die Idee von Design Patterns ist nicht neu. Gamma et al. (Gamma, Helm, Johnson, & Vlissides, 1994) Veröffentlichung liegt schon Jahre zurück. Trotzdem zeigt sich, dass immer noch viel Potenzial der Idee nicht genutzt wird. Wie in Kapitel 2.4 dargestellt, gibt es einige Forschungsbereiche mit dem Ziel, die Verwendung von Design Patterns zu vereinfachen. Es liegt in der Natur von Mustern, nicht immer einfach und verständlich zu sein. Bei Design Patterns wird dies beim Einsatz schnell deutlich. So schrieb die GoF.

„The Design Patterns require neither unusual language features nor amazing programming tricks with which to astound your friends and managers. All can be implemented in standard object-oriented languages, though they might take a little more work than ad hoc solutions. But the extra effort invariably pays dividends in increased flexibility and reusability“ (Gamma, Helm, Johnson, & Vlissides, 1994)

Um Design Patterns zu verwenden, müssen diese auf die verwendete Programmiersprache und Situation angepasst werden. Wie Gamma et al. beschreiben, ist dies mit mehr Aufwand verbunden als eine Lösung ohne Entwurfsmuster (ad hoc). Gerade dieser Adaptionsprozess verlangt viel Wissen über das genutzte Design Pattern. Aus diesem Grund wird eine Empfehlungsmethode benötigt, die den Entwickler bei der komplexen Auswahl von Entwurfsmustern unterstützt. Das Augenmerk einer solchen Methode muss auf der Erkennung von Quellcodezeilen liegen, die verbessert werden können, und im zweiten Schritt in der Auswahlhilfe für ein Design Patterns das die gefundene Stelle verbessern kann.

Die in dieser Arbeit beschriebene Methode zur Erkennung von Entwurfsmusterkandidaten kann eingesetzt werden, sowohl bei der Implementierung von neuen Anwendungen, indem der aktuelle Quellcodestand immer wieder analysiert wird, als auch bei schon veröffentlichten Programmen, zu Wartungszwecken oder nach Erweiterungen. Wichtig ist, dass eine Quellcodebasis vorhanden ist. Grundlage der Analyse stellt der Quellcode dar.

3.2 Builder Pattern

„Separate the construction of a complex object from its representation so that the same construction process can create different representations. " nach Gamma et al. (Gamma, Helm, Johnson, & Vlissides, 1994)

Typ: Creational

Erkennungsebene: Methoden

Das Builder-Pattern erlaubt die Konstruktion komplexer Objekte, indem das Objekt sich selbst erzeugt. Dabei übernimmt das Pattern die Aufgabe des Erzeugens der einzelnen Objekte. Dieses ist in der Lage, verschiedene Objektvariationen zu erstellen, die sich in ihrer Zusammensetzung unterscheiden. Jeder konkrete Erzeugungsprozess verwendet einen Satz von Bestandteilen in unterschiedlichster Zusammensetzung. Die Grundmenge dieser Bestandteile ist dabei immer die gleiche.

Das folgende Beispiel-Szenario für das Builder Pattern entstammt dem Buch „Design Pattern" von Freeman et al. (Freeman, Robson, Bates, & Sierra, 2004) als Teil der Head-First-Buchreihe. In diesem Beispiel wird eine Buchungs- und Planungssoftware für einen Themenpark entworfen. Gäste sollen in der Lage sein, ihren Aufenthalt individuell zu gestalten, dazu gehören Hotel- oder Restaurantreservierungen, Eintrittskarten, etc. Folgende Abbildung 5 zeigt eine Variationsmöglichkeit der Tagesplanung. Jeder der gezeigten Punkte kann für einen Besuch individuell gestaltet werden.

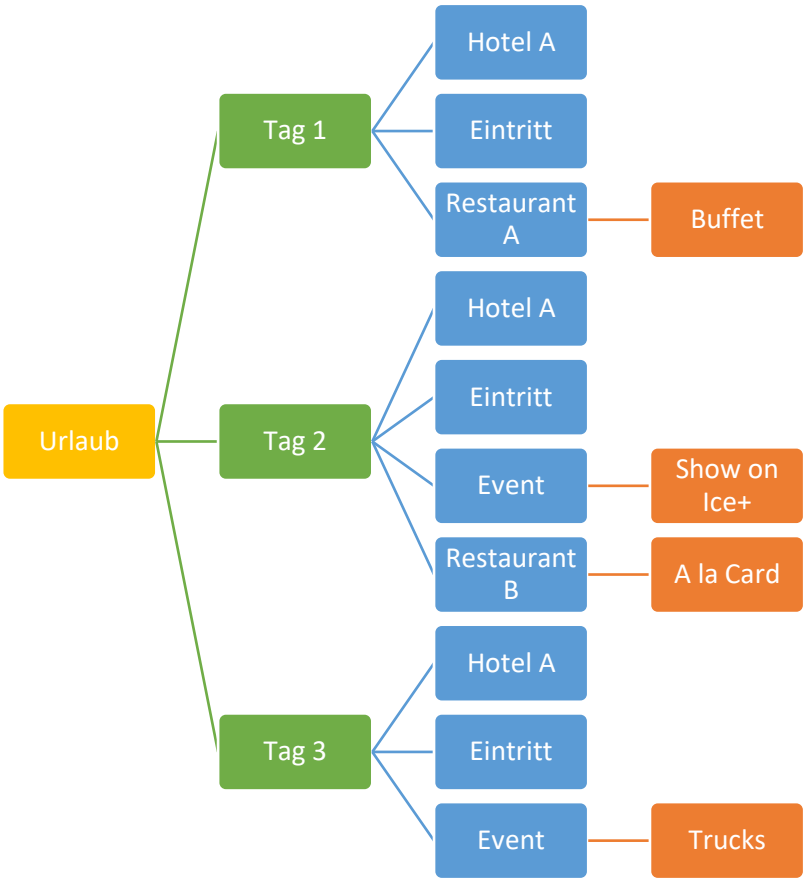


Abbildung 5 Mögliche Planung eines Themenparkbesuches

Somit gibt es eine nahezu unendliche Zahl an Kombinationsmöglichkeiten, die ein Gast buchen kann. Um die gezeigte Struktur und das notwendige Verhalten implementieren zu können, benötigt ein Programm eine flexible Datenstruktur, die immer wieder mit neuen Variationsmöglichkeiten aufgebaut werden kann, ohne die Erstellungsschritte anpassen zu müssen.

Zur flexiblen Erstellung solcher komplexen Datenstrukturen bietet sich ein Builder-Entwurfsmuster an, da es die Erstellungsschritte in kleine Teilaufgaben aufteilt und diese immer wieder neu zusammengesetzt werden können. So werden die Aufgaben Hotel, Events oder Tickets buchen in einzelnen Methoden implementiert. Je nach Vorgang können somit neue Urlaubsobjekte erstellt werden, welche unterschiedlichen Buchungen beinhalten. Obwohl das Objekt immer das gleiche ist sind die Einzelteile anders. Für unterschiedliche Themenparks existieren dann andere konkrete Builder. Wie in Abbildung 5 dargestellt, existiert für jede Buchungsmöglichkeit im Themenpark eine Methode, die diese Teilaufgabe implementiert. Beim Aufrufen einer solchen Methode wird die neue Buchung direkt in die Datenstruktur eingearbeitet.

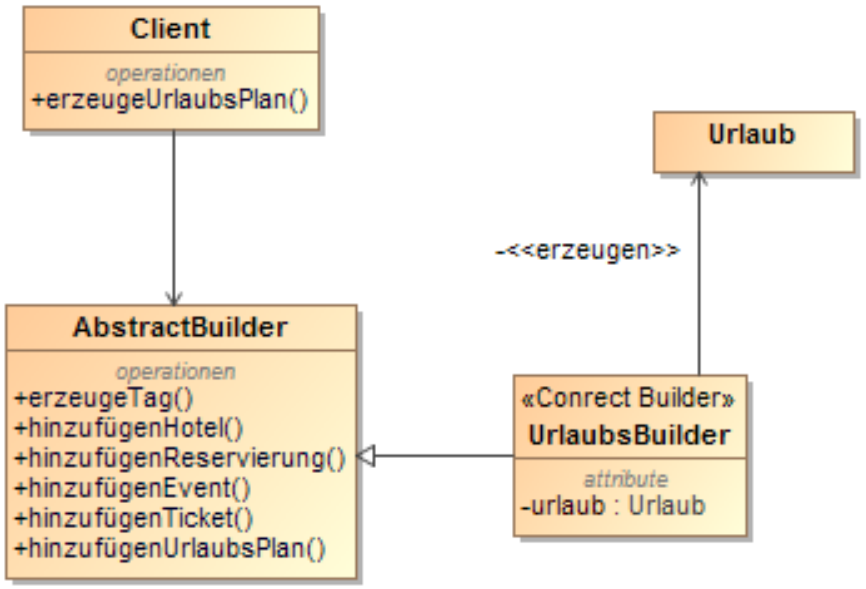


Abbildung 6 Darstellung des Builder Patterns der Planungssoftware

Ohne das Pattern würden das Programm viele einzelne Objekte benötigen, die in einer Struktur gespeichert würde.

3.3 Decorator Pattern

„Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality." nach Gamma (Gamma, Helm, Johnson, & Vlissides, 1994)

Typ: Structural

Erkennungsebene: Klassen

Ein Decorator-Design Pattern, auch Wrapper genannt, findet Anwendung, um Funktionalitäten oder Verhalten einem Objekt zur Laufzeit erweitern zu können. Die Funktionsweise des Patterns ist einfach zu beschreiben: Um das zu erweiternde Objekt wird ein weiteres Objekt gelegt, das das gewünschte Verhalten besitzt. So können sowohl die Methoden der Grundklasse als auch die Methoden der umhüllenden Klasse in demselben Objekt verwendet werden. Die Grundklasse kann dabei um mehrere Hüllen erweitert werden, bis alle benötigten Verhaltensmuster vorhanden sind.

Wichtigste Grundlage für die Verwendung des Patterns stellt die Tatsache dar, dass alle verwendeten Klassen miteinander verwandt sein müssen, d. h. alle von der gleichen Basisklasse abstammen.

Als Beispielimplementierung dient die häufig verwendete Java-IO-API. Ein Java-Reader-Objekt oder ein *Outputstream* existieren nur in jeweils einer Grundversion. Beim Instanziieren von Objekten des Typs *BufferedReader* oder *LineNumberReader* werden entsprechende andere Klassen um die Grundklasse gehüllt. Die gleiche Methode wird von anderen Klassen innerhalb von Java IO angewandt. Entsprechend einfach gestaltet es sich für Entwickler, ein Objekt mit dem gewünschten Verhalten zusammenzubauen.

Das genannt Beispiel, basierend auf der Java *Outputstream*-Klasse, verwendet zur Ausgabe eines Objektes einen Puffer und schreibt die Werte zeilenweise heraus. Auf diese Weise kann die Ausgabe verschiedenartig genutzt werden nur indem neues Verhalte hinzugefügt wird. All diese Eigenschaften werden Abbildung 7 zeigt ein beispielhaftes Objekt mit dem Grundtyp Reader.

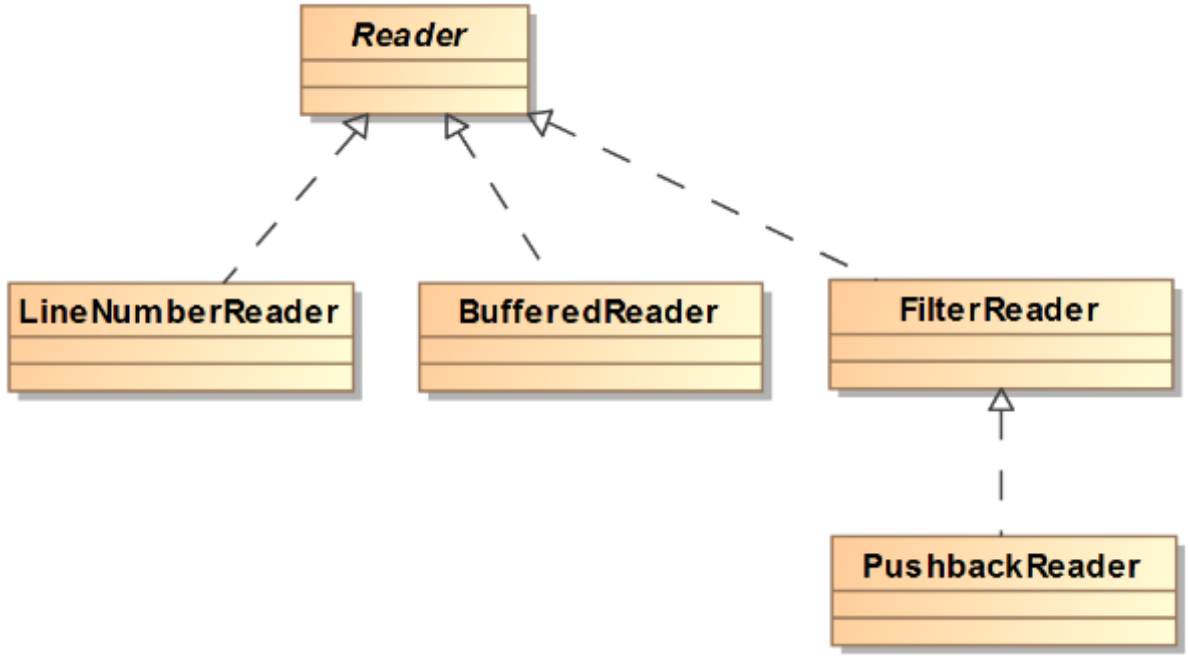


Abbildung 7: Aufbau eines Objektes, das durch einen Decorator mehrmals erweitert wurde

Anwendungsfälle für das Entwurfsmuster sind unter anderem das dynamische Hinzufügen eines bestimmten Verhaltens zu einem Objekt, ohne das Grundobjekt verändern zu müssen. Durch die Verwendung des Decorators müssen weniger Subklassen erzeugt werden, die das Verhalten der Grundklasse erweitern.

3.4 Facade Pattern

„Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use." nach Gamma (Gamma, Helm, Johnson, & Vlissides, 1994)

Typ: Object Structural

Erkennungsebene: Package

Das Facade-Pattern ermöglicht es, ein System in klar abgetrennte Teile zu zerlegen. Die so neu entstehenden Teilsysteme kommunizieren nur über eine Facade mit dem gesamten System. Dies

verringert die Abhängigkeit des Teilsystems vom Gesamtsystem und minimiert den Kommunikationsaufwand. Beides führt zu einer Verbesserung der Austauschbarkeit. Die Facade stellt ein Interface für Klassen zur Verfügung, die in das Teilsystem kommunizieren wollen.

Das Beispiel für das Facade Entwurfsmuster basiert ebenfalls auf dem Buch „Head-First Design Pattern“ von Freeman (Freeman, Robson, Bates, & Sierra, 2004). In diesem Szenario geht es um ein Home-Entertainment-System, das alle gängigen Geräte in einem solchen System verbindet, um abhängige Aufgaben (z. B. Projektor aktivieren, um den DVD-Player zu nutzen) zu kapseln (siehe Abbildung 13), ohne dass der Benutzer sich darüber Gedanken machen muss. Damit dies möglich wird, werden alle Aufgaben hinter der Klasse *HomeControl* gekapselt. Mit dem Aufruf der Methoden *startMovieNight* werden alle notwendigen Schritte durchgeführt, ohne dass diese von außen bekannt sind. Werden hinter der Facade Geräte ersetzt, z. B. ein Streaming Client anstelle eines DVD-Players, müssen die Befehle nur hinter der Facade geändert werden; der Aufruf von außen bleibt gleich, was das System unabhängig von den Subsystemen macht.

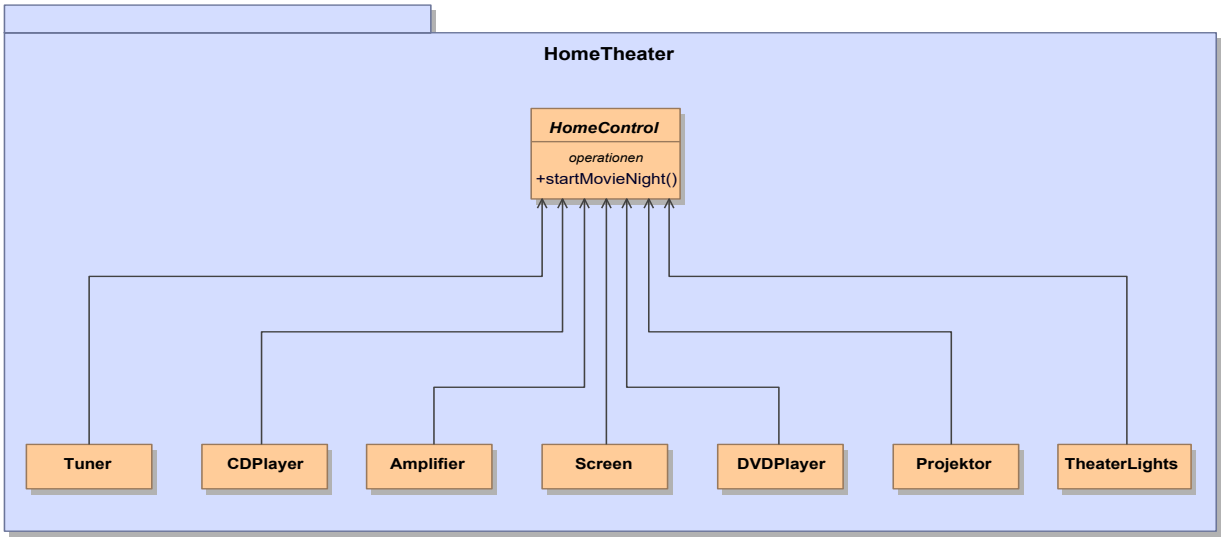


Abbildung 8 Home-Entertainment-System mit Facade Pattern

3.5 Mediator Pattern

„Define an object that encapsulates how a set of objects interact.“ nach Gamma (Gamma, Helm, Johnson, & Vlissides, 1994)

Typ: Behavioural

Erkennungsebene: Klassen

Die Hauptaufgabe des Mediator-Patterns (auch Vermittler genannt) liegt darin, die Kommunikation zwischen verschiedenen Objekten zu vereinfachen und ihre Kontrolle an einem Ort zu vereinen. Dies geschieht, indem alle Kommunikation über eine Mediator-Klasse abgewickelt wird. Dazu wird diese als zentrale Stelle eingesetzt, was dazu führt, dass alle beteiligten Objekte nur noch den Mediator kennen. Der Vorteil dieses Aufbaus liegt in der losen Kopplung zwischen den Objekten, was es Entwicklern erlaubt, einzelne Teile des Systems schneller zu tauschen und Kommunikationswege flexibel anzupassen.

Das folgende Beispiel wurde aus Holzners (Holzner, 2006) Buch „Design Pattern für Dummies“ entnommen; es illustriert die Anwendung eines Mediator-Entwurfsmuster (siehe Abbildung 9). In diesem Szenario wird ein einfacher Online-Shop implementiert, der aus vier Seiten (Welcome,

Shop, Purchase, Exit) besteht. Ein Kunde kann frei zwischen den verschiedenen Seiten hin und her navigieren. Ein Mediator übernimmt die Kommunikation zwischen den einzelnen Seiten, indem er jedes Kommunikationsverhalten in sich kapselt. Die Seiten kennen sich somit nicht untereinander und das Einfügen einer weiteren Seite muss nur im Mediator selbst bekannt gemacht werden.

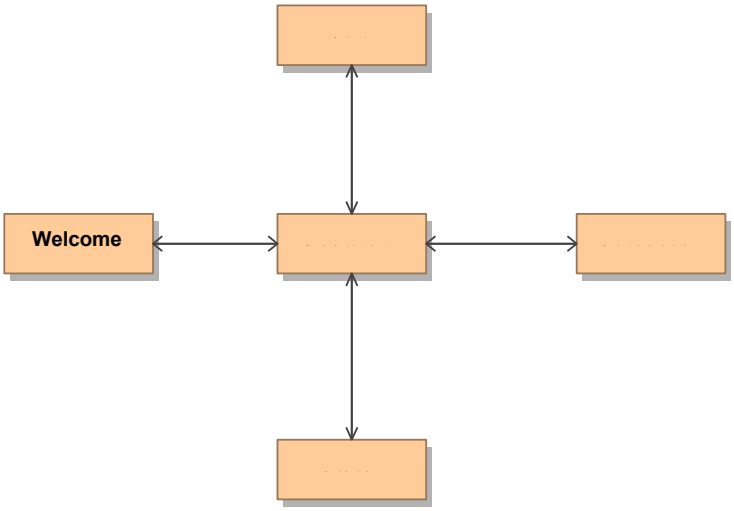


Abbildung 9 Online-Shop Beispiel mit Mediator Pattern

3.6 Strategy Pattern

„Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it." nach Gamma et al. (Gamma, Helm, Johnson, & Vlissides, 1994)

Typ: Behavioural

Erkennungsebenen: Klassen/Methoden

Das Strategy Pattern dient der Abgrenzung von unabhängigen Algorithmen (meist Strategien genannt) vom eigentlichen Programm. Die einzelnen Strategien sind unabhängig voneinander in ihrer Ausführung, besitzen jedoch eine gemeinsame Basis, d. h. sie benötigen ähnliche Eingabewerte und erzeugen gleiche Ergebnistypen. Durch die klare Abgrenzung können die einzelnen Strategien schnell und flexibel ausgetauscht werden.

Das Anwendungsbeispiel für ein Strategy Entwurfsmuster stammt aus der API von AWT bzw. Swing (vgl. Java Doc (Oracle, 2016) und (Hauer, 2016)). Bei der Verwendung des *LayoutManager* in Swing kann der Entwickler zwischen verschiedenen Layouts wählen. Die eigentliche Implementierung der einzelnen Layouts, wie Elemente auf der Seite anzuordnen sind und wie das Verhalten des Layouts ist, wird in einer eigenen Klasse pro Layout gekapselt. Je nach gewähltem Layout, das über einen Parameter im Konstruktor des *LayoutManager* gewählt wird, entscheidet Java zur Laufzeit, welche Strategie für diesen *Context* verwendet werden soll. Jede Layout-Strategie implementiert die gleichen Methoden – nur mit anderem Verhalten. Der zugehörige *Context* wird von der Klasse „Container" implementiert. Abbildung 10 zeigt den Aufbau des Design Patterns in Swing anhand eines UML-Klassendiagramms.

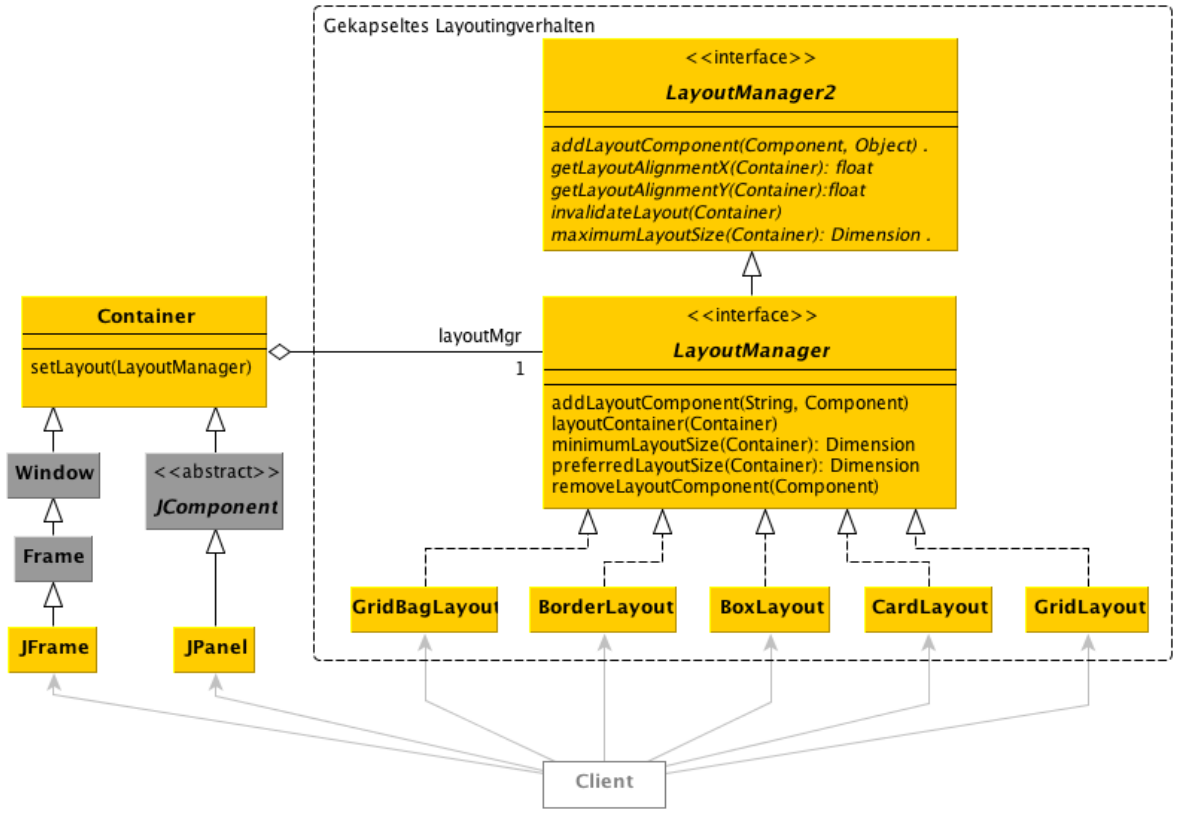


Abbildung 10 Strategy Entwurfsmuster bei AWT/Swing (Hauer, 2016)

3.7 State Pattern

„Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.“ nach Gamma et al. (Gamma, Helm, Johnson, & Vlissides, 1994)

Type: Behavioural

Erkennungsebene: Methoden

Das State Pattern beschreibt ein Vorgehen, einem Objekt verschiedene Verhaltensweisen zu geben und diese zur Laufzeit auswählen zu können, abhängig vom Zustand des Objekts. Verhalten bedeutet in diesem Fall, dass die Zustände, die ein Objekt annehmen kann, voneinander abhängig sind. Das Programm ist jederzeit in der Lage, zu überprüfen, in welchem Zustand es sich aktuell befindet, und kann entsprechend sein Verhalten ändern. Die Verantwortung für die Überprüfung des Zustands und somit das Zuweisen des entsprechenden Verhaltens liegt zentral. Damit ist der Aufwand für Änderungen minimal. Das eigentliche Verhalten ist immer in eigenständigen Klassen modelliert, was das Hinzufügen von neuen Verhalten erleichtert. Für die abfragenden Klassen läuft dieser Verhaltenswechsel, von einer Klasse zu einer anderen, unbemerkt ab.

Um die Verwendung eines State Patterns zu demonstrieren, wird ein weiteres Beispiel aus dem Buch „Head First“ verwendet (Freeman, Robson, Bates, & Sierra, 2004). In diesem Szenario wird eine Applikation entwickelt, die einen Kaugummi-Automaten mit vier Zuständen modelliert implementiert (siehe Abbildung 11), wobei jeder Zustand einen Schritt im Kaufprozess abbildet.

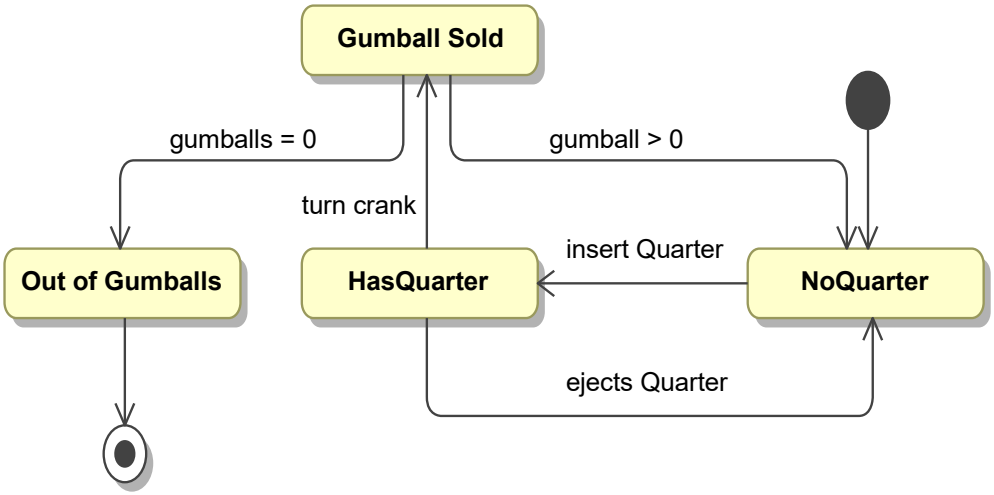


Abbildung 11 Zustandsdiagramm des Kaugummi-Automaten

Das verwendete State Pattern verringert die Komplexität der Implementierung, indem jeder Zustand in einer eigenen Klasse implementiert wird. Dabei erben diese abgeleiteten Klassen immer von einem generalisierten Zustand (siehe Abbildung 12), welcher zu jeder Transition eine Methode anbietet. Eine Transition wird in einem Zustand nur überschrieben, wenn diese dort auch vorkommt. Weitere Zustände in den Automaten einzufügen, gestaltet sich so einfach, da nur die Transitionen betroffen sind, die auf den neuen Zustand zeigen sollen.

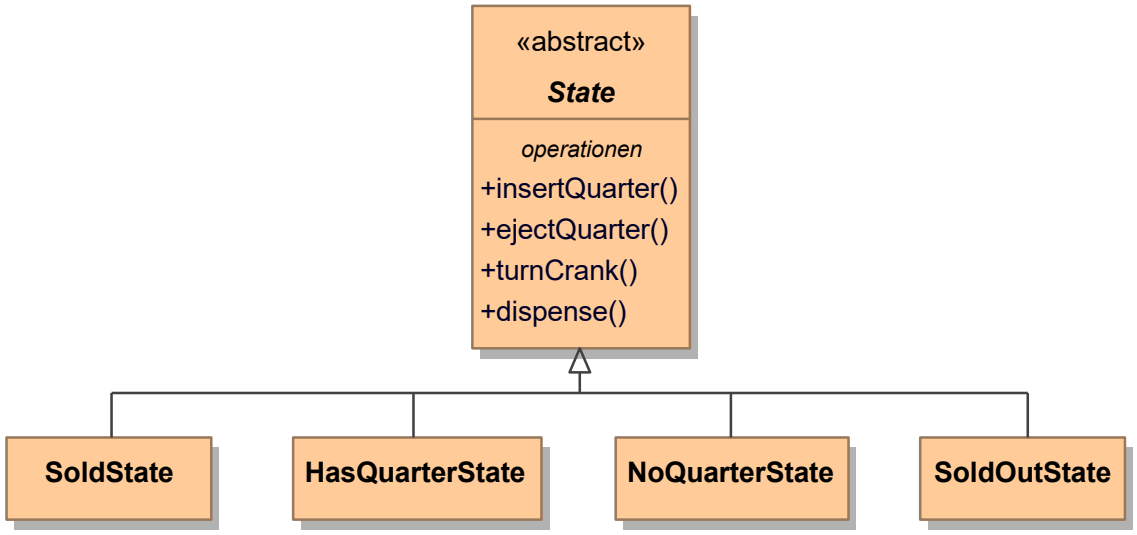


Abbildung 12 Klassendiagramm der Zustände

4 Erkennen von Entwurfsmusterkandidaten

„Es gibt ein Maß in den Dingen, es gibt letztlich feste Grenzen..."
- Horaz (65-8 v.Chr.), röm. Dichter

Die Bestimmung der Qualität einer Software und des darunterliegenden Quellcodes stellt sich heute noch als eine komplexe Aufgabe heraus, deren Bestandteile nicht klar definiert sind. Schon die Detektion von Design-Smells gestaltet sich immer noch schwierig. Design Smells haben direkten Einfluss auf die Software Qualität, da sie schon das Fundament der Anwendung instabil machen. Der Begriff Design Smells wird von Suryanarayana et al. (Suryanarayana, Samarthyam, & Sharma, 2014) wie folgt definiert:

"structures in the design that indicate violation of fundamental design principles and negatively impact design quality" (Suryanarayana, Samarthyam, & Sharma, 2014)

Gibt es für die Detektion von Code Smells Programme wie PMD (PMD, 2015), die Zeile für Zeile nach bestimmen Code-Mustern absuchen, bleibt die Suche nach Design Smells weiterhin komplex. Bei der Unterstützung im Einsatz von Design Patterns zeichnet sich ein noch düstereres Bild. Aktuelle Ansätze (vgl. Kapitel 2.4) sind nur in der Lage, entweder schon implementierte Patterns zu identifizieren oder über komplizierte Frage-Antwort-Systeme einen Vorschlag zu liefern. Mag letztgenannte Methode den Softwaredesigner in frühen Phasen der Entwicklung unterstützen, so hilft es Programmierern nicht, das Problem im Quellcode zu finden und zu verstehen. Das Verständnis, wo ein Entwurfsmuster unterstützen kann und wo nicht, muss immer noch mühsam erarbeitet werden. Eine Erkennung von Entwurfsmusterkandidaten direkt im Quellcode würde helfen, die Qualität von Quellcode zu verbessern.
Dieses Kapitel der Arbeit beschäftigt sich mit der Aufgabe, wie eine Erkennung von Kandidaten im Allgemeinen aussehen kann, ohne bereits auf einzelne Design Patterns einzugehen. Dazu wird nachfolgend ein allgemeines Vorgehen beschrieben, welches zu einem späteren Zeitpunkt relativ einfach zum Aufspüren verschiedenster Entwurfsmusterkandidaten angepasst werden kann.

4.1 Definition der Regeln

Um die Aufgaben klar einzugrenzen, die sich hinter einer Pattern-Erkennungsregel verbergen, wurde die folgende Definition erarbeitet.

> Definition einer Erkennungsregel:
>
> Jede Erkennungsregel beschreibt eine Variante der Identifikation von Quellcodestellen (auch Kandidaten genannt) für das zugehörige Design Pattern. Die Regel selbst besteht aus der Regelbeschreibung, den Metriken zur Erkennung und den Grenzwerten zur Feststellung der Kandidaten-Tauglichkeit. Für ein Design Pattern können mehrere Erkennungsregeln existieren.

Eine Design Pattern-Analyse hat zum Ziel, mehrere Bestandteile festzulegen, die alle für eine Kandidatenerkennungsregel benötigt werden. Zu allererst liefert die Analyse eine textuelle Beschreibung des Design Patterns, seines Verhaltens und seiner Struktur. Aufbauend auf dieser Beschreibung wird eine Erkennungsregel definiert, die ebenfalls textuell beschreibt, wie und an

welcher Stelle ein Kandidat im Quellcode identifiziert werden kann. Basierend auf dieser Erkennungsregel wird ein Satz an Metriken festgelegt, die bei der Erkennung der Kandidaten anhand von verschiedenen Merkmalen aus dem Quelltext (z. B. Anzahl Case-Anweisungen), angewandt werden.

Der erste Schritt zur Festlegung einer Pattern-Erkennungsregel liegt in der Analyse von Quellcodestellen, die durch das Entwurfsmuster verbessert werden können, für die eine Regel aufgestellt werden soll. Es gilt zu verstehen, wie diese Quellcodestellen automatisch zu identifizieren sind und welche Merkmale sie ausmachen. Der zweite Schritt beinhaltet die Ergebnisse der Design-Patter Untersuchung aus Kapitel 3 zu Verhalten und Aufbau. Es gilt nun zu verstehen, wie eine solche Codestelle ohne Design Pattern implementiert werden könnte. Diese Lösungsansätze bilden die Grundlage für die Kandidatenauswahl, da ein Entwurfsmuster in vielen Fällen die bessere Wahl wäre.

Mit anderen Worten: Wie würde das im Quellcode identifizierte Problem gelöst, ohne dass ein Entwurfsmuster zum Einsatz kam, obwohl dessen Einsatz die optimale Lösung dargestellt hätte? Diese gefundenen Lösungen fließen später in die Erkennung von Kandidaten ein. Als Beispiel für diese Art des Vorgehens dient ein Design Pattern-Kandidat, der in einem Open Source Project gefundenen wurde (vgl. Kapitel 7). Ein System kann mehrere Zustände besitzen, welche von außen festgelegt werden. Auf jeden Zustandswechsel soll eine Reaktion des Systems stattfinden und entsprechende Logik ausgeführt werden. Ein Entwickler mit Wissen im Bereich Entwurfsmuster würde dieses Problem mit einem State Pattern lösen. Doch zur Erkennung von State-Kandidaten sind Lösungen gefragt, die gerade dies nicht tun. Das folgende Klassendiagramm in Abbildung 13 zeigt eine solche Lösung, die auf der Verwendung einer Switch-Anweisung beruht.

```
public class Systemzustand {
        private int aktueller Zustand = 0;
        private enum MoeglicheZustaende
        public void ZustandFeststellen(int neuerZustand) {
                switch(neuerZustand){
                        case 0:
                                ZustandA();
                                break;
                        case 1:
                                ZustandB();
                                break;
                        case 2:
                                ZustandC();
                                break;
                        default:
                                System.out.println("i liegt nicht zwischen null und drei");

                }
        }
        public void ZustandA() {
                …
        }
        public void ZustandB() {
                …
        }
        public void ZustandC() {
                …
        }
}
```

Abbildung 13 Darstellung der Implementierung einer Zustandsmaschine

Dieses Beispiel könnte im Quellcode mit einer ineffizienten Lösung wie folgt implementiert werden. Die Klasse *Systemzustand* beschreibt die möglichen Zustände (A, B, C) eines Systems und es existiert für jeden Zustand eine Methode mit entsprechender Logik. Die Zustände sind zur besseren Lesbarkeit in einer Enumeration (*MoeglicheZustaende*) vordefiniert. Soll sich der Zustand verändern, wird von außen eine Methode (*ZustandFeststellen*) aufgerufen und der neue Zustand mitgegeben. Innerhalb der Methode existiert eine Switch-Anweisung mit je einem Case für jeden Zustand.

Zur Erkennung und zur Festlegung des möglichen Verbesserungspotenzials solcher Kandidaten gehört zu jeder Regel ein Satz an Metriken, welche benötigt werden, um die Gültigkeit eines Kandidaten festzustellen. Metriken symbolisieren ein Merkmal eines Kandidaten. Dabei kann eine Metrik zu verschiedenen Regeln gehören. In vielen Fällen sind die Metriken Abfragen, ob ein Verhalten oder eine Struktur vorhanden ist. Zum Beispiel bei der Abfrage ob ein Interface für die Klasse vorhanden ist oder nicht kann das Ergebnis der Metrik mit Booleschen Werten dargestellt werden. Bei einigen Metriken kommen numerische Werte zum Einsatz. Diese sind dann notwendig, wenn nach einer Anzahl von Strukturen gesucht wird, z. B. eine bestimmte Anzahl an Case-Anweisungen. Der Typ einer Metrik wird durch die Analyse der oben genannten Eigenschaften eines Entwurfsmusters bestimmt. Zur Bestimmung des Verbesserungsbedarfes an der gefundenen Quellcodestelle werden für alle Metriken Grenzwerte festgelegt. Die Komplexität der Erkennungsregeln und die Anzahl der notwendigen Metriken variieren je nach Komplexität des analysierten Design Patterns.

Eine allgemeine Beschreibung der Bestandteile findet sich in Tabelle 12.

Tabelle 12 Bestandteile einer Erkennungsregel

Bestandteil	Beschreibung
Design Pattern-Analyse	Textuelle Beschreibung (Struktur, Verhalten) des untersuchten Entwurfmusters
Erkennungsregel	Textuelle Beschreibung zur Erkennung
Metriken	Merkmale im Quellcode die zur Erkennung vermessen werden
Grenzwerte	Grenzen für jede Metrik um den Grad der möglichen Verbesserung bestimmen zu können.

Die Beschreibung der einzelnen Design Pattern-Erkennungsregeln für spezifische Pattern sind in Kapitel 4.5 beschrieben.

4.2 Merobase

Um die Tauglichkeit, also das Verbesserungspotenzial, von Kandidaten zu bestimmen, werden Grenzwerte benötigt Die Grundlage nahezu aller im Zuge dieser Arbeit definierten Grenzwerte ist der Index der Komponenten-Suchmaschine Merobase (Janjic, Hummel, Schumacher, & Atkinson, 2013). Zusammengesetzt ist der Index mit Projekten aus einigen der bekanntesten *Internet Software Repositories* wie *sourceforge.net* oder *apache.org*. Dabei wurden Projekte aus verschiedenen *Concurrent Version Systems* (CVS) oder *Subversion-Repositorien* (SVN) heruntergeladen und in einer Datenbank gesammelt. Auf dieser Grundlage wurde der Merobase-Index erstellt. Der aktuell verfügbare Index (vom 06.02.2010) beinhaltet über 2,4 Millionen Java- und über 4,3 Millionen Class-Dateien. Eine detaillierte Aufstellung der Index-Inhalte zeigt die folgende Tabelle 13.

Tabelle 13: Inhalt des Merobase-Indexes (Janjic, Hummel, Schumacher, & Atkinson, 2013)

Dateityp	Anzahl
Java-Quellcode-Dateien	2.429.999
Java-Class-Dateien	203.689
.jar-Container	27.168
Java-Quellcode in.jar	40.692
Java-Class in.jar	4.342.376
Java-Methoden	22.262.954
LoC	182.224.390

Für eine Suche über den Index stellt Merobase zwei Wege zur Verfügung.

1. Merobase Index Browser Tool: Die erste Suchmöglichkeit verwendet eine einfache Lucene-Textsuche. Ein Suchstring kann dabei aus einfachen verknüpften Wörtern oder regulären Ausdrücken bestehen.
2. Merobase.com: Die Website stellt eine vergleichbare Funktionalität wie das Tool bereit. Sie ist seit Mitte 2014 offline.

Über diese beiden Wege können, mit Hilfe von drei verschiedenen Methoden, die indizierten Projekte durchsucht werden.

1. **Einfache Textsuche:** Die erste Suchmöglichkeit verwendet eine einfache Lucene-Textsuche. Ein Suchstring kann dabei aus einfachen verknüpften Wörtern oder regulären Ausdrücken bestehen.
2. **Merobase Query Language:** Zur Abfrage würde innerhalb von Merobase eine eigne entwickelte Sprache entwickelt. Diese nennt sich Merobase Query Language (MQL).
3. **Parser-Suche:** Merobase ist in der Lage, Java- oder C#-Quellcode zu parsen. Somit gibt es dem Entwickler die Möglichkeit nach bestimmten Methodenköpfen zu suchen.

Zusätzlich bietet der Index eine Reihe von Feldern, die eine Detailsuche ermöglichen. Unter Verwendung der Felder ist es z. B. möglich, die Programmiersprache einzugrenzen. Einen Auszug der möglichen Felder zeigt die folgende Tabelle 14.

Tabelle 14: Mögliche Suchfelder für den Merobase-Index (Janjic, Hummel, Schumacher, & Atkinson, 2013)

Feld:	Beschreibung:	Beispiel:
lang	Auswahl der benötigen Programmiersprache	*lang: Java*
implements	Nur Klassen suchen, die das angegebene Interface implementieren.	*implements:IState*
extends	Nur Klassen auswählen, die von der angegebenen Klasse erben	*extends:Builder*
type	Eine Auswahlmöglichkeit, ob die auszuwählende Datei eine Klasse oder ein Interface sein sollte	*type:interface*
form	Quellcode oder Binärcode	*form: source*

Suchanfragen über den Merobase-Index

Der Merobase-Index wurde nach bereits implementierten Design-Patterns durchsucht und die Ergebnisse manuell ausgewertet. Die folgende Tabelle 15 zeigt einen Überblick über die gefundenen Design Patterns im Merobase-Index und die konkreten Suchstrings. Bevor die Merkmale des gesuchten Patterns gemessen werden können, muss die Ergebnisliste noch gefiltert werden. Aus der Liste werden alle Design Pattern-Implementierungen entfernt, die nicht auf der Definition von Gamma beruhen, z. B. ein Bridge-Pattern-Interface, das von keiner Klasse implementiert wird. Bei der Festlegung der Grenzwerte ist es ein Ziel für jede Erkennungsregel, mindestens 50 verschiedene zugehörige Design Pattern-Implementierungen zu vermessen. Dabei dürfen nie mehr als zwei Implementierungen aus dem gleichen Projekt entnommen werden.

Tabelle 15: Übersicht über die gefundenen Design Patterns in der Merobase pro Pattern und mit Suchstring mit ‚I' als Anfangsbuchstaben

Pattern	**Builder**	**Facade**
Suchstring	I*Builder lang:java type:interface (protocol:svn OR protocol:CVS)	I*Facade lang:java type:interface *(protocol:svn OR protocol:CVS)*
Gefundene Entwurfsmuster Implementierungen*	354	63
Pattern	**Decorator**	**Mediator**
Suchstring	I*Decorator lang:java type:interface (protocol:svn OR protocol:CVS)	I*Mediator lang:java type:class (protocol:svn OR protocol:CVS)
Gefundene Entwurfsmuster Implementierungen*	108	392
Pattern	**State**	**Strategy**
Suchstring	I*State lang:java type:interface (protocol:svn OR protocol:CVS)	I*Strategy lang:java type:interface (protocol:svn OR protocol:CVS)
Gefundene Entwurfsmuster Implementierungen*	801	286

** Ergebnisse vor der manuellen Überprüfung.*

Quervergleich mit Design Pattern-Erkennungswerkzeugen

Wie zuvor erwähnt, wurden die in der Merobase gefundenen Design Patterns manuell auf ihre korrekte Implementierung nach Gamma et al. evaluiert, d. h. der Quellcode jedes ausgewählten Ergebnisses wurde daraufhin überprüft. Nur eine solche Vorgehensweise erlaubt das Sicherstellen der Auswahl von Design Patterns nach Gamma. Implementierungen, die sich nicht an Gamma et al. (Gamma, Helm, Johnson, & Vlissides, 1994) halten, würden die Grenzwerte verfälschen, da die Analyse und Definition der Regeln auf der Gang of Four beruhen.

Die identifizierten Design Pattern-Implementierungen wurden einer weiteren Evaluation unterzogen. Dieser Schritt sollte die Richtigkeit der Ergebnisse festigen. Dazu wurden die eigenen Ergebnisse mit den Ergebnissen einem Design Pattern-Erkennungswerkzeuge quervergliche. Diese Werkzeuge waren Pattern4 von Chatzigeorgiou et al. (Chatzigeorgiou, Tsantalis, & Stephanides, 2006) und DeMIMA von Guéhéneuc et al. (Guéhéneuc & Antoniol, DeMIMA: A Multilayered Approach for Design Pattern Identification, 2008)

Wie in Kapitel 2.4 bereits diskutiert, liefern die verschiedenen Werkzeuge sehr unterschiedliche Ergebnisse, wie Dong et al. (Dong, Zhao, & Peng, 2009) nachwiesen, was auf die unterschiedlichen Ansätze zurückzuführen ist und darauf, dass es sich um Forschungsprojekte handelt. Deshalb ist davon ausgegangen worden, dass es keine 100%-Übereinstimmung bei Ergebnissen gibt. Doch es sollte eine möglichst große Schnittmenge zwischen den in der Merobase identifizierten Pattern und denen der Tools geben – unter der Annahme, dass die meisten gefundenen Design Patterns der Tools korrekte Implementierungen sind.

Nur Pattern4[2] stand zum Zeitpunkt der Evaluation als lauffähige Version zur Verfügung. Bei DeMIMA[3] standen verschiedenste Suchergebnisse von unterschiedlichen Open Source-Anwendungen bereit. Aus diesem Umstand ergaben sich zwei Vorgehensarten.

Durch Pattern4 konnten Projekte, bei denen Design Pattern-Implementierungen über die Merobase gefunden wurden, direkt mit dem Werkzeug ein weiteres Mal analysiert werden. Nach der Analyse wurde ein Vergleich der beiden Ergebnismengen durchgeführt. Jedes identifizierte Pattern aus Merobase sollte durch Pattern4 ebenfalls gefunden werden. Es gab eine Einschränkung bei der Nutzung von Pattern4. Das Tool unterstützt nur eine Teilmenge der benötigten Design Patterns, diese sind Decorator, State und Strategy. Bei State und Strategy Pattern wird in der Ergebnissemenge nicht getrennt. Interessant war auch, dass Merobase mit dem Quellcode arbeitet und Pattern4 im Vergleich die Binärdateien benötigt. Zudem gab es keine Möglichkeit, die Analyse von Pattern4 durch Parameter etc. zu verbessern.

Der Quervergleich mit den Beispielen des Decorator-Patterns lieferte folgendes Ergebnis (Tabelle 16); „Identifiziert“ bedeutet, dass das Tool in der Lage war, das Pattern zu erkennen.

Tabelle 16: Analyseergebnisse für das Decorator-Design Pattern

Programm	Design Pattern-Fundstelle	Pattern4
JHotDraw	DecoratorFigure	Identifiziert
Yawl Editor	JoinDecorator	Nicht Identifiziert
	NetGraphModel	Nicht Identifiziert
Wicket	IAjaxCallDecorator	Identifiziert
	CryptFactoryCachingDecorator	Identifiziert
DBEdit	internal (Klassenname)	Nicht Identifiziert
VilAug	VisualizationControllerDecorator	Nicht Identifiziert
	ElementDialogDecorator	Nicht Identifiziert

Von den acht Decorator-Design Patterns aus der Merobase, welche manuell geprüft wurden, konnte Pattern4 nur drei wiedererkennen, was ca. 37% entspricht. Beim Vergleich mit den State Pattern-Ergebnissen zeichnet sich ein anderes Bild ab, wie Tabelle 17 zeigt. Hier wurden vier von fünf (80%) State Patterns wiedererkannt.

[2] http://java.uom.gr/~nikos/pattern-detection.html
[3] http://www.ptidej.net/downloads/replications/tse08/

Tabelle 17: Analyseergebnisse für das State-Design Pattern

Programm	Design Pattern-Fundstelle	Pattern4
NPlayer	State.java	Identifiziert
CUF	State.java	Identifiziert
Battlestars	State.java	Nicht Identifiziert
EasyMock	IMocksControlState	Identifiziert
vesuf	State.java	Identifiziert

Ein ähnliches Bild ergab die Analyse (Tabelle 13) der Strategy Pattern-Ergebnisse. Hier wurden sechs von sieben wiedererkannt. Daraus lässt sich schließen, dass der Erkennungsalgorithmus für State und Strategy gute Ergebnisse liefert. Im Gegensatz dazu sind die Ergebnisse für das Decorator-Pattern wenig aussagekräftig. Insgesamt wurden 13 von 20 Design Patterns von Pattern4 wiedererkannt, was 65% entspricht.

Tabelle 18: Analyseergebnisse für das Strategy-Design Pattern

Programm	Design Pattern-Fundstelle	Pattern4
Design by Contract Utilities	ILoggingStrategy	Identifiziert
	IThrowStrategy	Identifiziert
Concept Explorer	LatticeImplicationCalculator	Nicht Identifiziert
Su Doku Solver	IStrategy	Identifiziert
Sat4J	LearningStrategy	Identifiziert
	RestartStrategy	Identifiziert
Glazedlists	TextSearchStrategy	Identifiziert

Die Ergebnisse zeigen, dass nur eine manuelle Evaluierung des Quellcode eine klare Erkennung der Design Patterns erlaubt. Der Vergleich der Ergebnisse war nur möglich, weil bekannt war, dass sich in den Projekten Design Patterns befinden. Eine Suche ohne diese Information könnte zu *false positives* führen und würde somit den Grenzwert verfälschen.

Ausschlaggebend für die Ermittlung der Grenzwerte war immer das bei Merobase gefundene und manuelle überprüfte Ergebnis, da nur im Quellcode die notwendigen Metriken ermittelt werden konnten. Es wurden immer fünf Beispielprojekte verglichen.

4.3 Festlegung der Grenzwerte

Dieser Abschnitt beschreibt die allgemeine Herangehensweise zur Festlegung der Grenzwerte für die verschiedenen Metriken der Erkennungsregeln. Erst durch die Anwendung von Grenzwerten auf die Regeln wird es möglich sein, einen Kandidaten final auf eine Art und Weise einzuordnen um den Entwickler eine Gewichtung für das mögliche Verbesserungspotenzial aufzuzeigen. Jede Metrik besitzt einen oder mehrere spezifische Grenzwerte, welche verschiedenster Ausprägung, wie Boolean oder numerisch, sein können.

4.3.1 Vorgehen

Das Vorgehen zur Definition von Grenzwerten der verschiedenen Metriken beinhaltet vier allgemeine Schritte und wird in Abbildung 14 dargestellt. Diese Schritte werden für jedes Pattern individualisiert. Die Anwendung eines der verschiedenen bekannten Entwurfsmuster-Erkennungswerkzeuge die in Kapitel 2.4 und 7 beschreiben werden, war keine Option, da, wie oben ebenfalls erwähnt, die Resultate nicht stabil sind.

Des Weiteren gab es kein Tool, das die hier verwendeten Design Patterns unterstützt. Es wären somit mehrere Tools notwendig, die verschiedenste Ansätze verfolgen. Aus diesem Grund wurde entschieden, das nachfolgende Verfahren zu verwenden.

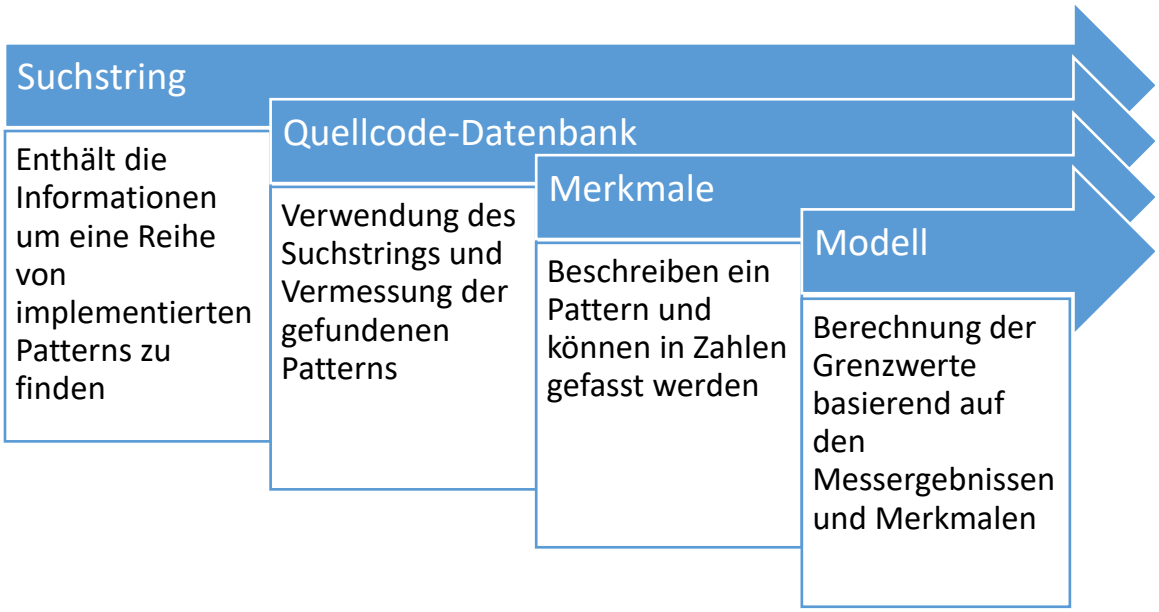


Abbildung 14: Prozess für das Allgemeine Vorgehen bei dem Erkennungsregeldesign

Es beginnt mit der Festlegung eines Suchstrings zur Identifikation von bereits implementierten Design Patterns. Dieser Suchstring wird verwendet, um in Merobase, bereits implementiere Entwurfsmuster-Implementierungen zu identifizieren.

Im nächsten Schritt werden besondere Merkmale eines Design Patterns bestimmt. Ein solches Merkmal könnte sein, dass z. B. mehrere Klassen die gleiche Anzahl an Methoden besitzen. Sie ergeben sich aus der Analyse zuvor. Aus den so gewonnenen Daten wird pro Pattern ein Erkennungsmodell erstellt. Im Folgenden werden die einzelnen Schritte näher beschrieben. Eine detaillierte Beschreibung für die einzelnen Design Pattern-Analysen ist in Kapitel 4.5 zu finden.

4.3.2 Suchstring festlegen

Das Auffinden von bereits implementierten Design Patterns bildet das Fundament für die Festlegung von Grenzwerten, welche für die Bestimmung des Verbesserungsbedarfs eines Kandidaten benötigt werden. Wie in Kapitel 2.4 beschrieben, gestalten sich die Suche und Identifikation von bereits implementierten Design Patterns immer noch schwierig. Zwar existieren einige Detektionswerkzeuge, aber die Ergebnisse solcher Detektionswerkzeuge sind nicht stabil genug für eine Analyse. Ein Problem stellt die Menge an false-positive oder true-negatives dar, deren Menge je nach Detektionsansatz teilweise recht groß ist.
Der genutzte Ansatz zum Auffinden von Entwurfsmustern kann somit nicht auf diese Werkzeuge zurückgreifen. Deshalb nutzt die Analyse eine textuelle Beschreibung der gesuchten Design Pattern-Struktur. Zur Festlegung wird die Zusammensetzung eines Patterns, ausgehend von Gammas Definition, analysiert; z. B. besitzt ein State Pattern immer ein Interface, welches alle konkreten States verwenden. Außerdem werden ein paar Annahmen getroffen, um die Anzahl der Ergebnisse einzugrenzen. Auf das Beispiel State angewandt, beinhalten die Interfaces von State häufig das Wort „State“ selbst. Einige Java-Konventionen könnten bei bestimmten Patterns berücksichtig werden, z. B. dass ein Interface in Java immer mit dem Buchstaben ‚I‘ beginnen sollte.

Kombiniert werden die Merkmale und Annahmen mit regulären Ausdrücken. Ausgehend von diesen Punkten, könnte ein Suchstring wie folgt definiert sein:

*I*Patternname lang:java type:interface*

Diese Suchstrings werden auf den Merobase-Suchindex angewandt, um entsprechende Beispielimplementierungen finden. Trotzdem müssen die gefundenen Ergebnisse weiter manuell kontrolliert werden, ob es sich dabei um Design Pattern-Implementierungen nach Gamma et al. (Gamma, Helm, Johnson, & Vlissides, 1994) handelt.

Formel 1: Beispielsuchstring

Technische Anmerkungen:

Es bleibt zu erwähnen, dass die von Merobase verwendete *Apache Lucene Search Engine* eine Platzhaltersuche (*wildcard search*) nur dann zulässt, wenn der Suchstring mit einem alphanumerischen Zeichen beginnt. Aus diesem Grunde verwenden manche angewendeten Suchstrings X*Patternname, wobei X für jeden beliebigen der 26-Buchstaben des Alphabetes stehen kann, die durchexerziert wurden.

Merobase sucht nicht nur 100% übereinstimmende Ergebnisse, sondern weicht vom Suchstring ab. Alle Ergebnisse werden von Merobase direkt nach ihrer Relevanz sortiert, d.h. je mehr das Ergebnis mit dem Suchstring übereinstimmt, desto höher die Relevanz in%.

Eine grundlegende Einschränkung wurde vorab für jede Suche getroffen, nämlich, dass die Analyse auf die Programmiersprache Java beschränkt wird. Java besitzt eine große Verbreitung, wie der *RedMonk Language Index* von 2014 zeigt (RedMonk, 2015). Diese Studie listet Java als zweitmeist verwendete Sprache innerhalb der Projektplattform GitHub auf. Ebenfalls stark vertreten ist Java auch im Index von Merobase, in dem 79,6% der Dateien auf Java basieren; zum Vergleich: Es basieren nur ca. 14,9% auf C# und C++ (Janjic, Hummel, Schumacher, & Atkinson, 2013). Die Festlegung der Programmiersprache geschieht über das Feld Lang (lang:java) und wird in dem Suchstring entsprechend gesetzt.

4.3.3 Pattern-Merkmale

Grundlage der Merkmalssuche bildet das Ergebnis der Design Pattern-Analyse, um Merkmale zu finden, die dieses Pattern beschreiben. Es ist dabei wichtig, dass die Merkmale sowohl in einem implementierten Pattern als auch in einem Kandidaten gefunden werden können. Nur durch diese Konstellation ist es möglich, die Grenzwerte von implementierten Design Patterns zu messen und auf die Kandidaten anzuwenden. Aus diesen Merkmalen werden später die Grenzwerte für die Kandidaten-Erkennung ermittelt. Ein solches Merkmal für implementierte Design Patterns könnte zum Beispiel sein:

- die Anzahl bereitgestellter Methoden eines Interface,
- die Anzahl der Klassen, die ein Interface implementieren, oder
- die Anzahl der Klassen, die auf eine Klasse zugreifen, je nach untersuchtem Pattern.

Je nach Entwurfsmuster ist die Auslegung der gewählten Merkmale für implementierte Design Patterns zu den Merkmalen für Kandidaten anders. Als Beispiel dient das Merkmal, wie viele Klassen ein bestimmtes Interface realisieren, welches in einem State Pattern angewandt wird. Das genannte Merkmal zeigt die Anzahl an States im Pattern. Bei einem State Kandidaten wird hierzu

die Anzahl an Case-Anweisungen pro Switch, die in verschiedenen Methoden innerhalb der Klasse gleich sind, genutzt. Die pattern-spezifischen Merkmale werden detailliert in den Abschnitten der einzelnen Design Pattern-Erkennungsregeln dargestellt (vgl. Kapitel 4.5).

4.4 Modell

Im letzten Schritt wird aus den gemessenen Merkmalswerten und aus der Art, wie die Merkmale des Patterns in einem Kandidaten repräsentiert werden, ein Erkennungsmodel für jedes Pattern erstellt. Ein Modell kann je nach Anzahl an Merkmalen mehrere Attribute und somit Grenzwerte besitzen. Für jedes Merkmal wird die statistische Verteilung berechnet, bestehend aus Minimum, 1. Quartil, Durchschnitt, Median, 3. Quartil und Maximum.

So entstand für jede Erkennungsregel ein dreistufiges Empfehlungsmodel. Je höher die Stufe, desto mehr wird der Einsatz eines Design Patterns an der entsprechenden Quellcodestelle *empfohlen*. Im Folgenden werden die Definitionen der drei Stufen kurz erläutert.

- **Möglich**: Auf der niedrigsten Stufe erscheint die Verwendung des Design Patterns als *möglich*, d. h. das angegebene Pattern wäre an der Codestelle denkbar, doch der Einsatz kann mit großem Zusatzaufwand verbunden sein oder der Einsatz würde der Code-Übersichtlichkeit nur begrenzt dienen.
- **Sinnvoll**: Auf der mittleren Stufe erscheint die Verwendung als *sinnvoll*, da das Pattern den Quellcode übersichtlicher gestalten sollte. Es bleibt ein Restaufwand, der vor dem Einsatz des Patterns zu analysieren ist. Wird die Stelle im Quellcode in Zukunft weiter ausgebaut, sollte das Pattern eingesetzt werden.
- **Empfohlen:** Auf der höchsten Stufe wird der Einsatz des Patterns *empfohlen*. Die Ergebnisse liegen alle über den ermittelten Grenzwerten. Der Einsatz würde den Code übersichtlicher gestalten und die Wartung bzw. Lesbarkeit vereinfachen. Der Zusatzaufwand ist dadurch in den meisten Fällen gerechtfertigt.

Die Festlegung der Grenzwerte pro Stufe geschieht pro Erkennungsregel. Ein allgemeines Vorgehen kann nicht definiert werden, da die Design Patterns und die verwendeten Metriken zur Erkennung zu unterschiedlich sind. In den meisten Fällen orientiert sich die mittlere Stufe (sinnvoll) am statistischen Mittelpunkt (Median oder Durchschnitt) der ermittelten Metrik-Ergebnisse. In einigen Fällen haben die Erkennungsregeln einen Mindestwert, wie z. B. „mindestens 2 Klassen müssen über den Mediator kommunizieren“. Dieses Vorgehen, eine Untergrenze einzufügen, wird notwendig, wenn man betrachtet, dass ein Mediator mit nur einem Teilnehmer oder eine State-Maschine mit nur einem State fachlich nur wenig Sinn ergibt. Die unteren Grenzwerte für die niedrigste Stufe (möglich) liegen dann bei diesen Mindestwerten.
Die Modelle der einzelnen Design Patterns werden im entsprechenden Abschnitt diskutiert.

4.5 Verwendete Technologien

„Gebt mir einen Hebel, der lang genug, und einen Angelpunkt, der stark genug ist, dann kann ich die Welt mit einer Hand bewegen."

- Archimedes (287 - 212 v. Chr.), griechischer Physiker, Mathematiker und Mechaniker

Dieses Kapitel beschreibt einige grundlegende Themen, die für das Verständnis der Arbeit benötigt werden. Zum einen werden einige Grundlagen kurz beschrieben, zum anderen werden genutzte Technologien erklärt. Aufbauend auf Grundlagen und Technologien wird am Ende des Kapitels das Design Pattern Candidate Detection Tools oder DPCDT beschrieben. Das Ziel des DPCDT ist eine automatisierte statische Code-Analyse auf dem Abstract Syntax Tree (AST), einer Repräsentation des Quellcodes bei der Übersetzung, nach Codemustern, die darauf hinweisen, dass an entsprechender Stelle ein Entwurfsmuster den Code verbessern könnte. Aufbauend auf dem Open Source-Tool PMD wird das DPCDT als eine Erweiterung dafür entwickelt. Am Ende bekommt der Entwickler ein Ergebnis, bestehend aus der Stelle im Quellcode die verbessert werden soll, dem vorgeschlagenen Entwurfsmuster und dem möglichen Grad der Verbesserung.

4.5.1 Pattern Erkennung durch Statische Code-Analyse

Eine Analyse von Quellcode, bei der der Programmcode weder kompiliert noch in irgendeiner anderen Form übersetzt wird, wird statische Code-Analyse genannt (Sneed, Seidl, & Baumgartner, 2010). Ein Vorteil dieser Art von Analyse stellt der hohe Detaillierungsgrad dar. Ein Programm auf Quellcode-Ebene umfasst die größtmögliche Menge an Informationen über Ablauf, Logik und Struktur, die keine Dokumentation oder Anforderungsdokument erreichen kann. Ein Problem stellt die Analysierbarkeit von Quellcode für Menschen dar, weshalb die meisten Informationen extrahiert und aufbereitet werden müssen, bevor eine Analyse beginnen kann. Eine Analyse auf dieser tiefen Ebene wird ermöglicht durch drei Arten von Elementen:

- die Strukturen innerhalb des Quellcodes
- die Code-Elemente zur Implementierung von Logik (bspw. Variablen, Verzweigungsanweisungen)
- die Verbindungen zwischen den Elementen (bspw. zwischen Klassen).

Um diese Code-Informationen zu erfassen, nutzt der in der vorliegenden Arbeit diskutierte Ansatz den AST.

4.5.2 Pattern Erkennung auf dem Abstract Syntax Tree

Der Kompiliervorgang eines Programmes von Quellcode zu Maschinencode besteht aus sechs Phasen (siehe Abbildung 15), welche in zwei Abschnitte eingeteilt werden: Analyse und Synthese. Im ersten Abschnitt der Analyse wird der Quellcode untersucht, um Fehler in der Struktur oder Sprachverwendung zu identifizieren. Darauffolgend wird der Quellcode in Maschinencode übersetzt. Der erste Abschnitt wird auch Front End und der zweite Back End genannt.

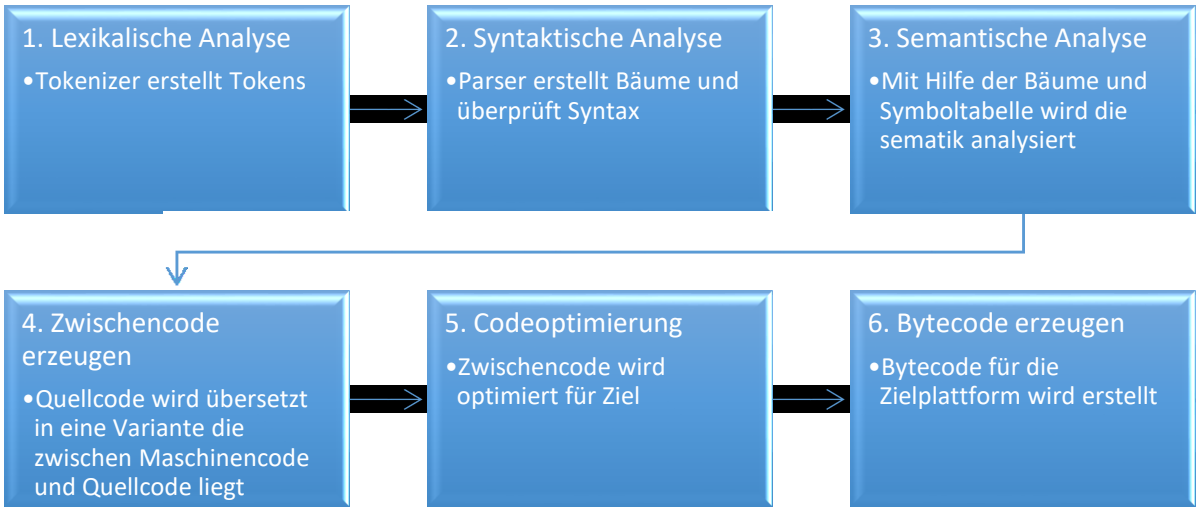


Abbildung 15 Phasen eines Compilers

Das Front End beinhaltet drei der sechs Phasen. In der ersten Phase, der lexikalischen Analyse, wird der Quellcode von einem Tokenizer in einzelne Tokens zerlegt. Jeder Token repräsentiert hierbei einen bestimmten Teil des Quellcodes, wie ein Schlüsselwort oder einen Operanden, sowie dessen Position im Quellcode. All diese Informationen werden in der Symboltabelle gespeichert. Phase zwei umfasst die syntaktische Analyse. In dieser Phase werden vom Parser aus den zuvor gesammelten Tokens Syntax-Bäume erstellt. Die Erkennung von Entwurfsmusterkandidaten findet hauptsächlich auf der Grundlage von Informationen aus einem dieser Syntax-Bäume, dem AST, statt. Die Definition des AST nach Aho et al. (Aho, Lam, Sethi, & Ullman, 2006) verdeutlicht seine Funktion:

„One form, called abstract syntax trees or simply syntax trees, represents the hierarchical syntactic structure of the source program." (Aho, Lam, Sethi, & Ullman, 2006)

Wie aus dem Namen schon zu schließen ist, repräsentiert der AST eine Baumstruktur, bei der jeder Knotentyp einen syntaktischen Teil der zugrundeliegenden Programmiersprache abbildet.

AST vs. CST

Neben dem AST existiert noch eine weitere Baumstruktur, der Concrete Syntax Tree (CST) oder auch Parse Tree genannt (Aho, Lam, Sethi, & Ullman, 2006). Aho et al. beschreiben den CST wie folgt.

„A parse tree pictorially shows how the start symbol of a grammar derives a string in the language." (Aho, Lam, Sethi, & Ullman, 2006)

Ein CST stellt die Grammatik eines Quellcodes als Baumstruktur dar. Dabei repräsentiert jeder Knoten ein Symbol der Sprache. Die Wurzel stellt ein nichtterminales Startsymbol dar, d.h. sie werden während der Ableitung durch Terminal-Symbole ersetzt. Diese Terminal-Symbole stellen die Endknoten eines CST dar. Blätter zwischen dem Wurzelknoten und dem Endknoten sind immer Nichtterminale.

Gemäß Aho et al. unterscheiden sich CST und AST wie folgt.

"However, in the syntax tree, interior nodes represent programming constructs while in the parse tree, the interior nodes represent nonterminals." (Aho, Lam, Sethi, & Ullman, 2006)

Wie in der Definition dargelegt, liegt der Unterschied im Inhalt der Knoten. Im AST repräsentieren sie Teile der Quellcodes wie Variablennamen oder Key-Wörter. Knoten vom CST repräsentieren nichtterminale Symbole (z. B. Expression oder Symbol) und somit die Sprache selbst. Der CST wird verwendet um die Korrektheit des Quellcodes mit der Sprache zu überprüfen und der AST, um die Struktur des Quellcodes zu analysieren.

Die semantische Analyse stellt die dritte Phase im Kompiliervorgang dar. In dieser Phase wird der AST zusammen mit der Symboltabelle verwendet, um die Semantik des Quellcodes gegenüber den festgelegten Sprachkonventionen zu überprüfen, etwa, ob für genutzte Operanden die richtigen Operatoren verwendet wurden, also z. B., ob die Case-Bedingung eines Switches auch dem Typ der Switch-Bedingung entspricht. Die in dieser Arbeit verwendeten AST-Beispiele basieren auf der Java-Language-Spezifikation (JLS) 4.0 (Oracle, 2015), auf der auch Java 7 beruht. Sie beinhaltet die Definition von 75 verschiedenen AST-Knoten.

Ein Zwischencode wird in Phase vier erstellt. Dieses Derivat ist schon sehr nahe dem eigentlichen Maschinecode oder Bytecodes. Je nach Compiler und Sprache kann ein Zwischencode verschiedene Formen haben, sowohl eine Baumstruktur als auch eine Abwandlung vom Maschinencode, die später von einem Interpreter oder einer virtuellen Maschine in Bytecode umgewandelt wird.

Der Zwischencode wird in Phase fünf verbessert. Dabei verbessert der Compiler den Quellcode an bestimmten Stellen, um z. B. um den Programmablauf zu beschleunigen. So kann es sein, dass Datentypen ausgetauscht oder Teile von If-Anweisungen zur Verbesserung verändert werden. In der letzten Phase wird der eigentliche Bytecode für die entsprechende Zielplattform erzeugt.

Die Verwendung des AST stellt die ideale Ausgangslage für die Analyse und Erkennung von Entwurfsmusterkandidaten im Quellcode dar. Im AST sind alle wichtigen Informationen über die Struktur und das Design eines Quellcodes in kompakter Form dargestellt. Dazu gehören Informationen über Attribute, Klassen, Methoden, Packages, Methodenaufrufe etc., welche durch verschiedene Knotenarten im AST beschrieben werden. Gerade die Aufteilungen in verschiedenste

AST-Knoten, wie z. B. Kommentare, Switches, Cases u. v. m., begünstigen eine Erkennung, weil durch seine Anordnung als Baum bestimmte Knotentypen schnell gefunden werden können.

Zusätzlich bietet die Verwendung einer Baumstruktur den Vorteil, dass eine Hierarchie zwischen den Knoten existiert. In einem AST stehen Klassenknoten immer über den Methoden, was darauf zurückzuführen ist, dass eine Methode immer zu einer Klasse gehört. Des Weiteren gehört eine Variable (nicht zu verwechseln mit dem Attribut das zu Klassen gehört) immer zu einer Methode.

In Java erstellt der Compiler für jede Klasse einen separaten AST, damit werden die Syntax und Struktur aus Sicht der gerade aktuellen Klasse geprüft. Die folgende Abbildung 16 zeigt den Teilabschnitt der Klassendefinition der Klasse Rental.java. Abbildung 17 zeigt die AST-Repräsentation eines Attributes in der gleichen Klasse. Diese Klasse ist Teil von Martin Fowlers VideoStore-Beispiel. (Fowler, 2002).

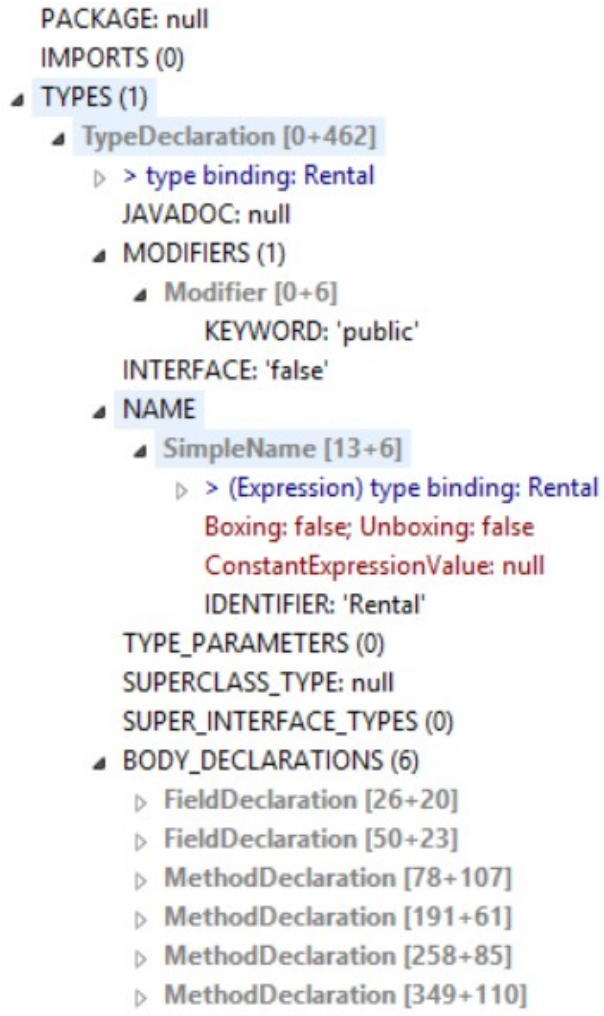


Abbildung 16 Teil-AST der Klassen-Deklaration von Rental.java

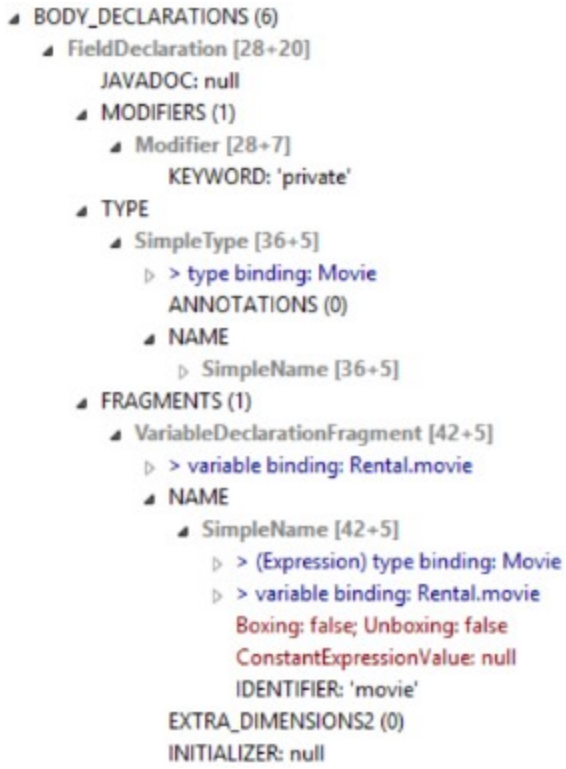


Abbildung 17 Teil-AST eines Attributes in der Klasse Rental.java

Das Beispiel verdeutlicht, dass der AST nur aus Sicht der Klasse Rental erstellt wurde. Es existiert nur eine Klassen-Deklaration mit dem Knoten *TypeDeclaration* (siehe Abbildung 16)). Zuerst kommt der Knoten Package, wo zu sehen ist, dass diese Klasse zu keinem Package gehört. In Java gibt es keinen eigenen AST für Packages, in dem alle Klassen enthalten sind, sondern nur der AST der

Klasse weiß, zu welchem Package er gehört. Der Package-Name wird pro Klasse gespeichert. Darauf folgt im Beispiel AST eine Auflistung der importierten Packages. Auch hier gibt es im Beispiel keinen Import. Daraufhin folgt die eigentliche Klassen-Deklaration unter *TypeDeclaration*. Zuerst kommen Informationen über die Klasse wie Sichtbarkeit (Modifiers), ob ein Interface und welches Interface verwendet wurde, welche vererbte Klasse verwendet wurde, und der Klassenname der vererbten Klasse. Alle Namen von Methoden, Klassen etc. sind immer unter dem Knotentypen *SimpleName* zu finden. Unter *BODY_DECLARATIONS* sind alle Attribute und Methoden der Klasse aufgelistet. *Abbildung 17* zeigt beispielhaft die Deklaration des Attributes Movie mit dem Type Movie, einer anderen Klasse. Der Type des Attributes wird unter dem Knoten-Type abgelegt. Wie zu sehen ist, gibt es nur einen *SimpleName* mit dem Namen Movie, aber keine klare Verbindung, wo die Klasse Movie zu finden ist, welche den Type definiert. Somit weiß die Klasse Rental zwar, dass sie ein Attribut von Type Movie besitzt, aber mehr ist von der Klasse Movie nicht aus dem AST zu lesen: Unter dem Knoten *FRAGEMENTS* ist der Name und, wenn genutzt, der Initialwert des Attributes gespeichert. Knoten vom Type *MethodDeclaration* beinhalten die einzelnen Methoden einer Klasse. Es gibt noch Knoten vom Type *MethodInvocation*, wie in Abbildung 18 zu sehen. Dieser beinhaltet unter *EXPRESSION* das Objekt, welches die aufzurufende Methode beinhaltet, unter *NAME* den Namen der Methode und unter ARGUMENTS die zu übergebenden Parameter.

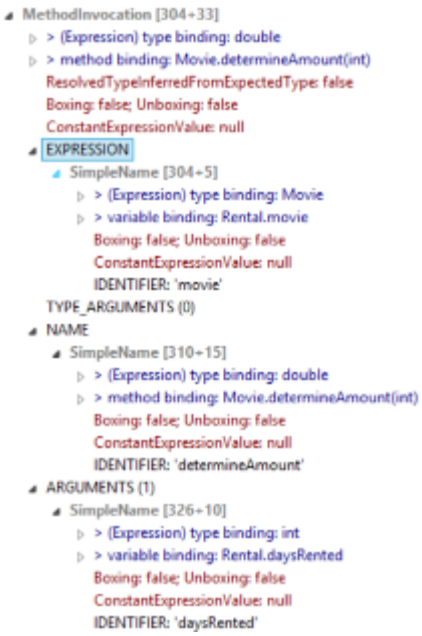


Abbildung 18 AST Knoten MethodInvocation

Schlussfolgernd zeigt jeder AST nur die Sicht auf das Programm, basierend auf der Perspektive der aktuell generierten Klasse. Bei einigen Betrachtungen kann dies zu Missverständnissen führen, weil z. B. eine Klasse einmal ein Objekt-Knoten ist und ein anderes Mal ein Klassen-Knoten. Dies führt im Zuge der Analyse eines Programms dazu, dass keine übergreifenden Programmknoten oder gar Package-Knoten existieren. Außerdem fehlen Information darüber, wie Klassen außerhalb der erstellten Sicht mit der Klasse kollaborieren. Dieses Problem kann durch eine Kombination der verschiedenen AST-Sichten gelöst werden. Mit Hilfe der Information, welches Objekt bei einem Methodenaufruf genutzt wird, kann festgestellt werden, unter welcher Variablen oder welchem Attribut dieses Objekt initialisiert wurde. Aus dem Attribut bzw. der Variable kann der Klassenname gelesen werden, was zu Klasse führt. All diese Informationen sind im AST gespeichert, sie müssen nur entsprechend verbunden werden.

4.5.3 Verwendung von stark verbunden Komponenten (SVK)

Zur Identifikation von komplexen Verbindungsnetzwerken zwischen Quellcode-Elementen wie Klassen oder Packages, welche bei einigen Entwurfsmusterkandidaten auftreten, wird eine Methode aus der Graphentheorie angewandt. Bei der Erkennung werden die Verbindungen zwischen den Klassen innerhalb eines Packages in einen Graphen transformiert. Die Klassen stellen hierbei die einzelnen Knoten dar. Aus den Methodenaufrufen von einer Klasse zu einer anderen oder aus einem Attribut einer Klasse, die den Type einer anderen Klasse besitzen (z. B. Klasse A besitzt ein Attribut private B objB) und somit eine Verbindung zwischen zwei Klassen herstellen, ergeben sich die Kanten des Graphen. Ausgenommen sind Attribute, die einen basic type besitzen, wie z. B. Integer oder Float. Auf diesen Graphen wird der Algorithmus zur Erkennung von stark verbundenen Komponenten (SVK) angewandt. SVK werden verwendet, um komplexe Strukturen oder Verbindungen zwischen verschiedenen Elementen wie Klassen oder Packages zu identifizieren. Gerade komplexes Kommunikationsverhalten, wo jede Klasse mit jeder anderen kommuniziert, kann ein Indiz für schlechtes Design sein, was durch ein Entwurfsmuster wie Mediator (vgl. Kapitel 5.4) oder Facade (vgl. Kapitel 5.3) verbessert werden kann. Die Definition einer SVK lautet:

„A directed graph is called strongly connected if there is a path in each direction between each pair of vertices of the graph. In a directed graph G that may not itself be strongly connected, a pair of vertices u and v are said to be strongly connected to each other if there is a path in each direction between them." (Wolsey & Nemhauser, 2014)

Die folgende Abbildung zeigt einen gerichteten Graphen mit vier SVKs.

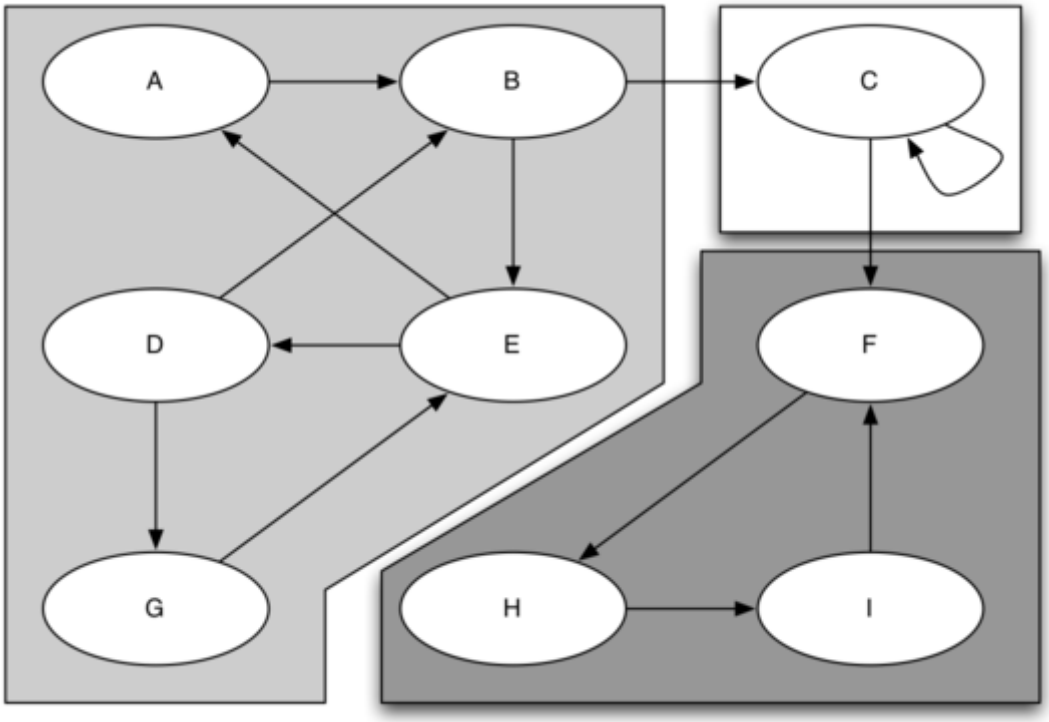


Abbildung 19 Gerichteter Graph mit SVKs[4]

[4] http://interactivepython.org/runestone/static/pythonds/_images/scc1.png

4.6 PMD, die Grundlage der Code Analyse

Als technologisches Fundament für das Erkennen von allen Entwurfsmusterkandidaten dient das Open Source-Quellcode-Analyse-Tool PMD (Louridas, 2006), welches auf Java basiert und dessen Toolname PMD (PMD, 2015) kein Akronym ist. PMD 4.3 erlaubt die Analyse von Java-Quellcode zur Identifikation der folgenden potenziellen Codefehler- und Strukturprobleme-Kategorien. Das Tool ist in der Lage, mehrere Sprachen zu analysieren. Neben Java unterstützt es noch *JSP*, *ecmascript*, *jsp*, *xml* und *xsl*. Bekannte Java-Codeprobleme sind in Tabelle 19 näher beschrieben.

Tabelle 19: Auswahl Möglicher Codefehler und Strukturprobleme, die PMD erkennen kann

Kategorie	Beschreibung
Dead Code	Quellcode, der nie ausgeführt wird oder Variablen, die nie gesetzt werden.
Cloned Code	Identische Zeilen von Quellcode, die mehrmals im Programm verwendet werden.
Komplizierte Ausdrücke	IF-/Switch-Anweisungen oder Schleifenbedingungen, die eine gewisse Komplexität überschreiten.
Bugs	Verschiedene Code-Strukturen oder -Konstellationen, die zu einem Fehler führen können, wie bspw. leere Catch-Blöcke.
Suboptimaler Code	Code-Strukturen, die zwar verwendet werden, aber verschwenderisch mit Ressourcen umgehen.

Neben diesem Problem ist PMD noch in der Lage, viele weitere Probleme zu identifizieren. Eine vollständige Liste der aktuellen PMD-Regeln für Java und andere Sprachen findet sich auf der PMD-Webseite (PMD, 2015). Zur Erweiterung der Regeln stellt PMD eine flexible Regeldefinition zur Verfügung, die jeden Anwender in die Lage versetzt, neue Erkennungsarten zu entwickeln oder bestehende zu erweitern. Grundlage jeder Regelerweiterung ist das von PMD implementierte Visitor Pattern für den AST: Mit diesem Pattern wird der Entwickler in die Lage versetzt, spezifische Knotentypen eines Programmcodes zu besuchen und zu analysieren. Es existieren zwei Varianten, um Regeln zu definieren.

1. **XPath Rules**: Diese Variante der Regelerweiterungen verwendet die Abfragesprache XPath (W3C, 2015). Die Abfragesprache wurde entwickelt um Knoten in einem XML-Dokument zu suchen oder direkt anzusteuern. In PMD wird der AST als XML dargestellt, was den Einsatz von XPath erlaubt. Der verwendete XPath-Ausdruck baut auf dem Knoten des AST-Baumes auf. Mit Ausdrücken wie *//FieldDeclaration* ist ein Entwickler in der Lage bestimmt Strukturen im Code zu identifizieren. Als Beispiel: *//MethodDeclaration//LocalVariableDeclaration* gibt alle lokalen Variablen-Deklarationen in einem Java-Programm aus. Es gibt noch weitere Möglichkeiten, den Detailgrad der XPath-Suche zu erhöhen. So können bestimme Namen gesucht werden oder auch Mengenangaben von Knoten, die sich in einem Knoten befinden. Diese Möglichkeit der Erweiterung eignet sich für einfache Regeln, die nach Strukturen im Code suchen. In PMD gibt es seit der Version 1.02 eine Regel *IfStmtsMustUseBraces* zum Auffinden von If-Anweisungen, die ohne Klammern geschrieben wurden. Sie ist in XPath geschrieben und hat folgenden Aufbau:

    - //IfStatement[count(*) < 3][not(Statement/Block)]

Zuerst wird über den XPath jeder AST-Knoten vom Type *ForStatement* angesteuert. Passiert der Visitor einen solchen Knoten, wird ausgewertet, ob das If-Statement weniger als 3 Zeilen besitzt, da bei mehr Zeilen eine Klammer gesetzt werden muss, damit nicht nur die erste Zeile nach der If-Anweisung zur Anweisung gehört. Nun wird geprüft, ob dieser Knoten Folgeknoten vom Type Statement oder Block hat, welche durch eine Klammer erzeugt werden. Ist dies nicht der Fall, so meldet PMD einen Fehler. Es besteht auch die Möglichkeit, mehrere dieser Regeln zu einem Regelset zusammenzuführen, was zu einer neuen Regel führt.

2. **AbstractJavaRule** In der zweiten Variante werden die gewünschten Regeln direkt in Java-Code geschrieben. Dazu muss die Klasse, welche die neue Regel beinhaltet, von der PMD-eigenen Klasse *AbstractJavaRule* erben. Diese abstrakte Klasse dient als Schnittstelle und bietet dem Programmierer verschiedene Methoden, die es ermöglichen, Plug-Ins zu implementieren und somit die PMD-internen Schnittstellen zum Visitor direkt anzusprechen. Jeder Knotentype von Java besitzt im Visitor eine eigene Methode, die vom Entwickler überschrieben werden kann. So wird die Methode *visit(ASTMethodDeclaration method, Object ruleCtx)* vom Visitor aufgerufen, wenn dieser einen entsprechenden Knoten passiert. Auf die Weise wird die passende Regel ausgeführt. Durch die Verwendung von Java zur Implementierung von Regeln können komplexe Erkennungsarten implementiert werden.

5 Entwicklung von Erkennungsregeln

„Man sieht oft etwas hundert Mal, tausend Mal, ehe man es zum allerersten Mal wirklich sieht."

- Christian Morgenstern (1871–1914), dt. Schriftsteller

Im Zuge dieser Arbeit wurden die 23 bekannten Patterns der Gang of Four (GoF), die auf Gammas (Gamma, Helm, Johnson, & Vlissides, 1994) Veröffentlichung basieren, untersucht. Wie zuvor erwähnt, wurden sechs Patterns (Builder, Decorator, Facade, Mediator, State, Strategy) ausgewählt, um Kandidaten-Erkennungsregeln für sie festzulegen. Neben den Erkennungsregeln werden in den folgenden Abschnitt noch der Prozess zur Festlegung der Grenzwerte und das allgemeine Vorgehen zur Erstellung von Regeln für die einzelnen Patterns beschrieben.

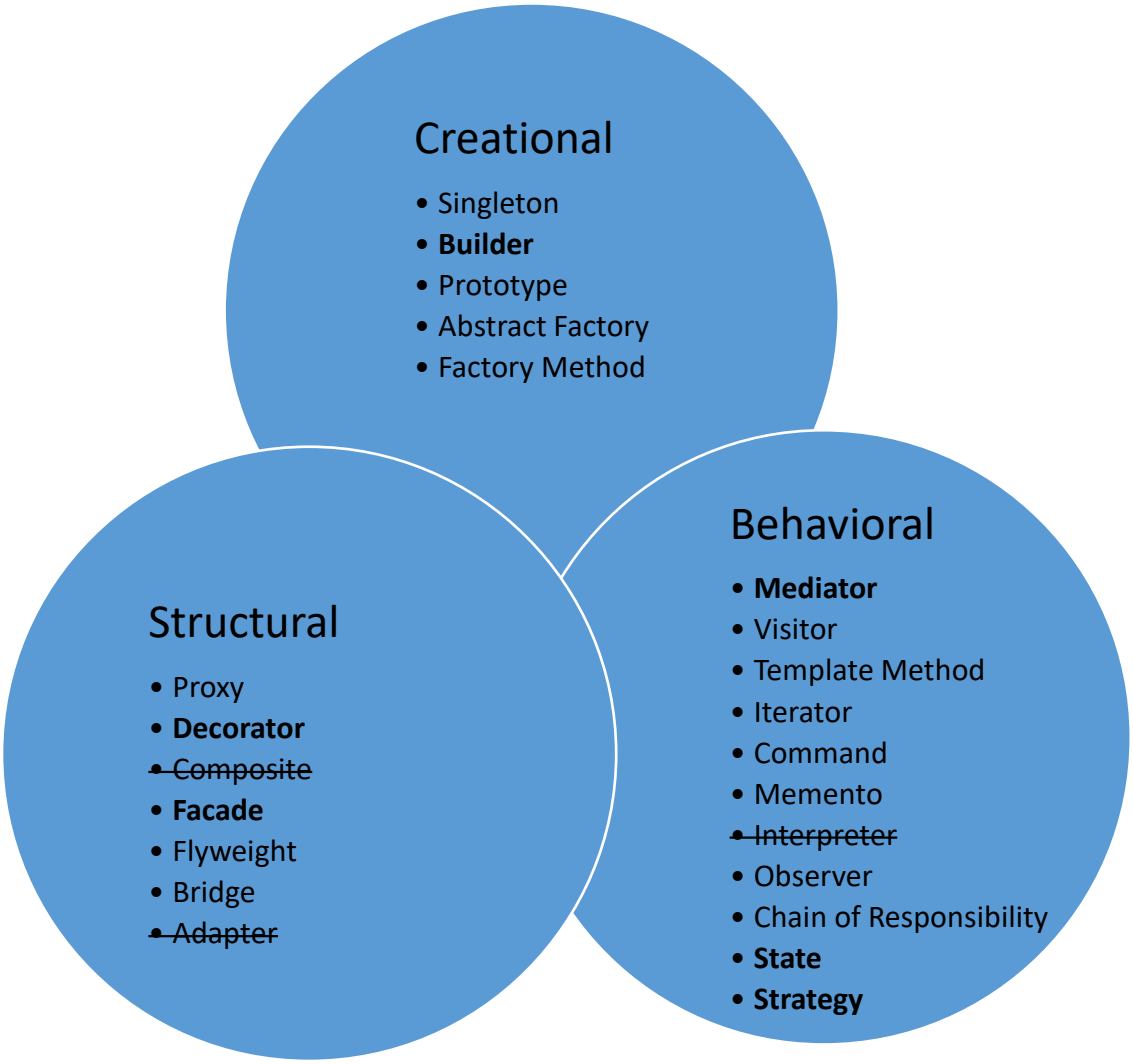


Abbildung 20: Übersicht der nach Gamma beschriebenen Entwurfsmuster (fett = Erkennungsregel definiert, durchgestrichen = Ausschluss, normal = Lösungsskizze)

Drei Design Patterns – Adapter, Composite und Interpreter – wurden nach einer ersten Analyse ausgeschlossen, weil für diese keine Erkennung im Quellcode möglich ist, sondern eine bewusste Entscheidung vom Entwickler notwendig ist, vgl. Kapitel 5.8.

5.1 Builder-Pattern

Nachfolgend wird anhand eines Beispiels aus dem Programm ArgoUML von Tigris (Version 0.34) diskutiert, wie ein Builder-Pattern-Kandidat ausgewählt und bewertet wird. Konkret geht es bei diesem Beispiel um die Klasse *UMLModelElementListModel*, welche unter anderem die Aufgabe hat, bis zu elf verschiedene Objektvarianten zu erzeugen. Bei diesen Variationen handelt es sich um Listen von Model-Objekten, die unterschiedliche UML-Diagramme beinhalten können. Neben den Modellen kann eine Liste noch bis zu drei Events beinhalten, die zu verschiedenen Aktionen auf der Liste ausgeführt werden, z. B. Hinzufügen oder Löschen von Modellen. Zuletzt kann noch das Löschverhalten der Liste angepasst werden. Die Verwendung eines Builder Pattern erlaubt die einfache Erstellung dieser komplexen Varianten, indem für jede Variante ein konkreter Erbauer erstellt wird. Das Hinzufügen weiter Varianten wird so vereinfacht und der Direktor muss nicht mehr wissen, welche Varianten es gibt.

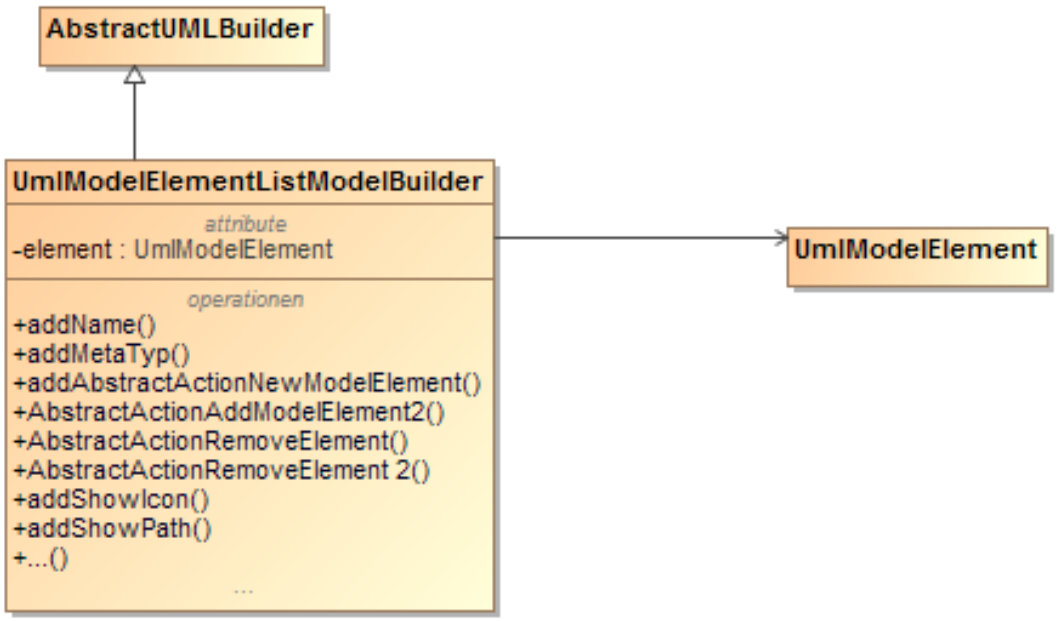


Abbildung 21 Mögliche Implementierung des UMLModelElementListModelBuilders

5.1.1 Erkennungsregel

Im Falle des Builder-Patterns wurde die folgende Regel zur Erkennung von Kandidaten abgeleitet.

R1.1 *Eine Klasse implementiert eine große Anzahl an Konstruktoren.*

Die Basis für diese Regel liegt in der Identifikation des Anti-Patterns Teleskop-Konstruktor im Quellcode (Freeman A. , 2015). Dieses Anti-Pattern zeichnet sich dadurch aus, dass eine Klasse eine Vielzahl an Konstruktoren oder Konstruktoren eine große Menge an Übergabeparametern besitzen. Bei der Analyse wird jede Klasse auf die Anzahl ihrer Konstruktoren hin untersucht. Überschreitet die Anzahl an Konstruktoren den definierten Grenzwert (Tabelle 20), so ist dies ein Indiz für einen Builder-Kandidaten. Jeder Konstruktor stellt hierbei einen Weg dar, das gleiche Objekt in verschiedenen Repräsentationen zu erstellen. Die Verwendung eines Builder-Patterns kann das komplexe System zur Erschaffung von Objekten verschiedenster Art vereinfachen. Außerdem erlaubt der Einsatz des Patterns neue Wege zur Erschaffung, um flexibel zu implementieren.

Tabelle 20 Metriken und Grenzwerte der Builder Regel

Metrik	Beschreibung	Grenzwert
M1.1 Anzahl Konstruktoren	Eine Klasse muss über eine gewisse Anzahl an Konstruktoren verfügen.	≥3

5.1.2 Grenzwerte

Aus den zur Verfügung stehenden Merobase-Informationen wurden für das Builder-Pattern die folgenden Grenzwerte ermittelt, gemäß des in Kapitel 4.3 definierten Vorgehens.

Die Suche basierte auf folgendem String:

*I*Builder lang:java type:interface (protocol:svn OR protocol:CVS)*

Das Vorgehen basiert auf der Annahme, dass alle Interfaces in Java sich an die bekannten Namenskonventionen halten und mit dem Buchstaben ‚I' beginnen. Außerdem wird davon ausgegangen, dass Entwickler das Wort *Builder* im Interfacenamen verwendet, um den Einsatz des Patterns zu kennzeichnen.

Festlegung der Grenzwerte

Durch die Verwendung des Suchstrings lieferte Merobase 354 mögliche Builder-Implementierungen. Aus dieser Menge an Ergebnissen wurden 50 Builder aus 46 verschiedenen Open Source-Projekten ausgewählt. (vgl. Kapitel 4.2) Diese Implementierungen entsprechen der GoF-Definition. Insgesamt wurden 164 konkrete Builder gefunden. Einige Beispiele implementieren das Builder-Interface in einem weiteren Interface, wie eine Art Zwischenschicht. In solchen Fällen wurden immer die Klassen gezählt, die auf der untersten Ebene eine Verbindung mit dem Builder-Interface besitzen, und die Zwischenschichten wurden ignoriert.

Wie in Tabelle 20 gezeigt, verfügt die Erkennungsregel für Builder über nur eine Metrik. Diese Metrik beschreibt die Anzahl an Konstruktoren, die eine Klasse besitzt. Der dazugehörige Grenzwert definiert, ab wann die Anzahl an Konstruktoren ausreicht, um ein Builder Pattern einzusetzen. Um diesen Grenzwert zu definieren, wurden in den oben gefundenen Beispielimplementierungen von Buildern die Anzahl an konkreten Buildern vermessen. Aus der Anzahl konkreter Builder wird dann der Grenzwert ermittelt. Die Verteilung der gefundenen Builder wird in Abbildung 22 dargestellt. Die x-Achse beschreibt, wie viele konkrete Builder ein gefundenes Entwurfsmuster besitzt, und die y-Achse, wie oft eine solche Konstellation gefunden wurde.

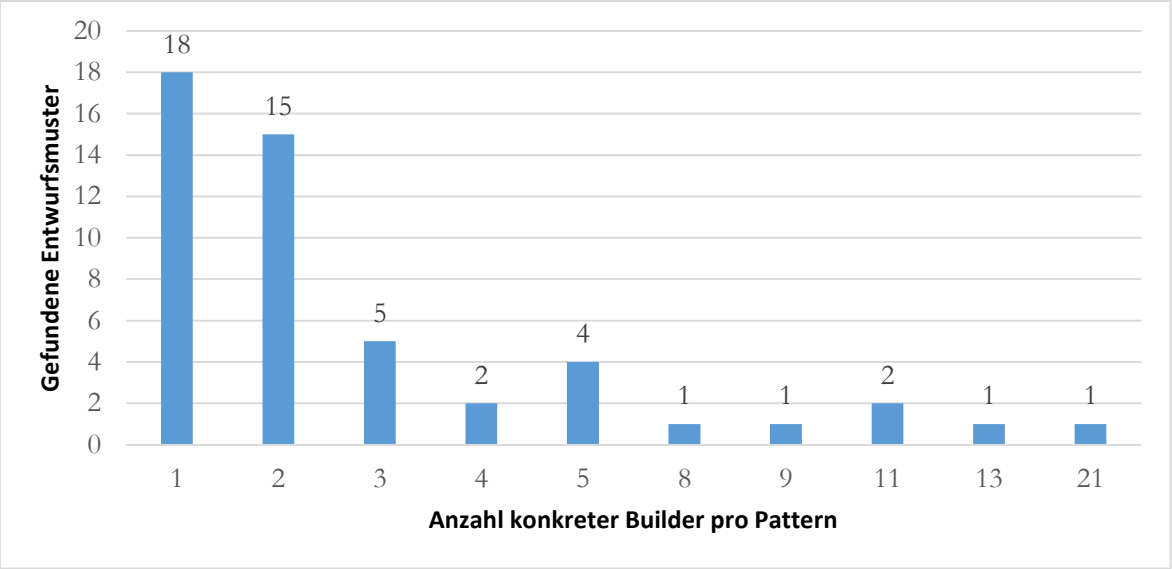


Abbildung 22: Anzahl und Menge der in Merobase gefundenen konkreten Builder

Ein Drittel der Implementierungen besitzt nur je einen konkreten Builder. Das kann auf zwei Gründen beruhen.

1. Die Programme implementieren ein Modul von einem anderen Projekt, welches das Builder-Interface vorgibt.

2. Das Programm selbst soll erweiterungsfähig sein.

Ebenfalls ein Drittel der Implementierungen besitzt je zwei konkrete Builder. Das letzte Drittel der Ergebnisse besitzt je drei bis 21 konkreten Builder. Tabelle 21 zeigt die statistische Verteilung der Ergebnisse.

Tabelle 21: Statistische Verteilung der gefundenen konkreten Builder

	Builder
Min.	1,00
1. Quartil	1,00
Median	2,00
Mittelwert	3,28
3. Quartil	3,00
Max.	21,00

Die Analyse der Ergebnisse ergibt ein interessantes Bild. Mit einem Mittelwert von 3,28 und dem 3. Quartil von 3,00, das genau auf diesem Mittelwert liegt, zeigt sich, dass nicht viele konkrete Builder pro Pattern implementiert werden. Von einem Grenzwert von nur einem Konstruktor ist logischerweise ganz abzusehen, da nahezu jede bekannte Klasse dann in die Kandidatenmenge fallen würde. Einen Grenzwert festzulegen, bei dem ab zwei Konstruktoren ein Builder Pattern möglich wäre, würde die Menge an Kandidaten sehr hoch ausfallen lassen. Eine erste mögliche Menge an Kandidaten für die unterste Stufe der Erkennung sollte somit der Mittelwert bilden. In Verbindung mit dem 3. Quartil bilden die beiden Werte eine sehr gute Grundlage, um eine sinnvolle Ergebnismenge zu erhalten, ab wann ein Entwickler überlegen sollte, ein Builder Pattern

einzusetzen. Der Mittelwert von 3,28 und das 3. Quartil von 3 werden als Ausgangspunkt für die Grenzwerte verwendet.

Darauf aufbauend folgt, dass ein Kandidat mindestens 3 Konstruktoren besitzen sollte, damit ein Builder *möglich* erscheint und somit dem Entwickler geraten wird, die Stelle genauer zu untersuchen. Ab vier Konstruktoren ist es *sinnvoll*, über den Einsatz des Patterns nachzudenken, wobei dies von Fall zu Fall entschieden werden muss. Ab fünf wird im Modell ein Builder-Pattern *empfohlen*. Die nachfolgende Tabelle 22 fasst die Stufen zusammen. Werden aus dem Ergebnis der Merobase-Analysen die Builder mit einem und zwei konkreten Buildern weggelassen, weil diese geringe Menge an konkreten Buildern das Entwurfsmuster nicht als sinnvoll erscheinen lässt, so fallen fast 65% der gefundenen Builder Patterns auf die Menge mit 3 bis 5 konkreten Buildern. Dieses Ergebnis spiegeln die Erkennungsstufen wider.

Tabelle 22: Erkennungsmodell mit Grenzwerten pro Erkennungsstufe

Builder	Erkennungsstufe
≥3	Möglich
≥4	Sinnvoll
≥5	Empfohlen

5.1.3 Beispielerkennung

Dieser Abschnitt beschreibt die einzelnen Schritte zur Erkennung eines Kandidaten im AST, der mit einem Builder-Pattern verbessert werden kann. Der Prozess (siehe Abbildung 23) beinhaltet drei Schritte.

- 1. Extraktion der Daten aus dem AST
- 2. Zählung der Konstruktoren
- 3. Abgleich der Daten mit den Grenzwerten

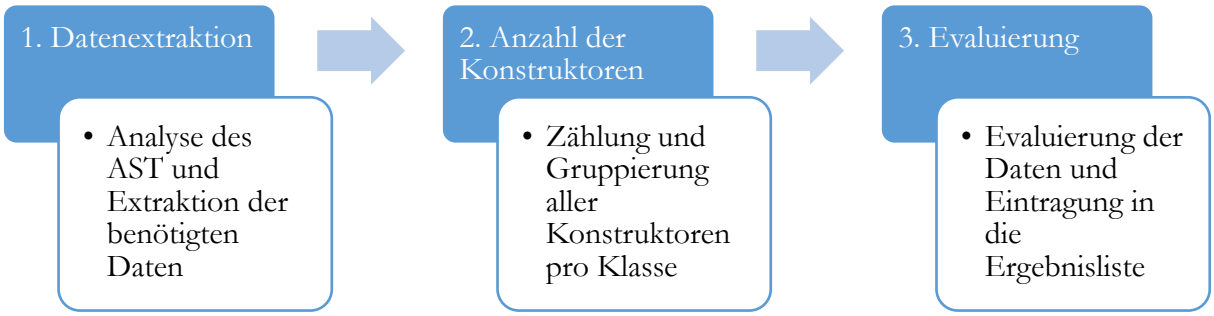


Abbildung 23: Detailprozess zur Erkennung von Builder-Kandidaten

Nachfolgend wird anhand des Beispiels vom Anfang des Kapitels die Erkennung im Detail diskutiert. Der gefundene Kandidat befindet sich, wie zuvor erwähnt, in der Klasse *UMLModelElementListModel* des Programmes ArgoUML (CollabNet, Inc., 2015). Diese Klasse besitzt elf unterschiedliche Konstruktoren, wovon ein Konstruktor mehr als vier Parameter anfordert.

Der 1. Schritt „Extraktion der Daten“ arbeitet vorrangig mit dem AST-Knoten *ASTConstructorDeclaration*. Der Informationsextraktionsprozess aus dem AST wird in Kapitel 4.5.2 näher beschrieben.

Der Knoten *ASTConstructorDeclaration* beinhaltet weitgehend die Informationen des Knotens *ASTMethodDeclaration*. Er identifiziert jedoch nur Konstruktoren. Wie bei anderen Extraktionen, werden zuerst die allgemeinen Positionsinformationen wie zugehöriger Package-Name und Klassennamen gesammelt. Zusätzlich werden alle Informationen der Übergabeparameter (Name, Typ, Art) aus dem AST extrahiert. Mit diesen Informationen wird für jeden Konstruktor ein Eintrag in der Datenbank erstellt. Die nachfolgende Tabelle 23 zeigt einen Ausschnitt aus der Methoden-Tabelle mit allen Konstruktoren der Klasse *UMLModelElementListModel*. Spalte 1 zeigt die ID der Methode – eine fortlaufende Nummer, die automatisch vergeben wird. Der Name der der zugehörigen Klasse in Spalte 2. Spalte 3 zeigt die Anzahl der Übergabeparameter. In der Tabelle 23 wird klar, dass die Klasse *UMLModelElementListModel* 11 Konstruktoren besitzt die unterschiedlichste Paramater aufweisen und somit ein Objekt auf unterschiedliche Weise zusammenbaut.

Tabelle 23: Alle Konstruktoren der Klasse mit ID 1 aus der Methoden-Tabelle

Konstruktor ID	Klasse	Anzahl Parameter
2	*UMLModelElementListModel*	0
3	*UMLModelElementListModel*	1
4	*UMLModelElementListModel*	3
5	*UMLModelElementListModel*	4
6	*UMLModelElementListModel*	2
7	*UMLModelElementListModel*	5
8	*UMLModelElementListModel*	3
9	*UMLModelElementListModel*	3
10	*UMLModelElementListModel*	4
11	*UMLModelElementListModel*	4
12	*UMLModelElementListModel*	3

Für die Regel R1.1 wird die Anzahl der Konstruktoren pro Klasse benötigt. Es gibt dabei keine Unterscheidung zwischen öffentlichen und privaten Konstruktoren. Die hohe Anzahl an Konstruktoren der Klasse ist ein Indiz darauf, dass die Klasse ein Objekt von sich selbst in unterschiedlichen Arten zusammenbauen kann.

Evaluierung

Basierend auf den Informationen handelt es sich um einen möglichen Kandidaten für ein Builder-Pattern. Durch seine 11 Konstruktoren bekommt der Kandidat die Stufe *empfohlen*. In der nachfolgenden Tabelle 24 werden alle Konstruktoren und ihre Parameter des Kandidaten kurz dargestellt. Eine mögliche Implementierung mit dem Builder Pattern illustriert Abbildung 21 auf Seite - 59 -.

Tabelle 24: Methodenköpfe der Konstruktoren

Konstruktoren: UMLModelElementListModel
[]
[String name]
[final String name, final boolean showIcon, final boolean showPath]
[final String name, final boolean showIcon, final boolean showPath, final Object metaType]
[String name, Object theMetaType]
[final String name, final Object theMetaType, final AbstractActionAddModelElement2 addAction, final AbstractActionNewModelElement newAction, final AbstractActionRemoveElement removeAction]
[final String name, final Object theMetaType, final AbstractActionAddModelElement2 addAction]
[final String name, final Object theMetaType, final AbstractActionNewModelElement newAction]
[final String name, final Object theMetaType, final AbstractActionAddModelElement2 addAction, final AbstractActionRemoveElement removeAction]
[final String name, final Object theMetaType, final AbstractActionAddModelElement2 addAction, final AbstractActionNewModelElement newAction]
[String name, Object theMetaType, boolean reverseTheDropConnection]

5.2 Decorator-Pattern

Wie in den Kapiteln zuvor, wird auch für das Decorator Pattern ein Beispiel zur Erklärung verwendet. Für die Detaildiskussion zur Identifikation eines Decorator-Pattern-Kandidaten wurde ein Beispiel aus dem Open Source-Project *ArgoUML* (CollabNet, Inc., 2015) (Version 0.34) ausgewählt. In diesem Projekt wurde ein Kandidat identifiziert, bei dem 86 Klassen, direkt oder indirekt, von der Hauptklasse *AbstractArgoJPanel* erben. Hauptaufgabe dieser Klasse ist das Bereitstellen von allgemeinen Funktionen für die von ihr abgeleiteten Panels, die in ArgoUML angezeigt werden können. Die mögliche Verbesserung des genannten Kandidaten als Decorator-Entwurfsmuster wird in Abbildung 24 illustriert.

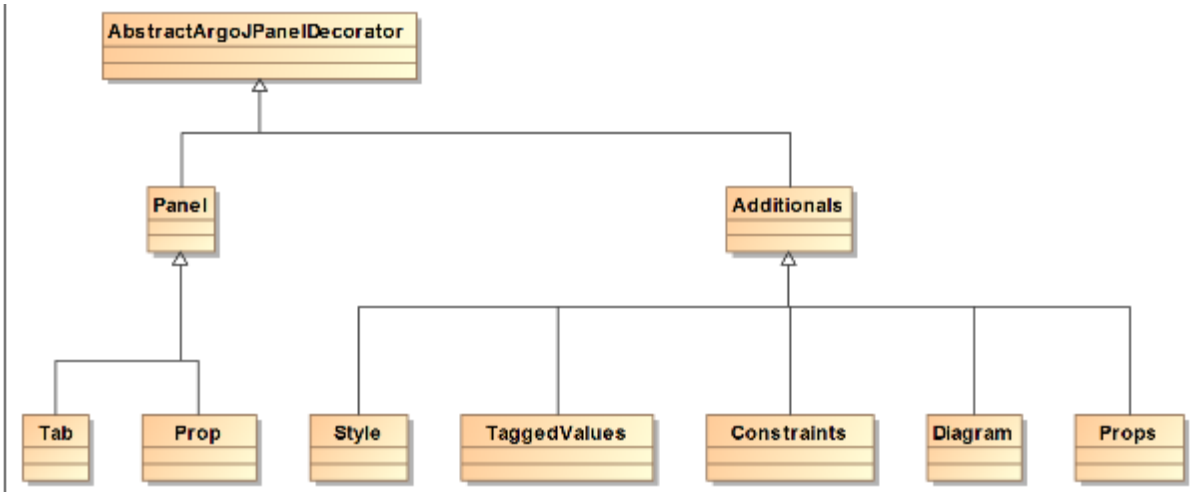


Abbildung 24 Mögliche Verbesserung AbstractArgoJPanel Klasse als Decorator-Pattern

Die im Folgenden beschriebene Decorator-Erkennungsregel stellte sich während der Evalation mit OpenSource-Systemen als nicht ideal für dieses Pattern heraus. Es wurden zwar alle Kriterien für die Erkennung erfüllt, jedoch könnten, je nach Kandidat auch andere Patterns die gefundene Stelle optimieren bzw. wäre der Decorator teilweise keine sinnvolle Optimierung. Eine Erkennung nur über große Klassenhierarchien allein, liefert nicht genug Informationen um klar auf einen Decorator-Kandidaten zu schließen. Ohne Veränderung an der Regel ist eine klare Identifikation nur durch manuelle Überprüfung oder durch Interviews (siehe 7.4) möglich. Eine Erweiterung der Regel war im Rahmen dieser Arbeit nicht mehr möglich, erste Verbesserungsideen werden aber am Ende dieses Kapitels diskutiert.

5.2.1 Erkennungsregel

Basierend auf der Definition und der Verwendung des Design Patterns wurde die folgende Erkennungsregel definiert.

R1.2 Es gibt eine Klassenhierarchie mit vielen Subklassen auf der ersten Ebene.

Die Regel zur Erkennung eines Decorator baut auf der Annahme auf, dass eine Grundklasse immer wieder vererbt wird, um neues Verhalten hinzuzufügen. Dementsprechend wird nur die erste Vererbungsebene betrachtet. Dies entspricht dem Verwendungszweck des Decorator-Design Patterns. Im Vergleich zur Verwendung des Patterns werden aber keine Subklassen an die Grundklasse angehängt, sondern für jedes neue Verhalten eine neue Subklasse erstellt. Aus dem Vorgehen resultiert eine große Menge an Subklassen. Außerdem wird in diesem Fall viel Verhalten redundant implementiert. Um die Menge an Subklassen zu bestimmen, wird eine Vererbungshierarchie aufgebaut. Diese enthält die Informationen darüber, welche Klasse von welcher anderen Klasse erbt.

Die verwendeten Metriken sind wie folgt festgelegt worden.

Tabelle 25: Verwendete Metriken für die Erkennung von Decorator-Kandidaten

Metrik	Beschreibung	Grenzwert	Anwendung in Regel
M1.1 Klassenhierarchie	Das Programm enthält eine Klassenhierarchie.	True	R1.2
M1.2 Anzahl der Subklassen	Die Klassenhierarchie enthält mehrere Subklassen auf der ersten Ebene.	≥4	R1.2

5.2.2 Grenzwerte

Der folgende Abschnitt beschreibt die Auswertung der ermittelten Ergebnisse aus der Merobase und das daraus entwickelte Erkennungsmodell.

Die Merobase-Analyse basierte auf folgendem Suchstring:

*X*Decorator lang:java type:interface (protocol:svn OR protocol:CVS)*

Die Suche wird wie in Kapitel 4.3.2 durchgeführt. Basierend auf der Implementierung von Gamma et al. Verwendet der Dekorierer eine Klasse und kein Interface, deshalb müssen alle Buchstaben in Betracht gezogen werden.

Festlegung der Grenzwerte

Die Suche in Merobase lieferte 108 Ergebnisse. Daraus wurden die ersten 50 korrekt implementierten Decorator-Pattern-Ergebnisse ausgewählt, die aus 41 verschiedenen Projekten stammen. In diesen ausgewählten Beispielen werden 339 Dekorationsmöglichkeiten implementiert. Bei der Analyse ergab sich folgende Verteilung der Ergebnisse.

Tabelle 26: Statistische Werte der Decorators aus dem Merobase-Index

	Decorators
Min	2
1. Quartil	2
Median	4
Mittelwert	6,78
3. Quartil	7,75
Max	34

Bei der Analyse ergab sich, dass 34% der Decorator-Patterns nur zwei konkrete Decorators besitzen. Dies markiert die unterste akzeptierte Grenze in der Analyse. Eine Verwendung des Decorator-Patterns mit nur einem Decorator ist unlogisch, da in einem solchen Szenario nur eine kleine neue Funktionalität an ein Objekt angeheftet werden würde. Drei und sieben konkrete Implementierungen finden sich bei 10% der Ergebnisse. In 22% der Fälle haben die Implementierungen 10 oder mehr Decorators. Die restlichen Ergebnisse der Fälle zwischen vier und 30 konkreten Decorators nehmen zwischen 2% und 6% der Ergebnismenge ein. Das Maximum von 34 konkreten Decorators wurde nur in einem Beispiel gefunden.

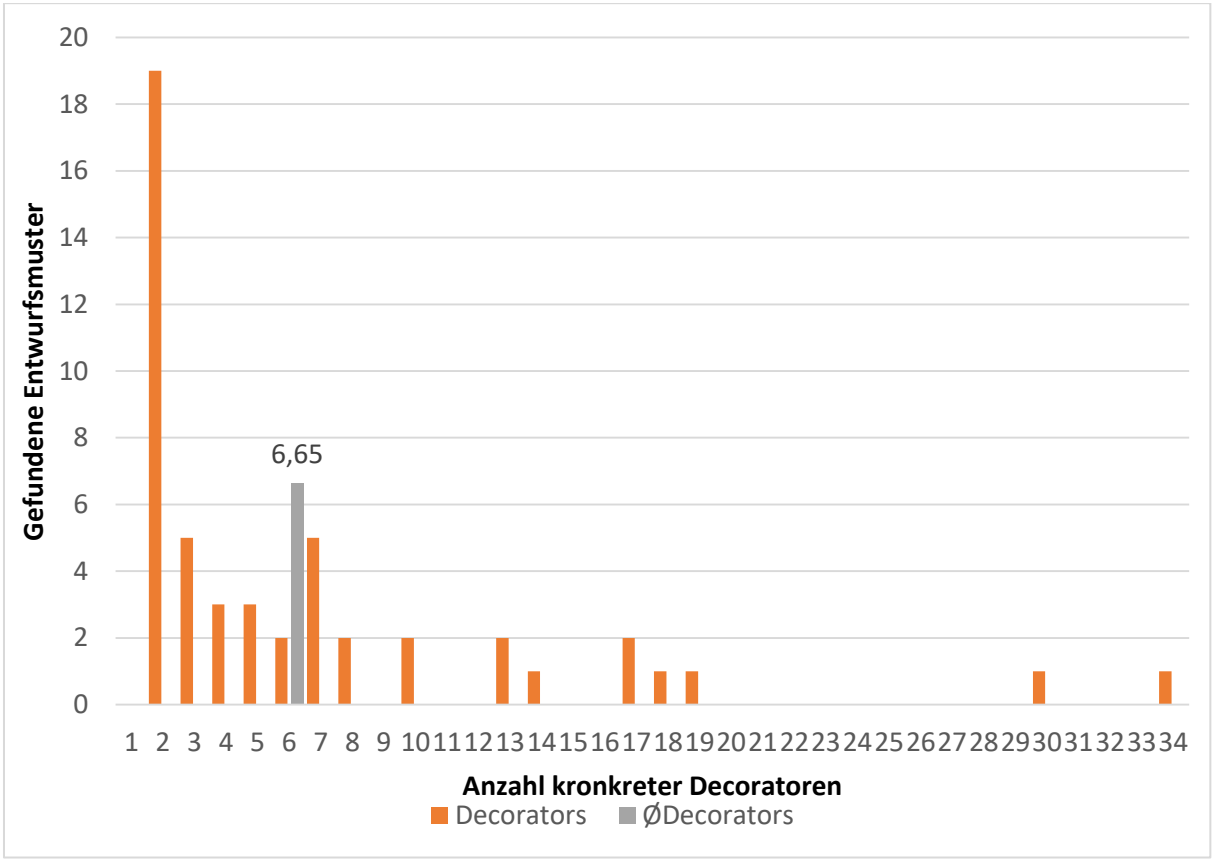


Abbildung 25: Histogramm der Decorator-Ergebnisse inklusive Durchschnitt

Das Modell zur Erkennung sieht, aufbauend auf den ermittelten Daten, wie folgt aus:

- Als untere Grenze und damit als Grenze für die Stufe *möglich* dient der Median von vier Subklassen.
- Der Grenzwert für die Erkennungsstufe *sinnvoll* liegt, ausgehend vom Mittelwert, bei sechs Subklassen der Ebene 1 in einer Klassenhierarchie.
- Sollte ein Kandidat mehr als 8 Subklassen besitzen, wird der Einsatz *empfohlen*. Das 3. Quartil von 7,75 dient hier als Grenzwert.

Daraus wurde das folgende Modell in Tabelle 27 erarbeitet.

Tabelle 27: Grenzwerte und Erkennungsstufen für Decorator-Kandidaten

Grenzwerte	Erkennungsstufe
≥4	Möglich
≥6	Sinnvoll
≥8	Empfohlen

5.2.3 Beispielerkennung

Analog zu den vorangegangenen Regeln wird diese Erkennungsregel ebenfalls an einem Beispiel im Detail diskutiert. Zur Erkennung eines Decorator-Kandidaten wird eine Klassenhierarchie benötigt. Grundlange für diese Hierarchie stellen die Vererbungsverbindungen zwischen den Klassen dar, d. h. die Frage, welche Klasse von einer anderen erbt. Abgebildet wird diese Hierarchie in einer Baumstruktur, wobei die Wurzel des Baumes der generalisiertesten Klasse entspricht. Zum Aufbau des Baumes und dessen Auswertung kommt das Open Source-Framework *jGraph* (Naveh, 2015) zum Einsatz. Ein Baum stellt hier nur eine spezialisierte Version des Graphen dar. Dieses Vorgehen erlaubt eine spätere Erweiterung um andere Analysemethoden und Messungen von vererbten Klassen zur Erkennung von anderen Design Patterns. Die komplette Verarbeitung benötigt fünf Prozessschritte (siehe Abbildung 26).

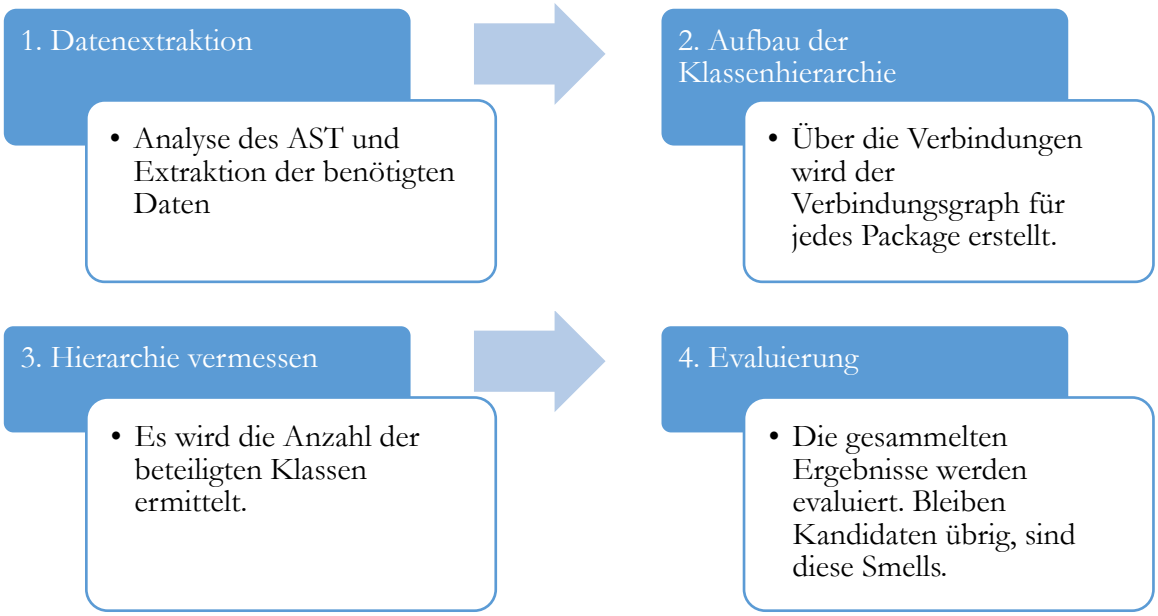


Abbildung 26: Detailprozess zur Erkennung von Decorator-Kandidaten

Im ersten Schritt wurden die Daten aus dem AST extrahiert. Im Fokus dieser Analyse stehen die Informationen über Vererbung. Von der Klasse Nr. 214 *AbstractArgoJPanel* erben zwölf Klassen,

Bei der Extraktion der Daten können nur Beziehungen ersten Grades festgestellt werden, d. h. nur die Klasse, von der geerbt wird, steht im AST. Solche Vererbungen werden innerhalb einer Klasse mit dem Knoten *ASTExtendsList* gekennzeichnet, welcher den Namen der Superklasse beinhaltet. Da der Knoten nur Informationen darüber enthält, von welcher Klasse die aktuelle Klasse erbt, können auf diesem Wege keine Informationen darüber abgelesen werden, ob die Superklasse dieser Klasse wiederum von einer anderen erbt. Dies wird möglich, indem alle Vererbungsbeziehungen von allen Klassen in der Datenbank gespeichert werden. Am Ende der Datenextraktion existiert eine Tabelle, in der alle Klassen verzeichnet sind, die von einer anderen Klasse erben, sowie die Klasse, von der geerbt wird. Die folgende Tabelle 28 zeigt einen Auszug der Tabelle.

Tabelle 28: Auszug aus der Tabelle, die alle Vererbungen beinhaltet

ID der erbenden Klassen	Name	Superklassen-ID
214	AbstractArgoJPanel	214
434	TabChecklist	214
3403	StylePanel	213
3404	StylePanelFig	3403
3420	StylePanelFigNodeModelElement	3404
3427	StylePanelFigText	3404
4454	StylePanelFigClass	3420

Da der Knoten *ASTExtendsList* nur den Namen beinhaltet, werden die restlichen Informationen (Methoden, Attribute etc.) ergänzt, sobald der Visitor die Superklasse direkt besucht.

Aus den Daten, die in Schritt 1 ermittelt wurden, werden in diesem Schritt 2 mehrere Vererbungshierarchien aufgebaut. Diese Hierarchien beinhalten die wichtigsten Informationen über die Klassen (Verbindungen, IDs etc.) und darüber, wie tief die Vererbung reicht. So kann eine universelle Klasse über mehrere Ebenen immer weiter spezialisiert werden. Die folgende Abbildung 27 zeigt einen Ausschnitt aus der Vererbungshierarchie der Klasse 214 (*AbstractArgoJPanel).* Wie zu sehen ist, besitzt die Klasse einige Ebenen und auch eine gewisse Breite.

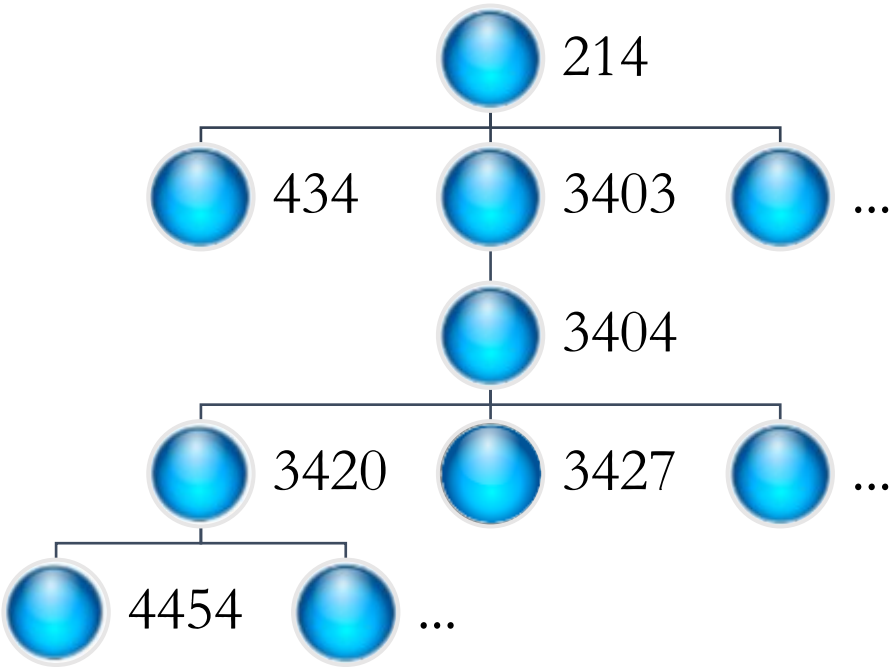


Abbildung 27: Vererbungshierarchie der Klasse 213

Als Repräsentation zur Speicherung der Klassenhierarchien dient ein Graph. Jede Hauptklasse bildet die Wurzel eines Baumes. Die einzelnen Knoten verweisen auf die vererbten Klassen. Da in Java keine Mehrfachvererbung möglich ist, entstehen bei der Analyse mehrere unabhängige Hierarchiebäume.

Für die Feststellung, ob ein Design Pattern-Kandidat vorliegt, werden die einzelnen Hierarchien vermessen. Mögliche Messverfahren sind hier:

1. Breite der Hierarchie: Wie viele Klassen erben von der Hauptklasse pro Ebene? Je mehr Klassen eine Ebene hat, desto breiter ist sie.
2. Tiefe der Hierarchie: Was ist der längste Weg durch die Vererbungshierarchie?
3. Anzahl der beteiligten Klassen in der Hierarchie. Wie oft werden die Informationen weitervererbt? Dies ist eine Verbindung aus 1. und 2.

Im Fall von Klasse 214 sind 86 Klassen an der Vererbung beteiligt, davon allein 12 auf der ersten Ebene. Es gibt eine Tiefe von mehr als 4 Ebenen.

Evaluierung

Durch die ermittelten Informationen kann im letzten Schritt festgestellt werden, ob es sich um einen Kandidaten handelt und welche Stufe dieser Kandidat besitzt. Mit 12 beteiligten Klassen auf der ersten Ebene liegt der gemessene Wert über dem Grenzwert von 8 der Metrik M1.2. Der Kandidat wird somit in die Erkennungsstufe *empfohlen* eingeordnet.

Eine mögliche Implementierung des Decorator-Entwurfsmuster um diese Quellcodestelle zu verbessern, wurde am Anfang in Abbildung 24 gezeigt.

Evaluationsunschärfe und Regelerweiterung

Um die Erkennung von Decorator-Kandidaten zu schärfen, ist wie am Anfang des Kapitels angesprochen, eine Regelerweiterung notwendig. Diese könnte wie folgt aussehen.

Ein Decorator bietet die Möglichkeit, dynamisch Verhalten in Objekte einzufügen. Damit können Subklassen mit doppelten Funktionalitäten vermieden werden. Die entsprechenden Klassen-Hierarchien werden heute schon identifiziert. Es gilt nun festzustellen ob diese Klassen gleiche Funktionalitäten implementieren. Als Beispiel diene ein Programm zum Bestellen von Pizzas. Jede Pizza wird durch ein Objekt repräsentiert. Jede Pizza erbt von der Klasse *Margaritha* (Mozzarella und Tomaten), Da es Zutaten gibt, die auf mehreren Arten vorkommen (z.B. Schinken), wird das Verhalten mehrfach implementiert, z.B. für eine Pizza *SchinkenAnanas* und eine Pizza *SchinkenPilze*. Beides sind eigenständige Klassen, die von Margaritha erben und Verhalten doppelt implementieren. Die Regel könnte nun so erweitert werden, dass die Namen der Klassen als Identifikationsmerkmal mitverwendet werden. Sobald Klassennamen in der Hierarchie gleiche Substrings aufweisen (z.B. Schinken) könnte dies als weiteres Indiz für einen Decorator-Kandidaten verwendet werden. Die zuvor festgelegten Grenzwerte können weiter verwendet werden.

Das oben genannte Beispiel erfüllt diese Regeln in einigen Subklassen: von der Klasse StylePanel gibt es mehrere Unterklassen (siehe Teilauszug in der folgenden Tabelle). Wie zu sehen ist, wird der Name der Klasse immer erweitert. Die angefügten Substrings deuten auf die neuen Funktionalitäten hin. Es bestand also die Möglichkeit, dass die Klassennamen ein Hinweis sind für die mehrfache Implementierung von Funktionalitäten, wie z.B. *StypelPanelFig* und *StylePnaleFigNode* wo die zweite Klasse noch die Funktionalität für eine Node anfügt. Dieser Namensstruktur wurde jedoch nicht bei jeder Klasse des Kandidaten verwendet. Mit der Hilfe der zugehörigen JavaDoc wurde die Funktionalitätserweiterung bestätigt.

ID	Name	Beschreibung aus JavaDoc
3403	StylePanel	The Presentation panel - formerly called style panel
3404	StylePanelFig	The basic stylepanel for a Fig which allows the user to see and adjust the common attributes of a Fig: the boundaries box, line and fill color information and the stereotype view combo box.
3420	StylePanelFigNode ModelElement	Stylepanel which provides base style information for modelelements, e.g. shadow width, the path checkbox
3427	StylePanelFigText	StylePanel class which provides additional parameters for changing Text elements provided by GEF. TODO: i18n
4454	StylePanelFigClass	Stylepanel which adds an attributes and operations checkbox and depends on figClass

Da die Schwäche der Regel erst spät in der Evaluation erkannt wurde, konnten die eigentlichen Ergebnisse nicht mehr angepasst werden. Die folgenden Resultate für den Decorator basieren daher alle noch auf der ersten Version der Erkennungsregel. Unschärfen werden an den entsprechenden Stellen diskutiert um, die Verbesserungsmöglichkeiten weiter heraus zu arbeiten.

Zusätzlich wurde manuell eine erste Evaluation mit der neuen Regelversion durchgeführt. Das Erkennen von Substrings stellt sich zwar als möglich, jedoch als nicht einfach heraus. Entwickler neigen nicht dazu, ihre Klassen immer klar zu benennen. Ein klares Positiv-Beispiel konnte daher im Verlauf der Evaluation nicht identifiziert werden. D.h., es liegt nahe, zukünftig den Algorithmus zum Erkennen von Decorator-Kandidaten weiter zu verbessern.

5.3 Facade-Pattern

Eine Facade findet ihre Verwendung, wenn ein Subsystem unabhängig vom Gesamten sein soll, damit das Subsystem schnell und flexibel ausgetauscht oder erweitert werden kann. Die folgende Detailbeschreibung einer Kandidatenerkennung für das Facade Pattern stützt sich auf das Open Source-Project *recoder* (Heuzeroth, Trifu, & Gutzmann, RECODER, 2015) in der Version 0.97. Auf Grund der gesammelten Informationen wurde für das Package *recoder.util* ein Facade-Kandidat vorgeschlagen. Gerade dieses genannte Package wird sehr häufig von anderen Subsystemen des Programmes benötigt. Es selbst ist aber nahezu unabhängig von anderen Subsystemen. Doch die internen Verbindungen zwischen den Klassen des Packages sind recht hoch. Fast jede Klasse hat zu jeder anderen eine Verbindung, dadurch ist ausgeschlossen dass es sich um eine reine Toolklasse mit eigenständen Werkzeugen handelt. Durch den Einsatz eines Facade-Patterns wird die Abhängigkeit das Packages *recoder.util*, durch aufrufende Systeme, verringert. Hinter der Facade können Teile verändert werden, ohne dass andere Subsysteme von der Veränderung betroffen sind. Es wäre sogar möglich, das gesamte Package gegen ein anderes auszutauschen, ohne andere Teilsysteme anpassen zu müssen. Die folgende Abbildung 28 stellt den Kandidaten nach der Verbesserung durch ein Facade-Pattern dar. Wie zu sehen ist, gehen alle Aufrufe nun über das Interface.

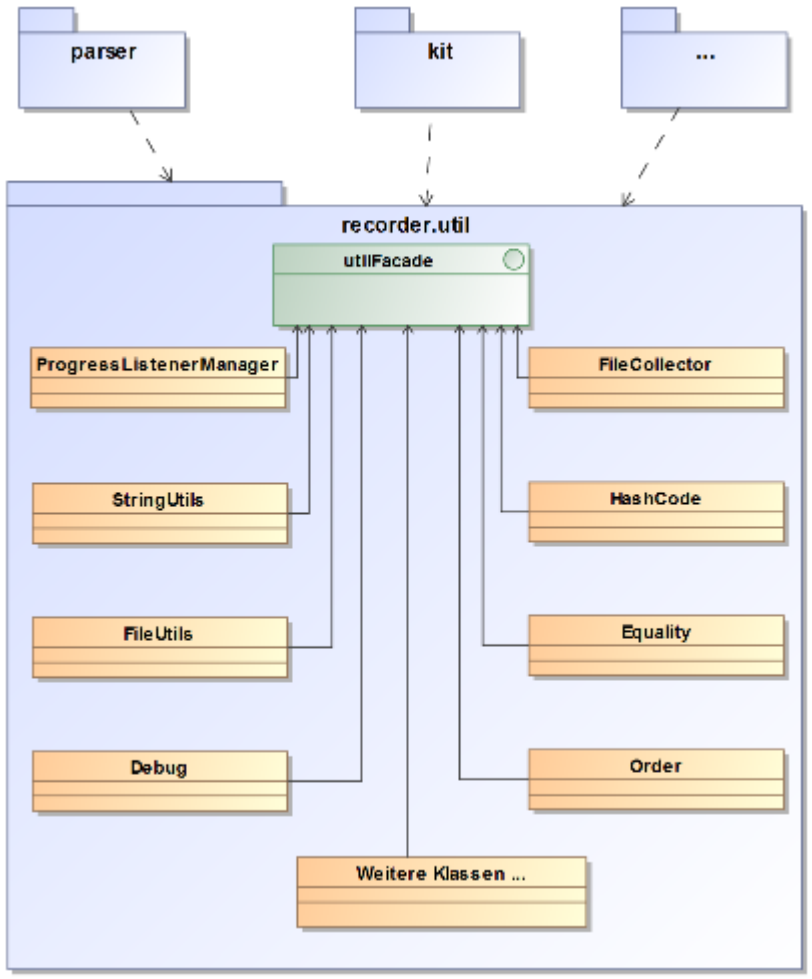


Abbildung 28 Vereinfachte Darstellung des recoder.util Package mit möglicher Facade

5.3.1 Erkennungsregel

Folgende Regel wurde auf Basis der oben genannten Verwendung für das Auffinden von Facade-Pattern-Kandidaten definiert.

R1.3 Ein Package wird von mehreren anderen Packages aufgerufen, doch es selbst ruft nur wenige andere Packages auf; außerdem sind die Klassen im Package stark miteinander verbunden.

Grundlage der Regeln sind die Verbindungen zwischen einzelnen Packages. Es wird davon ausgegangen, dass Packages existieren, deren Inhalte von überall innerhalb des Programmes aufgerufen werden, die aber selbst nur wenig mit dem Programm interagieren. Bei dieser Erkennung werden nur die Verbindungen zwischen Packages verwendet.

Aus Design-Sicht wäre es auch möglich, eine Facade innerhalb eines Packages einzusetzen, um Klassen im selben Package voneinander abzutrennen. Dies würde aber in den meisten Fällen gegen die Kapselung von Funktionen durch Packages sprechen. Einen weiteren wichtigen Aspekt der Regeln stellen die Interaktionen innerhalb des Packages dar. Komplettiert wird die Regel erst, wenn die Klassen im untersuchten Package viel miteinander kommunizieren. Des Weiteren werden für die Analyse mehr Details über die Verbindungen benötigt; so werden diese unterschieden in eingehende, ausgehende und interne Verbindungen. Tabelle 29 zeigt die Unterschiede.

Tabelle 29: Definition der verschiedenen Verbindungen

Verbindungstyp	Beschreibung
Eingehend	Eine Verbindung dieses Typs beschreibt einen Weg von einer Klasse A zu einer Klasse B innerhalb des Packages B. Dabei ruft die Klasse A eine Methode auf oder initialisiert ein Attribut von Klasse B. Die Klassen A und B liegen in unterschiedlichen Packages.
Ausgehend	Diese Verbindung hat die umgekehrte Richtung. Die Klasse B innerhalb des Packages holt Daten aus der Klasse A die außerhalb des Packages liegt.
Intern	Verbindungen, die innerhalb des gleichen Packages liegen, werden als intern bezeichnet. So ruft die Klasse B die Klasse C auf und beide befinden sich im gleichen Package.

Aus dieser Betrachtung ergibt sich eine Unterscheidung zu reinen Toolklassen (utility classes[5]). Toolklassensammlungen kommunizieren nur wenig untereinander, da Tools meist abgeschlossen sind. Eine Facade wäre hier schwer zu implementieren, da ein einheitliches Interface, das alle Tools umfasst, sehr komplex wäre. Zuletzt darf ein Facade-Kandidat nur wenige Aufrufe zu anderen Packages besitzen. Hier ist zu erwähnen, dass die Erkennungsregel nach möglichen Facade-Kandidaten sucht, welche eingehende Aufrufe abkapseln. Erst durch wenige ausgehende Verbindungen kann ein System als unabhängig betrachtet werden und ist somit ein sinnvoller Kandidat für eine Facade.

[5] https://msdn.microsoft.com/en-us/library/ee437010.aspx

Tabelle 30: Metriken zur Erkennung eines Facade-Kandidaten

Metrik	Beschreibung	Anwendung in Regel
M1.1 Eingehende Aufrufe	Anzahl direkter Verbindungen in das Package	R1.3
M1.2 Ausgehende Aufrufe	Anzahl direkter Verbindungen zu anderen Packages	R1.3
M1.3 Interner Zusammenhalt	Die Klassen innerhalb des Packages sind stark miteinander verbunden, d. h. die Klassen rufen viele andere interne Klassen auf oder werden von ihnen aufgerufen.	R1.3

5.3.2 Grenzwerte

Der folgende Abschnitt beschreibt die Schritte zur Festlegung von Grenzwerten. Die Regel bildet die Ausnahme beim allgemeinen Vorgehen: Es wurde nicht auf die Daten der Merobase zurückgegriffen.

Bei der Festlegung von Grenzwerten für das Facade-Pattern muss von dem normalen Vorgehen abgewichen werden. Wie bei den Patterns zuvor und auch danach sollte als Grundstock für die Grenzwerte bereits implementierte Facaden aus der Merobase-Index verwendet werden. Doch die ersten Suchergebnisse zeigten, dass es zwei Schwierigkeiten bei der Suche nach geeigneten Facade-Klassen gibt.

1. Die Anzahl der gefundenen Ergebnisse mit dem Suchstring: *X*Facade lang:java type:class (protocol:svn OR protocol:CVS)* ist sehr gering. X steht hierbei für die verschiedenen Buchstaben von A bis Z.

2. Die Qualität der gefundenen Facaden und der darin enthaltenen Patterns entsprach nicht der Qualität, die für diese Auswertung notwendig wäre. Eine implementiere Facade muss der Vorgabe von Gamma et al. (Gamma, Helm, Johnson, & Vlissides, 1994) entsprechen. So lautet die Definition der Facade: „Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.“ nach Gamma (Gamma, Helm, Johnson, & Vlissides, 1994). Diese Definition kann folgendermaßen interpretiert werden. Eine Facade verbirgt ein Subsystem aus mehreren Klassen vor dem Gesamtsystem. Dabei ist die Facade eine abstrakte Schnittstelle (Interface oder abstrakte Klasse). Aufgabe dieser Schnittstelle ist es, den Zugriff auf das Subsystem zu vereinfachen.

   Die gefundenen Facade Pattern-Implementierungen wichen teilweise sehr von dieser Struktur ab. Es fehlten die Methoden in der Schnittstelle, die eigentlich die Menge an Schnittstellen innerhalb des Subsystems vereinfachen sollten. Des Weiteren wurden einige Interfaces bereitgestellt, welche aber im Quellcode keine Anwendung fanden. Somit bietet das Interface keine Vereinfachung der Schnittstellen im Subsystem, sondern nur eine Hülse. Eine solche Implementierung mag zweckmäßig sein, wenn in einer späteren Version die Funktionalität hinter der Facade noch erweitert werden soll. Doch in der aktuellen Implementierung ergibt der Einsatz des Patterns, ausgehend vom vorliegenden Quellcode, wenig Sinn.

Aus diesen zwei Gründen (fehlende Qualität und geringe Auswahl) wurde das Vorgehen für die Festlegung der Grenzwerte abgewandelt. Anstelle der Merobase wurden bekannte Java-Module, wie SLF4J oder Apache Batik, analysiert, die den Entwicklern eine Facade zur Verfügung stellen. Über die zugehörigen Dokumentationen der Projekte wurde dies identifiziert. Nach der Identifikation solcher Module wurden Open Source-Projekte gesucht, die diese implementieren und somit als Beispielimplementierungen dienen für die Grenzwertbestimmung. Wie zuvor, sollte keines der ausgewählten Projekte (siehe Kapitel 7.2.1) mehr als zwei Pattern Implementierungen für die Ermittlung liefern. Dies gilt auch für die verwendeten Facaden, da eine mehrfache Verwendung keinen Unterschied aufzeigen würde, wenn das Modul selbst immer das gleiche ist.

Festlegung der Grenzwerte

Durch die Umstellung des Vorgehens konnten elf Open Source-Projekte (siehe Anahng 0) identifiziert werden. Die Einschränkung, dass jedes Modul nur zwei Mal verwendet werden darf, um eine gewisse Diversität zu garantieren, schränkte die Anzahl der möglichen Projekte mit Pattern-Implementierung ein. Doch die Ergebnisse zeigten schon bei 11 Beispielen stabile Werte. Tabelle 31 zeigt einen Überblick der Analyseergebnisse. Die Tabelle zeigt die eingehenden Verbindungen (Spalte 1) von Klassen außerhalb der Packages zur Facade, die ausgehenden Verbindungen von Klassen hinter der Facade zu anderen Klassen außerhalb (Spalte 2), die internen Verbindungen der Klassen hinter der Facade untereinander (Spalte 3), die Anzahl der Klassen (Spalte 4) sowie die relativen Zahlen von eingehenden, ausgehenden und internen Verbindungen zur Klassenanzahl (Spalten 5, 6, 7).

Tabelle 31: Übersicht der verschiedenen Verbindungen und der Relation zur Klassenanzahl hinter der Facade

	Eingehende	Ausgehend	Intern	Anzahl Klassen hinter der Facade
Min	1	0	2	4
1. Quartil	2,25	0	5	6
Median	6	0	21	9,5
Mittelwert	33,00	4,00	113,80	42,80
3. Quartil	17,5	6	230	78,25
Max	232	15	460	139
	Ein / Klassen	**Aus / Klassen**		**Intern /Klassen**
Min	10%	0%		50%
1. Quartil	20%	0%		51%
Median	38%	0%		187%
Mittelwert	79%	6%		212%
3. Quartil	138%	10%		379%
Max	222%	30%		455%

Bei den eingehenden Verbindungen zeigt sich die größte Streuung. Eine Facade besitzt zwischen 1 und 232 Verbindungen von außenstehenden Klassen. Das Verhältnis zwischen den Verbindungen, die von außen auf die Facade kommen, zu der Gesamtanzahl an Klassen die innerhalb der Facade, zwischen 10% und 222%.

Der relative Mittelwert zwischen ausgehenden Verbindungen aus dem Packages heraus und Verbindungen zwischen Klassen innerhalb des Packages liegt bei 6% und erreicht maximal einen Wert von 30%. 5 von 10 Projekten haben gar keine Verbindung nach außen. Das liegt auch an der

Art der Implementierung. Ist die Facade Teil eines externen Modules, das nur angebunden wird, sollte es keine ausgehenden Verbindungen geben. Wird das Modul aber als Teil des Projektes entwickelt, können bestimmte Funktionalitäten vom Gesamtprojekt aufgerufen werden. Bei den internen Verbindungen zwischen den package-eigenen Klassen existiert kein Wert unter 50%, d.h. jeder Klasse hat eine Verbindung zu jeder 2. Klasse im Package. Dies zeigt schon die hohe Verbundenheit zwischen den Klassen. Im Mittelwert werden schon 212% erreicht. Bei einem solchen Wert besitzt jede Klasse mindesten zwei Verbindungen zu jeder anderen. 455% werden im Maximum erreicht. Eine solch hohe Verbundenheit zwischen den Klassen wird nur bei Modulen gemessen, die eine sehr hohe Komplexität besitzen und intern viele Funktionen implementieren, die gegenseitig von Nutzen sind.

Die Festlegung der Grenzwerte erfolgt nach den relativen Ergebnissen, basierend auf dem Verhältnis zwischen der Anzahl der Verbindungen und der Anzahl interner Klassen. Einen absoluten Wert festzulegen, erweist sich als schwierig, da die Verbindungsanzahl sehr von der Anzahl der Klassen abhängig ist. Für die drei Stufen wurden folgende Grenzwerte festgelegt.

Tabelle 32: Grenzwerte und Erkennungsstufen für Facade-Kandidaten basierend auf den Messergebnissen

Eingehende	Ausgehende	Interne	Erkennungsstufe
≥20%	≤30%	≥51%	Möglich
≥38%	≤10%	≥187%	Sinnvoll
≥79%	≤6%	≥212%	Empfohlen

Anzumerken ist, dass der Wert für ausgehende Klassen immer kleiner wird, weil eine hohe Anzahl ausgehender Verbindungen eine Abkapslung der Klassen sehr aufwändig machen kann. Aus diesem Grund sind kleinere Zahlen in dieser Metrik sinnvoller. Im Gegensatz dazu steigen pro Erkennungsstufe die möglichen eingehenden Verbindungen. Ein Package mit vielen eingehenden und wenigen ausgehenden Verbindungen eignet sich besser für ein Facade-Pattern. Den letzten Punkt stellen die internen Verbindungen dar. Ist jede Klasse mit jeder zweiten Klasse innerhalb des Packages verbunden (≥50%), kann eine reine Toolsammlung ausgeschlossen werden. Zeigt sich eine starke Kopplung, bei der jede Klasse mehrmals mit jeder anderen interagiert, so sollte das Package auf jeden Fall abgekapselt werden. Der Einsatz einer Facade würde einen Austausch oder Änderungen in der Funktionalität flexibler gestalten.

5.3.3 Beispielerkennung

Im Vergleich zu anderen Regeln bedarf es für die Erkennung von Facade-Kandidaten nur einiger weniger Arbeitsschritte. Die notwendigen drei Schritte werden in Abbildung 29 kurz angerissen. Die folgenden Abschnitte beschreiben die Erkennung des zuvor schon angesprochenen Facaden-Kandidaten im Open Source Projekt *recoder*.

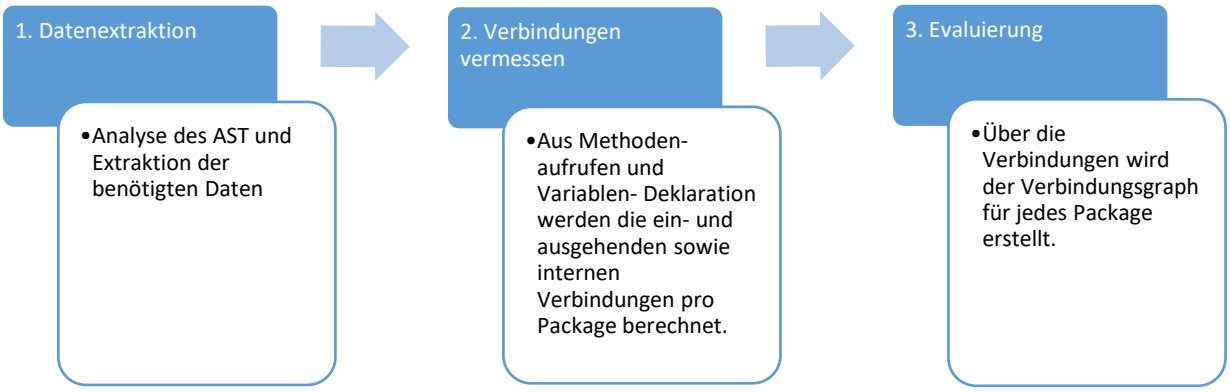


Abbildung 29: Detailprozess zur Erkennung von Facade-Kandidaten

Analog zu anderen Regeln werden zuerst die wichtigsten Informationen aus dem AST des zugrundeliegenden Quellcodes extrahiert. Hierzu gehören Daten über Methodenaufrufe zwischen Klassen, Package- und Klassen-Namen und genutzte Attribute, die keine Basistypen besitzen. Nachfolgend einige Auszüge aus den extrahierten Daten.

Zuerst wird ein Package ausgewählt:

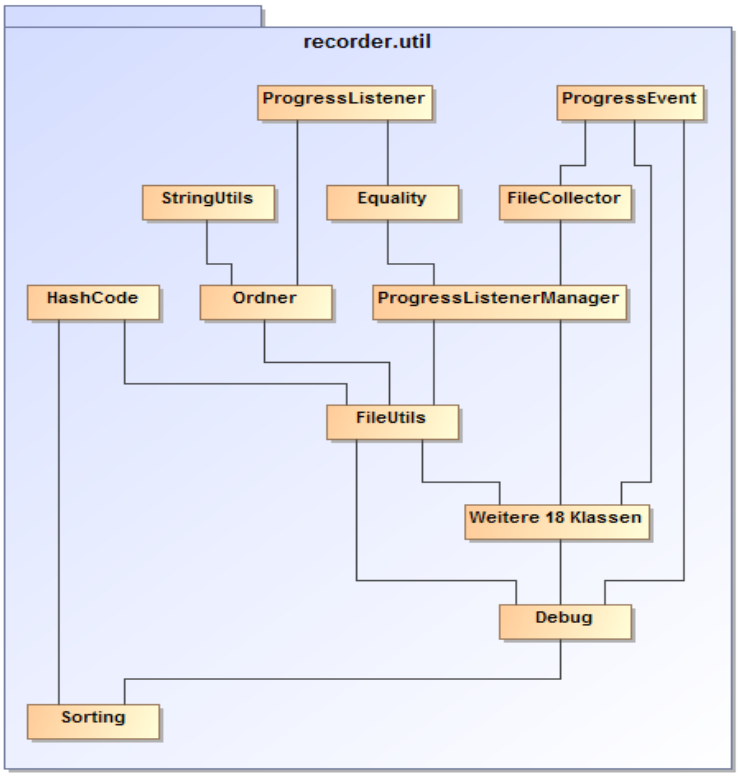
Tabelle 33: Auszug aus der Package-Datentabelle

ID	Name
39	recoder.util

Für dieses Package werden alle Klassen aus der Datenbank gezogen. Tabelle 34 zeigt einen Auszug der Klassen, die zum Package 39 gehören.

Tabelle 34: Klassen, die zum Package mit der ID 39 gehören und Skizze der Verbindungen innerhalb das Packages

ID	NAME
62	Order
106	Debug
460	FileUtils
465	ProgressListenerManager
492	StringUtils
506	FileCollector
732	HashCode
733	Equality
1419	ProgressEvent
1641	Sorting
1642	Index
1679	ProgressListener
…	(18 weitere Klassen)



Zuletzt ein Auszug aus den Methodenaufrufen der Klassen (506), die zu Package 39 gehören. Tabelle 35 zeigt den Namen des Attributes, die ID der Klasse, aus der es stammt, und die Klasse, in der es gefunden wurde.

Tabelle 35: Auszug der Methodenaufrufe innerhalb des Package 39

ID	Attributsname	Herkunftsklassen-ID	Zugehörige Klasse
15862	stack	13	506
15863	current	13	506
15864	count	42	506
15865	stack	13	506
15866	newStack	13	506
15867	System	40	506
15868	current	13	506
15869	content	9	506
…	(weitere 9 Verbindungen)		

In diesem Schritt werden die Verbindungsdaten zusammengestellt. Methodenaufrufe sowie verwendete Attribute werden zu Verbindungen zwischen Klassen aufgearbeitet. Um eine Methode auf einer anderen Klasse aufzurufen wird eine Referenz benötigt. Dieser Zusammenhang wird hier hergestellt. So werden die in Tabelle 35 gezeigten Attributs- und Variablendaten zu Verbindungen zwischen Klassen verarbeitet. Ebenfalls werden die Methodenaufrufe verwendet, vgl. Tabelle 36. Für das Package 39 und die Klasse 506 würde eine solche Verbindungstabelle wie folgt aussehen.

Tabelle 36: Verbindungstabelle von Ursprungsklasse und -Package zu Zielklasse sowie Anzahl der Verbindungen

ID	Ursprungs Klasse	Ursprungs Package	Ziel Klasse	COUNT
299	506	39	503	1
1196	506	39	13	10
1237	506	39	9	3
1980	506	39	40	2
2017	506	39	42	2

Evaluierung

Bei der Durchführung der Erkennung kam es für das oben genannte Package zu folgenden Verbindungszahlen (siehe Tabelle 37).

Tabelle 37: Ergebnis pro Metrik aus der Analyse

Interne Klassen	M1.1	M1.2	M1.3
30	Eingehende Verbindungen: 322	Ausgehende Verbindungen: 4	Interne Verbindungen: 47
	M1.1: 1073%	M1.2: 0,13%	M1.3: 156%

Aus den ermittelten Daten und dem Vergleich mit den Grenzwerten ergibt sich ein Facade-Kandidat, der *sinnvoll* erscheint. Einzig die wenigen internen Verbindungen sorgen für eine Einstufung als *sinnvoll* anstatt *empfohlen*.

5.4 Mediator-Pattern

In diesem Kapitel geht es um die Erkennung von Mediator Pattern-Kandidaten. Für die Diskussion wurde ein Kandidat aus dem Open Source Editor JEdit (Version 5.1) verwendet. Dieser Kandidat wurde im Package *org.gjt.sp.jedit.search,* identifiziert. In diesem Package existieren vier Klassen *SearchAndReplace, HyperSearchResults, SearchDialog* und *ReplacheActionHandler,* deren Kommunikationsverhalten durch ein Mediator Pattern verbessert werden kann. Abbildung 30 zeigt die Kommunikationswege zwischen den genannten vier Klassen mit Mediator Pattern. Ohne das Pattern existiert von nahezu jeder der genannten Klassen zu jeder anderen ein Kommunikationsweg. Der Einsatz eines Mediators ermöglicht es, das Kommunikationsverhalten in einer neuen Klasse zu zentralisieren. Dadurch können neue Klassen schnell in die Kommunikation eingebunden werden.

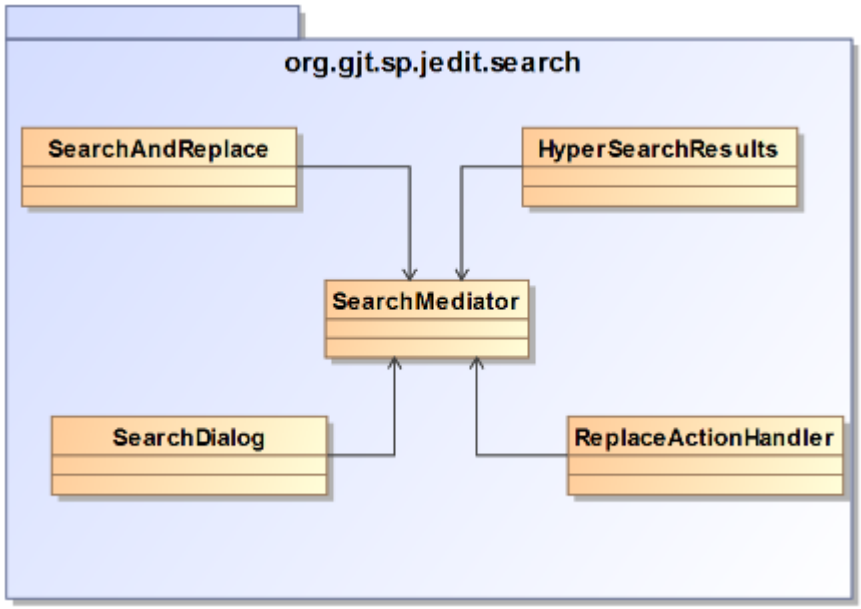


Abbildung 30 Package org.gjt.sp.jedit.search mit möglicher Mediator-Klasse

5.4.1 Erkennungsregel

Die Regel zur Erkennung möglicher Quellcodezeilen, bei denen das Mediator-Pattern angewandt wird, definiert sich wie folgt.

R1.4 Mehrere Klassen innerhalb eines Packages kommunizieren in einem komplexen Geflecht untereinander.

Aufbauend auf dem Anwendungsfall für das Pattern, wurde das Regeldesign darauf ausgelegt komplizierte Kommunikationsverbindungen innerhalb eines Packages zu identifizieren. Bei der Suche werden nur die Unique-Verbindungen innerhalb eines Packages beziffert, d. h. Verbindungen zwischen zwei Klassen werden nur einmalig pro Richtung gezählt. Im Falle eines eingesetzten Mediator Patterns wird jede Art der Kommunikation zwischen den gewählten Klassen über die Mediator-Klasse führen. Daraus folgt, dass nur der Mediator bei einer Änderung im Kommunikationsablauf angepasst werden muss.

Bei der Verwendung des Mediator Patterns ist eine besondere Umsicht notwendig, da die falsche Anwendung zum Anti-Pattern God-Class (Smith & Williams, 2000) führen kann. Eine God-Class vereinigt einen Großteil der Funktionalität eines Programmes und wird somit zum Zentrum des Programms. Jede Änderung in der Kommunikation würde auch diese Klasse betreffen – egal, wie

gering diese ist. Deshalb sollten nur die Klassen, die eine komplexe Kommunikationsbeziehung besitzen, in die Implementierung des Mediator Patterns einbezogen werden. Dieser Umstand wird von den Regeln berücksichtigt, da nur komplexe Geflechte zu einem Kandidaten führen.

Die Regel verwendet dazu die folgenden Metriken:

Tabelle 38: Übersicht der für die Regel verwendeten Metriken

Metrik	Beschreibung	Grenzwert	Anwendung in Regel
M1.1 Anzahl stark verbundener Komponenten	Im Graphen gibt es eine stark verbundene Komponente (SVK, Definition siehe Kapitel 4.5.3) aus Klassen.	True	R1.4
M1.2 Anzahl beteiligter Klassen	Die Anzahl der Klassen in der SVK überschreitet einen Schwellwert.	≥2	R1.4
M1.3 gemeinsames Package	Alle Klassen in der SVK gehören zu einem Package.	True	R1.4

5.4.2 Grenzwerte

Die Grenzwerte für Mediator-Kandidaten wurden wie folgt bestimmt:

Für die Identifikation entsprechender Mediator-Beispiele wurde folgender Suchstring mit den Buchstaben A-Z am Anfang in der Merobase verwendet.

*X*Mediator lang:java type:class (protocol:svn OR protocol:CVS)*

Die Basis der Suche liegt in der Annahme, dass ein implementiertes Mediator-Pattern eine Klasse mit dem Namen Mediator besitzt.

Festlegung der Grenzwerte

Basierend auf dem oben genannten Suchstring ergaben sich, wenn für X die Buchstaben A bis Z verwendet wurden, 392 Ergebnisse. Aus diesem Pool von Ergebnissen wurden 50 Mediatoren durch eine manuelle Prüfung des Quellcodes und seiner Struktur ausgewählt. Jeder der gewählten Mediatoren beruht auf der Definition von Gamma et al. Ergebnisse, die nicht der Implementierung entsprochen haben, wurden nicht in die Grenzwertbestimmung aufgenommen. Insgesamt stammen die 50 Mediatoren aus 40 verschiedenen Open Source-Projekten unterschiedlicher Größe. Abbildung 31 zeigt die Verteilung der gefundenen Ergebnisse in Bezug auf die Anzahl am Clients pro Mediator.

Zur Definition des Grenzwertes für die Mediator-Regel R1.1 wurde ein Merkmal aus dem Entwurfsmuster gewählt: die Anzahl durch den Mediator verbundener Klassen. Dieses Merkmal stellt die wichtigste Aufgabe des Patterns dar. Bei der Vermessung der gefundenen Mediatoren werden nur projekteigene Klassen einbezogen, das schließt Systemklassen wie String etc. aus.

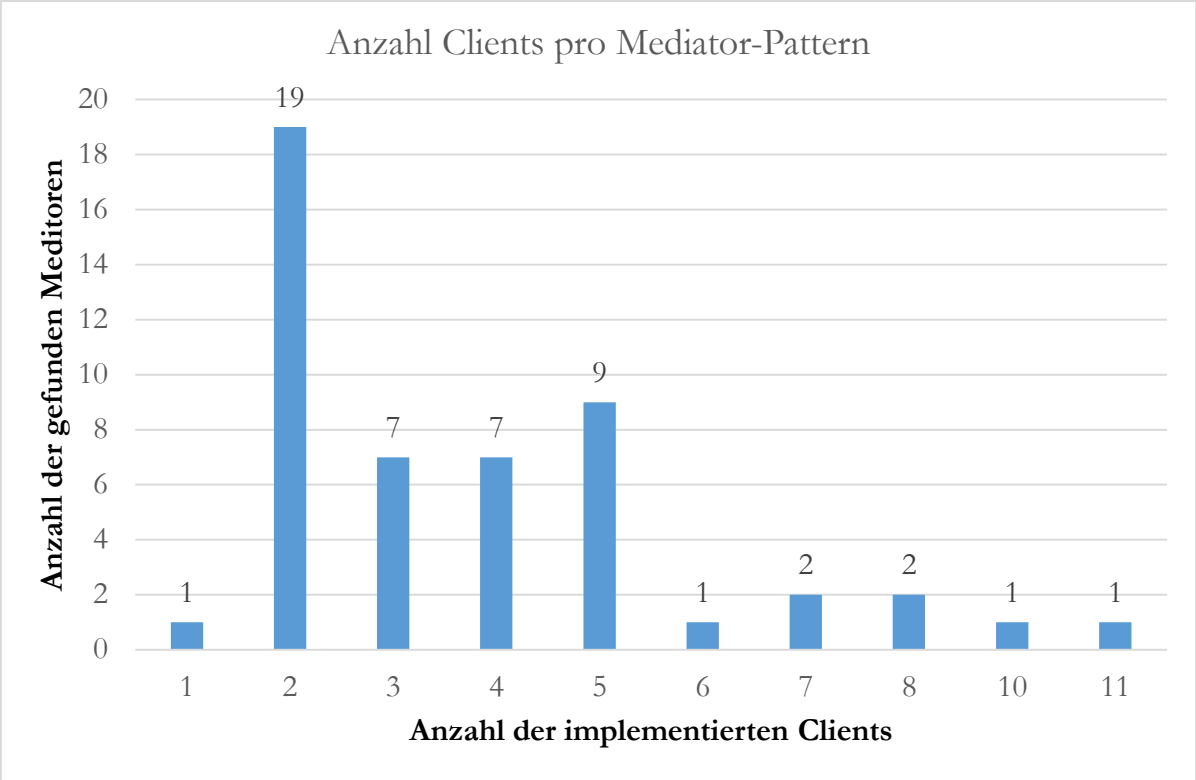


Abbildung 31: Anzahl und Häufigkeit von Mediator-Patterns im Merobase-Index

Die Auswertung der Ergebnisse ergab eine durchschnittliche Anzahl von 3,8 Clients pro Mediator. Am stärksten werden zwei Clients angesprochen (38%). Mediatoren mit mehr als fünf Clients sind selten, da von den 50 Messungen nur sieben über fünf Clients besaßen. 84% der Ergebnisse liegen zwischen zwei und fünf Clients.

Tabelle 39: Statistische Verteilung der Mediator-Suchergebnisse

	Clients
Min	1,00
1. Quartil	2,00
Median	3,00
Mittelwert	3,80
3. Quartil	5,00
Max	11,00

Entsprechend der Ergebnisse wurde ein Grenzwert definiert. Der Mittelwert wurde als mittlere Grenze bestimmt. Bei einem Mittelwert von 3,80 ergibt sich eine Grenze von drei Klassen pro stark Verbundene Komponenten (SVK) (siehe Kapitel 4.5.3). Entsprechend dieser Grenze wurde ein Modell definiert. Bei einem Kandidaten mit zwei Klassen ist ein Mediator möglich. Ab drei Klassen ist er sinnvoll. Ab vier Klassen wird ein Mediator *empfohlen*. Tabelle 40 fasst die Erkennungsstufen zusammen.

Tabelle 40: Grenzwerte und Erkennungsstufe für Mediator-Kandidaten

Grenzwerte	Erkennungsstufe
2	Möglich
3	Sinnvoll
≥4	Empfohlen

5.4.3 Beispielerkennung

Die Erkennung für die Mediator-Regel benötigt fünf Arbeitsschritte (siehe Abbildung 32) und verwendet einen Graphen, in den die Unique-Verbindungen zwischen den Klassen als Kanten und die Klassen selbst als Knoten eingetragen werden. Durch den Algorithmus zur Erkennung von SVK werden entsprechend komplexe Strukturen im Graphen identifiziert. Jede SVK stellt herbei eine Codestelle dar, die durch ein Mediator-Pattern verbessert werden kann.

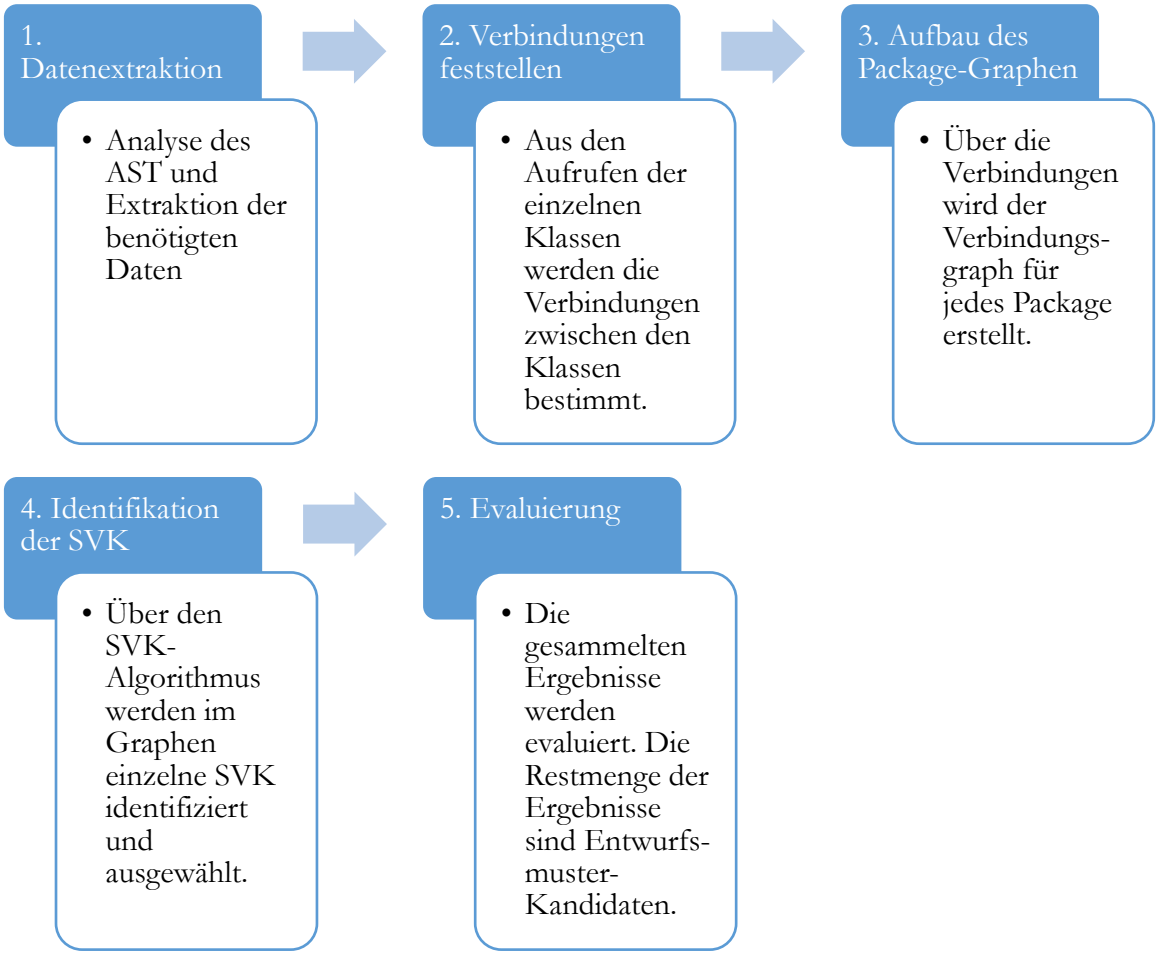


Abbildung 32: Detailprozess zur Mediator-Kandidaten-Erkennung

Im ersten Schritt der Erkennung werden die zur Analyse benötigten Daten aus dem AST des Quellcodes extrahiert. Für das Erkennen eines Kandidaten benötigt das Analysetool alle Verbindungen zwischen Klassen. Am Ende der Analyse stehen in der Datenbank:

1. das in der weiteren Analyse zu untersuchende Package;

Tabelle 41: Ausgewähltes Package

ID	Name
33	org.gjt.sp.jedit.search

2. alle zugehörigen Klassen;

Tabelle 42: Zum Package 33 gehörende Klassen

ID	Name	Active Class	Package-ID
784	SearchAndReplace	True	33
785	SearchFileSet	True	33
786	DirectoryListSet	True	33
836	SearchBar	True	33
972	SearchDialog	True	33
39 weitere Ergebnisse			

3. die Methodenaufrufe von einer Klasse zu anderen mit der Methode, der Startzeile im Code und den dazugehörigen Attributen. Aufrufe zwischen den Klassen basieren auf den zu den Klassen gehörenden Objekten, die für Methodenaufrufe verwendet werden, oder den Attributen/Variablen einer Klasse, die in einer anderen erzeugt wurden. Tabelle 43 beinhaltet diese Informationen für das Package 33. *ClassType* besagt, von welcher Klasse die aufgerufene Methode stammt. *Attribute* nennt die ID zum zugehörigen Attribut innerhalb der Klasse. *Class ID* bestimmt die Klasse, in der der Methodenaufruf ausgelöst wird, und *Startline* nennt die Zeile des Aufrufes.

Tabelle 43: Methodenaufrufe in der Klasse 784, die zu Package 33 gehört

ID	Name	Attributs Type	Method ID	Target Class ID
24825	search.equals	3362	12851	6158
24826	SearchAndReplace.search	784	12852	6158
24827	SearchAndReplace.search	784	12852	6158
24828	EditBus.send	282	12853	6158
24829	replace.equals	3363	12854	6160
24830	SearchAndReplace.replace	784	12852	6160
24831	SearchAndReplace.replace	784	12852	6160
…				

Aus den in Schritt 1 gesammelten Informationen werden in Schritt 2 die Verbindungen zwischen den einzelnen Klassen berechnet. Diese Analyse nutzt im Wesentlichen die Informationen aus Tabelle 43. Zusätzlich werden die folgenden Informationen gesammelt:

- die Klasse und das Package, in dem der Aufruf stattfindet (ClassOrigin/PackageOrigin);
- das Ziel des Aufrufes (ClassTarget/PackageTarget);

- die Summe der Verbindungen in diese Richtung in der Klasse.

Die folgende Tabelle 44 und Abbildung 33 zeigen einen Ausschnitt der Analyse der Verbindungen innerhalb des Packages 33. Die Tabelle zeigt die Aufrufe, ausgehend vom Ursprung zum Ziel, und die Anzahl der Aufrufe zwischen den Klassen.

Tabelle 44: Verbindungstabelle des Packages 33 mit internen Verbindungen

ID	Ursprungs-Klasse	Ursprungs-Package	Ziel-Klasse	Ziel-Package	Anzahl
3602	784	33	785	33	1
5241	784	33	3299	33	1
9492	784	33	3359	33	1
10745	784	33	784	33	1
13137	784	33	972	33	1
38209	784	33	972	33	1
38214	784	33	3299	33	5
42 weitere Ergebnisse					

In Schritt 3 wird der Graph selbst aufgebaut. Grundlage eines Graphen sind immer die Verbindungen innerhalb eines Packages. Verbindungen aus dem Package herausfinden keine Berücksichtigung. Ebenso wird jede Verbindung nur einmalig im Graphen eingezeichnet. Damit entsteht für jedes Package ein individueller Graph.

Identifiziert werden die SVK über das Framework *JGraph*. Der Algorithmus von *JGraph* basiert auf Cormen et al. (Cormen, Leiserson, Rivest, Stein, & others, 2001). Im Zuge der Analyse werden alle Teile des Graphen entfernt, die keine SVK bilden.

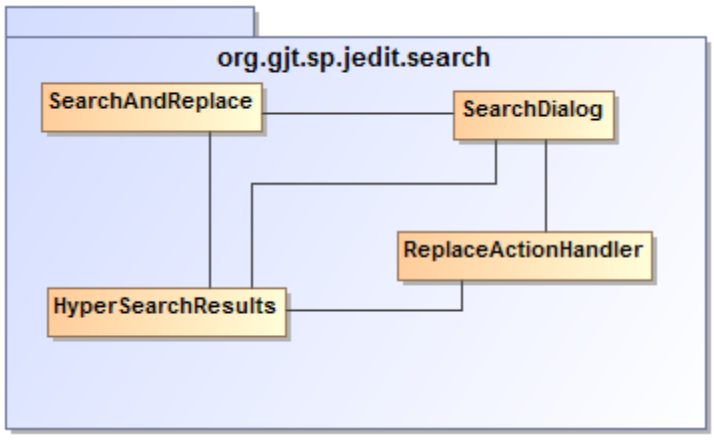


Abbildung 33 Skizze der Verbindungen

Evaluierung

Aus dem gewonnenen Wissen über die SVK im Package 33 ergibt sich ein Kandidat für ein Mediator-Pattern, der die Erkennungsstufe *sinnvoll* erhält. Diese Stufe resultiert aus den vier beteiligten Klassen.

Es bleibt noch der Unterschiede zwischen dieser Regel und der der Facade zu erläutern. Ein Unterschied ist die betrachtete Analyseebene. Die Analyse für Facade-Kandidaten findet auf der Package-Ebene und nicht auf der Klassenebene statt. Wie zuvor beschrieben, werden Facade-Kandidaten nur auf dieser Ebene gesucht. Damit unterscheidet sich die Auswertung der Verbindungsinformationen von der beim Mediator.

5.5 Strategy Pattern

Das Pattern sollte dann angewendet werden, wenn ein Programm mehrere Algorithmen kennt, durch die in unterschiedlicher Weise ein ähnliches Resultat erzielt wird. Zum besseren Verständnis des Pattern-Verhaltens dient der folgende Beispielkandidat, der mit einem Strategy Pattern verbessert werden kann. Das Beispiel stammt aus dem Projekt *JEdit* (Version 5.1). Die Klasse *TextUtilities* besitzt zwei Methoden (*findWordEnd, findWordStart*). Jede dieser Methoden implementiert einen Switch mit drei gleichen Case-Bedingungen. Des Weiteren wird die Variable *type* in beiden Switches innerhalb des Kopfes verwendet. Die Variable bekommt ihren Wert aus anderen Teilen der Applikation. Durch das Setzten der Variable wird die entsprechende Strategie ausgewählt. Durch die Anwendung des Strategy Pattern werden die verschiedenen Algorithmen in unabhängige Klassen gekapselt. Aktuell sind diese in den verschiedenen Methoden verteilt. Neue Algorithmen können dann schnell hinzugefügt oder auch entfernt werden.

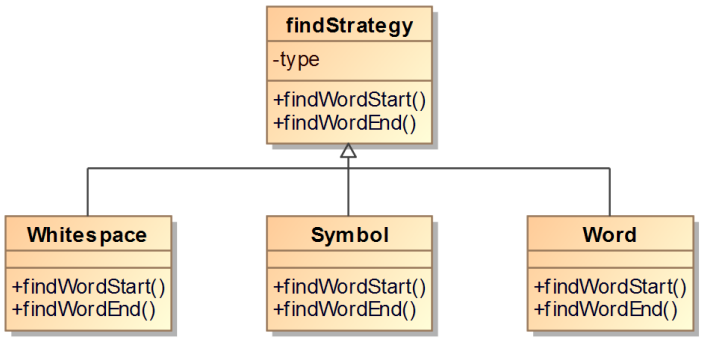


Abbildung 34 JEdit Strategy Pattern für TextUtilites Klasse

5.5.1 Erkennungsregel

Basierend auf der Definition des Patterns, wurde die folgende Regel erarbeitet, um mögliche Stellen im Quellcode, die durch das Strategy Pattern verbessert werden können, zu identifizierten:

R1.5 *Innerhalb einer Klasse existieren mehrere Methoden. Diese Methoden besitzen alle eine identische Switch-Anweisung, welche immer die gleiche Anzahl an Case-Fällen verwendet.*

Diese Regel basiert auf der Annahme, dass die Auswahl von verschiedenen Algorithmen mit der Hilfe von Switch-Anweisungen umgesetzt wird. Dazu wird in jeder Methode, die einen Algorithmus wählen muss, mindestens eine Switch-Anweisung implementiert. Jede dieser Anweisungen verwendet die gleiche Variable oder den gleichen Parameter in ihrem Kopf. Mit der Variable wird der Algorithmus bestimmt. Die eigentlichen Algorithmen werden dann innerhalb des entsprechenden Cases aufgerufen. Das Strategy Pattern würde die Auswahl vereinfachen, da nur noch das passende Objekt aufgerufen werden muss. Außerdem müssten bei Anpassungen nicht alle Switch-Anweisungen angepasst werden.

Es wurden den Regeln folgende Metriken zugewiesen.

Tabelle 45: Zugehörige Metriken zur Strategy-Kandidaten-Identifikation

Metrik	Beschreibung	Grenzwert	Anwendung in Regel
M1.1 Anzahl an Case-Bedingungen	Eine Methode muss eine gewisse Anzahl an zusammenhängenden Case-Bedingungen besitzen.	≥2	R1.5
M1.2 Anzahl an Methoden	Die gleiche Abfrage der Bedingungen muss in mehreren Methoden vorkommen.	≥2	R1.5
M1.3 Gemeinsame Klasse	Die Methoden mit der identischen Abfrage müssen in der gleichen Klasse liegen.	True	R1.5
M1.4 Gleiche Anzahl an Bedingungen	Alle Switch-Anweisungen müssen die genau gleiche Anzahl an Case-Bedingungen aufweisen.	True	R1.5
M1.5 Übereinstimmende Bedingungen	Neben der gleichen Anzahl müssen auch die Case-Bedingungen zu 100% übereinstimmen.	True	R1.5
M1.6 Gleiches Attribut / Parameter im Kopf	Alle ausgewählten Abfragen müssen das gleiche Attribut oder den gleichen Parameter als Bedingungsvariable im Kopf der Switch-Anweisung verwenden.	True	R1.5

5.5.2 Grenzwerte

Zur Erkennung und Bewertung entsprechender Kandidaten werden Grenzwerte für die genannten Metriken benötigt. Der folgende Abschnitt beschreibt die Auswertung und die Ermittlung dieser Grenzwerte aus dem Merobase-Index und das daraus entwickelte Erkennungsmodell.

Die Suche in der Merobase nach bereits implementierten Strategy Patterns basiert auf folgendem Suchstring:

*I*Strategy lang:java type:interface*

(protocol:svn OR protocol:CVS)

Der Suchstring basiert auf der Annahme, dass alle Strategien, die nach dem Muster der GoF implementiert wurden, ein Interface als Oberklasse verwenden. Des Weiteren wird davon ausgegangen, dass diese Oberklasse das Wort ‚Strategy' im Klassennamen beinhaltet. Außerdem

gilt die Annahme, dass alle Interfaces in Java sich an die bekannten Namenskonventionen[6] halten und mit ‚I' beginnen.

Festlegung der Grenzwerte

Die Suche innerhalb der Merobase lieferte 286 Ergebnisse. Daraus wurden die ersten 53 korrekt implementierten Strategy Patterns ausgewählt. Diese Auswahl stammt aus 35 verschiedenen Projekten. Innerhalb dieser Strategy Patterns wurden 182 konkrete Strategien bzw. Algorithmen implementiert.

Für die Festlegung der Grenzwerte für die Erkennungsregel R1.1 wurden zwei Merkmale ausgewählt, die in den vorliegenden Implementierungen gemessen wurden. Das erste Merkmal ist die Anzahl konkreter Strategien, die ein Pattern implementiert. Dieses Merkmal stellt den Grenzwert dar, ab wann eine identifizierte Quellcodestelle genug Algorithmen besitzt, damit ein Strategy Pattern effektiv eingesetzt werden kann. Das zweite Merkmal identifiziert die Anzahl implementierter Methoden pro Interface und somit die Anzahl an möglichen Operationen auf den Strategien. Bei der Erkennung wird dieser Wert benötigt, um festzulegen, in wie vielen Methoden die Algorithmen-Auswahl vorhanden sein sollte.

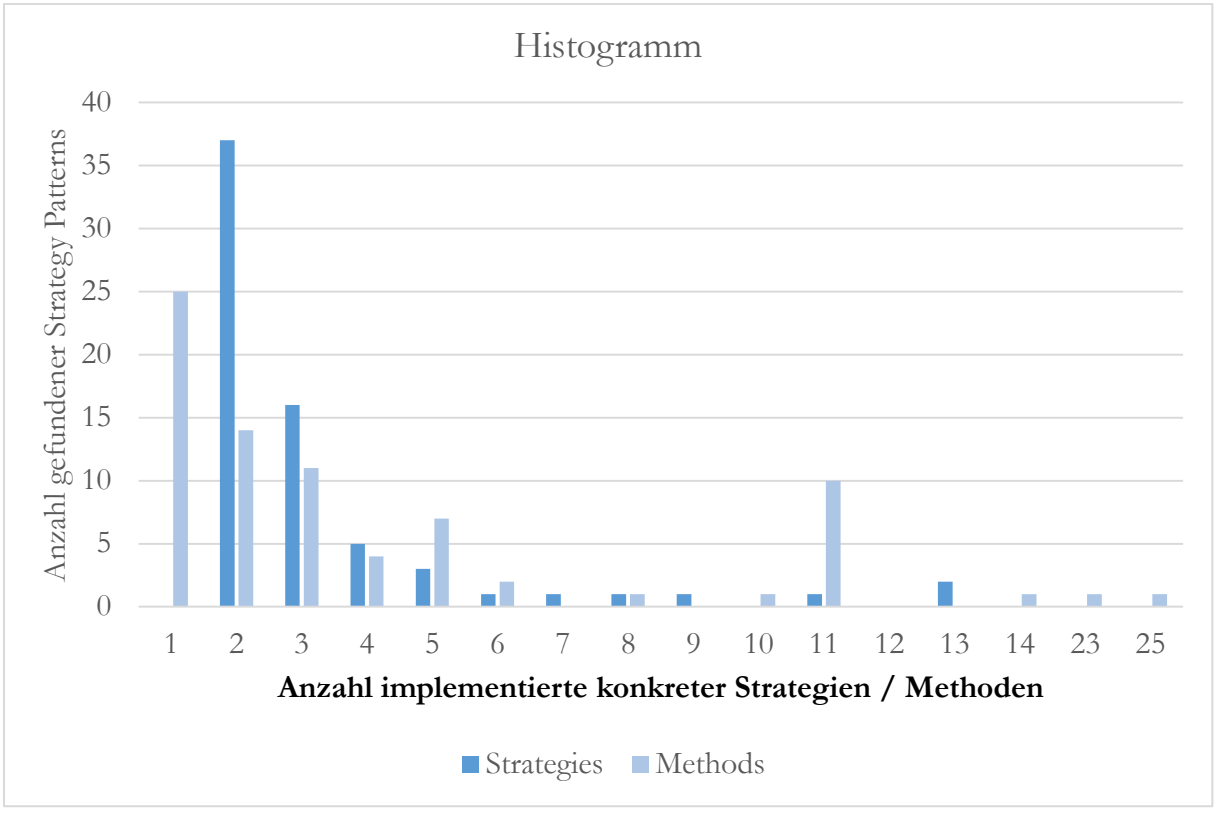


Abbildung 35: Übersicht der ausgewählten konkret implementierten Strategien und Methoden aus dem Merobase-Index

Aus den Messungen ergibt sich, dass ein Strategy Pattern durchschnittlich 3,4 Algorithmen oder konkrete Strategien implementiert. Die Auswertung der Methoden des Interfaces ergab, dass ein solches Pattern den konkreten Strategien durchschnittlich drei Methoden vorgibt. Die nachfolgende Tabelle 46 zeigt die statistisch interessanten Werte der analysierten Strategy Patterns.

[6] https://wiki.eclipse.org/Naming_Conventions#Classes_and_Interfaces

Tabelle 46: Statistische Werte der ausgewählten Strategy Patterns

	Strategien	Methoden
Min	2,00	1,00
1. Quartil	2,00	1,00
Median	2,00	2,00
Mittelwert	3,29	3,37
3. Quartil	3,00	4,00
Max	13,00	25,00

Aus den vorliegenden Daten ergeben sich folgende Grenzwerte, basierend auf den Mittelwerten. Damit eine Klasse als Strategy Pattern-Kandidat erkannt wird, sollte sie mindestens zwei Methoden, die mindestens drei identische Switch-Anweisungen implementieren, oder drei Methoden mit zwei Switch-Anweisungen beinhalten. Daraus wurde das folgende Modell in Tabelle 47 erarbeitet.

Tabelle 47: Erkennungsstufen und Grenzwerte für einen Strategy-Kandidaten

M1.1	M1.3	Erkennungsstufe
≥2	≥2	Möglich
≥3	≥3	Sinnvoll
≥4	≥4	Empfohlen

Sobald beide Merkmale den Wert von 2 oder größer besitzen und die anderen Metriken erfüllt sind, ist eine Quellcodestelle erkannt, bei der der Einsatz eines Strategy Patterns möglich (M) wäre. Besitzt der Kandidat mindestens 3 identische Cases (und dies in 3 Methoden), ist der Kandidat als *sinnvoll* (S) einzustufen. Sind beide größer gleich 4, wird der Einsatz des Patterns *empfohlen* (E). Tabelle 48 verdeutlicht das Modell graphisch.

Tabelle 48: Erkennungsmodell für Strategy-Kandidaten (M=Möglich, S=Sinnvoll, E=Empfohlen)

		Metrik M1.3								
Metrik M1.1	**9**		M	S	E	E	E	E	E	E
	8		M	S	E	E	E	E	E	E
	7		M	S	E	E	E	E	E	E
	6		M	S	E	E	E	E	E	E
	5		M	S	E	E	E	E	E	E
	4		M	S	E	E	E	E	E	E
	3		M	S	S	S	S	S	S	S
	2		M	M	M	M	M	M	M	M
	1									
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

5.5.3 Beispielerkennung

Im nachfolgenden Abschnitt wird der Erkennungsalgorithmus mit Hilfe des zuvor genannten Praxisbeispiels im Detail diskutiert. Zur Erkennung des Kandidaten für ein Strategy Pattern werden sechs Schritte durchlaufen (siehe Abbildung 36). Die Datenmenge wird dabei Schritt für Schritt reduziert, bis nur noch die Kandidaten übrigbleiben. Die Extraktion der Daten in Schritt 1 wird für alle Erkennungsregeln gemeinsam durchgeführt, um die Laufzeit der Erkennung zu minimieren. Die Schritte 2 bis 6 werden für jeden möglichen Kandidaten separat ausgeführt, so dass z. B. jede Klasse, die in Schritt 2 identifiziert wird, einzeln in Schritt 3 weiterverarbeitet wird.

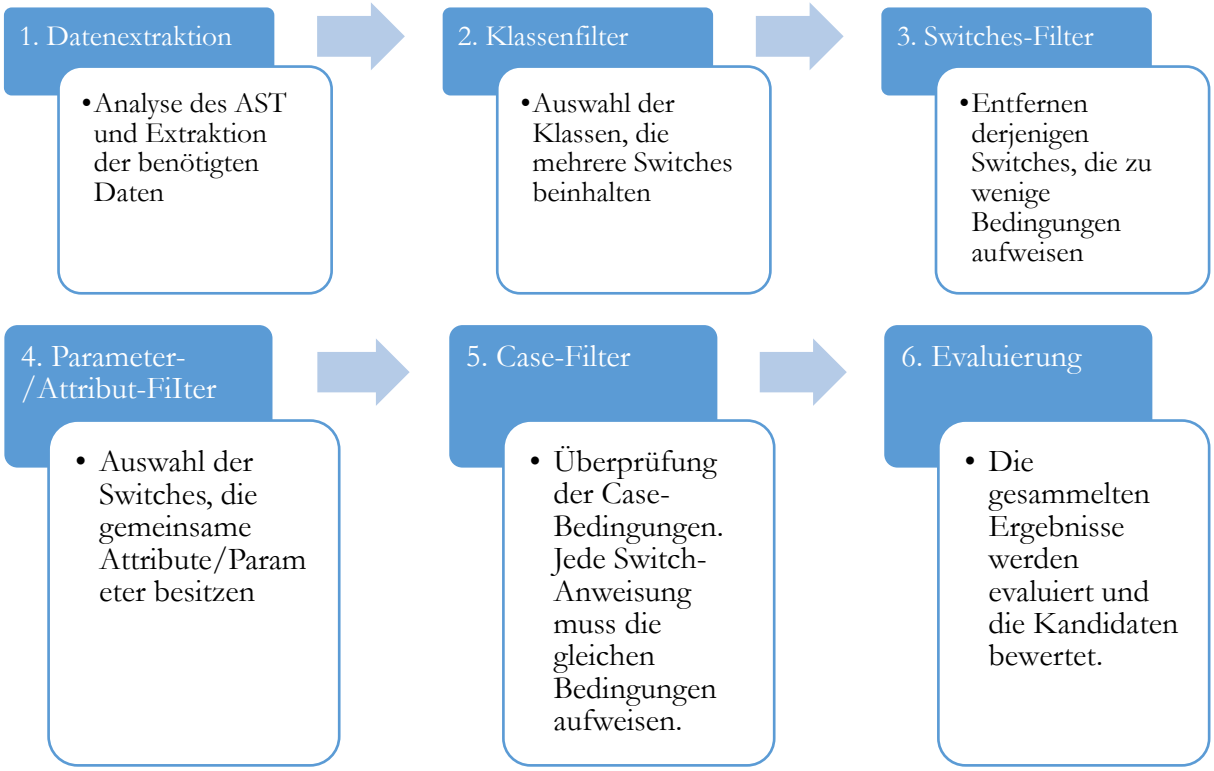


Abbildung 36: Detailprozess zur Erkennung von Strategy Pattern-Kandidaten

Die folgenden Snippets aus dem Projekt *JEdit* (Version 5.1) zeigen die Klasse *TextUtilities,* welche zwei Methoden besitzt (*findWordEnd, findWordStart*).

Das erste Snippet zeigt die Methode *findWortStart*. Mit der Hilfe der Variable *ch* und dem Parameter *noWordSep* wird die Variable *type* initialisiert. Diese dient dann der Switch-Anweisung als Bedingung.

```
public static int findWordStart(CharSequence line, int pos, String noWordSep,
                boolean joinNonWordChars, boolean camelCasedWords,
                boolean eatWhitespace, boolean eatOnlyAfterWord) {
                char ch = line.charAt(pos);
                if(noWordSep == null)
                        noWordSep = "";
                int type = getCharType(ch, noWordSep);
                for(int i = pos; i ≥ 0; i--) {
                        char lastCh = ch;
                        ch = line.charAt(i);
                        switch(type) {
                        case WHITESPACE:
                                if(Character.isWhitespace(ch))
                                        break;
                                else if (!eatOnlyAfterWord)    {
                                        return i + 1;
                                }
                                else if (Character.isLetterOrDigit(ch) | |
                                        noWordSep.indexOf(ch) != -1){
                                        type = WORD_CHAR;
                                }
                                else
                                        type = SYMBOL;
                                break;
                        case WORD_CHAR:
                                if (camelCasedWords && Character.isUpperCase(ch)
                                        && !Character.isUpperCase(lastCh)
                                        && Character.isLetterOrDigit(lastCh)) {
                                        return i;
                                }
                                else if (camelCasedWords && !Character.isUpperCase(ch)
                                        && Character.isUpperCase(lastCh)) {
                                        return i + 1;
                                }
                                else if(Character.isLetterOrDigit(ch) | |
                                        noWordSep.indexOf(ch) != -1) {
                                        break;
                                }
                                else if(Character.isWhitespace(ch)
```

Das zweite Snippet zeigt die Methode *findWortEnd*. In Übereinstimmung mit der Methode zuvor wird die Variable *type* initialisiert und als Bedingung im Switch verwendet.

```
public static int findWordEnd(CharSequence line,
                                int pos,
                                String noWordSep,
                                boolean joinNonWordChars,
                                boolean camelCasedWords,
                                boolean eatWhitespace) {
                if(pos != 0)
                        pos--;
                char ch = line.charAt(pos);
                if(noWordSep == null)
                        noWordSep = "";
                int type = getCharType(ch, noWordSep);
                for(int i = pos; i < line.length(); i++) {
                        char lastCh = ch;
                        ch = line.charAt(i);
                        switch(type) {
                        case WHITESPACE:
                                if(Character.isWhitespace(ch))
                                        break;
                                else
                                        return i;
                        case WORD_CHAR:
                                if (camelCasedWords && i > pos + 1
                                        && !Character.isUpperCase(ch)
                                        && Character.isLetterOrDigit(ch)
                                        && Character.isUpperCase(lastCh)) {
                                        return i - 1;
                                } else if (camelCasedWords && Character.isUpperCase(ch)
                                        && !Character.isUpperCase(lastCh)) {
                                        return i;
                                } else if(Character.isLetterOrDigit(ch) | |
                                        noWordSep.indexOf(ch) != -1) {
                                        break;
                                } else if(Character.isWhitespace(ch)
                                        && eatWhitespace) {
                                        type = WHITESPACE;
                                        break;
                                } else
                                        return i;
```

```
                                && eatWhitespace && !eatOnlyAfterWord) {
                                type = WHITESPACE;
                                break;
                        }
                        else
                                return i + 1;
                case SYMBOL:
                        if(!joinNonWordChars && pos != i)
                                return i + 1;
                        if(Character.isWhitespace(ch)) {
                                if(eatWhitespace && !eatOnlyAfterWord){
                                        type = WHITESPACE;
                                        break;
                                } else
                                        return i + 1;
                        } else if(Character.isLetterOrDigit(ch) ||
                                noWordSep.indexOf(ch) != -1)
                        {
                                return i + 1;
                        } else {
                                break;
                        }
                }
        }
        return 0;
}
```

```
                case SYMBOL:
                        if(!joinNonWordChars && i != pos)
                                return i;.
                        if(Character.isWhitespace(ch)) {
                                if(eatWhitespace)    {
                                        type = WHITESPACE;
                                        break;
                                } else
                                        return i;
                        }
                        else if(Character.isLetterOrDigit(ch) ||
                                noWordSep.indexOf(ch) != -1) {
                                return i;
                        } else {
                                break;
                        }
                }
        }
        return line.length();
}
```

Beide Methoden implementieren eine Switch-Anweisung mit drei Cases, wobei immer die 3 gleichen Bedingungen verwendet werden, WHITESPACE, WORD_CHAR und SYMBOL (siehe Abbildung 37). Die Rümpfe der Cases unterscheiden sich in beiden Methoden.

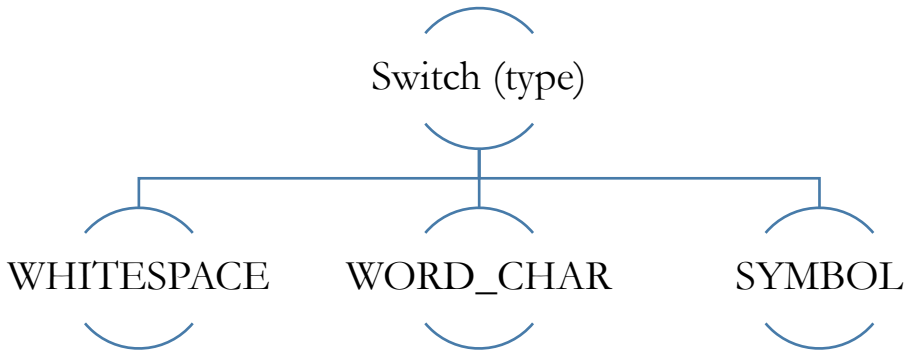


Abbildung 37: Graphische Darstellung der potenziellen Strategien aus dem Kandidaten

In diesem Abschnitt werden nur die Schritte zur Extraktion von Switch-/Case-Knoten aus dem AST diskutiert. Die allgemeine Beschreibung des Extraktionsprozesses von Informationen und AST werden in Kapitel 4.5.2 erklärt.

Ein Switch wird unter PMD im AST als *ASTSwitchStatement*-Knoten identifiziert. Erreicht der PMD-Visitor einen solchen Knoten, beginnt der Extraktionsprozess. Zuerst werden die allgemeinen Informationen zur Identifikation der Position gesammelt. Dazu gehören Name und Parameterliste der Methode, in der das Switch liegt, der zugehörige Klassenname und der Package-Name. Als nächstes wird die genutzte Variable aus dem Switch-Kopf ausgelesen. Sie liegt in einem *ASTName*-Knoten unter dem Knoten *ASTExpression*. Dieser Knoten wird darauf analysiert, ob es sich um ein Attribut, eine Variable oder einen Methodenaufruf handelt. Bei Attributen werden die Keywords „super“ und „this“ aus dem Namen entfernt. Daraufhin werden die einzelnen Cases aus dem AST geholt. Jedes Case wird als *ASTSwitchLabel* repräsentiert. Hierbei sind die Gesamtanzahl der Knoten und die Bedingung der einzelnen Cases wichtig.

Alle gewonnenen Daten werden für die Verwendung in späteren Schritten in einer Datenbank abgelegt. Dabei werden diverse Querverbindungen zu Attributen, Parametern und Methoden gezogen. Die folgende Tabelle 49 zeigt ein Teilergebnis aus der Analyse der Beispielklasse.

Tabelle 49: Anzahl des Auftretens einer Methode und Variable in Verbindung mit einem Switch

ID	Name	Anzahl des Auftretens	ID der Methode	ID der Attribute
1	SWITCH	3	1	7
2	SWITCH	3	18	7

Nun werden die gesammelten Informationen analysiert. Dazu wird Schritt für Schritt die Datenmenge verkleinert, indem Fälle entfernt werden, die keine Kandidaten ergeben können. Zuerst werden die Klassen identifiziert, die mehr als eine Switch-Anweisung besitzen und somit über dem untersten Grenzwert liegen. Tabelle 50 zeigt die Anzahl der Switch-Anweisungen pro Klasse aus dem Beispiel.

Tabelle 50: Anzahl der Switch-Anweisungen pro Klasse

Anzahl der Switches	Klassen-ID
3	// *TextUtilities* **2**
4	4
3	7
2	8

Die Klasse *TextUtilities* wird durch die ID 2 repräsentiert. Wie der Auszug zeigt, besitzt die Klasse zwei Switch-Anweisungen.

In diesem Schritt werden alle Switches in der Klasse mit ID 2 entfernt, die weniger als 3 Case-Anweisungen implementieren. Des Weiteren werden die Switches nach der Anzahl ihrer Cases gruppiert. Diese Gruppen sind nötig, um die verschiedenen Switches in einer Klasse unterscheiden zu können. Das Ergebnis ist in Tabelle 51 dargestellt.

Tabelle 51: Case-Anweisungen, die identisch sind und in mehreren Klasse vorkommen

Anzahl der Cases	Vorkommen in Klasse	Klassen-ID
3	2	2

Die Klasse besitzt zwei Switch-Anweisungen, und jede davon beinhaltet drei Case-Bedingungen.

Aus der Teilmenge von Schritt 3 werden nun die möglichen Kandidaten herausgefiltert, die ein gemeinsames Attribut oder einen gemeinsamen Parameter nutzen. Die Verbindung zwischen Switch-Kopf-Variable und -Attribut bzw. -Parameter wurde bei Schritt 1 gesammelt. Der Vergleich der Parameter, die im Switch-Kopf verwendet wurden, zeigt eine Übereinstimmung . Es wird ein Parameter in mehreren Switch-Anweisungen und mehreren Klassen verwendet wurde (siehe Tabelle 52). Bei den Attributen wird ein Attribut in zwei Switch-Anweisungen implementiert.

Tabelle 52: Attribut, das in mehreren Switch-Anweisungen genutzt wurde

Anzahl gemeinsame Nutzung	Attribut-ID
2	7

Wie in der oben gezeigten Tabelle veranschaulicht, wird das Attribut 7 (Name *type* vom Typ *int*) innerhalb der Klasse 2 von 2 Switches verwendet, deren Anzahl an Case-Fällen 3 ist.

Für die verbleibenden Switch-Anweisungen müssen nun die Case-Knoten überprüft werden, damit immer die exakt gleichen Bedingungen verwendet werden. Die Reihenfolge ihres Auftretens spielt bei diesem Vergleich keine Rolle. Die Namen werden aus der Datenbank abgefragt und mittels einer *HashMap* verglichen. Im Falle von Switch-ID 1 und 2 sind diese gleich (vgl. Tabelle 53).

Tabelle 53: Vergleich der Case-Anweisungen der beiden Switches

Switch-ID 1	Switch-ID 2
WHITESPACE	WHITESPACE
WORD_CHAR	WORD_CHAR
SYMBOL	SYMBOL

Evaluierung

Die Datensätze, die aus der Gesamtmenge von Schritt 2 übrig sind, stellen die gefundenen Kandidaten dar und werden vom Programm in die Ergebnismenge mit anderen Kandidaten aufgenommen. Da es sich nur um zwei Switch-Anweisungen mit drei identischen Cases handelt, wird der Kandidat als *möglich* eingestuft.

5.6 State Pattern

Die Verwendung und richtige Implementierung einer *Finite State Machine* (FSM) ist in der Praxis, durch ihren speziellen Anwendungsfall, eher selten. Das folgende Beispiel zeigt den idealen Kandidaten für ein State Pattern, da in den verwendeten Projekten kein entsprechender Kandidat gefunden wurde. Es basiert auf dem Beispiel des Buches „Head First“ von Freeman et al. (Freeman, Robson, Bates, & Sierra, 2004) und wird in Kapitel 3.6 im Detail diskutiert. In Abbildung 38 wird die FSM des Beispiels dargestellt.

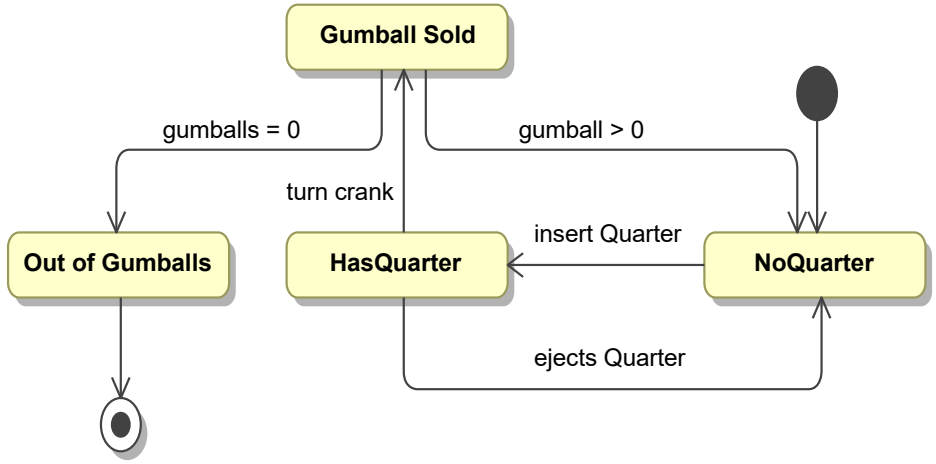


Abbildung 38: State-Diagramm der FSM eines Kaugummi-Automaten

5.6.1 Unterschied zwischen Strategy und State-Erkennungsregel

Die Unterschiede im strukturellen Aufbau und dem Verhalten zwischen einem Strategy- und einem State-Entwurfsmuster sind minimal. Bei ihren grundlegenden Verhalten liegt die größte Differenz in den Beziehungen zwischen den zur Verfügung stehenden Strategien bzw. States. Während beim Strategy Pattern die einzelnen Auswahlmöglichkeiten völlig unabhängig voneinander sind, besitzen sie bei den State Patterns immer eine Abhängigkeit untereinander, die die notwendigen Zustandsübergänge abbildet.

Zwangsläufig gibt es eine Nähe in den Erkennungsregeln, welche im folgenden Kapitel im Detail betrachtet werden. Der Unterschied bei der Erkennung ist minimal. Dadurch sind einige Schritte in der Erkennung gleich. Auf Grund dieser Verwandtschaft der beiden Patterns werden sie beide zusammen diskutiert. Es ist zu beachten, dass jeder State-Pattern-Kandidat ebenfalls ein Strategy-Kandidat sein könnte.

Einige Pattern-Mining-Studien kombinieren aus diesem Grund ihre Ergebnisse für dieses Problem. Die Erkennungsregel für das State Pattern ist eine Spezialisierung der Strategy-Regel. Die State-Erkennungsregel wurde entworfen, um State-Kandidaten von Strategy-Kandidaten zu unterscheiden. Sollte ein State-Kandidat nicht einwandfrei erkannt werden können, bleibt der Kandidat ein Strategy-Kandidat. Wie bei allen anderen Erkennungsregeln bleibt es dem Entwickler überlassen, zu entscheiden, ob und welches Pattern verwendet werden sollte.

5.6.2 Erkennungsregel

Die Regel zum Identifizieren von State Pattern-Kandidaten wurde wie folgt definiert.

R1.6 In mehreren Methoden gibt es eine lange Switch-Anweisung, welche immer die gleichen Fälle in den Case-Bedingungen abfragt und für diese Abfrage immer die/den gleiche(n) Variable/Parameter im Switch-Kopf verwendet. Diese(r) Variable/Parameter wird auch im Body jeder Case-Bedingung verändert.

Eine komplexe Switch-Anweisung ist der erste Indikator für einen State-Kandidaten. Hinter solchen Strukturen können sich unterschiedliche Programmverhalten verbergen, die je nach Zustand einer Variable unterschiedlich ausgeführt werden. Diese Art von Kontrollstrukturen wird explizit für eine einfache Verhaltenssteuerung genutzt, jedoch sollten sie nicht zu komplex und somit unübersichtlich werden.

Für einen State-Kandidaten müssen mehrere solcher Switch-Anweisungen existieren. Jede dieser Anweisungen implementiert die gleiche Verhaltensauswahl, indem alle die gleiche Anzahl und Zusammensetzung von Case-Bedingungen haben. Erst das mehrfache Auffinden innerhalb einer Klasse bestätigt einen Kandidaten. Es wird davon ausgegangen, dass eine Verhaltensauswahl immer nur in der gleichen Klasse implementiert wird, nie über das gesamte Programm verstreut. Die letzte Bedingung für einen Kandidaten ist, dass die gleiche Variable im Switch verwendet wird. Diese Variable steht für den Zustandswechsel. Je nach Inhalt der Variable wird ein anderes Verhalten ausgeführt. Die verwendeten Metriken sind wie folgt festgelegt worden.

Tabelle 54: Übersicht über die verwendeten Metriken zur State-Kandidaten-Erkennung

Metrik	Beschreibung	Grenzwert	Anwendung in Regel
M1.1 Anzahl Bedingungen	Eine Methode muss eine gewisse Anzahl an zusammenhängenden Case-Bedingungen besitzen.	>2	R1.6
M1.2 Anzahl Methoden	Die Abfrage der Bedingungen muss in mehreren Methoden vorkommen.	>2	R1.6
M1.2.1 Übereinstimmende Bedingungen	Die Case-Bedingungen müssen übereinstimmen.	True	R1.6
M1.2.12 Gleiche Anzahl Bedingungen	Alle Methoden müssen die genau gleiche Anzahl an Bedingungen aufweisen.	True	R1.6
M1.3 Gemeinsame Klasse	Die Methoden mit dem Switch müssen sich in der gleichen Klasse befinden.	True	R1.6
M1.5 Gleiches Attribut / Parameter im Kopf	Alle ausgewählten Abfragen müssen das gleiche Attribut oder den gleichen Parameter als Bedingungsvariable im Kopf der Switch-Anweisung verwenden.	True	R1.6
M1.6 Gleiches Attribute / Parameter wird innerhalb des Bedingungsbodys	Die in M1.5 verwendete Variable muss auch in den Bodys aller Case-Bedingungen verändert werden.	True	R1.6

5.6.3 Grenzwerte

Bei der Festlegung des Grenzwertes für einen State Pattern-Kandidaten wurde analog zu den anderen Grenzwerten vorgegangen. Die entsprechenden Schritte werden nachfolgend im Detail beschrieben.

Für die Merobase-Suche wurde der folgende Suchstring verwendet.

*I*State lang:java type:interface (protocol:svn OR protocol:CVS)*

Vergleichbar mit dem Suchstring des Strategy Patterns wurde dieser String erstellt, um alle Interfaces zu finden, die das Wort ‚State' beinhalten. Diese Suche impliziert wieder, dass viele Interfaces in Java mit dem Buchstaben ‚I' beginnen. Außerdem verwendet nach GoF eine State Pattern-Implementierung ein Interface für den Grundzustand oder Basiszustand.

Festlegung der Grenzwerte

Eine auf dem Suchstring basierende Analyse lieferte 801 mögliche State Patterns. Aus dieser Menge wurden 38 Implementierungen aus 37 verschiedenen Open Source-Projekten ausgewählt. Wie beim Facade-Pattern schwankte die Qualität der Implementierungen sehr, so dass eine geringe Zahl an Implementierungen ausgewählt wurde. Die verwendeten Merkmale zur Festlegung des Grenzwertes wurden analog zum Strategy Pattern verwendet. Es handelt sich um die Anzahl der implementierten States und die Anzahl der Methoden pro Interface. Abbildung 39 und Abbildung 40 zeigen die Verteilung der gefundenen Ergebnisse pro Merkmal.

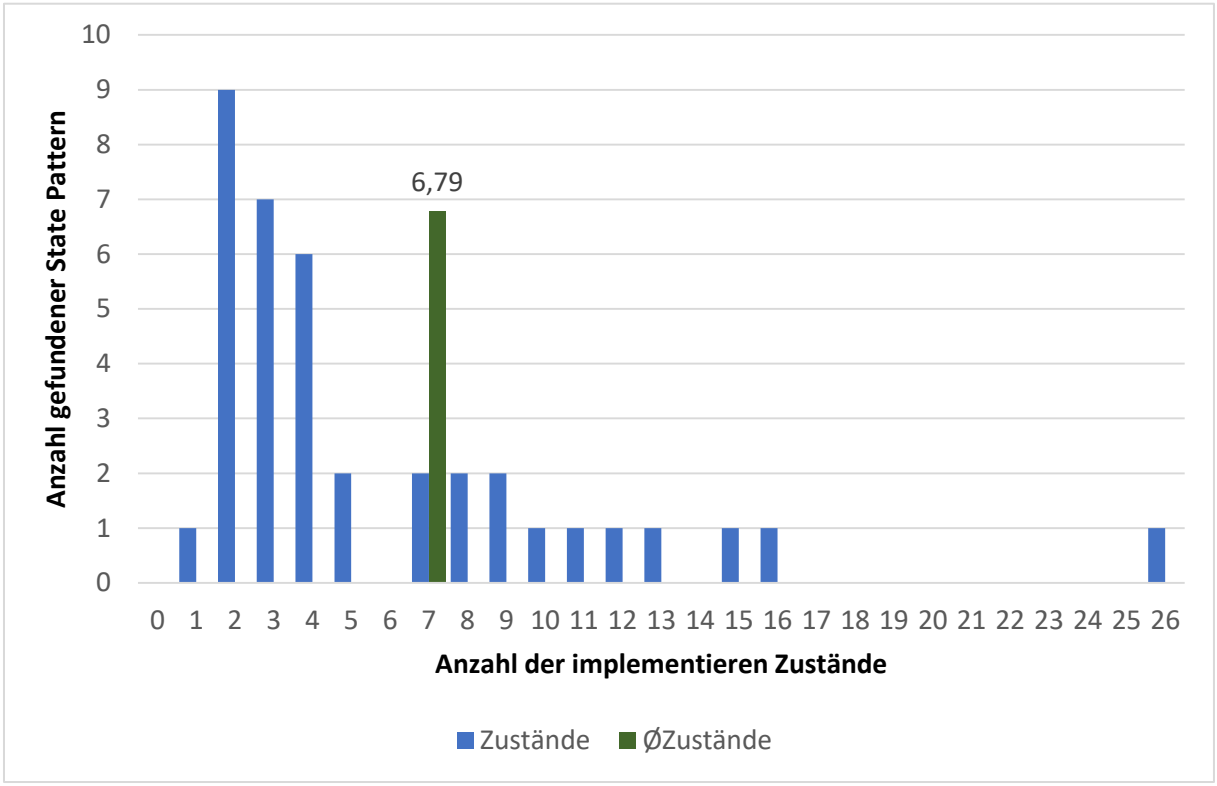


Abbildung 39: Häufigkeit der Anzahl an Zuständen pro untersuchtem State Pattern in der Merobase

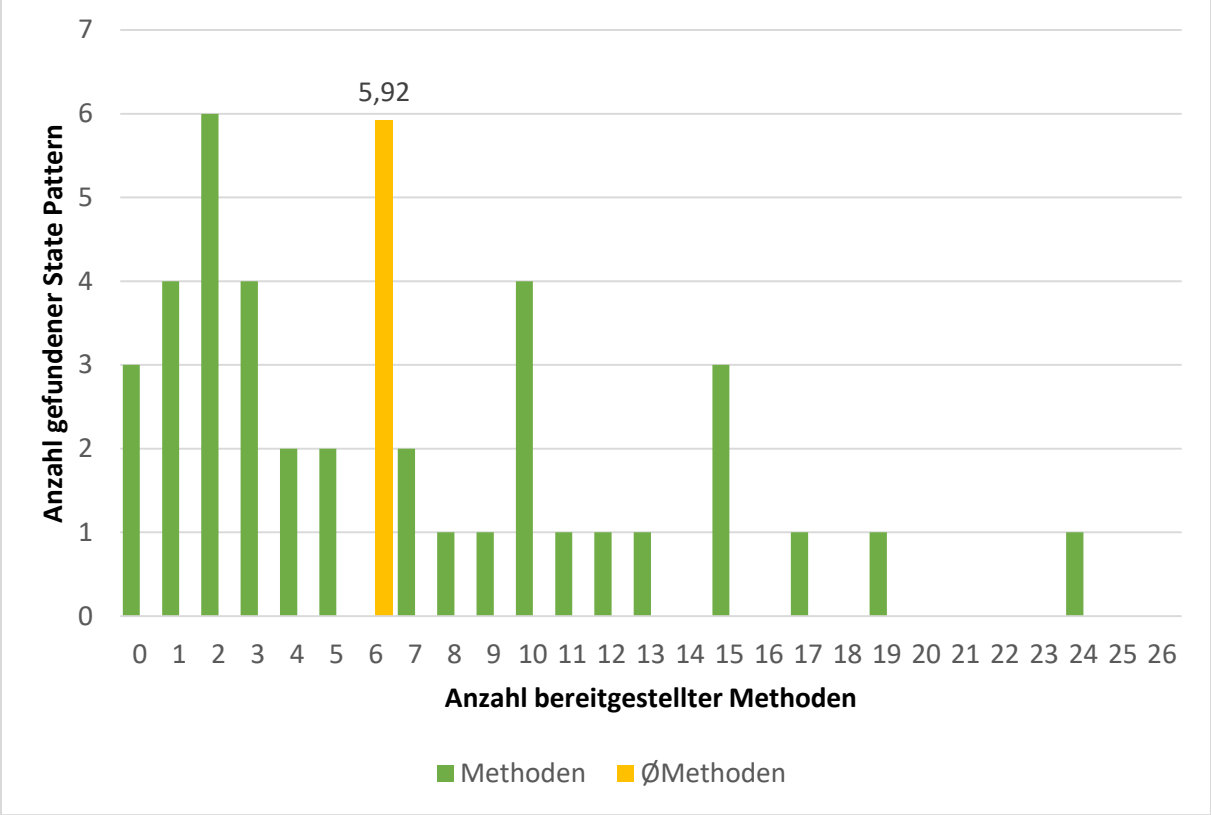


Abbildung 40: Häufigkeit der Anzahl an Methoden pro untersuchtem State Pattern in der Merobase

Ausgehend von diesen Daten ergeben sich folgende Mittelwerte: In 38 Design Patterns wurden im Schnitt 6,7 Zustände implementiert. In der Analyse gab es nur ein Pattern mit einem Zustand und ebenfalls nur ein Vorkommen mit 26 Zuständen. Im Mittel wurden 5,9 Methoden implementiert. In diesem Fall existierte ebenfalls ein Interface, das keine Methoden zur Verfügung stellte. Im Maximum werden 24 Methoden bereitgestellt. Die nachfolgende Tabelle 55 zeigt die statistische Verteilung der Ergebnisse.

Tabelle 55: Statistische Verteilung der Merobase-Ergebnisse für das State Pattern

	Methoden	Zustände
Minimum	0	1
1. Quartil	2	2,25
Median	4,5	4
Mittelwert	5,9	6,7
3. Quartil	10	8
Maximum	24	26

Die Auswertung der gesammelten Ergebnisse führte zu folgenden Grenzwerten: Für ein State Pattern sollten mindestens 3 Zustände vorhanden sein. Des Weiteren sollten für diese Zustände mindestens 3 Methoden bereitgestellt werden. Sind beide Grenzwerte (siehe Tabelle 56) erreicht, so wird ein State Pattern als *möglich* angesehen. Entsprechend wurde das Modell erarbeitet, das die Grenzwerte in die bekannten drei Erkennungsstufen einteilt.

Tabelle 56: Grenzwerte und Erkennungsstufen für das State Pattern

M1.1 Grenzwert	M1.2 Grenzwert	Erkennungsstufe
≥3	≥3	Möglich
≥5	≥4	Sinnvoll
≥7	≥6	Empfohlen

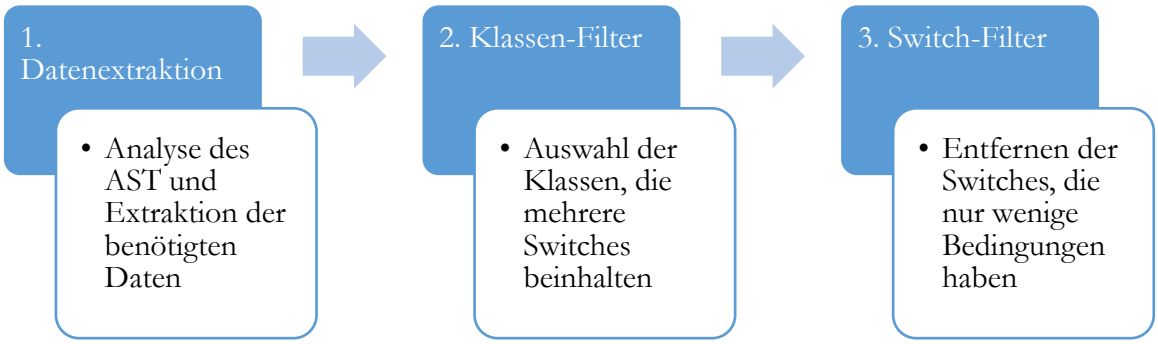
Sobald eine Klasse die Anzahl von mindestens 6 Methoden und 7 Zuständen aufweist, wird der Einsatz eines State Patterns empfohlen. Diese Erkennungsstufe basiert auf den Mittelwerten, wie es bei den Erkennungsregeln zuvor auch der Fall war. Sinnvoll erscheint ein Pattern, wenn 5 Zustände und 4 Methoden gefunden werden. Tabelle 57 fasst das Modell graphisch zusammen.

Tabelle 57: Erkennungsmodell für das State Pattern (M=Möglich, S=Sinnvoll, E=Empfohlen)

	Metrik M1.1									
Metrik M1.2	**9**			M	M	S	S	E	E	E
	8			M	M	S	S	E	E	E
	7			M	M	S	S	E	E	E
	6			M	M	S	S	E	E	E
	5			M	M	S	S	S	S	S
	4			M	M	S	S	S	S	S
	3			M	M	M	M	M	M	M
	2									
	1									
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

5.6.4 Beispielerkennung

Die ersten drei notwendigen Schritte zur Erkennung eines State-Kandidaten sind identisch mit denen für die Identifikation eines Strategy-Kandidaten. Aufbauend auf den so gewonnenen Informationen, werden drei weitere Schritte durchgeführt, die nur für die State-Kandidaten-Erkennung notwendig sind. Abbildung 41 illustriert die Schritte.



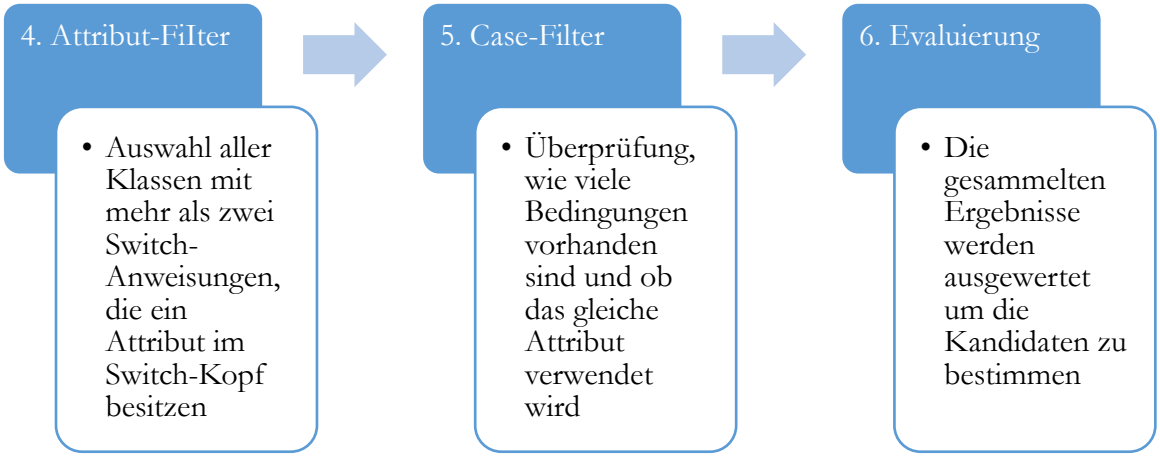


Abbildung 41: Detailprozess zur Erkennung von State Pattern-Kandidaten

Die Extraktion der Daten für einen State-Pattern-Kandidaten basiert auf der Extraktion der Strategy Pattern-Erkennung aus Kapitel 5.5. Wie bei der Analyse zur Erkennung von Strategy-Pattern-Kandidaten werden *ASTSwitchStatement*-Knoten gesucht, welche die Switch-Anweisungen repräsentieren und eine gewisse Menge an Cases (repräsentiert durch *ASTSwitchLabel*-Knoten) besitzen. Persistiert werden die Daten in den gleichen Tabellen wie beim Strategy Pattern.

In diesem Schritt werden noch die Informationen über die verwendeten Attribute einer Klasse gesammelt. Ein für die Analyse interessantes Attribut zeichnet sich dadurch aus, dass das Attribut zu einer Klasse innerhalb des untersuchen Projektes gehört. Solche Verbindungen fallen unter den *Type Methode Invocation* und werden im AST mit dem Knoten *ASTPrimaryExpression* dargestellt.

Wie Schritt 1 basieren auch der zweite und der dritte Schritt auf der Strategy-Pattern-Erkennung (siehe 5.5.1).

Ab dem Schritt vier unterscheidet sich das Vorgehen zur Erkennung bei beiden Patterns. Es werden aus der Tabelle die Switches ausgewählt, die mehr als zweimal in der gleichen Klasse vorkommen und das gleiche Attribut im Switch-Head verwenden. Im Falle der Klasse *GumballMachine* mit der ID 1 wurde ermittelt, dass es 5 Switches gibt, die alle das Attribut mit der ID 7 verwenden. Das Ergebnis zeigt Tabelle 58.

Tabelle 58: Ergebnis des Attribut-Filters

Anzahl	Name des Attributes	Attribut-ID	Klassenname	Klassen-ID
5	state	7	GumballMachine	1

Im folgenden Schritt geschieht ein Abgleich der verwendeten Attribute innerhalb der Cases mit dem Attribut im Switch-Kopf. Zuerst werden alle Attribute erfasst, die innerhalb der Methoden verwendet wurden. Bei jeder Switch-Anweisung und jedem Case-Knoten ist bekannt, in welcher Zeile sie beginnen und in welcher sie enden. Mit Hilfe dieser Information wird nun festgestellt, ob ein Attribut innerhalb jedes Cases verwendet wird. Für das Switch mit der ID 3 ergaben sich folgende Cases (Tabelle 59) und Attribute (Tabelle 60).

Tabelle 59: Ergebnis des Case-Filters angewandt auf die Daten von Schritt 1

Case-ID	Case-Anweisung	Startzeile	Endzeile	Switch-ID
11	HAS_QUARTER	90	93	3
12	NO_QUARTER	94	97	3
13	SOLD_OUT	98	101	3
14	SOLD	102	111	3

Tabelle 60: Ergebnis der Attributanalyse für die Switch-Anweisungen aus Schritt 1

Methodenaufruf	Attribut-ID	Zugehörige Methode	In Klasse	Zeile
state.setState	7	5	1	92
state.setState	7	5	1	95
state.setState	7	5	1	99
state.setState	7	5	1	107
state.setState	7	5	1	109

Evaluierung

Es wurden fünf identische Switch-Anweisungen in fünf Methoden in der Klasse *GumballMachine* identifiziert. Jede dieser Switch-Anweisungen verwendet in ihren Case-Knoten die Werte HAS_QUARTER, SOLD, NO_QUARTER und SOLD_OUT. Ein Vergleich mit den Grenzwerten ergibt somit einen State Pattern-Kandidaten, der die Erkennungsstufe *möglich* erhält.

5.7 Weitere Design Pattern-Regeln

Auf Grund des hohen Aufwandes standen nur die sechs zuvor genannten Design Patterns im Fokus einer genauen Analyse und nur für diese konnten Erkennungsregeln implementiert werden. Im Zuge der Pattern-Analyse wurden noch für andere GoF-Patterns Ideen zur Erstellung von Erkennungsregeln gesammelt. Die nachfolgenden Abschnitte beschreiben kurz die Design Patterns und die zugehörigen Erkennungsideen. Dabei handelt es sich ausschließlich um Ideen oder Skizzen, d. h. es wurden keine Metriken oder Grenzwerte festgelegt, geschweige denn Analysen im Quellcode auf Basis dieser Ideen durchgeführt. Alle folgenden Regelskizzen basieren auf den Definitionen von Gamma (Gamma, Helm, Johnson, & Vlissides, 1994).

5.7.1 Factory Method

Dieses Pattern stellt ein Interface zur Verfügung mit dem Objekte erzeugt werden können. Dabei entscheidet jedoch das Objekt wie sein genauer Aufbau, z.B. verwendete Unterobjekte und Startwerte, sein soll. So trennt das Pattern zwischen Erzeuger und Implementierung. Des Weiteren verwendet das Factory-Method-Pattern den Mechanismus der Vererbung. Objekte, die über ein solches Pattern erzeugt werden, sind somit immer miteinander verwandt und implementieren ähnliche Aufgaben, z.B. eine Anbindung an verschiedene Datenbanktypen oder Verbindungsarten. Normalerweise werden in der Factory nur Objekte erzeugt, die in den Blättern der Vererbungshierarchie liegen. Vorteil des Patterns ist, dass es die Schritte zur Erstellung des Objektes vor dem Anwender kapselt. Sie müssen nur dem Erzeuger bekannt sein. Damit können schnell neue Objektarten hinzugefügt oder Erstellungsschritte abgeändert werden.

Das nachfolgende Beispiel basiert auf einer Anregung aus dem Entwurfsmuster-Katalog von (Hauer, 2016). Eine Applikation benötigt für ihre Aufgaben verschiedene Office-Datei-Formate, von Excel bis Word. Es existieren drei Use Cases in denen die verschiedenen Datei-Formate benötigt werden können

1. Eine neue leere Datei des gewählten Formates soll erzeugt werden.

2. Basierend auf einem Template soll eine neue Datei mit vorgegebenen Strukturen erstellt werden.

3. Eine bereits vorhandene Datei soll geöffnet werden.

Um die Formate voneinander zu trennen, existiert für jedes eine eigene unabhängige Klasse, wie in Abbildung 42 dargestellt. Wird nun ein bestimmtes Format angefordert, muss die Entscheidung außerhalb der Format-Klassen getroffen werden, da keine der Klassen Kenntnis von den anderen Klassen besitzt. Eine solche Entscheidung wird, entweder in der Klasse *DateiGenerator* oder *DateiImporter*, mit der Hilfe von Switch- oder If-Anweisungen getroffen. Somit gibt es keinen zentralen Ort für die Erstellung.

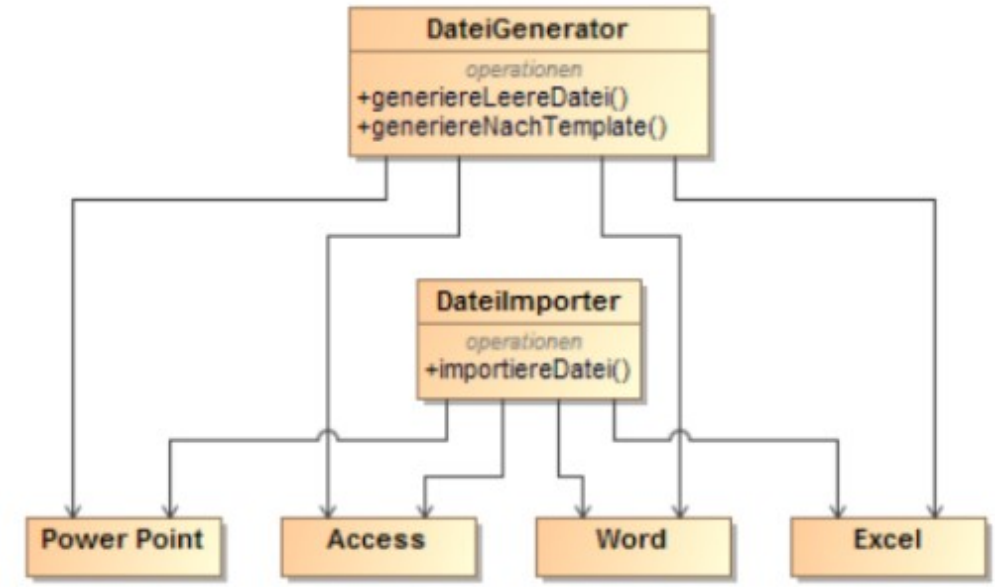


Abbildung 42 Multiple Erstellung von Objekten an verschiedenen Stellen.

Eine Lösung wäre der Einsatz des Factory Method Patterns, wie in Abbildung 43 gezeigt. Hier gibt es eine zentrale Stelle mit File-Objekt zur Erstellung von Objekten. Der Aufrufer muss lediglich seine Daten übergeben, die Auswahl des Typs übernimmt das Pattern.

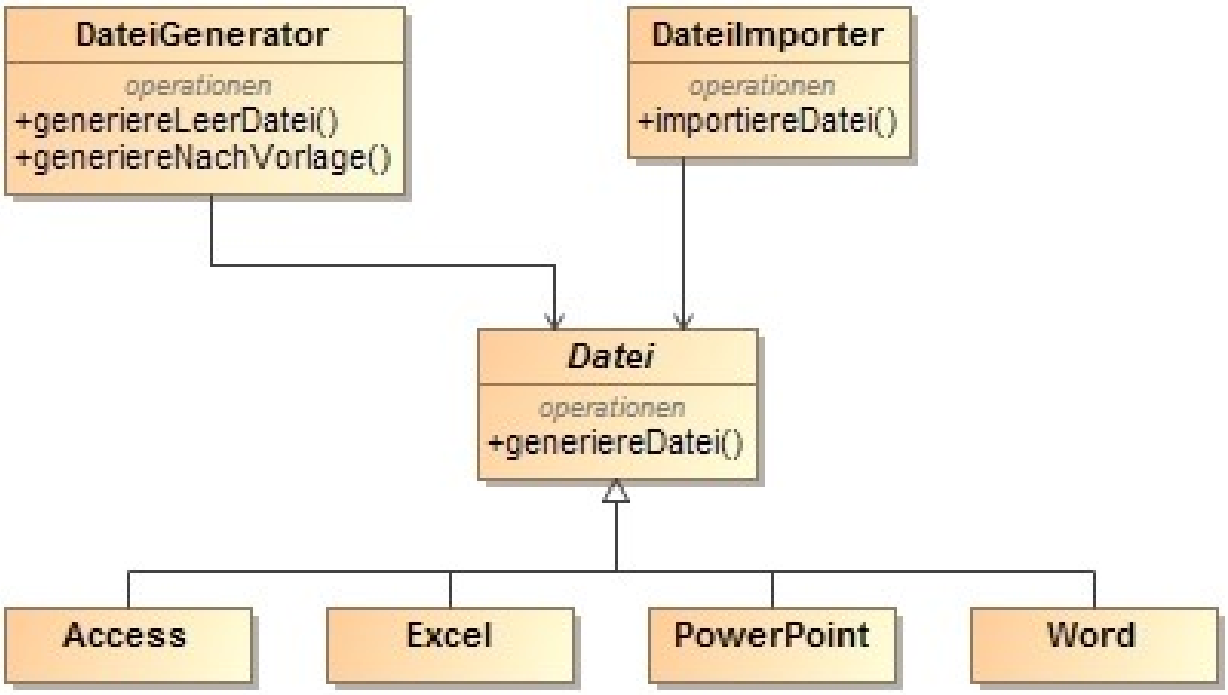


Abbildung 43 Factory Method Kandidaten-Skizze Nachher

Wie zu Beginn dieses Kapitels beschrieben, vereint das Pattern zusammengehörige Produkte. Dies geschieht über Vererbung, indem den Produkten eine gemeinsame Oberklasse gegeben wird. Der Einsatz von Vererbung stellt auch das Beispiel zur Entdeckung eines Kandidaten für genau dieses Pattern dar. Der erste Schritt zur Identifikation liegt in der Erkennung von Quellcodestellen, bei denen per Switch/IF-Anweisung entschieden wird, welcher Typ von einer Klasse erzeugt werden soll. Dabei müssen alle Klassen von der gleichen Oberklasse erben. Durch die Fallunterscheidung kann eine Aussage darüber getroffen werden, dass die Klassen mit einer bestimmten

Wahrscheinlichkeit eine ähnliche Logik implementieren. Somit werden je nach Entscheidung die notwendigen Schritte zu Erstellung durchgeführt. Gerade diese Auswahl durch Fallunterscheidungen, welcher Objekttyp für die Instanziierung verwendet wird, sollte an mehreren Stellen im Quelltext implementiert sein. Erst durch das mehrfache Auftreten der Fallunterscheidung wird klar, dass das Pattern den Quellcode verbessern kann. Mit Pattern wird eine zentrale Stelle zur Erzeugung geschaffen, die flexibel geändert werden kann. Ohne dies muss bei Erweiterungen mehrmals angepasst werden.

Tabelle 61 Merkmale der Factory Method Erkennung

Merkmal	Beschreibung
Vererbungshierarchie	Es existiert eine Vererbungshierarchie mit einer gewissen Menge an Klassen.
Fallunterscheidung	In mehreren Fallunterscheidungen wird entschieden welches Objekt der Hierarchie erstellt werden soll

5.7.2 Abstract Factory

Das Abstract Factory Pattern kann als Erweiterung der Method Factory gesehen werden. Anstatt einzelner Produkte werden in diesem Pattern ganze Produktfamilien, bestehend aus verschiedenen Produkten, auf einmal erzeugt. Eine einfache Beispielproduktfamilie stellt z.B. ein Auto dar. Dies besteht aus den Produkten Karosserie, Reifen und Motor. Im Quellcode wird jedes Produkt wieder durch eine eigene Klasse repräsentiert, wobei die einzelnen Klassen von einer gemeinsame Oberklasse erben. Auf diese Weise werden Produktfamilien abgebildet. Damit das Pattern effizient eingesetzt werden kann, müssen mehrere Produktfamilien im Programm vorkommen.

Im folgenden Beispiel Abbildung 44 (vgl. (Hauer, 2016)) werden drei Produktfamilien implementiert. Je nach Situation werden unterschiedliche Objekte benötigt, jedoch die Produktfamilien nie miteinander vermischt oder gleichzeitig erzeugt. Es gibt also keine Möglichkeit ein Objekt *Elefant* mit dem Objekt *Sand* zu erzeugen. Der Konstruktor innerhalb der Oberklasse (*Regenwald*, *Wüste* und *Polargebiet)* übernimmt die Aufgabe der Generation der Familie. Im Beispiel ist es nicht möglich, mehr als eine Produktfamilie auf einmal zu erstellen. Dies erlaubt den Rückschluss, dass sie ähnliche Typen implementieren. Entsprechend wird die Entscheidung, welche Familie benötigt wird, durch Switch- oder IF-Anweisungen außerhalb der Produktfamilien getroffen.

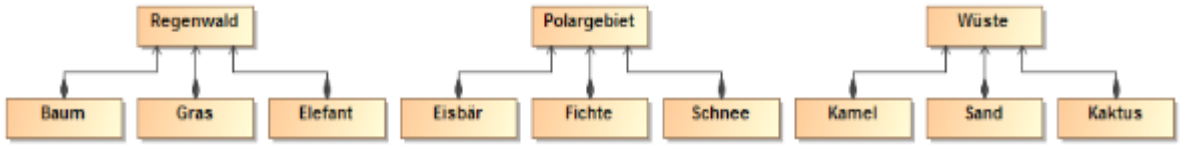


Abbildung 44 Kandidaten-Skizze Abstract Factory

Eine Abstract Factory ist in der Lage die Produktfamilien zusammenzufassen und das Generieren so zu vereinfachen, wie in Abbildung 45 dargestellt. Wie zu sehen ist, gibt es eine zentrale Stelle, an der Produktfamilien angefordert werden. Durch dieses Design können weitere Produktfamilien schnell angefügt oder nicht weiter benötigte entfernt werden.

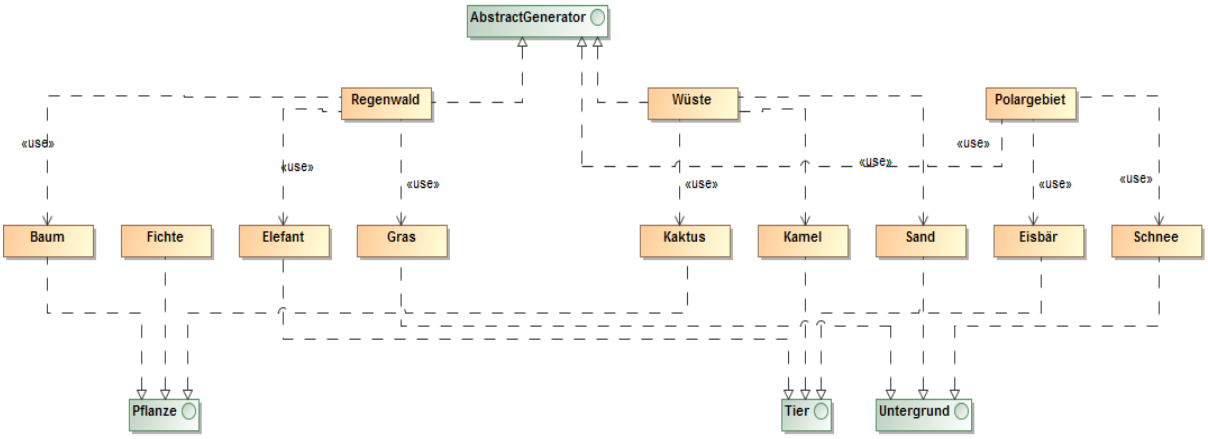


Abbildung 45 Abstract Factory Pattern

Im ersten Schritt des Erkennungsprozesses müssen somit die einzelnen Produktfamilien identifiziert werden. Diese Produktfamilien werden über ihre Verbindungen erkannt. Als Beispiel soll eine Landschaftsproduktfamilie dienen, bestehend aus dem Gebiet, einem passenden Tier, einer passenden Pflanze und dem passenden Bodentyp, wie in der folgenden Abbildung 46 illustriert.

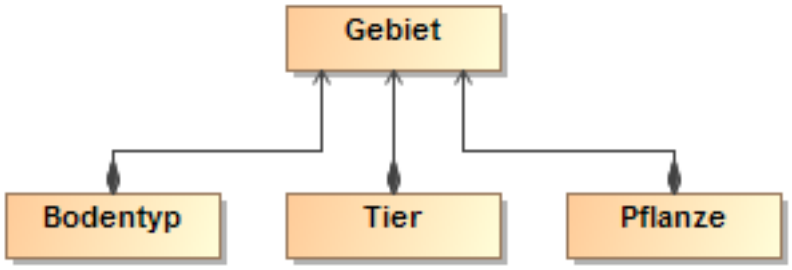


Abbildung 46 Beispiel Produktfamilie

Oft sind solche Verbindungen innerhalb einer Familie durch Kompositionen implementiert, da die Familie nur existieren kann, wenn alle Objekte erzeugt werden. Zur Erkennung werden also Klassen gesucht, bei deren Instanziierung die Generierung weiterer Klasse notwendig ist. Eine Instanziierung dieser Produktfamilie über andere Klassen als Gebiet ist nicht möglich. Somit gibt es keine Verbindungen zu den Konstruktoren Bodentyp, Tier und Pflanze von außerhalb. Sind diese Gegebenheiten erfüllt, handelt es sich um eine Produktfamilie. Die Entscheidung welche Familie erstellt wird, geschieht außerhalb und mit der Hilfe von Fallunterscheidungen. Existieren mehrere Produktfamilien in einer Applikation, sollte eine Abstract Factory verwendet werden.

Tabelle 62 Erkennungsmerkmale für das Abtract Factory Pattern

Merkmal	Beschreibung
Kompositionen	Es existieren mehrere Klassen, die bei der Erstellung weitere Klassen benötigen
Einziger Instanziierungspunkt	Instanziierung immer nur über eine Klasse im Verbund
Produktfamilien	Im System gibt es mehrere Produktfamilien
Fallunterscheidung	An mehreren Stellen wird per Fallunterscheidung entschieden, welche Familie erstellt werden soll

5.7.3 Prototype

Das Prototype Pattern beschreibt ein Muster, das es erlaubt eine Vielzahl von identischen Objekten schnell durch Klonen zu erzeugen, ohne dabei viele Ressourcen beim Instanziieren zu benötigen. Der Einsatz des Pattern sollte dann in Erwägung gezogen werden, wenn das Erzeugen eines Objektes teuer ist, d.h. das Objekt muss jedes Mal durch Logik in den Settern oder dem Konstruktor konfiguriert werden und nimmt somit Zeit und Ressourcen in Anspruch. Ein Beispiel eines solchen Objektes sind 3D-Objekte mit Bildtexturen. Das Laden einer Textur kann aufgrund der Größe einiges an Zeit beanspruchen. Ebenso das Positionieren des Objektes im Raum. Werden nun viele dieser Objekte erstellt, beeinträchtig dies die Laufzeit des Programmes.

Nachfolgende Abbildung 47 präsentiert das mögliche Verhalten basierend auf einem Prototype Szenario von Freeman et al. (Freeman, Robson, Bates, & Sierra, 2004). Ein Computerrollenspiel erzeugt dynamisch neue Monster-Objekte. Dabei unterscheiden sich die Monster, je nach Typ, voneinander. Auf Grund des Genres des Spiels werden viele verschiedene Objekte von der Klasse *Monster* benötigt. Jedes Mal, wenn ein neues Monster erzeugt wurde, muss dieses durch das Aufrufen von Gettern und Settern neu konfiguriert werden, was unnötige Zeit beansprucht.

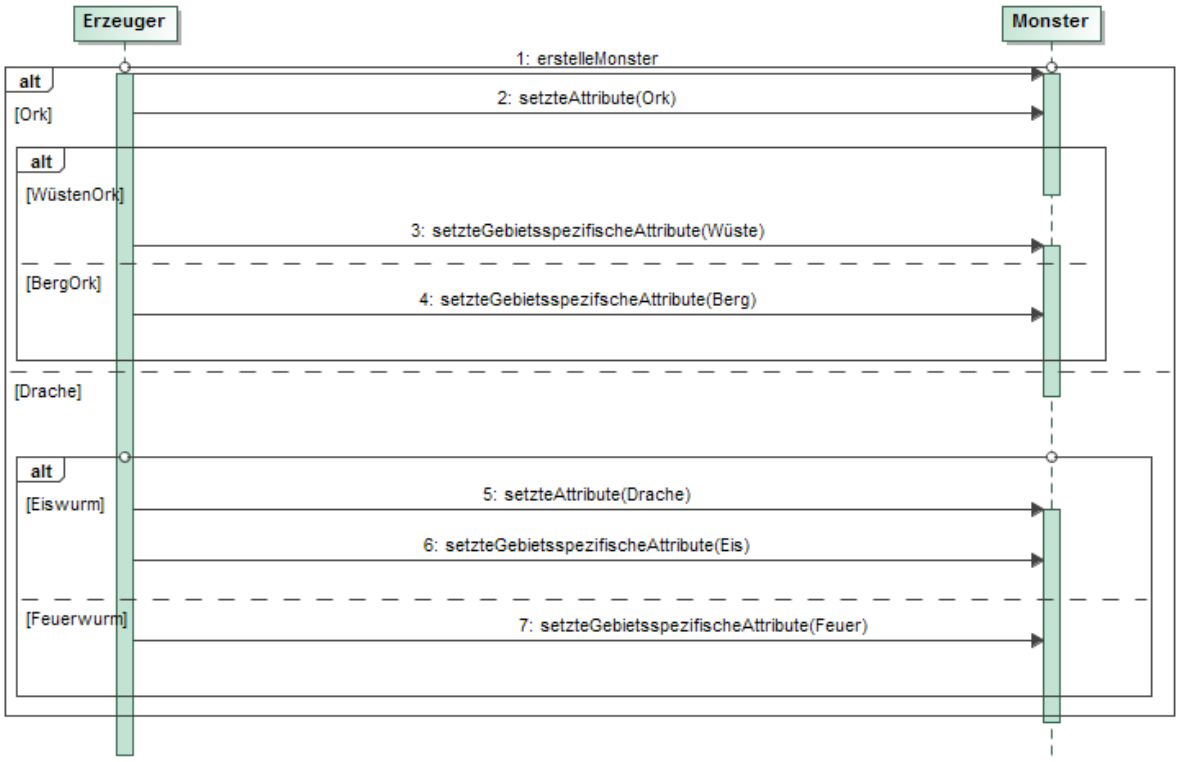


Abbildung 47 Prototype Kandidaten-Skizze

Dasselbe Szenario mit einem Prototype-Pattern wird in *Abbildung 48* und Abbildung 49 dargestellt. Hier besitzt jede Ausprägung von Monstern eine eigene Klasse. Jede Klasse stellt somit einen konkreten Prototype dar. Zu Beginn wird von jeder Klasse genau ein Objekt erzeugt. Werden weitere Objekt benötigt, geschieht dies über das Klonen von vorhandenen Objekten. Der Vorteil liegt nun darin, dass nur die Attribute geändert werden müssen, die sich vom Vorgängerobjekt unterscheiden.

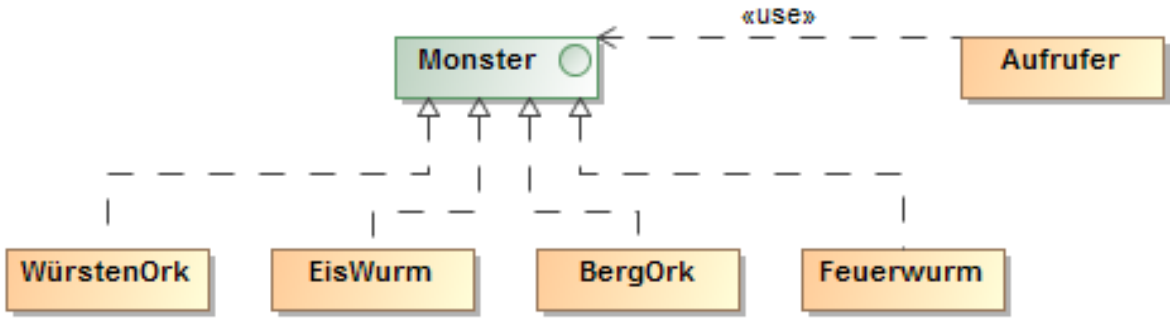


Abbildung 48 Monstergenerator mit Prototype Pattern

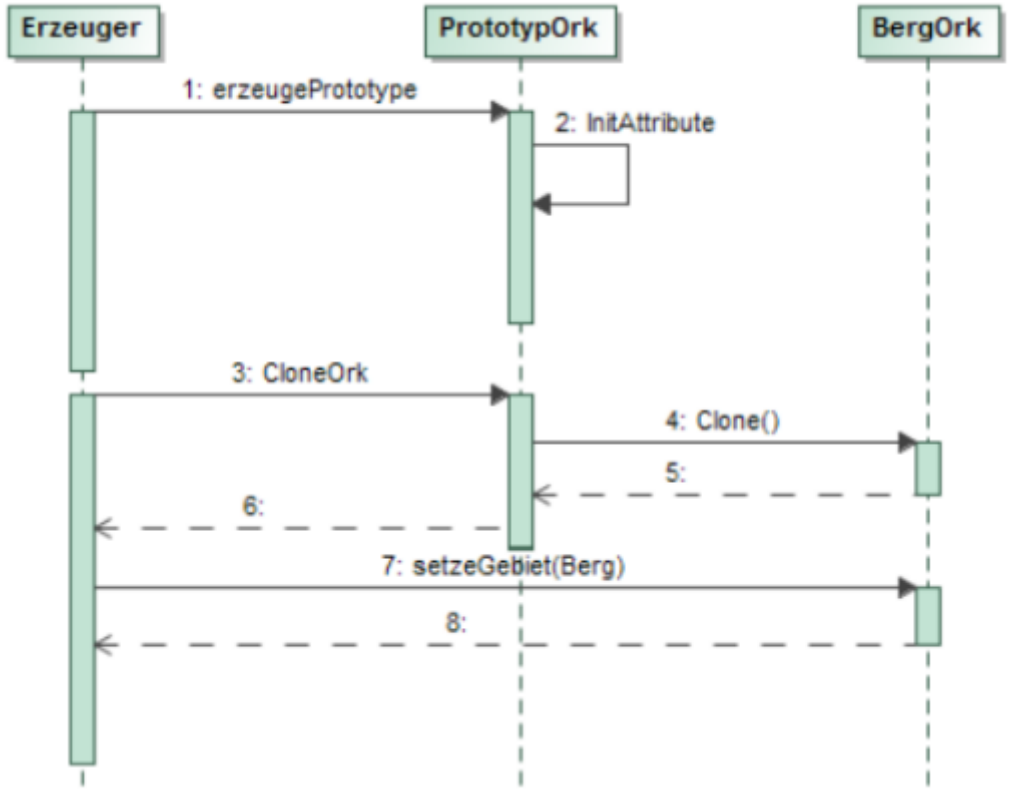


Abbildung 49 Monstergenerator mit Prototype-Pattern als Sequenzdiagramm

Die Erkennung eines Kandidaten erfolgt über die Anzahl an Konstruktorenaufrufen. Ein erstes Indiz liegt vor, wenn der Konstruktor häufig aufgerufen wird. Dies kann an unterschiedlichen Stellen im Quelltext oder innerhalb einer Schleife sein. Dieser Umstand deutet auf eine große Menge an benötigten Objekten hin. Im nächsten Schritt der Erkennung gilt es festzustellen, wie weit ein Objekt, nach seiner Erzeugung, individualisiert wird. Eine Individualisierung kann durch zwei Fälle durchgeführt werden. Einmal über eine hohe Anzahl an Parametern im Konstruktor. In einem anderen Fall wird gleich im Anschluss an die Erstellung des Objektes, d.h. nach dem Aufruf des Konstruktors, die Individualisierung durch das Aufrufen von vielen Setter-Methoden abgehandelt. Daraus folgt, dass nach dem Konstrukturaufruf viele Methodenaufrufe auf das neue Objekt durchgeführt werden müssen. Dabei handelt es sich meist nicht um einfache Setter, die nur Attribute setzten, sondern um komplex logische Abhandlungen. Beispielsweise gehören dazu Dateioperationen wie Laden, Netzwerkoperationen wie Verbindungsaufbau oder Datenbankoperationen wie Abfragen. Solche komplexe Abfragen können durch gezielte Suche nach der Verwendung oben genannter Operationen (z.B. Verwendung von Datenbankoperationen wie Select) identifiziert werden. Auch können Metriken wie LoC (Lines of Code) oder

zyklomatische Komplexität als Merkmale genutzt werden. Die Orchestrierung der Setter geschieht über verschachtelte Fallunterscheidungen.

Tabelle 63 Merkmale der Prototype Erkennung

Merkmal	Beschreibung
Konstruktorenaufrufe	Der Konstruktor einer Klasse wird häufig und an unterschiedlichen Stellen oder in Schleifen aufgerufen.
Individualisierung	Das neue Objekt wird durch Setter und andere Methodenaufrufe nach der Erzeugung individualisiert.
Komplexe Schritte	Einige Aufrufe der Individualisierung sind komplexe Operationen.

5.7.4 Singleton

Das Singleton-Entwurfsmuster hilft bei der Reduzierung der Anzahl an Objekten einer Klasse und somit bei der Reduzierung der benötigten Ressourcen. Ein weiterer Vorteil ist, dass die Anzahl der Instanzen durch das Pattern auf eine (oder ggf. wenige) reduziert wird. Somit müssen alle Verbraucher dieses eine Objekt verwenden. Der Zugriff auf das Objekt kann mit Hilfe des Patterns einfach kontrolliert werden. Das Pattern sollte anstelle von globalen Objekten oder tausender ähnlicher Objekte Anwendung finden.

Das folgende Beispiel, basierend auf dem Singleton-Szenario von Holzner (Holzner, 2006). In diesem Beispiel existiert eine Klasse für Datenbankaufrufe, die von verschiedenen Stellen im Programm genutzt wird. Es gibt aber mehrere Datenbankinstanzen der gleichen Datenbankart, (z.B. MariaDB) auf die das Programm zugreifen muss. Um den Zugriff für alle Programmteile einfach zu halten, besteht die Klasse aus statischen Attributen und Operationen. Auf diese Weise wird sichergestellt, dass die aufrufenden Klassen immer die gleichen Zugriffsdaten verwenden. Der Nachteil liegt in der starren bzw. statischen Struktur des Designs. Eine statische Variable bekommt ihren Wert beim Start des Programmes. Möchte ein Client eine Datenbankverbindung aufbauen, ruft er die statische Methode *getConnection(DatabaseName)* auf und teilt den Namen der gewünschten Datenbankverbindung mit. In der Methode wird per Fallunterscheidung bestimmt welche statische Variable zurückgegeben wird. Des Weiteren muss für jede Datenbankverbindung eine statische Variable angelegt werden. Eine Erweiterung der Klasse über Vererbung oder Decorator, um z.B. verschiedene Datenbankentypen (z.B. NoSQL) anzusprechen oder das dynamische Verändern der Zugriffsdaten zur Laufzeit, kann mit diesem Design nicht abgebildet werden. Schon deshalb, weil beim dynamischen Verändern der Zugriffsdaten mehrere Instanzen der Klasse instanziiert werden müssen, auch wenn nur eine pro Zugriffsdatensatz benötig wird. Statische Inhalte von Klassen können wie globale Variablen verwendet werden, was eine Zugriffsteuerung sehr komplex macht. Sollte das Design erweitert werden müssen, bleibt nichts anders übrig, als die statischen Methoden durch andere zu ersetzen (vgl. Abbildung 50).

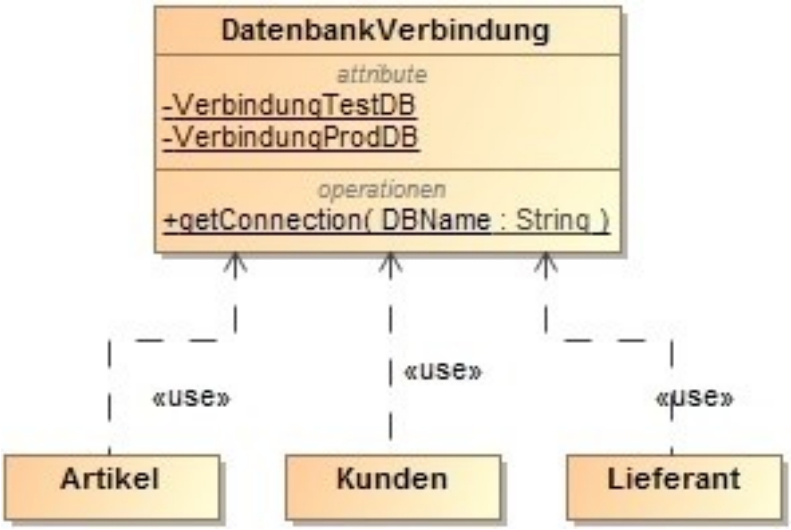


Abbildung 50 Singleton Kandidaten-Skizze mit statischen Methoden

Ein Singleton-Pattern würde diese Erweiterungen und dynamischen Veränderungen ermöglichen und trotzdem die Anzahl der Ressourcen minimal halten. Hier wird nur eine statische Variable verwendet, welche immer das gleiche Objekt zurück gibt (siehe Abbildung 51). Die eigentlichen Attribute und Methoden des Objektes sind dynamisch. Somit können die Attribute zur Laufzeit geändert werden. Auch wird es ermöglicht die Klasse zu vererben. Es ist damit nicht mehr nötig, die verschiedenen Verbindungen im Quellcode vor dem Programmstart zu hinterlegen.

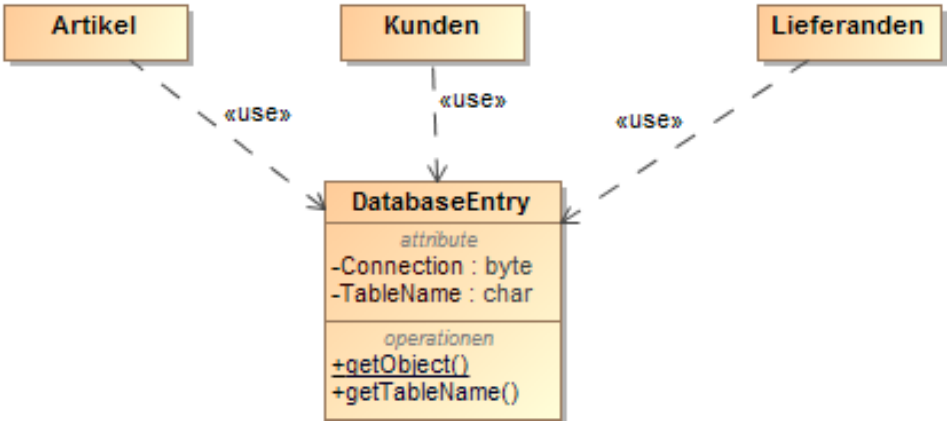


Abbildung 51 Singleton Pattern für Datenbankzugriff

Für die Identifikation eines Single-Pattern Kandidaten muss im Programm eine Klasse existieren, die von vielen anderen Klassen angesprochen wird. Dies impliziert, dass die Klasse über Daten verfügt, die für alle anderen wichtig sind. Sollte diese Klasse nun über statische Methoden und Variablen verfügen und nur diese von anderen Klassen zur Kommunikation mit der Kandidatenklasse verwendet werden, ist dies ein weiteres Indiz für einen Singleton-Kandidaten. Die Anwendung von statischen Inhalten (diese können zur Laufzeit nicht mehr verändert werden) soll die Informationen in der Klasse vor Veränderung schützen und unterstreicht somit die Wichtigkeit dieser für das Gesamtsystem. Andere Klassen im System sollen immer die gleichen Informationen bekommen, egal wann sie auf die Klasse zugreifen. So wird auch verhindert, dass mehrere Objekte mit unterschiedlichen Inhalten erzeugt werden. Von der Klasse wird nie ein Objekt erstellt, da es nur über statische Methoden angesprochen werden kann. Somit hat sie auch keine normalen Methoden oder Attribute. In den statischen Methoden, die alle anderen Klassen

aufrufen, gibt es eine Fallunterscheidung die benötigt wird, um zu bestimmen, welches der statischen Attribute als Rückgabewert dient.

Ebenfalls sinnvoll kann ein Singleton Pattern sein, wenn die Klasse mit statischen Methoden durch Vererbung erweitert werden sollte, um in speziellen Fällen weitere Funktionalitäten hinzuzufügen. Diese Entscheidung kann aber nur vom Entwickler getroffen werden. Dabei handelt es sich um eine Design Entscheidung die im Quellcode so nicht erkannt werden kann.

Tabelle 64 Merkmale der Singleton Erkennung

Merkmal	Beschreibung
Viele Zugriffe	Die Klasse wird von vielen anderen Klassen aufgerufen
Keine Objekte	Obwohl die Klasse häufig verwendet wird, gibt es kein Objekt von ihr.
Zugriff nur auf statische Inhalte	Klienten rufen immer die gleiche statische Methode auf.
Fallunterscheidung für statische Attribute	Es existieren mehrere statische Attribute, die alle von der gleichen Methode zurückgegeben werden. Zur Bestimmung des Rückgabewertes wird eine Fallunterscheidung verwendet.

5.7.5 Flyweight

Dieses Pattern verbessert den Ressourcenverbrauch von Objekten durch die Reduktion des Speicherbedarfes. Die Idee des Patterns ist die Separation von Attributen in zwei Typen, wobei jeder Typ in einer eigenen Klasse implementiert wird.

1. *Intrinsic (Redundante Daten, die mehrfach vorkommen und zusammengefasst werden können)*
2. *Extrinsic (Daten, die nicht zusammengefasst werden können, werden getrennt)*

Eine Klasse beinhaltet nur die Attribute, die dort benötigt werden. Die anderen Attribute werden von mehreren Klassen verwendet und sind somit separat in einer gemeinsam genutzten Klasse implementiert. Die so entstehende gemeinsame Klasse wird später mit den Klassen verknüpft, aus denen die Felder stammen. Auf diese Weise besitzen Klassen weniger Attribute, was dazu führt, dass ihre Objekte weniger Speicher benötigen. Die neu implementierten Klassen mit Intrinsic-Attributen benötigen durch die gemeinsame Nutzung ebenfalls weniger Speicherplatz.

Folgendes Beispiel demonstriert die Verwendung das Pattern. Von einer Beispielklasse *Student* wird eine sehr große Menge an Objekten erzeugt. Um einen spezifischen Studenten schnell zu finden, werden alle Objekte in einer HashMap hinterlegt. Um beim Programmstart von allen Studenten ein Objekt zu erstellen, wird eine Schleife verwendet. Durch die Beziehung der Klasse Student zu anderen Klassen wie Fächern und Adressen, werden für jedes Objekt noch weitere abhängige Objekte erzeugt. Von der Klasse Fächer werden mehrere Objekte erstellt und in einer Liste abgelegt. Jedes Fach besitzt zudem noch ein Objekt Note (siehe Abbildung 52). Durch diese Verschachtelung kommt es zu einer hohen Anzahl an Objekten.

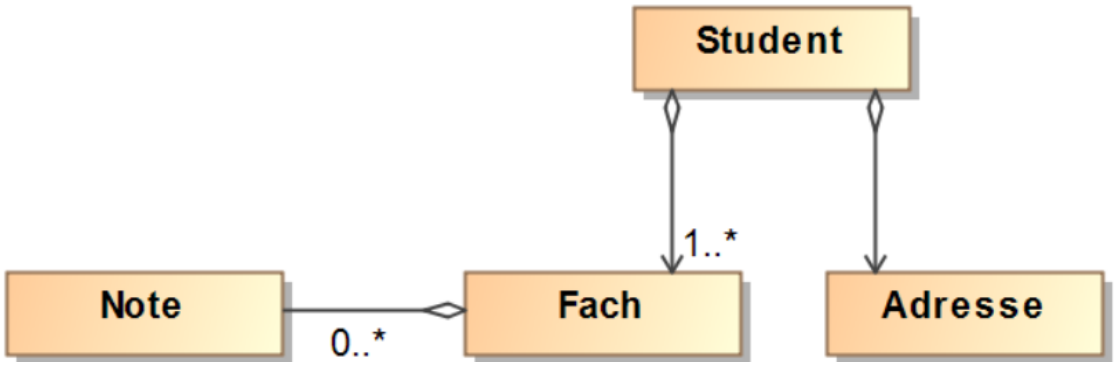


Abbildung 52 Flyeight Pattern-Skizze

Die Verwendung eines Flyweight Pattern (siehe Abbildung 53) reduziert die Anzahl der Objekte und somit den Speicherplatz. Dazu wird eine StudentFactory Klasse implementiert, die für das Erstellen der Objekte verantwortlich ist. Diese Factory kennt nur das Interface StudentFlyweight. Die Daten zur Erzeugung der Objekte werden von der Factory entgegengenommen und die entsprechenden Objektstrukturen erzeugt. Fächer und Noten sind intrinsische Zustände und könnten wiederverwendet werden, d.h. diese Daten sind redundant in der vorherigen Struktur. Redundant deshalb, weil ein Fach von mehreren Studenten besucht werden kann. Auch eine Note kann mehrmals vorkommen. Beide Daten können mehrmals verwendet werden, anstatt sie immer wieder neu zu erstellen, sog. Value Objects. Adresse und Student sind einzigartig, also extrinsisch. Solche Daten können nicht in einer Klasse für andere zusammengefasst werden. Nun speichert der Student zwar weiterhin die Fächer, doch es wird immer zuerst geprüft, ob das Fach schon existiert. Wenn ja, wird das vorhandene verwendet sowie die Note innerhalb des Faches.

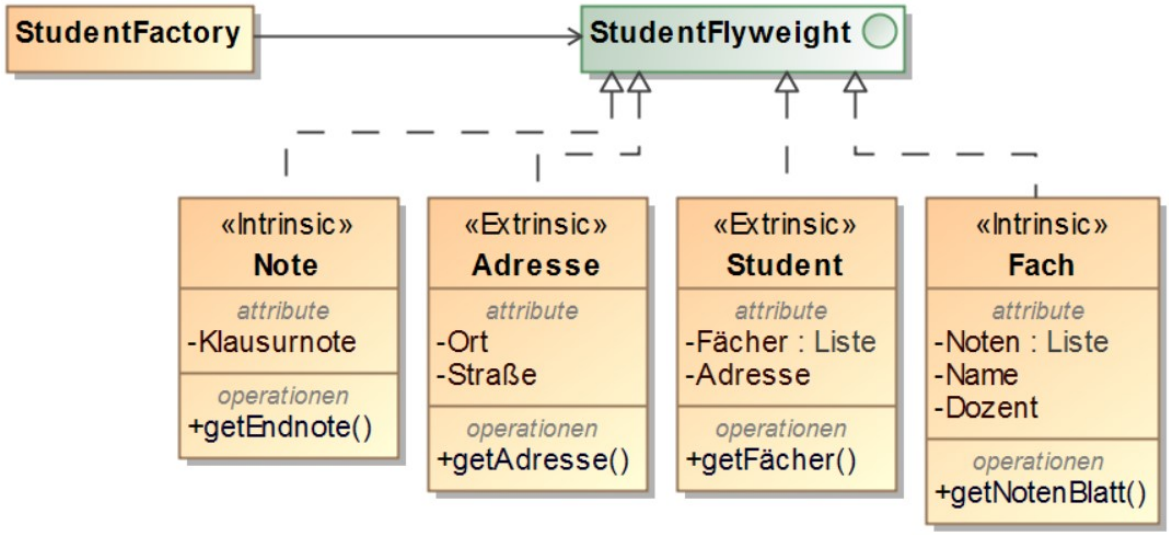


Abbildung 53 Flyweigth Student

Erster Schritt bei der Erkennung von Kandidaten liegt somit in der Feststellung der Anzahl der Instanzen einer Klasse. Je mehr Objekte von einer Klasse erzeugt werden, desto mehr Speicher wird auch benötigt. Über die Anzahl der Konstruktoraufrufe kann diese Anzahl grob bestimmt werden. Des Weiteren wäre noch möglich, den Speicher zur Laufzeit zu analysieren und die Anzahl der Objekte pro Klasse zu bestimmen. Das könnte unter Java mit dem Eclipse-Projekt Memory-

Analyzer[7] geschehen. Rein aus dem Quellcode ist eine genaue Abschätzung der Objektanzahl schwierig. Der Speicherbedarf ist besonders intensiv, wenn ein Objekt noch andere Objekte benötigt. Gerade das Instanziieren von Objekten, die viele abhängige Klassen mit sich bringen, benötigt meist viel Speicher. Besonders tritt dieser Effekt in Erscheinung, wenn es sich um 1 zu N Beziehungen zwischen den Klassen handelt. Eine solche Beziehung wird häufig über Collections abgebildet, da sich auf diese Weise Objekte einfach verwalten lassen. Dementsprechend sollte im zweiten Schritt der Erkennung nach den genannten Verbindungen in den zuvor ausgewählten Klassen gesucht werden. Als *intrinsic* können die abhängigen Klassen angesehen werden. Letztlich muss der Entwickler entscheiden, ob die Daten mehrfach verwendet werden. Der Kandidat zeigt in diesem Fall nur Strukturen auf, die daraufhin deuten. Nur durch eine Analyse der gespeicherten Daten kann klar werden, ob sie mehrfach vorkommen.

Tabelle 65 Merkmale der Flyweigth Erkennung

Merkmal	Beschreibung
Konstruktoraufrufe	Von einer Klasse werden viele Instanzen erstellt
Objektverbünde	Bei der Erzeugung eines Objektes können weitere abhängige Objekten in einer Liste erstellten werden.
1 : N Beziehungen	Es existieren 1:N Beziehungen zwischen dem Objekt und der Liste

5.7.6 Chain of Responsibility

Das Chain-of-Responsibility-Pattern dient der Trennung von Anfragen und Verarbeitung. In einem System werden Events erzeugt, deren Abarbeitung je nach Eventtyp unterschiedlich verarbeitet werden sollen. Bei der Verwendung des Patterns werden die verarbeiteten Objekte in einer Kette angeordnet. Das Event durchläuft die Kette von Objekt zu Objekt und kann von jedem Objekt bearbeitet werden, oder nur von einem. Das Objekt wird auf jeden Fall immer zum Nächsten weiter gereicht. Der Erzeuger kennt nur das erste Objekt in der Kette. Alle anderen Objekte in der Kette kennen immer nur ihren Nachfolger. Neue Befehle können somit einfach in die Kette eingefügt werden

Das folgende Beispiel eines E-Mail verarbeitenden Systems basiert auf einer Beschreibung von Freeman at el. (Freeman, Robson, Bates, & Sierra, 2004). Die E-Mails können Kunden via Webportal erzeugen und so ins System einleiten. Jede E-Mail wird beim Eingang in das System kategorisiert. Es gibt die Kategorien *Spam*, *Complain*, *NewLoc* und *Fan*. Dabei kann eine E-Mail mehrere Kategorien erhalten. Bei der weiteren Verarbeitung wird jede Kategorie anders behandelt. Abbildung 54 zeigt die Kategorieverarbeitung ohne Pattern. In dem gezeigten Fall, wird die Auswahl der Verarbeitungsschritte mit verschiedenen Optionsmöglichkeiten innerhalb einer Schleife durchgeführt. Es muss also eine Fallunterscheidung basierend auf der Kategorie getroffen werden. Innerhalb des Falles werden die E-Mails dann bearbeitet. Die Schleife wird solange durchlaufen, bis alle Kategorien abgearbeitet sind. Das Hinzufügen von weiteren Fällen erhöht die Komplexität des Systems. Da die Unterscheidung nicht zentral getroffen wird, existieren mehrere solche Fallunterscheidungen.

[7] http://www.eclipse.org/mat/

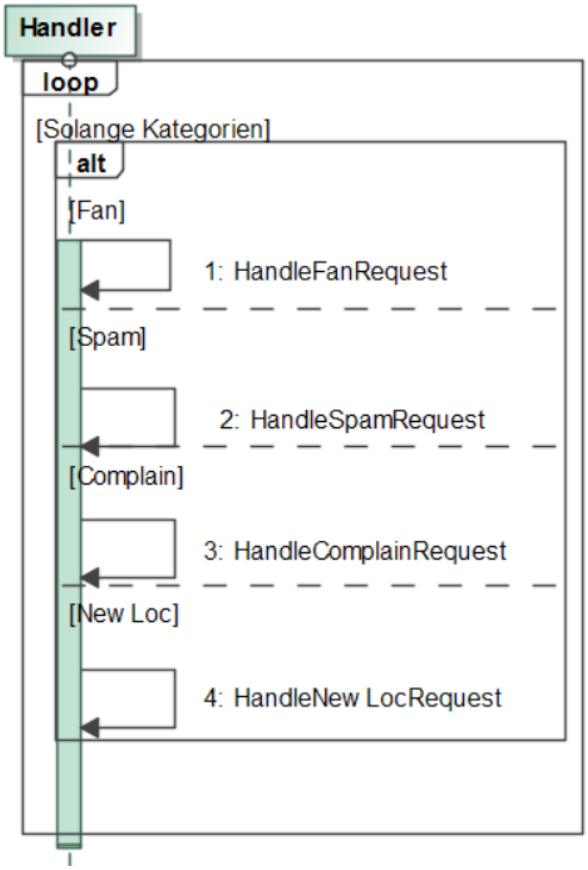


Abbildung 54 Event Verarbeitung eingehender E-Mails

Unter dem Einsatz eines Chain of Responsibility-Pattern würde die Verarbeitung in verschiedene Klassen ausgelagert und so flexibler gestaltet werden. Als Ansprechpartner für alle Klienten dient die Klasse Handler. Jede Kategorie wird in eine eigene Klasse implementiert. Neue Kategorien können, auch ohne Wissen des Aufrufers, flexibel eingebaut werden. Die E-Mail wird nur über die Methode *handleRequest* in die Kette eingeleitet, wodurch das Ergebnis später durch einen Teil der Kette zurück geliefert wird. Es ist für den Aufrufer nicht bekannt, welche Stelle die E-Mail bearbeitet hat. In der Abbildung 55 wird dargestellt, wie das Pattern die Aufgabe implementieren könnte.

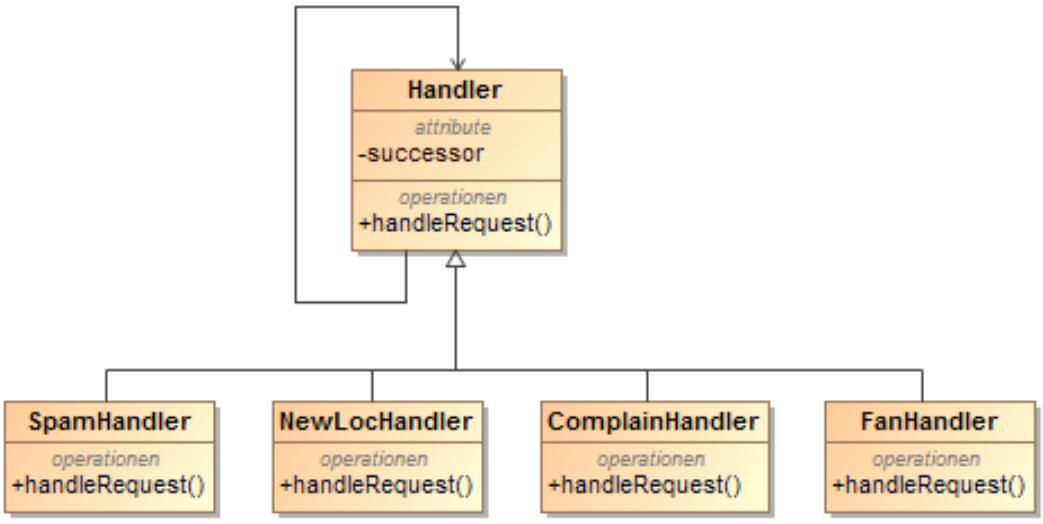


Abbildung 55 Event Verarbeitung mit Chain of Responsibility Pattern

Um einen Kandidaten zu identifizieren, müssen Fallunterscheidungen gefunden werden, die zur Auswahl von Verarbeitungswegen dienen. Dies bedeutet, dass entschieden wird, welche Methoden verwendet werden sollen, um das Objekt zu bearbeiten. In allen Fällen geht es um den gleichen Objekttyp (das Event). Die verwendeten Methoden innerhalb der Fälle sind ähnlich, nur in unterschiedlichen Zusammenstellungen. Da eine Verarbeitung nicht linear ablaufen muss, können Fallunterscheidungen verschachtelt sein. Nach der ersten Entscheidung müssen weitere getroffen werden, bis das Event abgearbeitet ist. Zur Identifikation sind im ersten Schritt mehrere Quellcodestellen zu identifizieren, die gleiche Fälle verarbeiten und die gleichen Methoden innerhalb dieser Fälle aufrufen. Aus der Menge der gefundenen möglichen Kandidaten werden nun die Fälle herausgesucht, die in ihrem Inneren immer wieder die gleichen Methoden aufrufen. Weisen diese gefundenen Quellcodestellen komplexe Fallunterscheidungen auf, ist ein Kandidat identifiziert.

Tabelle 66 Merkmale der Chain of Responsibility Erkennung

Merkmal	Beschreibung
Komplexe Fallunterscheidung	Es existieren mehrere verschachtelte If/If-Else Anweisungen im Code.
Gleiche Fälle	Die Fallunterscheidungen verwenden die gleichen Fälle.
Methoden aufrufe sind gleich	In den gleichen Fällen werden die gleichen Methoden aufgerufen.

5.7.7 Command

Komplexe Befehle, wie beispielsweise das Drucken eines Dokumentes (Daten konvertieren, Drucker initialisieren, aufheizen etc.) in kleine Einheiten zu separieren, ist die Aufgabe des Command-Patterns. Durch die Aufteilung können die einzelnen, einfacheren Befehle flexibel orchestriert und parametrisiert werden. Ein weiterer Vorteil ist, dass einzelne Befehle rückgängig gemacht werden können, da sie unabhängig von anderen sind.

Im nachfolgenden Beispiel, basierend auf einem Szenario von Hauer (Hauer, 2016) sollen verschiedene Drucker (Laser, Tine und Nadel) mit unterschiedlichen Sprachen (PS und PCL) angesprochen werden. Aufgrund der Unterschiede zwischen den Druckern bedarf es immer verschiedene Schritte zum Drucken eines Dokumentes. Da die Drucker nichtsdestotrotz viele ähnliche Eigenschaften haben, existiert eine Oberklasse *Drucker* von der alle Seitenbeschreibungssprachen (Page description language oder PDL) erben. Alle erbenden Klassen überschreiben die einzige Methode *druckeDokument,* die in der Oberklasse existiert. Jede PDL Klasse fügt noch eine Methode (konvertiereZuPCL/PS) zur Konvertierung des Dokumentes in die entsprechende Sprache hinzu, da jeder Drucker unterschiedliche Formate und Versionen benötigt. Diese Struktur (siehe Abbildung 56) führt dazu, dass jede Druckerklasse die Methoden *druckeDokument* und *konvertiereInX* selbst implementieren muss. Darüber hinaus besitzt jede Druckertypklasse eigene Methoden, die für das Drucken notwendig sind. In diesem Design werden alle notwendigen Schritte zum Drucken in den unterschiedlichen Druckklassen implementiert. Dies führt wiederum dazu, dass Klienten über *InstanceOf* bestimmen müssen, welcher Drucker gerade verwendet wird, um die richtigen Methoden zu erfolgreichen Drucken auszuführen. Das macht die Drucklogik für andere Klassen redundant und aufwändig.

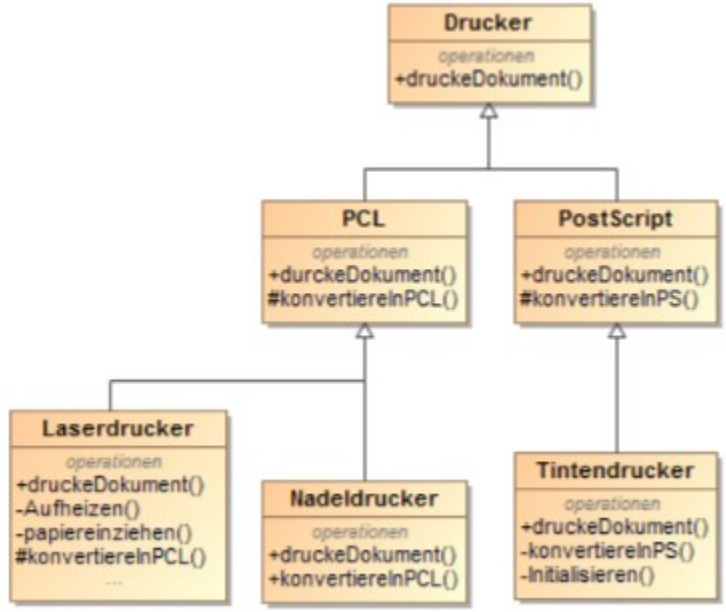


Abbildung 56 Druckerzuweisung ohne Pattern

Durch den Einsatz des Command-Patterns kann der Befehl zum Drucken zentralisiert und in einfache kleine Schritte zerlegt werden. Dazu wird das Interface *DruckerCommand* eingeführt, welches als zentraler Ansprechpunkt für alle Klienten, die ein Dokument ausgeben wollen, dient. Das Interface besitzt hierzu eine abstrakte Methode *druckeDokument*. Für jeden Druckertyp existiert eine konkrete Implementierung dieser Methode, die die jeweiligen Spezifika des Druckertyps umsetzt. Dadurch orchestriert nur diese Klasse die notwendigen Befehle, um den eigentlichen Drucker mit den nötigen Befehlen anzusprechen. Auf diese Weise werden die Methoden nicht immer überschrieben, sondern die Befehle werden nur neu zusammengestellt. Die Vererbung aus Abbildung 56 wird damit obsolet und durch eine flexible Klassenstruktur ersetzt. Zusätzlich bietet diese Abstraktion der konkreten Implementierung die Möglichkeit neue Drucker schnell einfügen zu können. Abbildung 57 zeigt eine mögliche Implementierung des Patterns als Klassendiagramm.

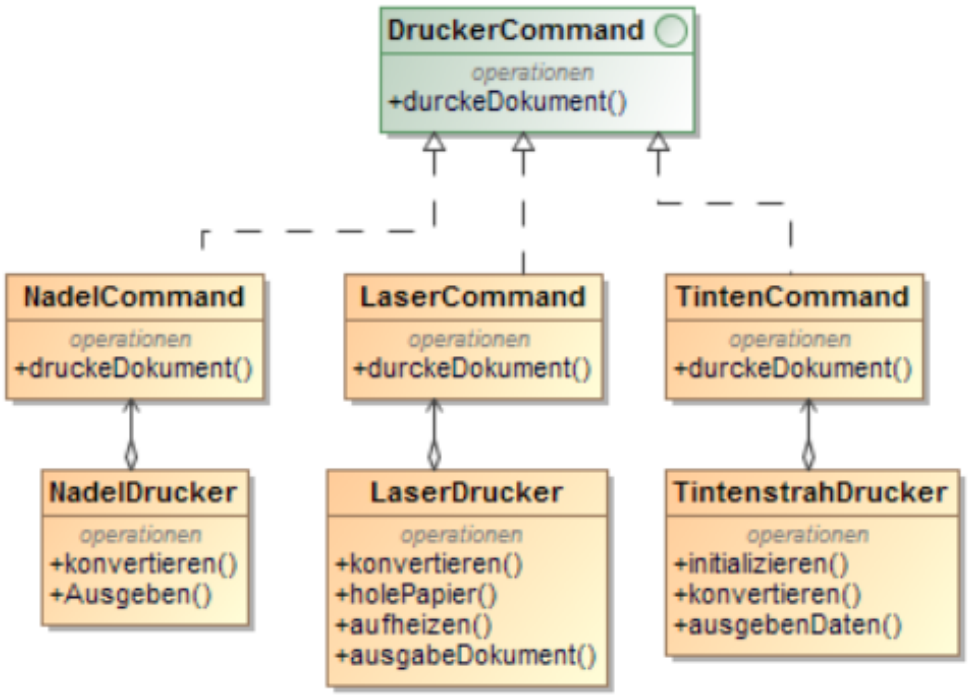


Abbildung 57 Command Pattern für Druckbefehl

Zur Identifikation des Command Patterns in existierendem Code sollten Stellen mit Vererbungshierarchien identifiziert werden, in der die Oberklasse keine oder wenige Methoden hat. Oberklassen, die überhaupt keine Methoden besitzen, dienen nur dazu die Unterklasse in eine Gruppe zu zwängen, welche eigentlich nicht notwendig ist. Sollte die Oberklasse Methoden enthalten und diese von allen erbenden Klassen überschrieben werden, ist das ein weiteres gutes Indiz für einen möglichen Kandidaten. Ein noch stärkeres Indiz wäre das Vorhandensein einer *NotImplementedException*, welche vom Entwickler verlangt die Methoden zu überschreiben. Das Überschreiben der Methoden führt dazu, dass die konkrete Ausprägung der genutzten Funktionen in jeder Subklasse neu zusammengestellt wird. Ein weiterer Anhaltspunkt sind Methoden in Subklassen, welche nur von einer überschriebenen Methode oder vom Konstruktor aufgerufen werden. Jede dieser Anzeichen in existierenden Klassen kann dazu führen, dass InstanceOf verwendet werden muss, um die richtigen Methoden finden zu können. Eine Übersicht der Erkennungsmerkmale befindet sich in Tabelle 67.

Tabelle 67 Merkmale der Command Erkennung

Merkmal	Beschreibung
Verwaiste Oberklasse	Es gibt eine Menge an Klassen, die von einer Oberklasse erben. Die Oberklasse besitzt aber nur wenige oder gar keine Methoden.
Überschriebene Methoden	Die erbenden Klassen überschreiben alle Methoden der Oberklasse.
Individuelle Methoden	Die Unterklassen implementieren neben der überschriebenen Methoden noch weitere Methoden, welche aber nur von den überschriebenen Methoden verwendet werden.
Instanzen-Entscheidung	Der Klient muss InstanceOf verwenden, um zu entscheiden, welche Klasse soll verwendet werden.

5.7.8 Iterator

Das Iterator-Pattern ist wahrscheinlich das am häufigsten eingesetzte Pattern in Java. Aufgabe des Patterns ist es eine Datenstruktur zu durchlaufen, ohne dass der Nutzer Wissen über den konkreten Aufbau benötigt. Mit dem Pattern können die Implementierung gekapselt und beispielweise auch mehrere Datenstrukturen verwendet werden, ohne dass der Benutzer es bemerkt. Weitläufig bekannt ist dieses Pattern durch das Java Collection Framework. Letzteres ist eine Sammlung von Klassen und Methoden für häufig verwendete Datenstrukturen. Dazu gehören unter anderem Listen, Sets und Maps in verschiedenen Ausprägungen. Jede Implementierung einer Datenstruktur im Framework bietet eine dazu passende Implementierung des Iterators an, um einfach die Daten durchlaufen zu können. Auf dieser Weise kann schnell und einfach auf die Daten in der Struktur zugegriffen werden. Seit Java 1.5 existiert eine For-Each Methode mit der die Collection direkt durchlaufen werden kann, ohne direkt auf den Iterator zugreifen zu müssen. Dieser wird im Hintergrund allerdings weiterhin verwendet. Der Iterator wird schon seit Java 1.2 angeboten, um jede Art von Containerklasse zu durchlaufen. Entwickler müssen die Schnittstelle nur noch selten selbst implementieren. Generell stehen den Entwicklern die verschiedenen Datenstrukturen mit allen notwendigen Methoden, wie dem Iterator, über Frameworks zur Verfügung. Der Iterator ist somit eines der wenigen Patterns, die überall eingesetzt werden. Das macht allerdings das Erkennen von Iterator-Kandidaten schwierig, da er oft schon impliziert verwendet wird. Weitere Einsatzgebiete zu finden ist ebenso diffizil. Es gibt ein paar wenige Fälle, in denen der Iterator neu implementiert werden muss.

Ein Fall tritt bei komplexen Datenstrukturen ein. Solche Strukturen können z.B. aus verschiedenen Collection-Arten bestehen. Ein einfaches Beispiel ist die Verwaltung von Softwarepaketen. Hierbei besteht ein Paket aus verschiedenen Tools und Programmen, wie beispielhaft in Abbildung 58 gezeigt. Um schnell auf ein benötigtes Paket zugreifen zu können werden die Namen als Key in einer HashMap abgelegt. Innerhalb der Map besitzt jeder Eintrag eine Liste, welche die verschiedenen Teile des Paketes beschreibt.

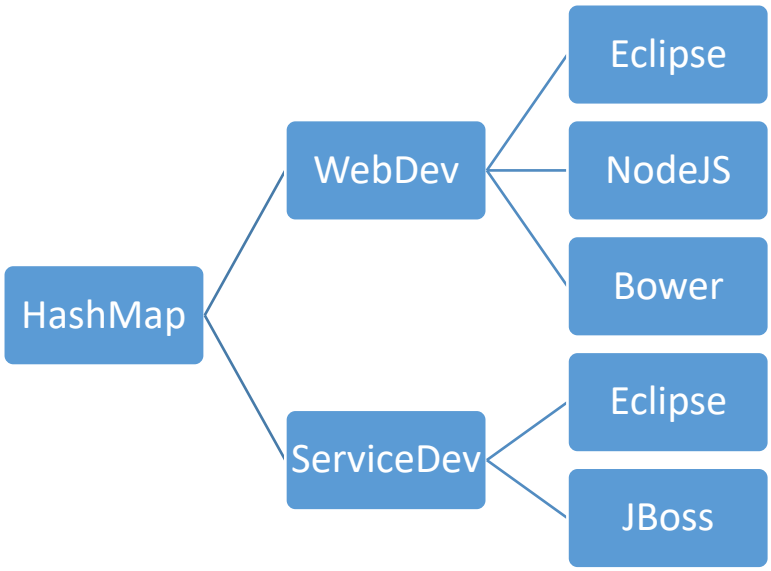


Abbildung 58 Beispieleinträge für die Softwarepackage Hashmap

Um eine gesamte Liste der verwendeten Software zu erstellen, muss die Struktur einmal durchlaufen werden. Der vorhandene Iterator im Java Collection Framework kann allerdings nicht für zwei verschiedene Strukturen, wie zum Beispiel Maps und Listen, gleichzeitig verwendet werden. Der Klient könnte die Struktur durchlaufen, indem er die Listeniteratioren in einer

Schleife des Hash-Iterators verwendet. Dadurch werden in einer inneren Schleife alle Einträge angesehen. Die Verantwortung liegt dabei beim Klienten, der auch die innere Struktur kennen muss. Flexibler und einfacher wäre es aber den Iterator für diesen Zweck anzupassen. Der Klient bekommt nur einen Iterator zurück geliefert, der die gesamte Struktur durchläuft. Ändert sich die Struktur im Hintergrund, würden die Klienten das nicht bemerken.

Durch die weite Verbreitung von Iteratoren im Collection Framework müsste dieser Spezialfall und weitere Fälle in einer weiteren Studie näher untersucht werden, um ihr Verbesserungspotenzial weiter zu untersuchen.

Tabelle 68 Erkennungsfälle für den Iterator

Merkmal	Beschreibung
Komplexe Datenstrukturen	Datenstruktur besteht aus mehreren Strukturarten wie Map oder List.

5.7.9 Memento

Das Memento-Pattern erlaubt das einfache Persistieren und Wiederherstellen von Objektzuständen. Dabei verhindert es, dass die eigentliche Kapselung durchbrochen wird, d.h. die Daten im Objekt können weiterhin nur über Getter und Setter abgefragt werden, ohne dass Details über die Implementierung nach außen gegeben werden müssen. Auf diese Weise bleibt die Objektidentität erhalten.

Ein Beispiel von Holzer (Holzner, 2006) liefert die Grundlagen für dieses Pattern. Damit Datenbankzustände nicht verloren gehen, werden diese in einem Objekt gespeichert. Die zugehörige Klasse wird in Abbildung 59 dargestellt und besitzt eine Methode *kopiereZustand*. Die Methode holt über Getter alle Attribute aus dem Objekt, um sie in ein neues Objekt der gleichen Klasse zu kopieren. Dabei gibt es zwei Möglichkeiten, das neue Objekt zu erstellen. Entweder im Konstruktor alle Attribute als Parameter zu übergeben oder nach der Instanziierung alle Setter aufzurufen. Zur späteren Wiederherstellung des Zustandes existiert die Methode *wiederherstellenZustand*. Ihr wird der alte Zustand übergeben aus dem die Attribute geholt und wieder im aktuellen Objekt gespeichert werden. Zur Feststellung, welche Identität ein Objekt gerade hat, werden die verschiedenen Attribute verglichen. Ein Klient muss das Speichern eines Zustandes aktiv anfordern. Das folgende Sequenzdiagramm (Abbildung 59) zeigt den Ablauf. Der Klient fordert die Klasse auf, ihren aktuellen Zustand zu sichern. Die Klasse erzeugt daraufhin von sich selbst ein neues Objekt und kopiert per Setter alle ihre Attribute in das neue Objekt (Objektidentität geht somit verloren). Später verlangt der Klient das Wiederherstellen des alten Zustandes. Die Daten werden von der Kopie per Getter zurückgeholt.

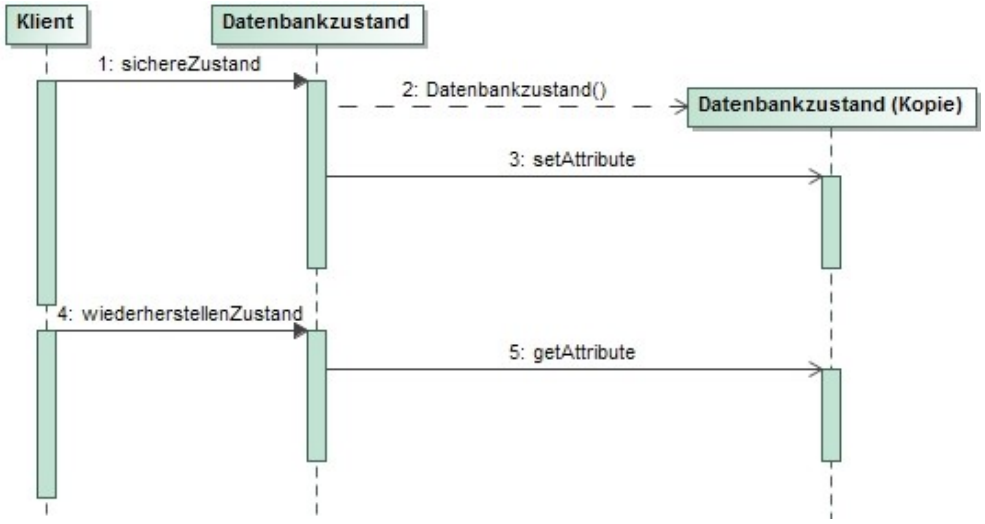


Abbildung 59 Zustände persistieren ohne Memento

Um das Vorgehen zu verbessern, bietet sich der Einsatz des Memento-Patterns an. Der aktuelle Zustand wird weiterhin in der Klasse *DatenbankZustand* gespeichert. Das eigentliche Speichern und Wiederherstellen von Objektzuständen wird jetzt in der Klasse *DBMemento* implementiert. In der Klasse *ZustandsVerantwortlicher* werden die eigentlichen alten Zustände verwaltet. Abspeichern und Wiederherstellen können unbemerkt von äußeren Anfragen durchgeführt werden. So kann bei jeder Anfrage von außen ein Zustand gespeichert werden, ohne dass der Klient etwas davon mitbekommt. Ebenfalls ist es möglich letzte Schritte zurückzunehmen, da alle Zustände zuvor gespeichert wurden. Abbildung 60 zeigt die Implementierung des Patterns.

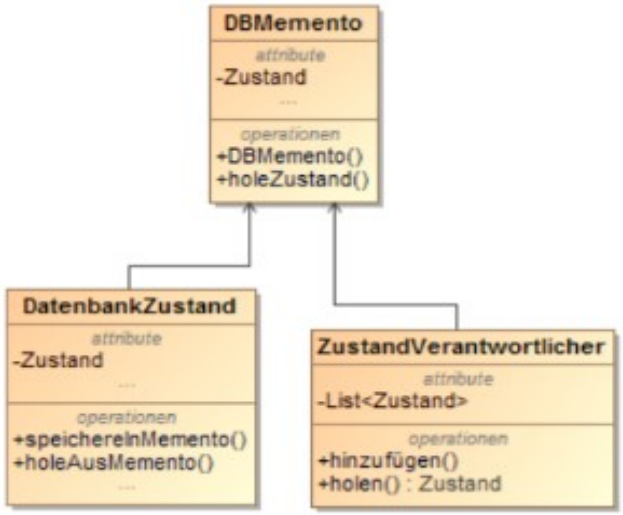


Abbildung 60 Zustandsspeicherung mit Memento Pattern

Die Identifikation eines Memento-Kandidaten basiert auf der Annahme, dass alle Attribute eines Objektes in ein anders Objekt des gleichen Typs kopiert werden. Die Kopie verwendet hierbei Getter oder Konstruktoren zur Übertragung der Daten. Eine ineffiziente Kopie fragt mithilfe von Gettern oder öffentlichen Attributen alle Daten einer Klasse ab. Der direkte Zugriff auf die Attribute könnte die Kapselung des Objektes verletzen. Die Werte werden dann entweder mittels eines Konstruktors in das neue Objekt übertragen oder es wird ein neues leeres Objekt erstellt und ein Setter oder alle Setter aufgerufen. Dieser Vorgang geschieht innerhalb einer Methode. Für einen Kandidaten muss auch der umgekehrte Fall vorliegen. Aus einem Objekt des gleichen Typs werden die Attribute gelesen und die eigenen Werte überschrieben. Dieser Vorgang sollte ebenfalls

innerhalb einer Methode durchgeführt werden. Eine Übersicht der Erkennungsmerkmale befindet sich in Tabelle 67.

Tabelle 69 Merkmale der Memento Erkennung

Merkmal	Beschreibung
Speichern	Es werden alle Getter oder direkt die Attribute einer Klasse aufgerufen und die Daten danach in einem Objekt des gleichen Typs abgelegt. Dies geschieht an einem zentralen Ort.
Wiederherstellen	Die Attribute des Objektes werden an einer zentralen Stelle durch die Attribute eines Objektes gleichen Typs überschrieben.

5.7.10 Observer

Publish-Subscribe- oder Single-Slot-Verfahren sind andere Bezeichnungen für das Observer-Pattern. Der Mechanismus des Patterns greift in das Kommunikationsverhalten von Objekten ein. Diese registrieren sich, um Nachrichten von bestimmen Stellen (Publisher) zu erhalten. Danach beobachten die angeschlossenen Objekte (Observer) den „Kommunikationskanal" und warten, bis eine Nachricht kommt. Der Publisher publiziert eine Nachricht für alle registrierten Objekte. Die wartenden Objekte müssen somit nicht periodisch nachfragen, ob sich ein Status geändert hat. Die Nachricht wird dann analysiert und von jedem Observer individuell interpretiert.

Das nachfolgende Beispiel basiert auf einem Szenario von Hauer (Hauer, 2016). Ein Verlagsprogramm soll immer dann informiert werden, wenn die Ausgabe eines Produkts erfolgt ist. Abbildung 61 zeigt eine mögliche Implentierung. Dazu muss das entsprechende Produkt von der Klasse *Verlag* immer wieder abgefragt werden. Hierfür gibt es in jedem Produkt die Methode *IstAusgeliefert*, welche bei Anfrage den aktuellen Status bekannt gibt. Ein eigener Thread, welcher durch die Klasse *Verlag* gestartet wird, sorgt für ein periodisches Polling, um den Zustand über die Methode aus den Produkten abzufragen. Mit *sleep* wird die Länge der Wartezeit für das Polling bestimmt.

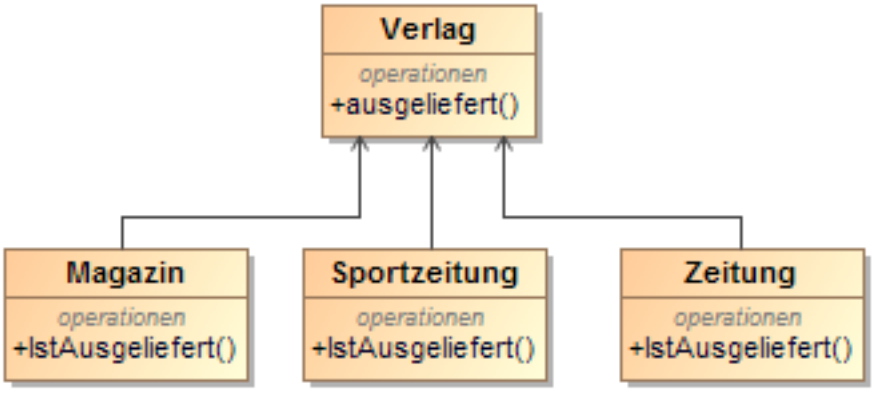


Abbildung 61 Verlagsprogramm wo die Klasse Verlag informiert wird

Die folgende Abbildung zeigt das Beispiel realisiert mit dem Observer Pattern. In diesem Fall registriert sich der Verlag als Observer bei den Produkten. Die Produkte generieren einen Event, wenn sie ausgeliefert wurden und pushen diesen an die Observer. Somit ist die Kommunikationsreihenfolge nun andersherum, nämlich von den Produkten zum Verlag. Das System läuft nun asynchron bzw. event-basiert, somit kann auf einen separaten Thread *sleep*

verzichtet werden. Ein zusätzlicher Vorteil ist, dass sich neue Verlage oder Produkte schnell an der Kommunikation beteiligen können.

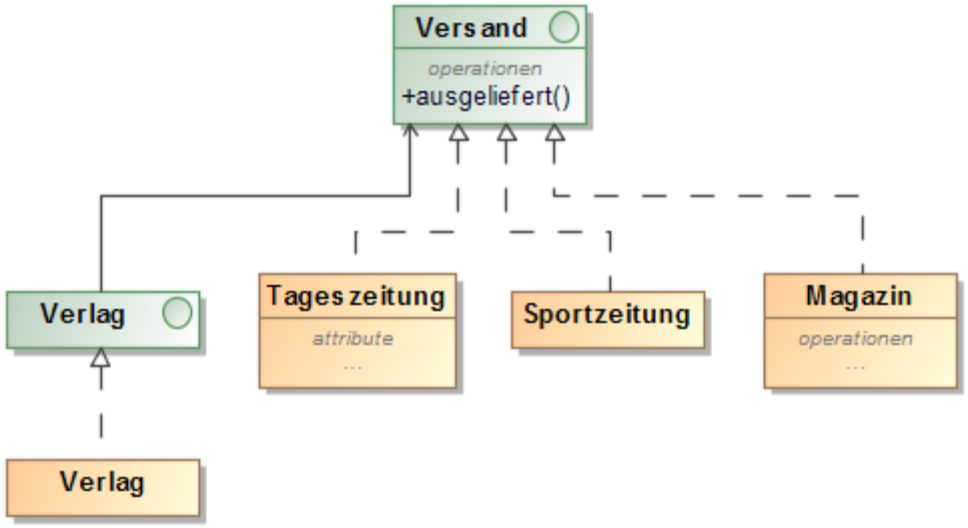


Abbildung 62 Decorator Beispiel mit Observer Pattern

Ein wichtiges Indiz zur Erkennung eines Kandidaten ist das Vorhandensein einer Schleife mit einem *sleep*-Befehl. Dies ist notwendig, um periodische Abfragen durchzuführen. Die pollende Methode wird innerhalb eines eigenen Threads ausgeführt, damit der Rest des Programms nicht blockiert wird. Dazu wird das Interface *Runnable* oder die Klasse *Thread* in Java verwendet. Mit dem Befehl *sleep* wird die Wartezeit zwischen den Abfragen justiert. Ein weiteres Merkmal, neben dem Vorhandensein des Befehls *sleep* innerhalb einer Schleife, ist eine bestehende Verbindung zwischen der Methode, in der *sleep* verwendet wird, zu anderen Klassen. Diese Verbindung sollte in der Annahme existieren, dass ein Überwacher regelmäßig den Zustand einer Gruppe Statusgeber kontrollieren möchte. Der Überwachte muss also alle Statusgeber kennen. Im letzten Schritt sollte überprüft werden, ob alle Verbindungen vom Überwacher zu den Statusgebern, aus einer Methode heraus, aufgebaut werden.

Tabelle 70 Merkmale der Observer Erkennung

Merkmal	Beschreibung
Periodisches Aufrufen	Die Methode zum Aufrufen der Statusempfänger wird innerhalb eines Threads ausgeführt, dessen Wartezeit mit Sleep bestimmt wird.
Verbindungen	Eine Klasse hat viele gerichtete Verbindungen zu anderen Klassen. Diese gehen alle von einer einzigen aufrufenden Methode aus.

5.7.11 Template Method

Ein Template Method erlaubt das Verändern des Verhaltens eines Algorithmus. Dabei wird eine Methode vorgegeben. Diese Methode beinhaltet den eigentlichen Algorithmus. An Stellen, die individualisierbar sind, werden sogenannte Template-Methoden aufgerufen. Für jedes Template kann in der Klasse eine Basisimplementierung existieren (welche z.B. nur einen Standardwert zurückgibt). Soll der Algorithmus nun angepasst werden, geschieht dies durch Vererbung und gezieltes Überschreiben der gewünschten Template-Methoden. Es ist jedoch nicht notwendig, alle Teile neu zu implementieren.

Das von Freeman at el. (Freeman, Robson, Bates, & Sierra, 2004) stammende Template-Method-Szenario ist Ausgangspunkt für das folgende Beispiel. Eine Software für Heißgetränkeautomaten liefert Kaffee- oder Teevariationen. Dabei existiert eine Methode *zubereiten*, welche den entsprechenden Algorithmus ausführt. Jede Getränkeart muss diesen Algorithmus und damit alle Schritte selbst implementieren. Viele Schritte beim Zubereiten von Tee und Kaffee sind gleich. So muss Wasser eingefüllt, heißgemacht und über die Substanz des Getränkes geschüttet werden. Doch in einigen kleinen Schritten, z.B. der Substanz, unterscheiden sie sich. Auch bei den Zusatzstoffen (Zitrone, Honig, Milch) gibt es Unterschiede. Die zugehörigen Klassen (*Kaffee* und *Tee*) überschreiben den Algorithmus aber immer komplett. (vgl. Abbildung 63)

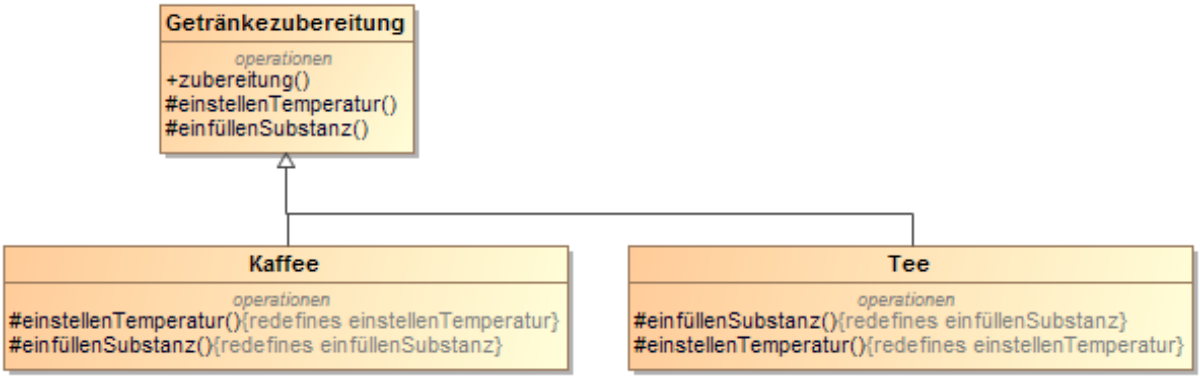


Abbildung 63 Getränkeautomat ohne Pattern

Bei einer Verbesserung durch das Template Method-Pattern würden die Schritte des Zubereitungs-Algorithmus (repräsentiert durch die Methode *zubreiten* in der Klasse *Getränkezubereitung*) in mehrere Teile separiert. Jeder Schritt wird als eine eigene Methode, wie *einstellenTemperatur* und *einfüllenSubstanz*, in der Basisklasse definiert. Dies sind die Templates für die Subklassen. Soll nun die Temperatur bei Tee anders eingestellt werden als bei Kaffee, muss die Klasse *Tee* nur die Methoden *einstellenTemperatur* überschreiben. Der Unterschied gegenüber der ursprünglichen Implementierung ist, dass nicht der gesamte Algorithmus zum Zubereiten überschrieben wird, sondern nur die Teile die jeweils individuell sind. Der eigentliche Algorithmus wird in der Oberklasse implementiert und nur durch die Subklasse individualisiert. So können immer wieder große Teile wiederverwendet werden. Dies macht auch das Anpassen von einzelnen Algorithmen einfacher. Die eigentliche Methode *zubereitung* wird nicht überschrieben. Das Klassendiagramm in Abbildung 64 zeigt die neue Methodenverteilung.

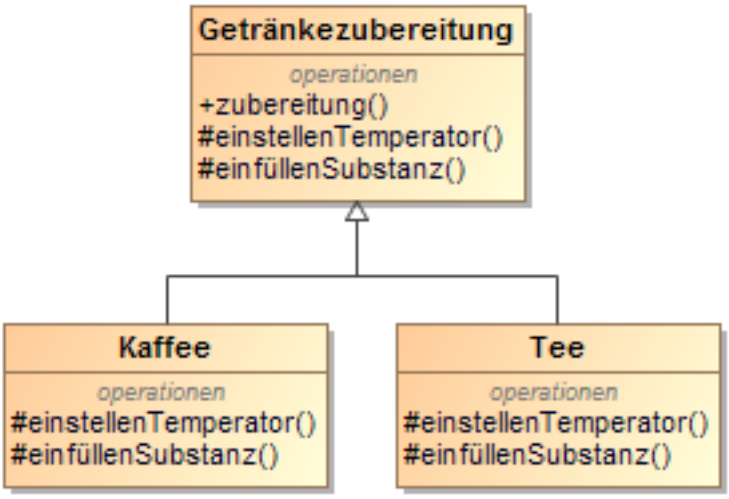


Abbildung 64 Getränkeautomat mit Pattern

Eine andere mögliche Identifikationsart für Kandidaten wäre das Verhalten über einen Parameter festzulegen. Irgendwo wird ein Attribut gesetzt und dieses bestimmt, welches Verhalten aufgerufen werden soll.

Ein Kandidat würde einen Algorithmus mehrfach und immer komplett in Subklassen neu implementieren. Es muss im Quellcode eine Oberklasse mit einer Methode existieren, wobei die Methode den Algorithmus für andere zur Verfügung stellt. Diese Klasse wird von einer Menge anderer Klassen geerbt. Es gibt nur eine Vererbungsebene und es wird immer eben diese eine Methode überschrieben, um den Algorithmus abzuändern, es handelt sich jedoch in den Grundzügen um den gleichen Algorithmus. Anhand des Objekttyps wird entschieden, welcher Algorithmus aufgerufen wird (vgl. Tabelle 71).

Tabelle 71 Merkmale der Template Method Erkennung

Merkmal	Beschreibung
Methodengleichheit	Ein Kandidat würde das Verhalten in unterschiedliche Subklassen ausgliedern und über eine Logik den entsprechenden Algorithmus zur Verfügung stellen.
Verbindung	Es muss im Quellcode eine Oberklasse existieren die eine Anzahl an Methoden zur Verfügung stellt. Diese Klasse wird von eine Menge andere Klassen geerbt.
Schleife	Es werden immer die gleichen Methoden in allen erbenden Klassen überschrieben. Anhand des Objekttyps wird entschieden, welcher Algorithmus aufgerufen wird.

5.7.12 Visitor

Der Beobachter (Visitor) erlaubt das Separieren von Algorithmen von ihren eigentlichen Strukturen. Dadurch können Methoden auf verschiedenen Objekten ausgeführt werden, ohne dass die Objekte in Beziehung zueinander stehen. Außerdem erlaubt das Pattern das schnelle Hinzufügen neuer Logik, welche in unterschiedlichen Klassen aufgerufen werden soll.

Ein solches Verhalten zeigt das folgende Beispiel das auf einem Szenario von Holzner (Holzner, 2006) aufbaut. In einer Personalsoftware soll ein Report über alle Personalkosten (Gehalt, Krankenversicherungsbeitrag) der Unternehmensangehörigen erstellt werden. Die Berechnung der Gehälter unterscheidet sich je nach Gruppe, ebenso die Versicherungskosten. Ein Problem stellt die Zuordnung der Mitarbeiter dar. Jeder Mitarbeiter muss sowohl seiner Anstellungsart, als auch seiner Krankenkassenart zugeordnet werden. Hierzu wurden zwei verschiedene Vererbungshierarchien implementiert. Eine stellt die Mitarbeiter und ihre Anstellungstypen dar, die andere Mitarbeiter und ihre Versicherungstypen. Da die Merkmale unterschiedlich sind und Mehrfachvererbung keine Option ist, existiert ein Mitarbeiter immer in beiden Hierarchien (vgl. Abbildung 65). Zur Berechnung müssen immer alle Elemente der beiden Hierarchien durchlaufen werden. Um die Struktur durchlaufen zu können, muss der Klient natürlich den internen Aufbau kennen. Das gewählte Design ist nicht flexibel. Sollten noch weitere Merkmale wie Rente oder Steuer hinzukommen, müssen weitere Hierarchien hinzugefügt werden.

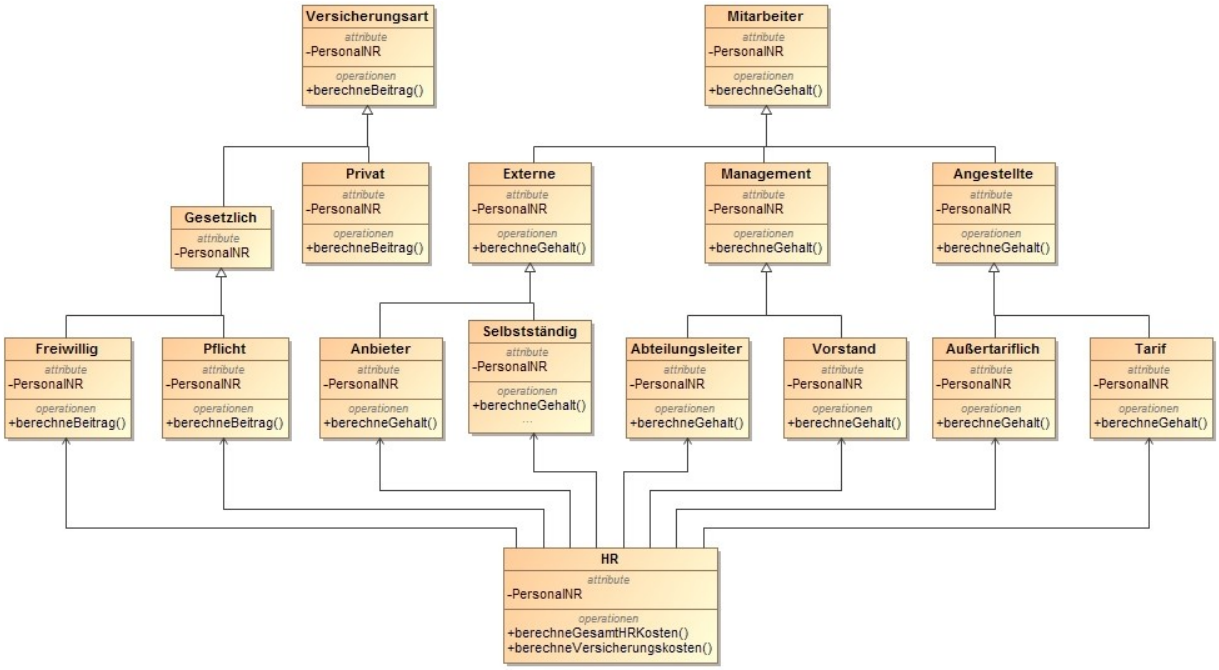


Abbildung 65 Mitarbeitersoftware ohne Pattern

Mit einem Visitor-Pattern kann dieses Verhalten verbessert werden. Die *HR* Klasse kennt nur den Visitor. Dieser durchläuft alle Elemente, ob *Mitarbeitergehalt* oder *–beitrag*. Beide Elemente akzeptieren den Visitor und liefern die gewünscht Zahl zurück. Dafür existiert für jedes Element ein konkreter Visitor, implementiert in *Gehaltsbesucher* und *Versicherungsbesucher*. Bei Bedarf kann auch nur eines der beiden Elemente durchlaufen werden. Das Klassendiagramm im Folgenden zeigt die Implementierung mit Visitor (Abbildung 66).

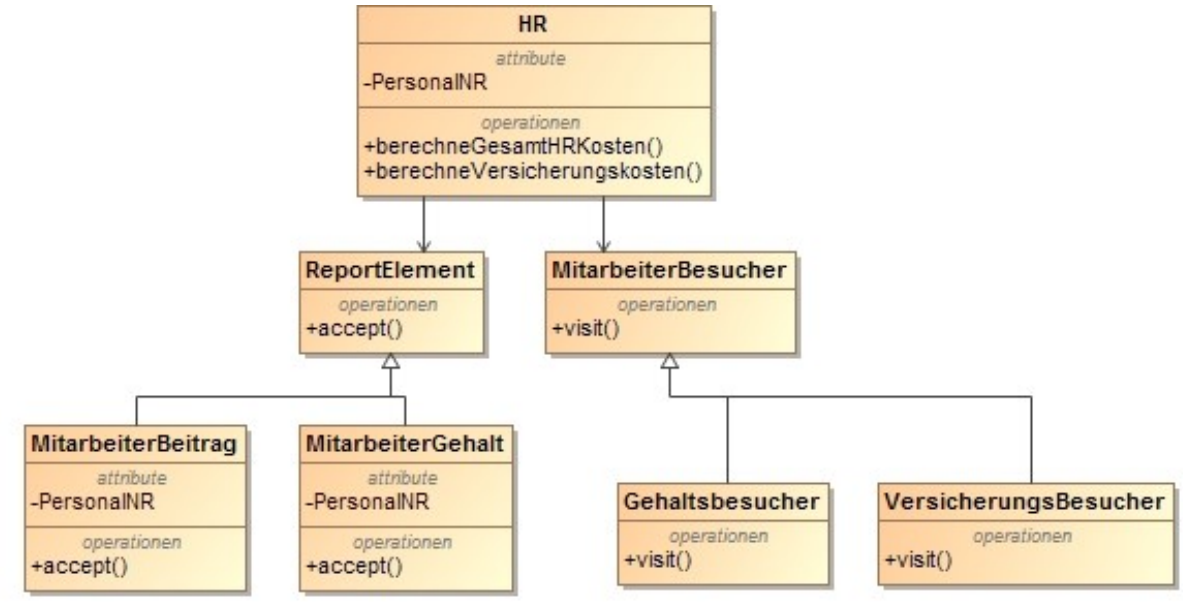


Abbildung 66 Personalsoftware mit Visitor Pattern

Der Einsatz sollte dann fokussiert werden, wenn viele Methoden mit ähnlichen Namen und Parameterlisten in unterschiedlichen Klassen existieren. Daraus folgt auch, dass eine Klasse eine gerichtete Verbindung (nur von ihr weg) zu vielen anderen Klassen hat. Diese Methoden werden über eine Schleife oder ein Array alle zentral und iterativ aufgerufen.

Tabelle 72 Merkmale der Visitor Erkennung

Merkmal	Beschreibung
Methodengleichheit	Es gibt in mehreren Klassen viele Methoden mit ähnlichen Namen und Parameterlisten.
Verbindung	Eine Klasse hat Verbindungen zu allen gefundenen Methoden.
Schleife	In einer Schleife werden alle Methoden aufgerufen.

5.8 Design Patterns ohne mögliche Erkennungsregeln

Dieses Kapitel beschreibt die Design Patterns, für die keine Regeln aufgestellt werden können. Es wird im Einzelnen diskutiert, warum aus Sicht des Autors eine Erkennung im Quellcode für diese Design Patterns nicht möglich ist. Nach aktuellem Stand können für die Patterns Composite, Adapter und Interpreter keine entsprechenden Erkennungsregeln definiert werden. Generell kann für diese drei Patterns gesagt werden, dass für ihren Einsatz Gegebenheiten außerhalb des untersuchten Codes berücksichtigt werden müssen. In den nächsten Abschnitten dieses Kapitel wird auf die drei Design Patterns näher eingegangen.

5.8.1 Adapter

„Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces." nach Gamma et al. (Gamma, Helm, Johnson, & Vlissides, 1994)

Das Adapter-Pattern behebt die Inkompatibilität zwischen zwei Schnittstellen. Dabei wird das Entwurfsmuster genau zwischen die beiden Schnittstellen gesetzt, um die Eingaben der einen

Schnittstelle in einem verständlichen Format für die andere Schnittstelle zu konvertieren. Entscheidet sich ein Entwickler, dieses Entwurfsmuster zu verwenden, basiert die Entscheidung auf Tatsachen, die im Quellcode so nicht zu finden sind. Ein Adapter konvertiert Nachrichten oder Objekte. Damit wird das Pattern nur dann verwendet, wenn ein Subsystem nicht die Sprache der anderen Systeme verwendet. Schon der Einsatz eines solchen Subsystems wird bewusst vom Entwickler getroffen. Bevor mit der Implementierung begonnen wird, steht damit fest, dass ein Adapter notwendig wird. Eine Erkennung im Quellcode, dass ein Adapter benötigt wird, ist somit sinnlos, da der Adapter schon vorhanden sein muss. Sollte dies nicht der Fall sein, so können Subsysteme nicht miteinander kommunizieren, was unlogisch wäre. Ein Entwickler muss bewusst die Entscheidung treffen, eine Komponente zu adaptieren. Deshalb beziehen sich arbeiten zum Thema Adapter-Pattern darauf dem Entwickler bei der automatisierten Erstellung eines Adapters zu unterstützen, wie z.B. Seiffert et al. (Seiffert & Hummel, 2015) beschreibt.

5.8.2 Composite

„Compose objects into tree structures to represent part-whole hierarchies. Composites let clients treat individual objects and compositions of objects uniformly." nach Gamma et al. (Gamma, Helm, Johnson, & Vlissides, 1994)

Das Composite-Pattern bildet Teil-Ganzes-Beziehungen von Objekten in Baumstrukturen ab. Durch den Einsatz des Patterns können Entwickler einfach auf komplexe Objektstrukturen zugreifen. Dabei verbirgt das Composite-Pattern die Komplexität so, dass keine Unterschiede mehr zwischen einem einfachen Objekt und einem komplexen Objekt erkennbar sind. Die Erkennung dieses Patterns stellt sich ebenfalls als zu komplex heraus. Eine Domain-Analyse während der Design-Phase identifiziert Teil-Ganzes-Beziehungen. Mit diesen Informationen sollten Analysten und Entwickler in der Lage sein, festzustellen, wo das Entwurfsmuster möglich und notwendig ist. Eine nachträgliche Erkennung gestaltet sich schwierig, weil die Teil-Ganzes-Beziehungen nie implementiert wurden. Aus einer reinen Code-Analyse sind solche Informationen nicht zu beziehen.

5.8.3 Interpreter

„Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language." nach Gamma et al. (Gamma, Helm, Johnson, & Vlissides, 1994)

Die Motivation zum Einsatz eines Interpreter-Patterns liegt in der Dynamisierung eines Programmablaufes z. B. in der Verwendung von Regeln durch eine bestimmte Grammatik (z. B. SQL). Durch ein solches Konstrukt sind Entwickler in der Lage, Abläufe im Programm einfach zu verändern oder das Verhalten des Programmes entsprechend der Benutzereingabe anzupassen. Die Definition der verwendeten Sprache bzw. die Entscheidung, welche Grammatik verwendet werden soll, wird schon in der Design-Phase der Entwicklung getroffen. In den meisten Fällen repräsentieren solche Grammatiken wichtige Abläufe innerhalb des Programmes. Bei einer nachträglichen Implementierung können hohe Aufwände entstehen. Auch bei diesem Pattern ist der Einsatz eine bewusste Entscheidung, die vor der Implementierung getroffen wird.

5.9 Zusammenfassung

In dieser Sektion werden die Regeln, Regelskizzen nochmals kurz tabellarisch zusammengefasst.

Pattern	Identifikatoren
Builder	Die Basis für diese Regel liegt in der Identifikation des Anti-Patterns Teleskop-Konstruktor im Quellcode, welches sich dadurch auszeichnet, dass eine Klasse eine Vielzahl an Konstruktoren oder Konstruktoren eine große Menge an Übergabeparametern besitzen.
Decorator *(Erkennung Unschaft)*	Die Regel zur Erkennung eines Decorator baut auf der Annahme auf, dass eine Grundklasse immer wieder vererbt wird, um neues Verhalten hinzuzufügen. Im Vergleich zur Verwendung des Patterns werden aber keine Subklassen an die Grundklasse angehängt, sondern für jedes neue Verhalten eine neue Subklasse erstellt. Aus dem Vorgehen resultiert eine große Menge an Subklassen. Außerdem wird in diesem Fall viel Verhalten redundant implementiert.
Facade	Grundlage der Regeln sind die Verbindungen zwischen einzelnen Packages. Es wird davon ausgegangen, dass Packages existieren, deren Inhalte von überall innerhalb des Programmes aufgerufen werden, die aber selbst nur wenig mit dem Programm interagieren. Bei dieser Erkennung werden nur die Verbindungen zwischen Packages verwendet
Mediator	Aufbauend auf dem Anwendungsfall für das Pattern, wurde das Regeldesign darauf ausgelegt komplizierte Kommunikationsverbindungen innerhalb eines Packages zu identifizieren. Bei der Suche werden nur die Unique-Verbindungen innerhalb eines Packages beziffert.
Strategy	Die Auswahl von verschiedenen Algorithmen mit Switch-Anweisungen. Dazu wird in jeder Methode, die einen Algorithmus wählen muss, mindestens eine Switch-Anweisung implementiert. Jede Anweisungen verwendet die gleiche Variable oder Parameter in ihrem Kopf. Mit dieser wird der Algorithmus bestimmt.
State	Mehrere Switch-Anweisungen existieren. Jede Anweisungen implementiert die gleiche Anzahl und Zusammensetzung von Case-Bedingungen haben. Die letzte Bedingung ist, dass die gleiche Variable im Switch verwendet wird.
Factory Method	Identifikation von mehreren Klassen, bei deren Erstellung durch Fallunterscheidung (IF / Switch) der benötigte Klassentype entschieden wird. Die Auswahl welcher Objekttyp für die Instanziierung verwendet wird, sollte an mehreren Stellen im Quelltext durchgeführt werden.
Abstract Factory	Produktfamilien über ihre Verbindungen identifizieren. Eine Klasse A hat nur Verbindung zur Klasse B. Klasse B hat neben Klasse A noch Verbindungen zu C und D, wobei B all diese Klassen auch instanziieren kann. Eine Instanziierung von B/C/D über andere Klassen als A ist nicht möglich. Mehrere Produktfamilien in einer Applikation. Mehrere Quellcodestellen existieren, die entscheiden, welche Produktfamilie instanziiert werden soll.
Prototype	Anzahl an Konstruktorenaufrufe. Wie weit ein Objekt, nach seiner Erzeugung, individualisiert wird. Dies kann einmal über eine hohe Anzahl an Parameter im Konstruktor geschehen. Ein anderes Mal werden für solche Anpassungen, durch Setter Methoden durchgeführt.
Singleton	Ersten sollten nur statische Methoden und statische Variablen in einer Klasse existieren. Zweitens sollte diese Klasse von vielen anderen Klassen aufgerufen werden.

Flyweight	Erster Schritt bei der Erkennung von Kandidaten liegt somit in der Feststellung der Anzahl der Instanzen einer Klasse. Des Weiteren wäre noch möglich den Speicher zur Laufzeit zu analysieren. Im zweiten Schritt der Erkennung wird nach Verbindungen in den zuvor ausgewählten Klassen gesucht werden.
Chain of Respons.	Im ersten Schritt mehrere Quellcodestellen zu identifizieren, die gleiche Fälle verarbeiten und die gleichen Methoden innerhalb dieser Fälle aufrufen. Aus der Menge der gefunden Ergebnisse, werden nun die Fälle herausgesucht die in den einzelnen Fällen immer wieder die gleiche Methoden aufrufen.
Command	Es gibt eine Oberklasse mit mehrere Unterklassen und beinhaltet nur wenige bis gar keine Methoden. Alle Subklassen überschrieben die erbenden Methoden. Die Subklassen besitzen Methoden, welch in der Überschriebenen Methode oder im Konstruktor aufgerufen werden. Die Aufrufende Klasse verwendet InstanceOf zu Entscheidung des Klassentyps.
Memento	Es werden alle Getter einer Klasse aufgerufen und die Daten in einem Objekt des gleichen Typs abgelegt. Dies geschieht an einem zentralen Ort Um die Daten im Objekt des gleichen Typs abzulegen werden die Setter einer Klasse an einem zentralen Ort aufgerufen.
Observer	Eine Klasse hat viele gerichtete Verbindungen zu anderen Klassen ausgehen von eine Methode. Die aufgerufenen Methoden haben den gleichen Namen und die gleichen Parameter. Alle Aufrufe werden in einer Schleife durchgeführt. Innerhalb der Schleife exisitert eine Fallunterscheidung in der entschieden wird welche Klassen gerade abgefragt werden. Ein Thread mit Sleep wird verwendet.
Template Method	Ein Kandidat würde das Verhalten in unterschiedliche Subklassen ausgliedern und über eine Logik den entsprechenden Algorithmus zur Verfügung stellen. Es muss im Quellcode eine Oberklasse existieren die eine Anzahl an Methoden zur Verfügung stellt. Diese Klasse wird von eine Menge andere Klassen geerbt. Es werden immer die gleichen Methoden in allen erbenden Klassen überschrieben. Anhand des Objekttyps wird entschieden, welcher Algorithmus aufgerufen wird.
Visitor	Der Einsatz sollte dann fokussiert werden, wenn viele Methoden mit ähnlichen Namen und Parameterlisten in unterschiedlichen Klassen existieren, die voneinander unabhängig sind. Draus folgt auch dass eine Klasse eine gerichtete Verbindung (nur von ihr weg) zu vielen anderen Klassen hat. Diese Methoden werden über eine Schleife oder ein Array alle zentral und iterativ aufgerufen.
Iterator	1. Bei Composed Datenstrukturen ein, z.B. mehrere Collection-Arten werden kombiniert. Der vorhandene Iterator kann nur durch die eigene Struktur iterieren. 2. Eine Klasse arbeitet mit zwei Datenstrukturen. Bei jeden Aufrufe muss so unterschieden werden, welche Struktur gerade notwendig ist.
Adapter	Keine Erkennung möglich, sondern bewusste Entscheidung durch den Entwickler.
Composite	Keine Erkennung möglich, sondern bewusste Entscheidung durch den Entwickler.
Interpreter	Keine Erkennung möglich, sondern bewusste Entscheidung durch den Entwickler.

6 Das Design Pattern Candidate Detection Tool

„Die Praxis sollte das Ergebnis des Nachdenkens sein, nicht umgekehrt."
- Hermann Hesse (1877 – 1962), dt. Schriftsteller

Um den vorgestellten Ansatz zu evaluieren und zu bewerten, wurden die Erkennungsregeln im DPCDT implementiert. Die in Kapitel 4 gewonnenen Erkenntnisse sind in die Entwicklung eines Empfehlungswerkzeugs eingeflossen. Ziel der Entwicklung war es, Entwicklern ein Werkzeug zur Verfügung zu stellen, dass sie bei der Auswahl und beim Einsatz von Patterns unterstützt. Mit diesem Werkzeug kann unabhängig von der verwendeten Entwicklungsumgebung jeder Java-Quellcode analysiert werden. Im Verlauf der Quellcode-Analyse sammelt das DPCDT Programmstrukturinformationen wie bspw. Verbindungen, Namen und Abhängigkeiten über die Elemente des Programms. Für jedes Design Pattern, das in Kapitel 4.5 diskutiert wurde, sind ferner die zuvor beschriebenen Regeln inklusive der vorgestellten Metriken hinterlegt. Die nachfolgenden Abschnitte beschreiben den Aufbau und die Prozessabläufe des Werkzeuges im Detail.

6.1 AST-Erkennungsmethoden im Detail

Die diversen Vorgehensweisen bei der Erkennung der verschiedenen Design Patterns können, bei einer abstrakten Betrachtung, in einem gemeinsamen Erkennungsmethodenmodell zusammengefasst werden. Dieses Modell und seine Bestandteile werden innerhalb dieses Abschnittes näher diskutiert und sind in Abbildung 67 graphisch dargestellt.

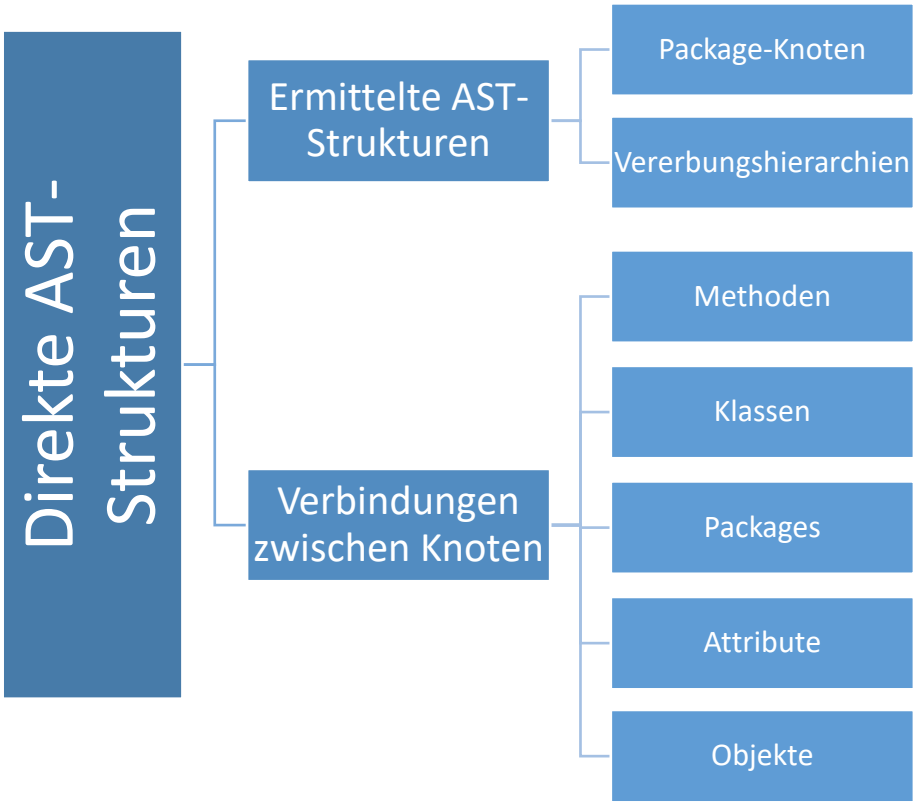


Abbildung 67: Darstellung der Erkennungsmethoden und der Daten, auf denen sie aufbauen

Die folgende Aufzählung erläutert die Unterschiede zwischen den vertikalen Ebenen sowie die Zusammenhänge zwischen den horizontalen Ebenen, die in Abbildung 67 dargestellt sind.

- Direkte AST-Strukturen: Die Erkennungsmethode, auf der einige Metriken aus Kapitel 4.5 aufbauen, verwendet direkt die AST-Knoten. Um Design Patterns zu erkennen, werden bestimmte Ansammlungen von Knoten im AST gesucht. Hierbei wird nach zwei Struktur-Arten gesucht:
    - einer Verkettung von verschiedenen AST-Knoten, z. B. eine Aneinanderreihung einer hohen Anzahl an Case-Knoten nach einem Switch, oder
    - einer Kombination von AST-Knoten mit einer bestimmen Reihenfolge oder eine Struktur, z. B. mehrere Knoten, die verschiedene Methoden repräsentieren, aber die gleichen Parameter übergeben.

- Ermittelte AST-Strukturen: Bei der Methode der ermittelten AST-Strukturen werden weitere Informationen einer Menge von direkten AST-Strukturen zur Pattern-Erkennung gewonnen. Wie bereits erläutert, existiert im AST kein übergreifender Package-Knoten. Es ist jedoch möglich, einen derartigen Knoten aus den Informationen der einzelnen Klassen zu erzeugen. Dies wird durch den Package-Namen des Knotens ermöglicht, den jede zugehörige Klasse enthält. Ein Auffinden gleicher Package-Namen im Knoten verschiedener Klassen ist ein Beispiel für erweiterte Informationsbeschaffung in AST-Strukturen. Die aus dem gesamten AST ermittelten Package-Namen bilden als zusammengesetzte Information den Package-Knoten, welcher jedoch nicht als echter Knoten definiert ist. Eine weitere Quelle zur Informationserhebung nach dieser Methode ist die Suche nach Vererbung der einzelnen Klassen. Für jede Klasse ist definiert, von welcher Klasse sie erbt. Aus diesen Informationen können Vererbungshierarchien aufgebaut werden, welche nötig sind, um die komplexen Beziehungen zwischen Klassen zu analysieren.

- Verbindungen: Diese Methode baut ebenfalls auf den Informationen, die aus dem AST extrahiert werden können, auf. Sie ermittelt Patterns anhand der Daten darüber, welche Verbindungen zwischen den einzelnen Knoten bestehen. Jede Klasse, für die ein AST generiert wird, kennt nur die Kommunikationspartner, mit denen sie direkt kommuniziert, d. h. nur gerichtete Verbindungen von der Klasse weg zu anderen Klassen. Dies beruht auf der Tatsache, dass der AST in den meisten Fällen nur für eine Klasse auf einmal generiert wird. Somit kann aus einer analysierten Klasse im AST nicht auf Verbindungen zu anderen Klassen geschlossen werden, ohne Querverbindungen zwischen verschiedenen Analysen zu ziehen. Kurzum kann aus einem AST einer Klasse A nicht abgelesen werden, ob Klasse B ein Objekt von ihr initialisiert.

  Um die Verbindungsinformationen der verschiedenen AST-Sichten analysieren zu können, werden die einzelnen Informationen in einem Modell kombiniert. Dieses Modell wird innerhalb dieser Arbeit Kommunikationsnetz genannt. Es beinhaltet die gerichteten Verbindungen aller Klassen. Die einzelnen Verbindungen können in verschiedenen Sichten analysiert werden. Auf der obersten Ebene werden die Verbindungen in den Packages gebündelt.

Es kann also eine Aussage darüber getroffen werden, welches Package zu welchem anderen wie viele und welche Verbindungen hat. In jedem Package werden die Verbindungen der einzelnen Klassen abgebildet, auch die Verbindungen aus dem Package heraus. Auf der untersten Ebene können die einzelnen Methodenaufrufe von Klasse zu Klasse analysiert werden.

6.2 Architektur des Design Pattern Candidate Detection Tools

Das *Design Pattern Candidate Detection Tool* (DPCDT) verwendet verschiedene Komponenten. Die Grundlage aller Komponenten bildet Java. Abbildung 68 zeigt die Architektur des DPCDT als Komponenten-Diagramm.

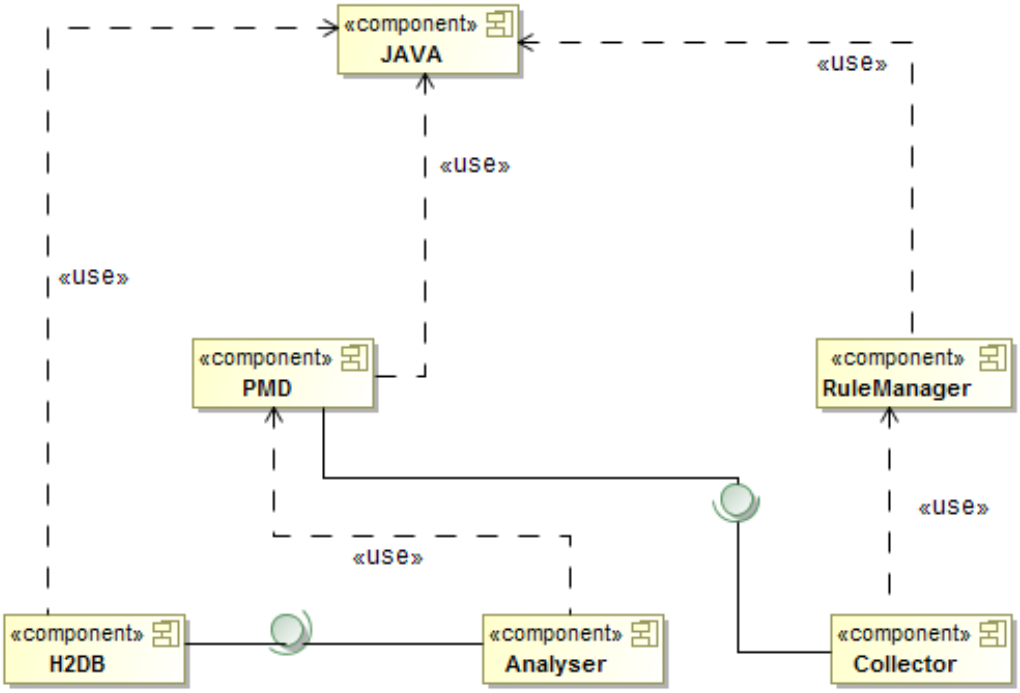


Abbildung 68: Komponenten-Diagramm des DPCDT

Die folgenden Unterkapitel beschreiben die Funktionen und die Zusammenhänge der in Abbildung 68 dargestellten Komponenten.

Für die Kandidatenidentifikation wurde eine neue Regelstruktur in PMD entwickelt. Für die Erweiterung von PMD zur Erkennung von Entwurfsmusterkandidaten wird die Klasse *AbstractJavaRule* vererbt. Diese Variante erlaubt das Erstellen von komplexen Erkennungsarten und erlaubt das Einbinden von weitern Java-Komponenten wie H2 InMemory DB.

Collector

Eine der entwickelten Komponenten zur Analyse ist der *Collector*. Diese Komponente dient als Client für den Visitor vom PMD der den AST durchläuft. Ihm werden alle relevanten Knoten übergeben und von diesem für die spätere Analyse gespeichert. Die Klasse *Collector* erbt von der Klasse *AbstractJavaRule (*vgl. Kapitel 4.5*)* um von PMD als Erweiterung akzeptiert zu werden. Jede Klasse, die von der Klasse *AbstractJavaRule* erbt, wird zum konkreten Visitor. Dadurch ist die Klasse *DesignPatternDetection* in der Lage für jeden AST-Knotentyp eine entsprechende Methode der Klasse *AbstractJavaRule* zu überschreiben. Sobald der Visitor auf einen Knotentype stößt, wird dieser der

Klasse *Collector* zur Analyse, wenn die entsprechende Methode überschrieben wurde. Für die Analyse werden die folgenden acht Methoden überschrieben.

- ASTPackageDeclaration:
  Diese Methode verarbeitet die Knoten, die Informationen des Keyword-Packages speichern. Das Keyword legt fest, zu welchem Package die Klasse gehört. Der Name wird benötigt, um die nachfolgenden Klassen, Methoden etc. einem Package zuzuordnen.

- ASTClassOrInterfaceDeclaration:
  Dieser AST-Knoten beinhaltet die Klassendeklaration mit Klassenname, Vererbung oder Interface. Die Informationen werden verwendet, um den Methoden später eine Klasse zuzuordnen und Klassenhierarchien aufzubauen.

- ASTMethodDeclaration:
  Die Methode verarbeitet die Knoten, welche die Übergabe- und Rückgabeparameter sowie den Namen einer Methode beinhalten. Diese Informationen werden benötigt, um gleiche Methoden in unterschiedlichen Klassen zu finden oder um festzustellen, ob andere Klassen diese Methoden aufrufen.

- ASTConstructorDeclaration:
  Der Konstruktor-Knoten ähnelt der Methodendeklaration, wird aber von einer separaten Methode verarbeitet. Auf diese Weise kann ein Konstruktor von einer normalen Methode unterschieden werden.

- ASTFieldDeclaration:
  Feld-Knoten sind Attribute der Klasse. Für die Analyse von Verbindungen werden nur Knoten benötigt, die komplexe Typen implementieren, deren Basisklasse Teil des Quellcodes ist.

- ASTPrimaryExpression:
  Diese Methode analysiert Knoten, die Methodenaufrufe von einer Klasse zu einer anderen darstellen. Solche Verbindungen werden für das Kommunikationsnetz benötigt. Die Knoten enthalten noch Informationen über die Art und Anzahl der Übergabeparameter und darüber, ob der Aufruf innerhalb der Klasse stattfindet (keyword this) oder in eine andere Klasse oder Superklasse geht.

- ASTLocalVariableDeclaration:
  Knoten, die Attribute innerhalb einer Methode repräsentieren, werden mit dieser Methode analysiert.

- ASTSwitchStatement:
  Diese Methode analysiert den Kopfknoten einer Switch-Anweisung. Kinder des Knotens sind die einzelnen Case-Anweisungen. Aus den Daten werden Informationen zum Verhalten des Programmes gewonnen.

Neben den beschriebenen Knoten besitzt der Collector noch eine Start- und eine Ende-Funktion. Diese beiden Funktionen werden zum Anfang bzw. zum Ende der Analyse des gesamten Baumes ausgeführt. Beim Starten wird die Datenbank initialisiert und am Ende der Analyse wird der

RuleManager aufgerufen, der die Analyseergebnisse verarbeitet und eine abschließende Bewertung durchführt.

RuleManager

Diese Komponente verwaltet alle implementierten Erkennungsregeln, bestehend aus den Grenzwerten und Metriken, die zur Vermessung und Kandidatenevaluierung benötigt werden. Der Manager führt die Regeln der Reihe nach aus. Die Verantwortung zur Abarbeitung wird dabei von den implementierten Regeln getragen. Um eine möglichst hohe Flexibilität bei der Erweiterung neuer Regeln zu bieten, verwendet der RuleManager ein Publish-Subcribe-Pattern in Form des Java-PorpertyChangeListeners. Die einzelnen Regeln werden im Listener des RuleManagers registriert und dieser benachrichtigt sie, sobald ihre Verarbeitung an der Reihe ist.

Rule/Threshold-Collection

Jede Regel wird in einer Klasse unabhängig von anderen Klassen implementiert. Damit können Regeln flexibel angepasst oder vom Benutzer gezielt aktiviert bzw. deaktiviert werden. Jede Regel gehört zur Rule/Threshold-Collection-Komponente. Dabei besteht jede Regel aus zwei Teilen. Teil eins analysiert die vom Collector gesammelten Daten. Teil zwei beinhaltet die Grenzwerte. In einigen Fällen, wie bei den Erkennungsregeln vom Mediator, werden zusätzliche Komponenten verwendet. Die beiden genannten Erkennungsregeln implementieren noch eine Komponente namens *jGraph*, mit der sich Graphen aufbauen lassen. Der Graph wird für den Aufbau der Klassenhierarchien und zur Feststellung von stark zusammenhängenden Komponenten innerhalb eines Kommunikationsnetzes verwendet.

In-Memory-Datenbank H2

Als Persistenz-Schicht und somit zur Aufbewahrung gesammelter Daten aus dem AST und den Analyseergebnissen der Regeln kommt die Open Source-Datenbanklösung H2 (Mueller, 2015) zum Einsatz. H2 implementiert ein relationales Datenbanksystem auf der Basis von Java. Ein Zugriff erfolgt über JDBC und es wird die Abfragesprache SQL unterstützt. Es gab mehrere Gründe, das H2-Projekt zu wählen. Erstens wurde bei der Implementierung von H2 auf eine einfache Architektur geachtet, wodurch schnelle Ausführungszeiten erreicht werden. Dies ist erforderlich, da für jeden gefundenen Knoten ein Zugriff auf die Datenbank erforderlich ist. Des Weiteren sind Kontrollabfragen während der Analyse des AST notwendig. Diese Abfragen verhindern, dass redundante Daten in die Datenbank geschrieben werden, z. B. kann der Name einer Klasse an verschiedenen Stellen vorkommen. Neben der Deklaration verwendet auch jedes Objekt der Klasse den Namen im entsprechenden Knoten. Um zu verhindern, dass die Klasse mehrmals in der Datenbank angelegt wird, ist es notwendig, jedes Mal abzufragen, ob die Klasse schon existiert. Je höher die Anzahl der Klassen, desto mehr Abfragen werden nötig, aufgrund der Anzahl an Einträgen.. H2 kann als In-Memory-Lösung konfiguriert werden, was die Performance solcher Abfragen verbessert. Im DPCDT wird die Datenbank zur Laufzeit im Hauptspeicher gehalten. Dies erhöht die Performance und verringert die Zeit der Analyse. Nach der Regelanalyse wird die Datenbank für spätere Abfragen auf der Festplatte gespeichert.

7 Evaluierung der Ergebnisse

„Zur Erkenntnis der Dinge braucht man nur zweierlei in Betracht zu ziehen, nämlich uns, die wir erkennen, und die Dinge selbst, die es zu erkennen gilt.“

- René Descartes (1596 - 1650), franz. Philosoph, Mathematiker, Naturforscher und Begründer des Rationalismus

Dieses Kapitel beschreibt den Aufbau, die durchgeführten Schritte und die Ergebnisse der Evaluierung des Verfahrens zur Erkennung von Entwurfsmusterkandidaten. Durch das Fehlen von äquivalenten Arbeiten aus diesem Themenbereich war es nicht möglich, eine vergleichende Studie durchzuführen. Aus diesem Grund wurde zur Evaluierung ein sechsstufigen Verfahren entwickelt, wie Abbildung 69 illustriert. Die hier durchgeführte Evaluation ist wesentlich größer als die bisher genannten Studien (siehe 2.4 und vgl. (Aversano, Canfora, Cerulo, Del Grosso, & Di Penta, 2007)).

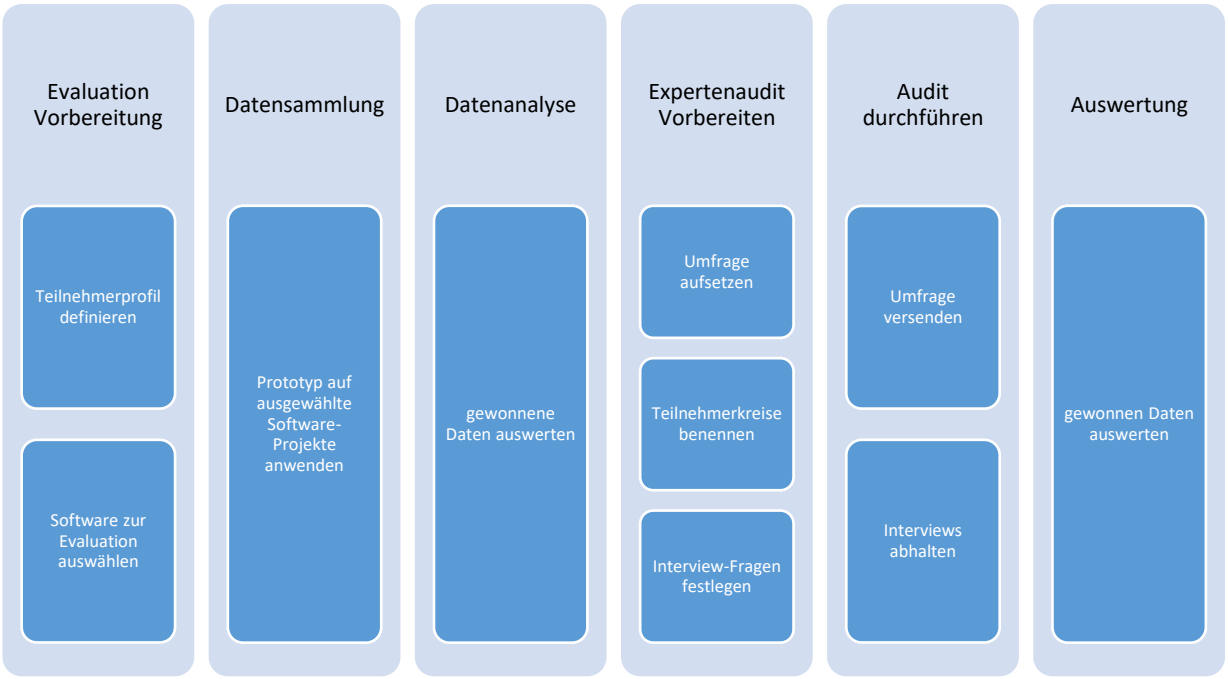


Abbildung 69: Aufbau der Evaluierung der DPCDT-Ergebnisse

Eine Evaluation von Design Patterns zeichnet sich durch eine besondere Komplexität aus. So beschrieben Heuzeroth et al. (Heuzeroth, Holl, Hogstrom, & Löwe, 2003) das Design von Software als schwer greifbar. Eine Evaluation von Entwurfsmusterkandidaten, deren Verbesserung massive Auswirkungen auf das Design hat, fällt somit in eine so schon schwer fassbare Eigenschaft der Software. Hinzu kommt, dass das Design einer Software als etwas gesehen wird, das in den wenigsten Fällen einfach als gut oder schlecht bewertet werden kann. Dies liegt an den vielen weichen Faktoren wie Erfahrung der Entwickler oder Komplexität der Problemstellung, die Einfluss auf das Design haben; wir sprechen hier hauptsächlich von menschlichen Faktoren. So wird ein Problem von unterschiedlichen Entwicklern auch unterschiedlich gelöst und somit auch unterschiedlich bewertet. Nichtsdestotrotz müssen Designs bewertet werden, was nur durch Menschen möglich ist. Die wichtigste Eigenschaft, die ein Entwickler zur Bewertung benötigt, ist Erfahrung.

Hierbei zählt nicht nur die Erfahrung in der Softwareentwicklung, sondern auch Domänenwissen über die Aufgabenstellung. Je höher die Erfahrung, umso besser die Qualität der Bewertung. Doch trotz aller Erfahrung und Kenntnis beschrieb Parnas et al. (Parnas & Clements, 1986) treffend den Einfluss von menschlichen Faktoren auf das Design von Software wie folgt:

„*Human errors can only be avoided if one can avoid the use of humans. Even after the concerns are separated, errors will be made.* " Parnos et al. (Parnas & Clements, 1986)

Eine Evaluation und besonders deren Ergebnis müssen berücksichtigen, dass die Bewertung von Entwurfsmusterkandidaten einen menschlichen Faktor beinhaltet. Es existiert kein Formalismus, der es erlaubt, die genannten weichen, Faktoren aus der Evaluation zu rechnen oder sie zu ignorieren. Aus diesem Grund werden die Kandidatenvorschläge der Evaluation von Menschen mit Programmiererfahrung evaluiert.Grundsätzlich handelt es sich bei dieser Evaluation um eine *positivist study* (vgl. (Klein & Myers, 1999)). Runeson et al. (Runeson & Höst, 2009) definieren diese Art der Studie wie folgt:

„A positivist case study searches evidence for formal propositions, measures variables, test hypotheses and draws inferences from a sample to a stated populations…"

Die hier genannte Hypothese der Evaluation basiert auf dem Grundziel dieser Arbeit, nämlich, dass es möglich ist, Entwurfsmusterkandidaten im Quellcode zu erkennen und zu bewerten. Um den Einfluss solcher Faktoren einzugrenzen, wurde der Kreis der Teilnehmer begrenzt. Dazu wurde ein Teilnehmerprofil entwickelt, das die Auswahl geeigneter Personen eingrenzt. Einer der wichtigsten Punkte ist, dass die Personen Grundwissen in Software-Entwicklung, dieses kann durch mehrjährige Arbeitserfahrung oder durch ein abgeschlossenes Studium nachgewiesen werden, sowie zu Design Patterns, im Speziellen zu den hier verwendeten Patterns, mitbringen müssen. Personen, die nicht in das Profil fallen, wurden nicht zur Umfrage zugelassen und ihre Ergebnisse nicht berücksichtigt. Die Auswahl dieses Teilnehmerkreises fördert eine einheitliche Herangehensweise an die Bewertung der Kandidaten-Beispiele, da alle Teilnehmer ein Grundverständnis für Probleme in der Softwareentwicklung besitzen.
Auch beim erwarteten Ergebnis spielen die weichen Faktoren eine Rolle. Sind ‚gut' und ‚schlecht' nicht einfach greifbar, so gibt es immer einige Zwischenmeinungen. Eine vollständige Zustimmung oder Ablehnung für ein Beispiel durch alle Teilnehmer ist sehr unwahrscheinlich. Es geht bei der Evaluation deshalb um das Bestimmen von Tendenzen: Stimmt der Großteil der Teilnehmer eher zu oder sind sie eher gegen die Verwendung des Patterns? Die Begrenzung des Teilnehmerkreises auf Personen, die Wissen in den zuvor genannten Bereichen besitzen, schränkt natürlich die Anzahl der Rückmeldungen ein. Viele Personen mit einem entsprechenden Fundus an Wissen sind schwer zu finden. Die Studie wurde deshalb nur in einem kleinen Kreis verteilt. Dieser Kreis fand sich in den Softwareentwicklungsabteilungen und -bereichen von verschiedenen Firmen und Universitätseinrichtungen. Zuerst wurden verschiedene Open Source-Programme mit dem DPCDT analysiert und die Ergebnisse statistisch evaluiert. Im zweiten Schritt wurden die ermittelten Ergebnisse von einer Gruppe aus 52 erfahrenen Java-Experten auditiert, indem diese die einzelnen Pattern-Kandidaten aus ihrer Sicht evaluierten. Zuletzt wurden die Meinungen und Ergebnisse aus dem Audit ausgewertet.
In den nachfolgenden Abschnitten werden Aufbau und Resultate der einzelnen Schritte im Detail besprochen.

7.1 Herausforderung bei der Evaluierung

Generell stellt die Evaluierung von interner Software-Qualität eine nicht zu unterschätzende Herausforderung dar. Die hier vorliegende Methode verwendet den Quellcode als Grundlage ihrer Analyse. Diese Basis beinhaltet die meisten verfügbaren Informationen über ein Programm, jedoch gibt es Grenzen, z.B. nichtfunktionale Anforderungen oder fachliche Vorgaben, die im Quellcode zwar umgesetzt sind, wobei aber aus den Codezeilen meist nicht hervorgeht, warum ein entsprechendes Design gewählt wurde. Daraus folgt, dass das DPCDT unter Umständen Kandidaten erkennt, die False-Positive sind, d.h. der Kandidat wird erkannt, macht aber an dieser Stelle wenig Sinn.

Schwieriger zu identifizieren sind offensichtlich False-Negative. Wenn das DPCDT einen möglichen Kandidaten nicht identifiziert, kann dieser nur durch manuelles Quellcodereview, welches aufwendig ist und nicht zu einem passenden Ergebnis kommen muss, gefunden werden. Grundlage von False-Negative können Szenarien sein, die nicht von den Erkennungsregeln berücksichtigt werden, z.B. kann ein Facade Pattern auch Sinn machen, wenn es mehrere Packages von Rest des Programmes abkapselt. Gerade solche seltenen Szenarien und Ausnahmen werden nicht immer von den Erkennungsregeln erfasst und sind ggf. als zukünftige Arbeiten zu betrachten, die den vorgestellten Ansatz erweitern können.

7.2 Evaluierung von Open Source-Projekten

Das in Kapitel 6 beschriebene DPCDT wurde verwendet, um 25 Open Source-Projekte zu analysieren. Durch diese Analysen wurden geeignete Entwurfsmusterkandidaten identifiziert. Bei der Untersuchung der ausgewählten Projekte wurden 3.632.734 LoC analysiert, welche in 8.240 Dateien implementiert waren, 44.775 Klassen umfassten und 318.277 Methoden beinhalteten. In anderen Studien sind es selten mehr als fünf Vergleichsprogramme.

7.2.1 Auswahl der verwendeten Projekte

Um eine aussagekräftige Evaluation zu erreichen, liegt der wichtigste Aspekt bei der Auswahl von Projekten darin, dass diese Projekte in einem produktiven Kontext sind, d.h. alle Projekte haben einen Reifegrad und einen Funktionsumfang, dass sie eingesetzt werden könnten. So beschreiben Yin et al (Yin, 1981) die Vorgabe bei der Auswahl von Objekten für eine Studie wie folgt:

„Contemporary phenomenon in its real-life context“ (Yin, 1981)

Dies bedeutet, jedes Projekt wird produktiv von mehreren Usern angewendet. Auf der anderen Seite beschreiben Runeson et al. (Runeson & Höst, 2009), dass eine Auswahl keine „Toy programs“ enthalten soll, weil diese keinen „real-life“-Kontext besitzen und deren Ergebnisse somit ungeeignet sind.

Für die Evaluation müssen die auszuwählenden Projekte neben den schon vorgestellten Aspekten noch die folgenden Merkmale erfüllen.

- **Quellcode frei verfügbar:** Damit das Programm analysiert werden kann, muss der komplette Quellcode verfügbar sein. Dies begrenzt, mangels Verfügbarkeit von anderen Projekten, die Auswahl auf Projekte, deren Code auf Plattformen wie GitHub (GitHub, Inc, 2015) oder Apache.org (The Apache Software Foundation, 2015) zur Verfügung gestellt wird und die damit unter dem Begriff Open Source zusammengefasst werden können. Da der Quellcode nicht weiterverwendet werden sollte, gab es keine Einschränkungen hinsichtlich der verwendeten Lizenzen.

- **Java:** Wie schon zuvor erwähnt, beschränkt sich die Analyse dieser Arbeit auf die Programmiersprache Java. Dementsprechend besteht die Sammlung nur aus Programmen, die hauptsächlich Java-Quellcode enthalten. Es gab keine Einschränkungen hinsichtlich des Programm-Umfangs. Auch hier ist die Menge an verfügbaren Projekten groß genug, sodass es keine Einbußen in der Qualität gibt.

- **Große Variation:** Software-Projekte unterscheiden sich in verschiedensten Merkmalen voneinander, wie z. B. Anwendungsgebiet. Die Auswahl an Projekten für die Evaluation wird eine große Menge an Variationsmöglichkeiten abdecken.

  Anwendungsgebiet: Die Auswahl an Projekten deckt verschiedenste Anwendungsgebiete ab – von E-Mail-Clients über Heimautomatisierungssysteme und Modellierung bis zu Spielen. Durch eine Vielzahl von Anwendungsgebieten werden unterschiedliche Software-Designs, Arten der Implementierung und Problemstellungen in die Evaluations eingebracht.

  Code Size: Ein weiteres Merkmal eines Projektes ist die Code Size, welche meist in LoC gemessen wird. So besitzt das kleinste Projekte der Auswahl, Apache Mina (Apache MINA, kein Datum), 144 Klassen und 6.338 LoC. Das größte Projekt, Wind (Athens Wireless Metropolitan Network, 2015), besitzt 668.968 LoC mit 3.325 Klassen. Im Vergleich dazu besitzt das Projekt Elasticsearch (Willnauer, 2015) nur 265.892 LoC, jedoch 5.674 Klassen. Für die LoC wurden nur die ausführbaren LoC gezählt, ohne Kommentare und Leerzeilen. Des Weiteren wurde die Anzahl der Funktionen und Dateien bestimmt. Alle diese Kennzahlen sind in Tabelle 73 aufgeführt. Die genannten Kenngrößen wurden mit dem Code-Analyse-Werkzeug Understand von SciTools (Scientific Toolworks, Inc., 2015) in der Version 3.0 gesammelt.

  Entwicklungsstand: Weitere Unterschiede zeigen sich beim Entwicklungsstand der einzelnen Projekte. Jüngstes Projekt nach Versionsnummer ist ArgoUML mit Version 0.34. Die höchste Versionsnummer besitzt Apache Wicket (Apache Wicket, 2015) mit 6.18.0. Ebenfalls wurden Projekte aus unterschiedlichen Anwendungsgebieten, z. B. Modellierung etc., ausgewählt. Keines der 25 Projekte implementiert ähnliche Funktionen und überschneidet sich höchstens ein wenig in ihrem Einsatzbereich.

  Herkunft: Die verschiedenen Projekte stammen aus vier unterschiedlichen Quellen: GitHub (GitHub, Inc, 2015), SourceForge (Dice Holdings, Inc, 2015), Apache.org (The Apache Software Foundation, 2015) und Tigris.org (CollabNet, 2015). Eine Verwendung unterschiedlicher Quellen bringt eine höhere Variation von beteiligten Entwicklern und Projektlaufzeiten mit sich. Das Alter der Projekte auf GitHub ist meist geringer als das bei SourceForge. Projekte von Apache besitzen in den meisten Fällen eine hohe Lebenszeit, so z. B. das Apache-Webserver-Projekt (HTTP Server Apache, 2015), dessen erste Version 1995 veröffentlicht wurde. Die unterschiedlichen Aspekte der Projekte werden in der Diskussion der Ergebnisse noch näher betrachtet.

- **In anderen Studien verwendet:** Es fanden sich Studien (siehe Kapitel 2) aus anderen Bereichen des Software-Engineerings wie Design Pattern-Erkennung oder Software-Metrik-Evaluierung. Einige dieser Studien verwenden ebenfalls Open Source-Projekte für ihre Analysen und Evaluierungen, wie z. B. ArgoUML oder jHotDraw (Aversano, Canfora, Cerulo, Del Grosso, & Di Penta, 2007). Diese wurde bei der Auswahl an Software für diese Evaluation berücksichtigt und einige Programme, die zuvor schon in anderen Studien verwendet wurden, sind auch hier enthalten. Durch dieses Vorgehen können zu einem späteren Zeitpunkt Ergebnisse gemeinsam evaluiert werden.

Unter Verwendung der oben genannten Einschlusskriterien entstand die folgende Liste von 25 Projekten. Die letzte Spalte beinhaltet die Version, die bei der Analyse verwendet wurde. Bei GitHub-Projekten wurde der Master des Tages der Vermessung verwendet.

Tabelle 73: Kenngrößen der ausgewählten Softwareprojekte

Projekt	Domäne	LoC	Klassen	Dateien	Funktionen	Version
ArgoUML	Modellierungstool	156.296	2.278	1.671	16.127	0.34
Columba	E-Mail-Client	94.946	1.764	1.427	9.279	1.4
JEdit	Editor	110.736	1.242	543	7.472	5.2
Apache Lucene	Bibliothek für Textsuche	332.897	4.067	2.258	22.259	4.10.3
JHotDraw	Malprogramm	80.140	1.066	679	7.663	5.6
Apache Ant	Build-Management	107.044	1.286	861	10.580	1.9.4
Apache Wicket	Web-Framework	143.177	3.063	1.880	14.561	6.18.0
Ganttproject	Office-Tool	54.392	1.416	639	6.954	2-6-1-r1499
Jrefactory	Entwickler-Tool	110.853	1.542	1.172	10.737	2.9.19
OpenHab	Heim-Automation	199.568	3.155	2.233	17.508	1.6
Freedomotic	Internet-of-Things-Framework	43.884	742	464	4.253	5.5.0
Jfreechart	Java-Graphen-Bibliothek	98.334	654	629	9.075	1.0.19+
Junit	Testing	9.263	278	195	1.404	r4.12
Recoder	Framework für Quellcode-Transformation	71.182	584	462	7.661	0.97
Jenkins	Continuous Integration Server	78.071	2.030	926	9.885	1.598
Wind	Web-Applikation für drahtlose Netze	668.968	3.325	2.620	58.321	1.0.1
Derby	Datenbank	345.618	1.972	1.743	26.492	10.11.1.1
Elasticsearch	Such- und Analyse-Engine	265.559	5.673	3.025	30.528	1.4.4
Freemind	Mind Manager-Clone	45.501	749	370	4.908	1.0.1
Hibernate	Object/Relational Mapper	35.509	745	625	4.367	4.5.2
Jabref	Referenzenverwaltung	92.864	1.623	695	6.568	2.10
Megamek	3D-Actionspiel	267.892	2.217	1.770	14.570	0.40.1
Mina	Network Application Framework	6.338	144	126	654	2.0.9
spring-core	Java-Framework	29.568	453	348	2.983	4.1.5
Triplea	Rundenbasiertes Strategiespiel	184.134	2.707	879	13.468	1.8.0.5
Gesamt		3.632.734	44.775	28.240	318.277	

Alle Analysen durch das DPCDT wurden in der gleichen Testumgebung durchgeführt. Als Messplattform diente ein Intel Core i7 4710HQ mit 16GB Hauptspeicher und einer M.2 256GB SSD-Festplatte. Je nach Größe des Projektes dauerte die Analyse zwischen 1 Minute und 3 Stunden. Je höher die Anzahl an Klassen, desto länger die Laufzeit der Analyse. Die Analysezeit wird beeinflusst von der Anzahl der Klassen und deren Verbindungen untereinander. Hintergrund der Erhöhung der Analysezeit ist die Prüfung der Klassen innerhalb der Datenbank. Damit eine Klasse oder eine Verbindung nicht redundant vom DPCDT gemessen wird, ist es immer wieder notwendig, zu überprüfen, ob das untersuchte Objekt schon bekannt ist. Aus diesem Grund sind viele Abfragen auf die Datenbank notwendig, was die Laufzeit verlängert.

7.2.2 Übersicht der Ergebnisse

Das DPCDT ermittelte insgesamt 1.913 Entwurfsmusterkandidaten in den 3,632 Millionen LoC. Davon wurden die meisten Vorschläge, 45,1% aller Kandidaten, für die Regel zur Erkennung von Builder-Pattern-Kandidaten entdeckt. Hier wurden 863 Kandidaten identifiziert. Die wenigsten Kandidaten wurden für das Mediator-Pattern erkannt. Entwurfsmuster wurde nur in 113 Fällen vorgeschlagen. Im Gesamten wurde für die hier behandelten Entwurfsmuster die in Tabelle 74 gezeigte Anzahl an Kandidaten gefunden. Die Tabelle zeigt sowohl die Gesamtanzahl an Kandidaten als auch die einzelnen Kandidaten nach Erkennungsstufen.

Tabelle 74: Ergebnisse des DPCDT nach Erkennungsstufen

	Gesamt	Empfohlen	Möglich	Sinnvoll	% an Gesamt
Builder	863	59	327	477	45,11%
Decorator	609	212	165	232	31,83%
Facade	117	31	72	14	6,12%
Mediator	113	53	60	0	5,91%
Strategy + State	211	23	157	31	11,03%
Gesamt	1913	378	781	754	100,00%

Nach der Durchführung aller Analysen ergaben sich die folgenden Einzelergebnisse, die in Tabelle 75 nach Projekt geordnet aufgeführt sind.

Tabelle 75: Einzelergebnisse nach Pattern-Art und Erkennungsstufe (E=Empfohlen, S=Sinnvoll und M=Möglich)

	Builder			Decorator			Facade			Mediator			Strategy			Gesamt
	E	**S**	**M**	**E**	**S**	**M**	**E**	**S**	**M**	**E**	**S**	**M**	**E**	**S**	**M**	
ArgoUML	3	9	18	10	9	15	0	4	2	2	2	0	0	2	1	77
Columba	1	20	16	13	10	13	3	2	1	1	2	0	0	3	0	85
Jedit	4	6	9	8	4	8	2	1	0	6	3	0	0	5	1	57
Lucene	7	27	51	32	24	25	2	2	0	3	2	0	0	26	6	207
JHotDraw	0	16	9	6	4	4	2	2	1	0	1	0	0	11	2	58
Ant	5	14	19	2	5	4	2	1	1	2	2	0	0	6	0	63
Wicket	11	39	57	8	10	13	3	0	1	0	2	0	0	1	0	145
Gantt-projekt	0	6	5	4	3	18	2	6	1	1	4	0	0	1	0	51
jrefactory	1	7	10	15	6	11	2	2	1	2	4	0	0	3	0	64
openhab	0	19	13	22	7	21	3	20	1	1	4	0	0	13	3	127
freedomotic	1	6	3	6	3	3	1	2	0	0	3	0	0	0	0	28
jfreechart	0	8	15	6	6	5	1	0	0	1	0	0	0	1	0	43
Junit	1	2	0	2	1	4	0	1	1	0	0	0	0	0	0	12
recoder	2	13	83	1	5	3	0	1	0	1	1	0	0	5	2	117
Jenkins	2	16	21	6	14	9	0	0	0	3	1	0	0	0	0	72
Wind	2	42	29	13	5	5	2	5	1	5	5	0	5	29	7	155
Derby	0	15	28	11	9	13	1	3	1	9	8	0	4	25	2	129
Elastic-search	12	25	51	15	14	23	0	7	1	1	3	0	0	4	0	156
freemind	0	1	2	4	2	10	0	1	0	3	3	0	0	1	0	27
hibernate	0	5	1	0	3	7	1	2	0	0	0	0	1	1	0	21
Jabref	2	16	9	9	7	6	0	4	0	3	2	0	0	0	0	58
megamek	4	7	11	9	5	6	2	3	0	5	2	0	13	14	7	88
Mina	0	3	2	0	1	0	0	0	0	0	0	0	0	0	0	6
spring-core	0	2	5	0	1	1	2	0	0	1	1	0	0	3	0	16
Triplea	1	3	10	10	7	5	0	3	1	3	5	0	0	3	0	51
Gesamt	59	327	477	212	165	232	31	72	14	53	60	0	23	157	31	1913

7.2.3 Diskussion der Kandidatenhäufigkeit

Dieses Kapitel diskutiert die Ergebnisse der einzelnen Messungen und stellt Vergleiche zwischen den Kennzahlen der einzelnen Projekte und den gefundenen Kandidaten an. Um einen Vergleich zu ermöglichen, wurde eine Rangliste, basierend auf der Gesamtsumme der gefundenen Kandidaten pro Design Pattern, und, für die meisten Kandidaten, nach Erkennungsstufe erstellt. Tabelle 76 zeigt die verschiedenen Ränge pro Pattern.

Tabelle 76: Rangfolge nach Anzahl der gefundenen Kandidaten und nach Erkennungsstufe

	Gesamt		Empfohlen		Möglich		Sinnvoll	
	Rang	Anzahl	Rang	In%	Rang	In%	Rang	In%
Builder	1	863	5	7%	4	38%	1	55%
Decorator	2	609	2	35%	5	27%	2	38%
Facade	4	117	3	26%	2	62%	4	12%
Mediator	5	113	1	47%	3	53%	5	0%
Strategy + State	3	211	4	11%	1	74%	3	15%
Gesamt		1913		20%		41%		39%

Von der reinen Anzahl an Kandidaten wurden mit 863 (vgl. Tabelle 76) die meisten für das Builder-Pattern identifiziert. Von diesen Kandidaten fallen nur 7% in die Kategorie *empfohlen*. Diese 7% besitzen somit alle sechs oder mehr Konstruktoren für eine einzige Klasse (vgl. mit Kapitel 4.5). Bei 55% der Kandidaten wurden drei Konstruktoren identifiziert, und basierend auf den festgelegten Grenzwerten wäre ein Builder somit *möglich*. In 1% der untersuchten 44.775 Klassen wurden somit mindestens 3 Konstruktoren für ein und diese selbe Klasse implementiert. Die Entscheidung obliegt dem Entwickler, ob die 3 Konstruktoren ausreichend sind für das Builder-Pattern. Das sollte von der Komplexität dieser Klassen abhängig gemacht werden.

An zweiter Stelle in der Rangfolge steht das Decorator-Design Pattern, mit 609 Kandidaten (vgl. Tabelle 76). Eine Erkennung basiert, wie in Kapitel 5.2 beschrieben, auf der Klassenebene. Je mehr Klassen ein Projekt hat, desto größer ist die Chance, einen Kandidaten zu finden – mit der Einschränkung, dass die Klassen eine Beziehung in Form von Vererbung besitzen müssen. Werden die Grenzwerte in die Betrachtung einbezogen, ergibt sich ein interessantes Bild. Die unterste Grenze für einen Decorator-Pattern-Kandidaten liegt bei 4 vererbten Klassen. Jede 74. Klasse gehört somit zu einer Vererbungshierarchie, die in der ersten Ebene mehr als 4 Klassen besitzt. Ein weiterer interessanter Punkt dieser Ergebnisse: 35% der Kandidaten besitzen die höchste Erkennungsstufe *empfohlen*. Damit kann die Aussage getroffen werden, dass 0,5% aller Klassen zu einer Hierarchie gehören, die mehr als 8 Klassen auf der ersten Ebene besitzt. Insgesamt zeigt das Ergebnis für dieses Pattern nahezu eine Gleichverteilung über die Kategorien mit 35% zu 27% zu 38%. Die Abbildung 70 zeigt die Verteilung nach Erkennungsstufen pro Pattern in Prozent.

An dritter Stelle folgen die gemeinsamen Ergebnisse für Strategy-Pattern mit 211 Kandidaten. 74% aller Kandidaten fallen hier in die Erkennungsstufe *sinnvoll*, weitere 15% unter *möglich* und 11% sind *empfohlen*. Jede der Strategien könnte ein State sein. Da die State-Erkennungsregel sehr detailliert gestaltet wurde und das State Pattern selbst nur selten verwendet wird, gab es keinen direkten *empfohlenen* Kandidaten für ein State Pattern. Da beide Patterns höchst komplex in ihrer Anwendung sind, werden sie von Entwicklern eher selten eingesetzt. Bei den 23 gefundenen Kandidaten (vgl. Tabelle 75) mit der Kategorie *empfohlen* besitzt, gemäß der Regel aus Kapitel 4.5, jeder Kandidat vier Methoden, und in jeder dieser Methoden gibt es mindestens ein Switch, das die

gleichen Cases verwendet. Es besteht die Möglichkeit, dass sich hier ein State-Kandidat verbirgt, der von seiner Struktur unsauber in das Switch eingefügt wurde.

Kandidaten für das Facade-Entwurfsmuster liegen mit 117 (vgl. Tabelle 76) an der 4. Stelle. Der Großteil der Kandidaten fällt in die Kategorie *sinnvoll*. Es wurden einige Kandidaten mit sehr hohen eingehenden und ausgehenden Verbindungen identifiziert, doch die dritte Metrik „Anzahl der internen Verbindungen" sorgt bei einigen Kandidaten für die Einstufung in eine niedrigere Erkennungsstufe. Eine Facade sorgt für eine klare Abtrennung, doch sie ist nicht überall zweckmäßig. Bei einigen der 12% der Packages mit der Erkennungsstufe *möglich* und hohen Werten in 2 der 3 Metriken sollte von den Entwicklern überprüft werden, ob Abhängigkeiten nicht auf andere Weise verringert werden könnten. Dies zeigt, dass auch ohne die Kategorie *empfohlen* oder ohne den Einsatz des vorgeschlagenen Design Patterns der Quellcode verbessert werden kann.

Knapp hinter der Facade liegen die 113 Kandidaten (vgl. Tabelle 76) des Mediator-Patterns. Interessanterweise wurden hier keine Kandidaten für die Erkennungsstufe *Möglich* gefundenen. Daraus folgt, dass die Klassen entweder mit wenigen anderen Partnern kommunizieren und damit unter der Schwelle von zwei Partnern liegen oder mindestens mit drei Partnern kommunizieren. Bei den gefundenen Kandidaten kann von einer 50-50-Verteilung gesprochen werden. Hier ist zu beachten, dass die Grenzen für die Erkennungsstufe *Möglich* sehr eng gezogen sind. Es müssen genau drei Partner sein. Somit können in der höchsten Kategorie auch Kandidaten sein, die sehr viele Kommunikationspartner haben. Es fand sich z. B. in den Apache-Derby-Ergebnissen ein Kandidat, bei dem 34 Klassen miteinander verbunden waren.

Die folgende Abbildung 70 illustriert noch einmal die wichtigsten Merkmale der Ergebnisse. Sie zeigt die Verteilung der gefunden Kandidaten in Prozent pro Kategorie (von links nach rechts: Sinnvoll, Möglich, Empfohlen) sowie deren Anzahl welche als Volumen der Kreise dargestellt werden.

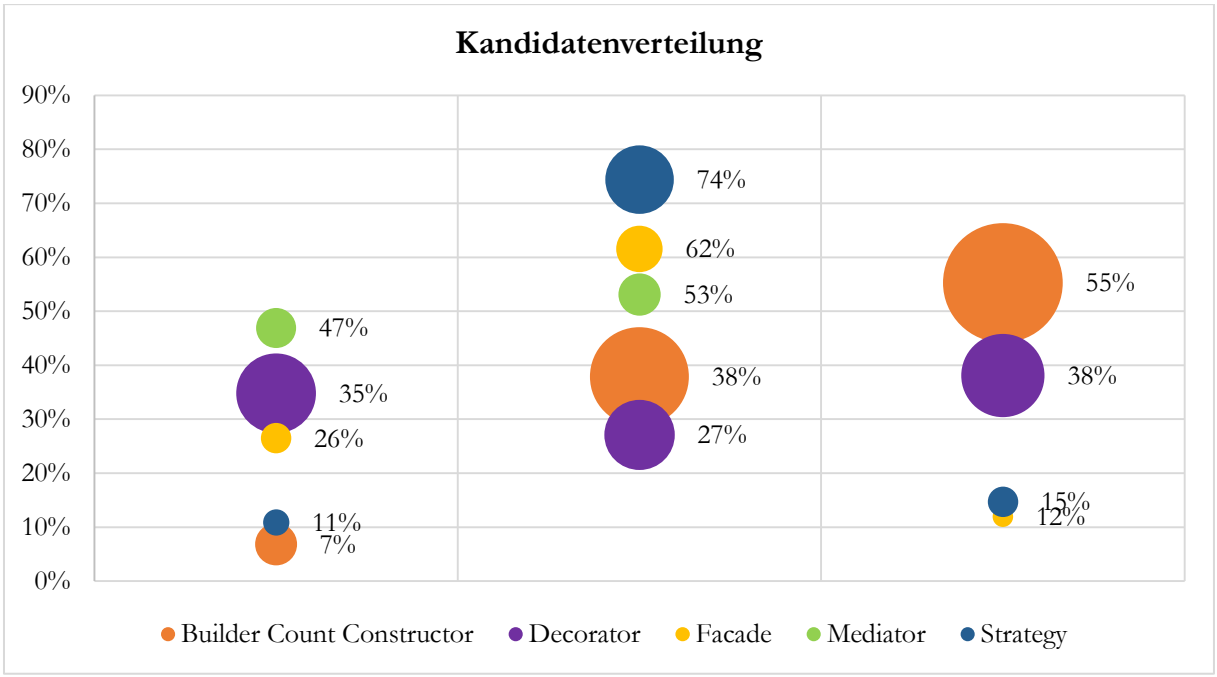


Abbildung 70: Verteilung des Auftretens der einzelnen Ergebnisse in Prozent.

7.2.4 Wechselbeziehung der Ergebnisse

Die erste Betrachtung führt zu einem interessanten Punkt der Diskussion, an dem zu klären ist, ob eine Wechselbeziehung zwischen der Größe der Programme, gemessen an LoC oder der Anzahl der Klassen, und der Anzahl an identifizierten Entwurfsmusterkandidaten besteht. Um diese Frage zu beantworten, wurde ein einfaches Model, basierend auf der GQM-Vorgehensweise, aufgestellt (Van Solingen, Basili, Caldiera, & Rombach, 2002).

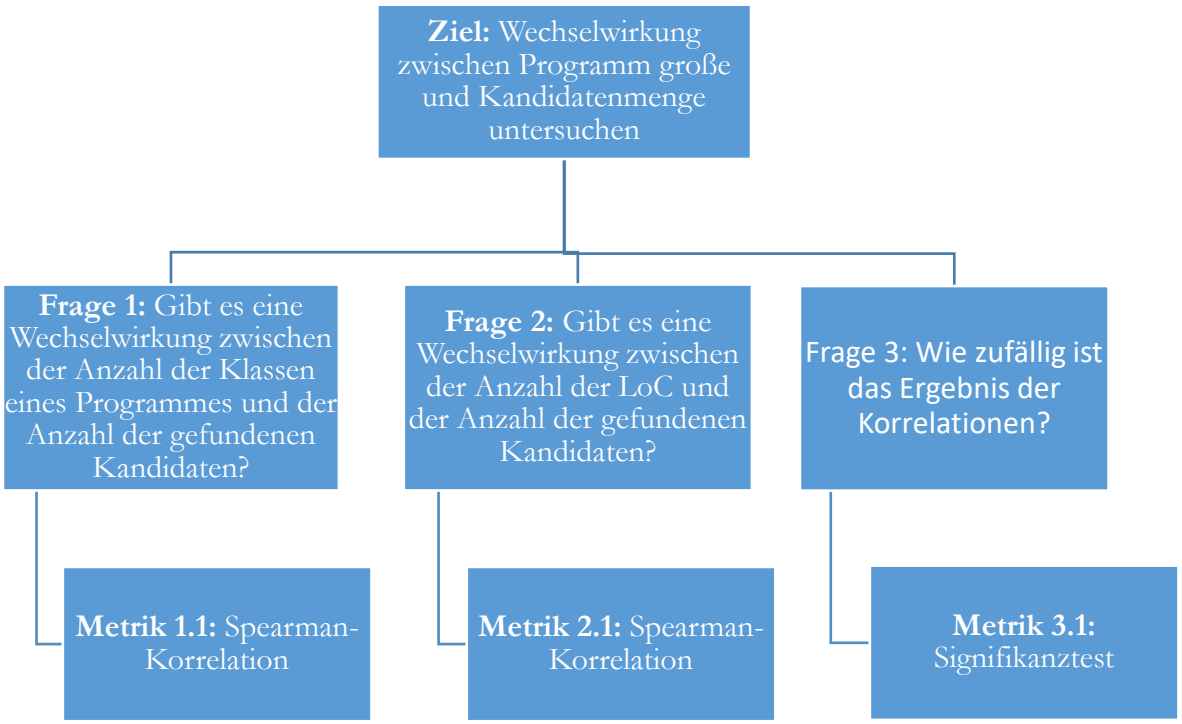


Abbildung 71: Goal-Question-Metrik-Modell zur Bestimmung der Wechselwirkung

Ziel: Wechselwirkung zwischen Programmgröße und Kandidatenmenge untersuchen

Das Ziel dieser Untersuchung liegt darin, herauszufinden, ob die Größe der ausgewählten Software-Projekte und die Menge an gefundenen Entwurfsmusterkandidaten in Zusammenhang stehen. So gilt es, festzustellen, ob ein Programm in einem Merkmal größer ist als die anderen bzw. im Vergleich mehr Kandidaten besitzt oder – im Umkehrschluss – ob in kleinen Programmen weniger Kandidaten gefunden werden. Die Größe kann hierbei mit der Anzahl Klassen oder der LoC eines Programmes bestimmt werden. Entsprechend sind die Ziele definiert. Sollte es keinen klaren Trend zwischen Kandidaten und Programmgröße geben, so könnte dies einen Nachweis für schlechte Quellcode-Qualität mit sich bringen, wenn bestimmte kleinere Programme mehr Kandidaten im Verhältnis aufweisen als größere.

Frage 1: Gibt es eine Wechselwirkung zwischen der Anzahl der Klassen eines Programmes und der Anzahl der gefundenen Kandidaten?

Die Berechnung der Spearman-Korrelation (auch Rangkorrelation genannt) (Zar, 1998) ergibt beim Verhältnis der LoC zu den Kandidaten einen Korrelationskoeffizienten von 0,83. Dieses Ergebnis deutet auf eine positive Korrelation hin. Daraus folgt: Je mehr LoC ein Projekt besitzt, desto höher die Chance, eine größere Menge an Kandidaten zu finden. Aus Sicht der

Erkennungsregeln kann diese Behauptung ebenfalls geschlussfolgert werden. So sind die verschiedenen Erkennungsregeln auf der Basis von Codestrukturen aufgebaut. Je mehr LoC ein Projekt nun besitzt, desto größer die Chance, dass die entsprechenden Strukturen vorhanden sind. Die folgende Abbildung 72 illustriert den Zusammenhang zwischen gefundenen Kandidaten und LoC der Projekte. Die gepunktete Linie zeigt das Ergebnis der linearen Regression.

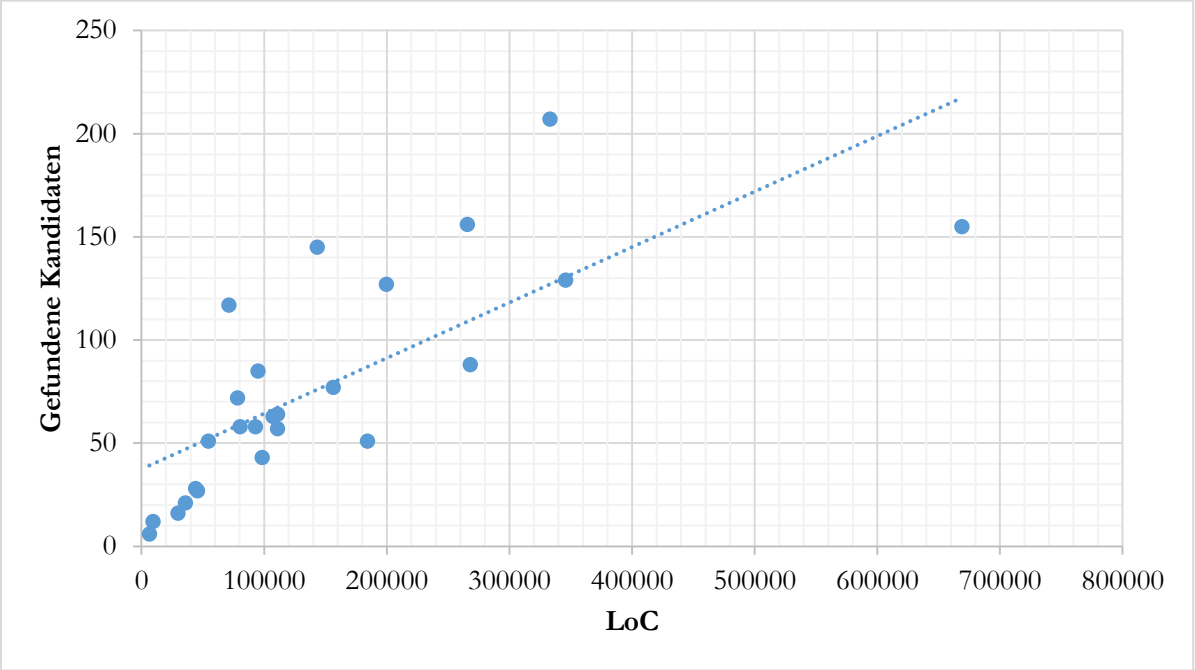


Abbildung 72: Ergebnis der Rangkorrelation zwischen Kandidaten und LoC

Ein ähnliches Bild zeigt die logarithmische Korrelation der Ergebnisse wie in der nachfolgenden Abbildung 72 zu sehen.

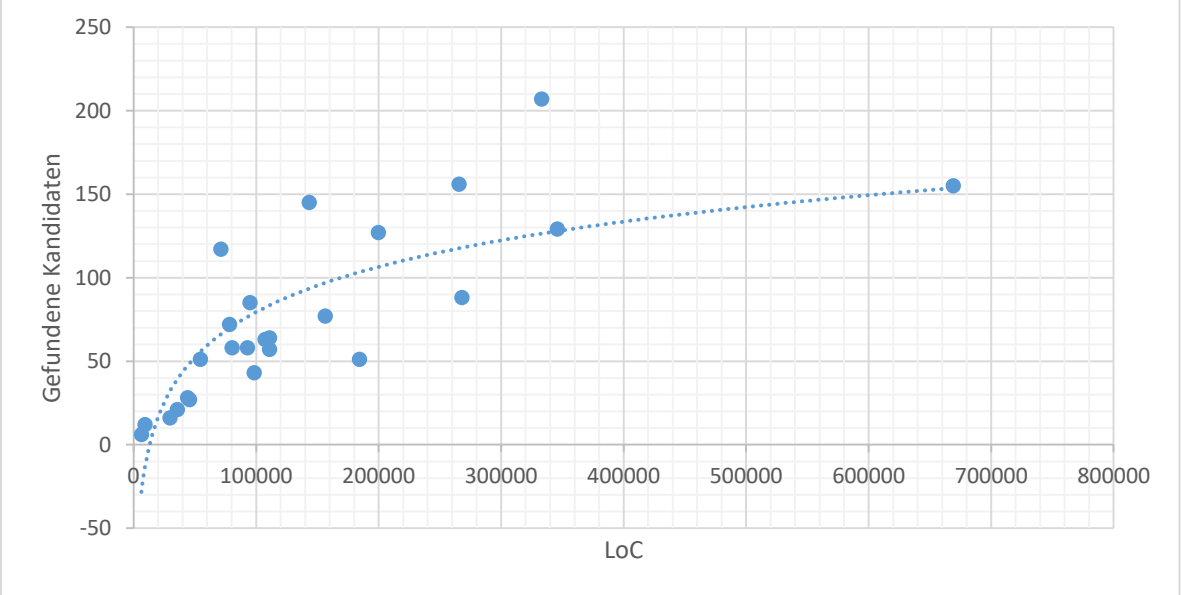


Abbildung 73 Ergebnis der Rangkorrelation zwischen Kandidaten und LoC mit logarithmischer Kurve

Frage 2: Gibt es eine Wechselwirkung zwischen der Anzahl der LoC und der Anzahl der gefundenen Kandidaten?

Die Berechnung der Rangkorrelation zwischen der Anzahl der Klassen und der Anzahl gefundener Kandidaten ergab einen Korrelationseffizienten von 0,95 und damit einen klaren Zusammenhang. Auch hier unterstützt die Sicht auf die Erkennungsregel das Ergebnis. Einige Regeln bauen direkt auf der Klassenebene auf, d. h. sie untersuchen die Beziehungen oder den Aufbau von Klassen. Ein Beispiel hierfür ist die Decorator-Erkennungsregel. Sie baut auf der Suche nach einer breiten Vererbung zwischen Klassen auf. Je mehr Klassen in einem System vorhanden sind, desto größer ist die Chance, eine größere Vererbungskette und somit einen Decorator-Kandidaten zu entdecken. Der Zusammenhang zwischen Klassen und Kandidaten wird in Abbildung 74 und Abbildung 75 graphisch verdeutlicht. Wie zuvor, zeigt die gepunktete Linie das Ergebnis der linearen Regression.

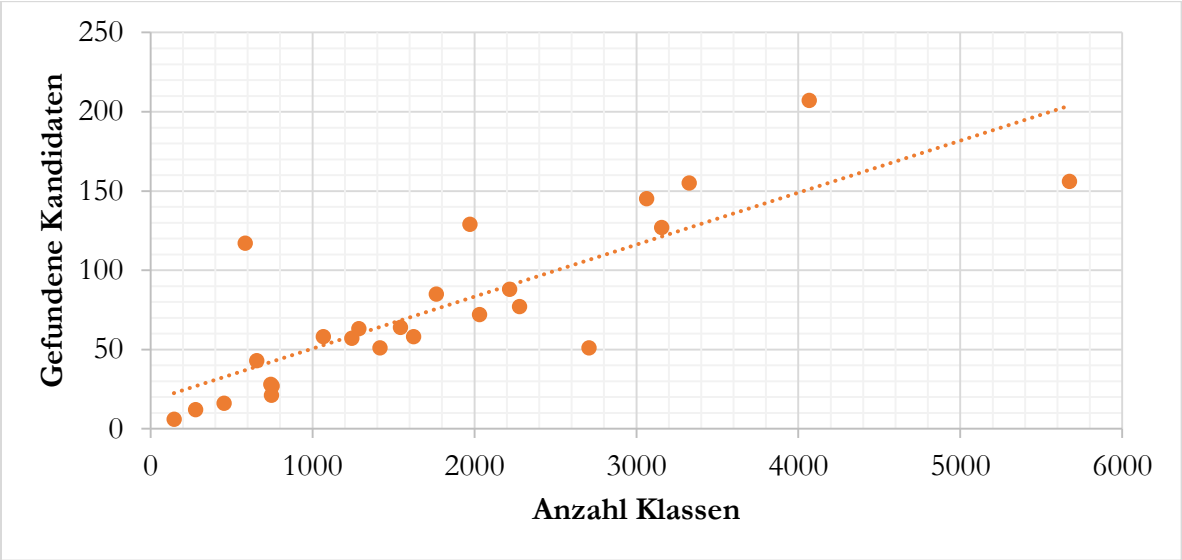


Abbildung 74: Ergebnis der Rangkorrelation zwischen Kandidaten und Klassen

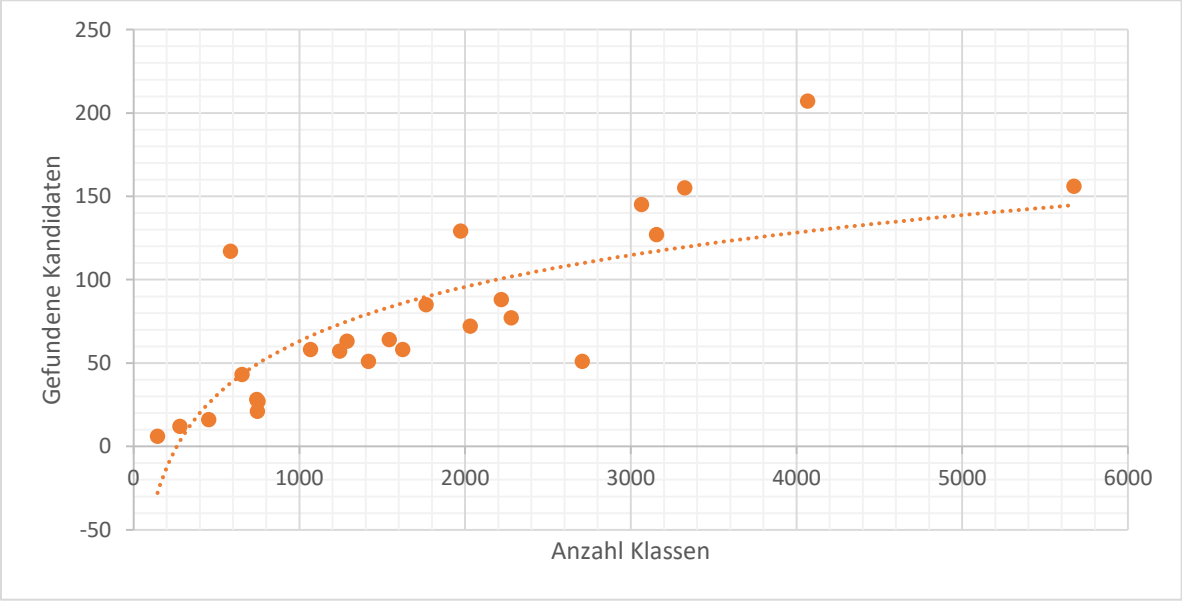


Abbildung 75 Ergebnis mit logarithmischer Korrelation

Frage 3: Wie zufällig ist das Ergebnis der Korrelationen?

Im Lauf der Beantwortung von Frage 1 und 2 wurde eine Reihe von Korrelationsergebnissen ermittelt. Es gilt nun, festzustellen, ob diese Ergebnisse eine Signifikanz aufweisen, d.h. die gemessenen Ergebnisse der Korrelation treten nicht zufällig auf. Um diese festzustellen, werden die beiden Korrelationsergebnisse miteinander verglichen. Dazu wurde eine Fisher-z-Transformation (Meng, Rosenthal, & Rubin, 1992) angewandt. Diese erlaubt es, zwei Korrelationskoeffizienten, auch bei der Rangkorrelation, miteinander zu vergleichen. Das Ergebnis wird als signifikant angesehen, wenn $p < 5\%$ ist. Die Ergebnisse dieser Tests sind in Tabelle 77 aufgeführt. Auch die Signifikanz ($p = 0$ und damit $> 5\%$) unterstützt die oben aufgeführten Aussagen.

Tabelle 77: Signifikanz-Test-Ergebnis für beide Korrelationen

Signifikanz-Test:	Ergebnis:
Prüfung auf lineare Unabhängigkeit	p = 0 (zweiseitig)
Vergleich zweier Korrelationskoeffizienten aus unabhängigen Stichproben	0,0006

Die Betrachtung einzelner Projekte zeigt Ausreißer, welche auf verschiedene Gründe zurückzuführen sind. Dazu können die Qualität der Arbeit einzelner Entwickler oder die Verwendung von Coding-Standards angeführt werden. Als Beispiele für Projekte, bei denen das Verhältnis von Kandidaten zu Merkmalen (LoC oder Klassen) entgegen der errechneten Korrelation liegt, können die zwei Projekte WiND und recoder verwendet werden. Das Projekt WiND (Athens Wireless Metropolitan Network, 2015), eine Datenbank für Funknetzwerke, das bei GitHub gehostet wird, war das größte hier analysierte Projekt. Es besteht aus 668.968 LoC und 3.325 Klassen, was 18,5% der Gesamten LoC und 7,4% der gesamten Menge an Klassen entspricht. Die Untersuchung brachte 159 Kandidaten für die verschiedenen Patterns hervor. Die 159 Kandidaten entsprechen 8,1% der gesamten gefundenen Kandidaten. Damit nimmt das Projekt zwar 20% der Gesamtmenge an untersuchten LoC ein, besitzt aber nur wenige Kandidaten im Vergleich zu anderen Projekten geringerer Größe. Eine klare Aussage zu treffen, warum dieses Projekt solche Werte liefert, gestaltet sich schwierig. Ein Punkt ist, dass das Projekt für eine ganz spezielle Aufgabe, nämlich als Datenbank für Funknutzwerke, entwickelt wurde. Diese Domäne bringt schon einige Designvorgaben mit sich, wie die Topologie der Netze. Solche Vorgaben helfen beim Design der Software. Des Weiteren wird des Projekt nur von wenigen Entwicklern vorangetrieben (<15), was eine Absprache vereinfacht. Hinzu kommt, dass die meisten dieser Nutzer laut E-Mail-Adressen aus dem Heimatland des Projektes stammen, was darauf hinweisen könnte, dass sie sich persönlich kennen. Dies würde eine Design-Absprache weiter vereinfachen. Es kann also darauf geschlossen werden, dass das gewählte Design Einfluss auf die Anzahl der Kandidaten in diesem Projekt hat.
Im Vergleich dazu stellt das Projekt recoder (Heuzeroth, Trifu, & Gutzmann, RECODER, 2015) nur 2% der Gesamtmenge an LoC. Bei dem Projekt handelt es sich um ein Framework zur Analyse und Transformation von Quellcode. In den 71.182 LoC wurden 6,1% aller Kandidaten identifiziert.

7.3 Experten-Audits

In der dieser Stufe der Evaluierung wurde eine Umfrage unter Software-Entwicklern auf Basis der Ergebnisse der Open Source-Projekte durchgeführt. Zum einen sollten die Ergebnisse aus den Analysen und die Grenzwerte so durch Experten evaluiert werden. Zum anderen wurden weitere Informationen über die Verwendung und das Wissen über Design Patterns gesammelt.

Eine Umfrage ist erforderlich geworden, da zu diesem Zeitpunkt keine ähnlichen Evaluierungen oder Studien über das Erkennen von Entwurfsmusterkandidaten existierten. Durch diesen Umstand war nur noch eine Evaluierung durch erfahrene Entwickler möglich. Um den Kreis der Teilnehmer einzugrenzen, wurde folgendes Profil entwickelt:

Teilnehmerprofil:

- **Kenntnisse in der Software-Entwicklung durch abgeschlossenes Studium mit IT-Schwerpunkt (z. B. Informatik oder Wirtschaftsinformatik) oder alternative mehrjährige Erfahrung >4 Jahre im Bereich Software-Entwicklung**
- **Grundwissen über Design Patterns ist vorhanden**
- **Basiskenntnisse (Funktion, Aufgaben und Design) über die sechs in dieser Studie verwendeten Entwurfsmuster sind vorhanden**
- **Programmiererfahrung in Java**
- **Verständnis von UML, um Diagramme lesen und verstehen zu können**

Im Zuge dieser Arbeit wurden 2 Studien durchgeführt. Die erste Studie war eine Stichprobe, um die Idee und die ersten Analyseergebnisse von Experten validieren zu lassen. Die zweite Studie beinhaltet alle Erkennungsregeln samt Analyseergebnissen durch das DPCDT und wurde am Ende der Arbeit durchgeführt. Sie hatte einen größeren Kreis an Teilnehmern.

Aufgabe der Teilnehmer war es, zu bewerten, ob die vom DPCDT gelieferten Vorschläge zu einer Verbesserung des Quellcodes im Hinblick auf die Qualitätskriterien der Veränderbarkeit und Wartbarkeit führen würden. Aussagen über die Performance oder generelle Tauglichkeit von Patterns waren nicht im Fokus.

Mit Hilfe von Quellcode-Auszügen bzw. kleinen UML-Diagrammen wurden die vom DPCDT identifizierten Quellcodestellen in der Studie visualisiert.

7.3.1 Vorstudie

Inhalt der Vorstudie waren die ersten Analyseergebnisse aus den Projekten ArgoUML, jEdit und Apache Lucene sowie die Bewertung der ersten Version einiger Erkennungsregeln. Ziel dieser Vorstudie war es, festzustellen, ob die Vorschläge der Alphaversion des DPCDT sowie das grundlegende Regeldesign aus Sicht der Teilnehmer sinnvoll waren. Die Umfrage bestand aus neun zu beurteilenden Entwurfsmusterkandidaten. Von diesen Patterns war eines ein Negativbeispiel (Frage 5), d. h. der Einsatz des vorgegebenen Design Patterns war aus Sicht der angewandten Erkennungsregeln und der zugrundeliegenden Kriterien nicht sinnvoll. Die Frage diente zur Kontrolle ob der Sachverhalt verstanden wurde.

Zusammen hatten die Teilnehmer mehr als 54 Jahre Programmiererfahrung in Java, was im Mittel 6 Jahre pro Person bedeutet. Von den Teilnehmern kannten 70% Design Patterns von der Universität, die anderen hatten sich durch Schulungen damit vertraut gemacht. Der Großteil der Teilnehmer arbeitete in der freien Wirtschaft als Entwickler, 3 Personen in der Wissenschaft und einer war Student. In der Selbsteinschätzung ihrer Kenntnisse zum Thema Design Patterns gaben

über 45% an, sie hätten Detailwissen. 33% besaßen erweiterte Kenntnisse und eine Person nur Grundwissen. All diese Daten sind nochmals graphisch in Abbildung 76 dargestellt.

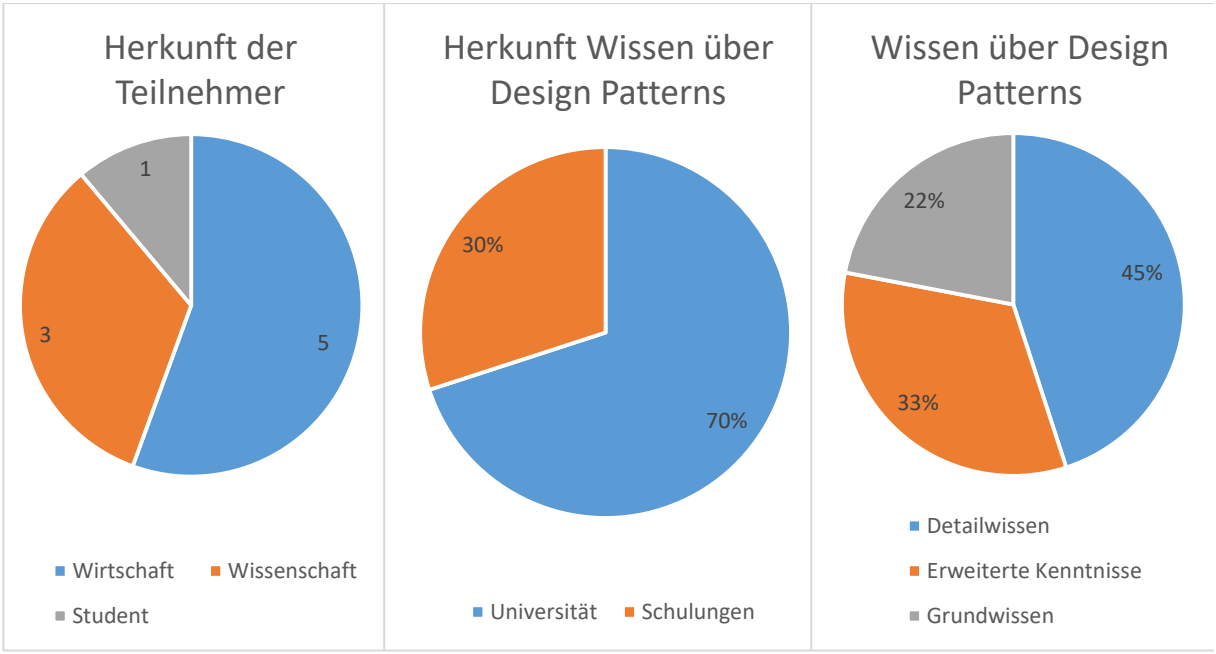


Abbildung 76 Teilnehmerdaten im Überblick

Die nachfolgende Abbildung 77 zeigt eine Übersicht über die Rückmeldungen zu den einzelnen Pattern-Kandidaten der Vorstudie.

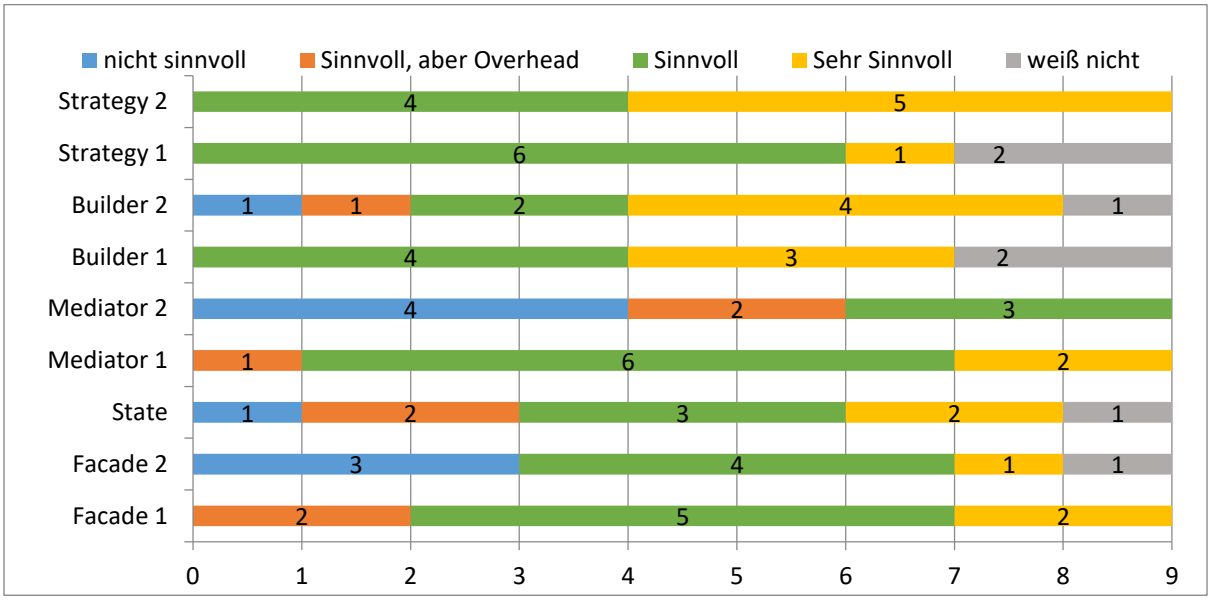


Abbildung 77: Rückmeldungen der Teilnehmer pro Frage

Bei den Fragen mit einem positiven Pattern-Kandidaten, d. h. dieser wurde vom DPCDT als solcher zurückgeliefert, gab es eine positive Rückmeldequote (sinnvoll oder sehr sinnvoll) von 66%. Die höchsten positiven Rückmeldequoten erhielten die Strategy Patterns mit 89%. Die verbleibenden 11% der Teilnehmer enthielten sich einer Meinung. Ein anderes Bild lieferte die Auswertung der Rückmeldung des State-Pattern-Kandidaten.

Hier waren nur 56% der Teilnehmer der Meinung, es wäre eine sinnvolle Entscheidung, das Entwurfsmuster anzuwenden. Mit den Stimmen für Enthaltung waren knapp 44% der Teilnehmer eher skeptisch gegenüber diesem Beispiel.
Die meisten negativen Rückmeldungen erhielt das zweite Mediator-Beispiel. Wie zuvor erwähnt, war dieses Pattern ein konstruierter False-Positive-Kandidat. In diesem Beispiel war die Anzahl an Verbindungen zwischen den einzelnen Klassen sehr gering und es gab nur wenige bidirektionale Verbindungen. Die Teilnehmer wurden vorher nicht darüber informiert, dass sich False-Positive-Beispiele in der Umfrage befinden. 2/3 der Teilnehmer sahen die Anwendung eines Mediator-Patterns in diesem Beispiel als unlogisch an.

Bei dem kleinen Teilnehmerkreis und den verwendeten experimentellen Grenzwerten zeigt die Vorstudie nur eine Tendenz auf. Wie auch immer, diese Tendenz zeigte eine positive Resonanz. Aus den Ergebnissen lässt sich ableiten, dass die Teilnehmer die Analysen nachvollziehen konnten und bei den meisten der Kandidaten eine Verbesserungsmöglichkeit wahrnahmen. Ein Punkt hierfür ist, dass viele Teilnehmer das False-Positive-Beispiel als sinnlos erkannten. Des Weiteren ist anzumerken, dass die Teilnehmer trotz experimenteller Beispiele eine Sinnhaftigkeit in der Idee der Entwurfsmusterkandidaten-Erkennung sahen.

7.3.2 Aufbau der Umfrage

Die Hauptstudie bestand aus 16 Fragen, die in der Auswertung in drei Abschnitte eingeteilt wurden. Der Umfragebogen und die Ergebnisse können am Ende dieser Arbeit gefunden werden. Abschnitt 1 umfasste Fragen zur Person selbst. Es wurden das Alter, die aktuelle Tätigkeit, der höchste Abschluss und der Themenschwerpunkt des Studiums/der Ausbildung abgefragt. Diese Antworten wurden später verwendet, um einige anonymisierte Cluster für die Auswertung zu bilden. Ziel der Umfrage war eine umfassende Evaluierung der Analyseergebnisse auf Sinnhaftigkeit, Validität und Akzeptanz.

Im zweiten Abschnitt wurden der Erfahrungsstand in der Programmierung in Jahren sowie der Wissensstand im Bereich Design Patterns abgefragt: Welche sind bekannt, welche Selbsteinschätzung wird zu den einzelnen Patterns getroffen und wie oft wurden Design Patterns schon von dem Teilnehmer in der Praxis verwendet? Ziel war es, die verschiedenen Teilnehmer in Erfahrungscluster einteilen zu können, um den Nachweis zu erbringen, dass die Umfrage eine große Diversität in den Teilnehmern erreicht, aber auch, um das Wissen der Experten evaluieren zu können.

Inhalt von Abschnitt 3 waren die in Kapitel 7.2 diskutierten Analyseergebnisse und die finalisierten Regeln. Für jede Regel wurde anhand eines Ergebnisses abgefragt, ob das vom DPCDT ausgewählte Entwurfsmuster an der gegebenen Stelle nach Ansicht des Teilnehmers sinnvoll ist oder nicht. Dabei wurden die Ergebnisse immer auf 2 Arten dargestellt: als UML-Diagramm und als textuelle Beschreibung.

7.3.3 Zielgruppe und Rückmeldequote

Ziel dieser Hauptstudie war eine Evaluierung durch eine größere Zielgruppe von Experten. Die Zielgruppe wurde erweitert um Architekten, Software-Analysten und IT-Projektleiter aus dem industriellen und wissenschaftlichen Umfeld. Ziel war es, die Ergebnisse des DPCDT durch mindestens 50 Experten mit unterschiedlichem Hintergrund und Erfahrungsniveau evaluieren zu lassen.

Die Umfrage wurde den Teilnehmern über den Online-Dienst Survio (Survio s.r.o., 2015) unter dem Titel „Pattern oder nicht Pattern oder Umfrage zur Verbesserung von Quellcode durch den

Einsatz von Design Patterns“ bereitgestellt. Die Umfrage wurde den Teilnehmern für 175 Tage bzw. vom 13.02.2015 bis zum 06.08.2015 zur Verfügung gestellt. Zugriff hatten nur Teilnehmer, die einen direkten Link zum Umfrage hatten. Damit nur Teilnehmer den Link erhalten, die dem Profil entsprechen, gab es eine Vorselektion. Zum einen wurden Teilnehmer durch die IT-Abteilung der Allianz (in Person von Abteilungsleiter Hr. Flasse) ausgewählt, die zu den Senior-Entwicklern dort zählen – darunter auch Teilnehmer, die nicht direkt bei der Allianz beschäftigt waren, um eine gewissen Diversität zu erhalten. In wissenschaftlichen Kreisen wurde die Umfrage gezielt an Forscher versendet, die mit dem Themengebiet zu tun hatten. Einige Personen leiteten den Link an Kollegen in ihrem Umfeld weiter. Dieser Umstand ließ sich nicht verhindern. Doch durch das sorgsame Selektieren wurde die Gefahr einer Unschärfe minimiert. Ein Zugang über Suchmaschinen wurde ausgeschlossen. Alle Fragen waren in deutscher Sprache.

In dem genannten Zeitraum besuchten 199 Besucher die Umfrage. Davon beantworteten 26,1% bzw. 52 Teilnehmer alle Fragen. Weitere 52 beendeten die Umfrage vorzeitig. Die restlichen 95 ließen sich die Umfrage nur anzeigen. Diese werden automatisch ignoriert. Die folgende Tabelle 78 illustriert die Teilnehmerinformationen.

Tabelle 78: Übersicht der Teilnehmer

199	52	52	95	26,1%
Besucher Insgesamt	Fertige Antworten	Unvollendete Antworten	Nur besucht	Abschlussquote insgesamt

Laut den Auswertungen benötigten die meisten Teilnehmer, die die Umfrage komplett abgeschlossen hatten, zwischen 10 und 30 Minuten. Ein Drittel schloss die Umfrage in weniger als 10 Minuten ab. Nur 17,3% der Teilnehmer benötigte mehr als 30 Minuten. Abbildung **78** und Abbildung **79** stellen das Gesamtbild dar.

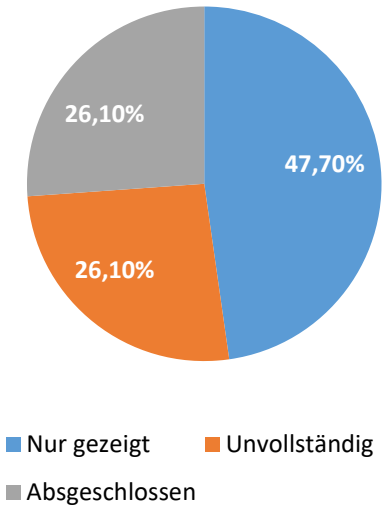


Abbildung 78: Besucher Total

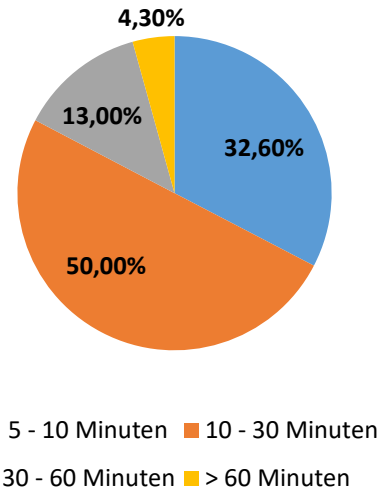


Abbildung 79: Durchschnittliche Zeit der Fertigstellung

7.3.4 Teilnehmer-Informationen

Nachfolgend werden einige Informationen zum Teilnehmerfeld aufgelistet, die zeigen, inwieweit die Umfrage verschiedenste Entwickler und Fähigkeitsgruppen erreicht hat. Eine Übersicht der Zahlen ist in der Abbildung 80 zusammengefasst.

Das durchschnittliche Alter der Teilnehmer lag bei 32,59 Jahren. Hierbei war der älteste Teilnehmer 48 und der jüngste 21. Von den 52 Teilnehmern hatten 49 Personen einen Studienabschluss und 3 eine abgeschlossene Ausbildung. Mehr als 53% der Teilnehmer verfügten über einen Abschluss in Informatik und weitere 26,9% über einen in Wirtschaftsinformatik. Die Hauptzielgruppe lag auf Informatikern und Wirtschaftsinformatikern, da diese mit dem abgefragten Themenbereich meist schon im Studium in Kontakt kamen. Im teilnehmenden Personenkreis gab es 25 Entwickler und 5 Architekten, die direkt am Quellcode arbeiten. Die andere große Gruppe waren Projektleiter (12), die mit Design und Analyse zu tun haben. Zusammengerechnet hatten die Teilnehmer 314 Jahre Programmiererfahrung. Im Durchschnitt waren dies 6,0 Jahre pro Person.

Tabelle 79: Teilnehmerinformationen im Überblick

Abschluss	Anzahl	%	Themen-gebiet	Anzahl	%	Tätigkeit	Anzahl	%
Berufsaus-bildung	3	5,8	Informatik	28	53,8	Entwickler	25	48,1
Bachelor	12	23,1	Wirtschafts-informatik	14	26,9	Architekt	5	9,6
Master	19	36,5	Mathematik	2	3,8	Wissenschaftler	5	9,6
Doktor	5	9,6	Physik	1	1,9	Projektleiter	12	23,1
Diplom	13	25	Andere	7	13,5	Führungskraft	1	1,9
						Andere	4	7,7

7.3.5 Wissen der Teilnehmer über Design Patterns

Im zweiten Abschnitt der Umfrage wurde das Wissen der Teilnehmer über Design Patterns abgefragt. Zuerst wurde darum gebeten, sein Wissen selbst einzuschätzen, und zwar über die 6 Design Patterns, für die Erkennungsregeln festgelegt wurden.

Den meisten Personen waren die Design Patterns in ihren Grundlagen bekannt. Das Ergebnis zeigt aber gut, dass ein Wissen über Design Patterns nicht in der Breite vorhanden ist, was ein Kritikpunkt beim Einsatz von Design Patterns war (vgl. Kapitel 1.1). Es zeigte sich auch, dass mindestens 30% der Teilnehmer ein bestimmtes Entwurfsmuster schon verwendet haben oder sogar Detailwissen besitzen. Eine Ausnahme gibt es bei dieser Betrachtungsweise: Detailwissen über das Mediator-Pattern und die Verwendung desselben waren sehr wenig verbreitet unter den Teilnehmern. Jedoch war es das Pattern, von dem die meisten zumindest schon einmal gehört hatten. Zusammenfassend kann gesagt werden: Der Teilnehmerkreis brachte einiges an Wissen im Gebiet der Design Patterns mit. Nur 14,8% der Teilnehmer kannten ein Entwurfsmuster nicht. Dafür hatten 37,5% Detailwissen oder wenigstens ein Pattern schon verwendet.

Tabelle 80: Wissen des Teilnehmerkreises in Zahlen

Antwort	Schon mal gehört		Grundwissen vorhanden		Schon verwendet		Detailwissen vorhanden		nicht bekannt	
	Anzahl	%	Anzahl	%	Anzahl	%	Anzahl	%	Anzahl	%
Builder	13	25	11	21,2	14	26,9	11	21,2	3	5,8
Decorator	12	23,1	12	23,1	11	21,2	8	15,4	9	17,3
Facade	7	13,5	7	13,5	20	38,5	11	21,2	7	13,5
Mediator	22	42,3	12	23,1	7	13,5	2	3,8	9	17,3
State	17	32,7	11	21,2	11	21,2	5	9,6	8	15,4
Strategy	13	25	12	23,1	11	21,2	6	11,5	10	19,2
Durchschnitt		26,9		20,9		23,8		13,8		14,8

Die nachfolgende Frage hatte zum Ziel, festzustellen, wie oft die Teilnehmer Design Patterns nach ihrer eigenen Einschätzung verwendet haben. 19,2% verwendeten Design Patterns einmalig oder gar nicht. Immerhin 32,7% verwendeten Design Patterns in mehreren Anwendungen und somit häufig. Bei der Antwortmöglichkeit „In mehreren Anwendungen (an wenigen Stellen)“ waren es noch 24,4%.
Es zeigt sich, dass der Großteil der Teilnehmer mehr als theoretische Erfahrung mit dem Themengebiet hat.

Tabelle 81: Generelle Verwendung von Design Patterns durch die Teilnehmer

Frage	Anzahl	%
Einmal oder gar nicht verwendet	10	19,2%
In mehreren Anwendungen (an wenigen Stellen)	22	24,4%
Häufig (mehrere Anwendungen)	17	32,7%
Bei jeder Gelegenheit	3	5,8%

7.3.6 Bewertung der Kandidaten

Im 3. und letzten Abschnitt wurden den Teilnehmern zu jedem der sechs hier näher untersuchen Design Patterns durch den Prototyp identifizierte Kandidaten vorgelegt. Die Aufgabe bestand darin, diese nach ihrer Sinnhaftigkeit zu bewerten. Alle Beispiele waren Ergebnisse der Untersuchungen des DPCDT. Dabei hatten die Teilnehmer die Möglichkeit, einen Kandidaten von 1, sinnvoller Einsatz des Patterns, bis 5, nicht sinnvoller Einsatz, zu bewerten. Hinzu kam noch die Nummer 6, was bedeutete, dass der Teilnehmer keine Antwort geben wollte. Damit nicht zeilenweise Quellcode abgebildet werden musste, wurden UML-Diagramme und Tabellen als Darstellungsform gewählt.

7.3.7 Beispielfrage

Kandidaten für das Mediator-Pattern werden unter der Annahme identifiziert, dass ein Block von Klassen innerhalb eines Packages viel untereinander kommuniziert und jede Klasse zu fast jeder anderen innerhalb dieses Kommunikationsblocks eine Verbindung besitzt.
Im Package *org.gjt.sp.jedit.search* bilden die vier Klassen *SearchAndReplace*, *HyperSearchResults*, *SearchDialog* und *ReplacheActionHandler* einen Kommunikationsblock. Nahezu jede Klasse kommuniziert mit den anderen. Das folgende Bild stellt den Sachverhalt dar.

Wie beurteilen Sie den Einsatz eines Mediator-Patterns im folgenden Beispiel?

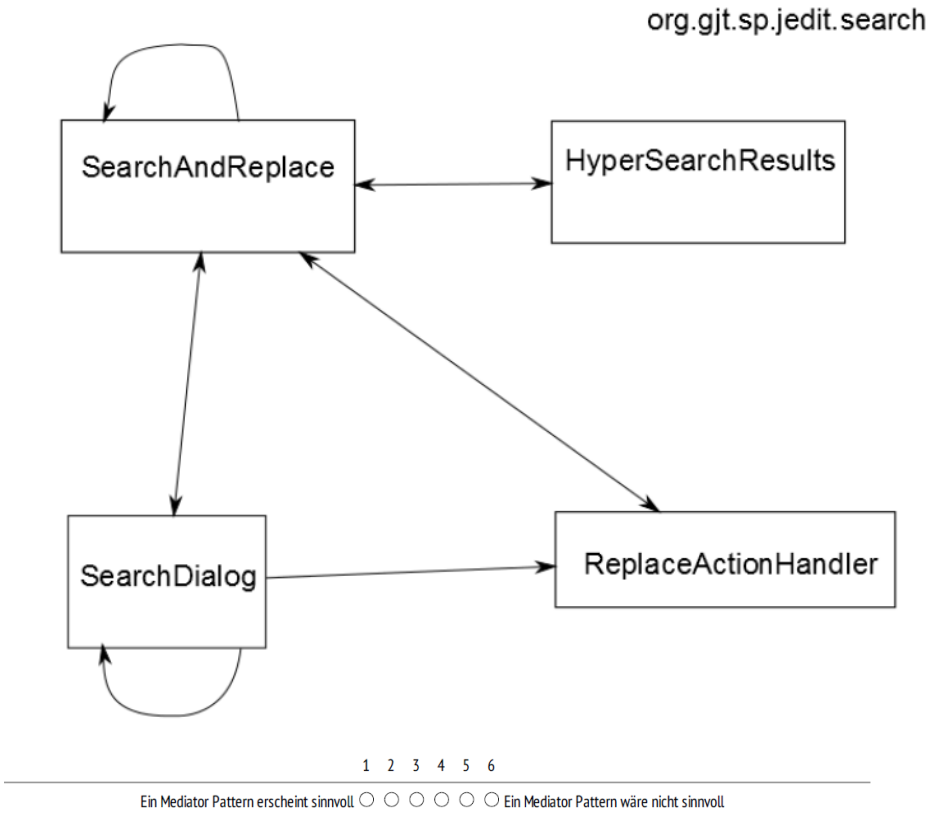


Abbildung 80: Mediator-Kandidatenfrage

7.3.8 Auswertung

Bei der Auswertung wurden die Rückmeldungen wie folgt betrachtet. Die Wertung 1 (sinnvoll) und 2 (eher sinnvoll) zählen als positive Rückmeldungen, Wertung 3 als neutrale und 4 (eher nicht sinnvoll) sowie 5 (nicht sinnvoll) als negative Rückmeldung. Die Stufen zwischen 1 und 2 bzw. 4 und 5 dienen nur der Gewichtung. Wie zuvor schon beschrieben, dient die Wertung 6 als „keine Meinung“. (siehe Fragebogen am Ende dieser Arbeit)

Bei der Frage der Entwurfsmusterkandidaten-Evaluierung lag die Rückmeldequote je nach Fragestellung zwischen 50 und 51 von 52 Teilnehmern. In Summe waren 51,68% der Rückmeldungen innerhalb der Wertung 1 und 2 und somit als positiv zu werten. Hier ging die Tendenz eher zur Wertung 2 mit 28,95%. 18,4% lagen im negativen Bereich. Eine klare Tendenz gab es im negativen Bereich nicht. Höchste positive Rückmeldung bekam der Facade-Pattern-Kandidat. Das Decorator-Beispiel wurde eher durchschnittlich bewertet. Eine Übersicht der Rückmeldungen pro Design Pattern-Kandidat wurde in der Tabelle 82 zusammengefasst. Die einzelnen Fragen und Rückmeldungen werden im Folgenden diskutiert.

Tabelle 82: Gesamtübersicht über die Rückmeldungen der Kandidatenbeispiele

Design Pattern-Kandidat	Sinnvoll (1)	Eher sinnvoll (2)	Neutral	Eher nicht sinnvoll (4)	Nicht sinnvoll (5)	Enthaltung (6)
Builder	17	20	4	3	1	6
Strategy	9	12	10	3	10	7
Facade	19	9	6	6	4	7
Decorator	8	19	8	5	4	6
Mediator	8	10	9	7	5	12
State	8	18	7	4	4	10
Anzahl Gesamt	69	88	44	28	28	48
Gesamt in%	22,62%	28,85%	14,43%	9,18%	9,18%	15,74%
	51,48%			18,36%		

Die Beispiele der Umfrage werden in den Entsprechenden Sektionen der einzelnen Erkennungsregeln unter Kapitel 5 diskutiert. Die Umfrage ist in Anhang **Fehler! Verweisquelle onnte nicht gefunden werden.** zu finden.

Das erste Beispiel beinhaltete einen Kandidaten für ein Builder-Pattern. Die gewählte Klasse *UMLModelElementListModel* besitzt 11 Konstruktoren mit variierender Anzahl und Typen an Übergabeparametern. Je nach Konstruktor kann dieser bis zu 5 Übergabeparameter akzeptieren. Somit hatte der Kandidat eine Erkennungsstufe von *empfohlen*.

Über 72% hielten den Vorschlag des DPCDT für sinnvoll. Nur 6 Teilnehmer enthielten sich der Bewertung. Nur 7,9% tendierten zu nicht sinnvoll. Damit erhielt der Kandidat eine deutlich positive Bewertung und stimmt damit mit der Erkennungsstufe überein.

Tabelle 83: Auswertung der Ergebnisse zum Builder-Kandidaten

	1	2	3	4	5	6	
Ein Builder-Pattern erscheint sinnvoll.	17	20	4	3	1	6	Ein Builder-Pattern wäre nicht sinnvoll.
	33,3%	39,2%	7,8%	5,9%	2,0%	11,8%	

Die zweite Frage beinhaltete einen Kandidaten für ein Strategy-Design Pattern, welches auf der Annahme basiert, dass eine Klasse mehrere Methoden besitzt, die identische Switch-Anweisungen mit einer gewissen Menge an Cases implementieren. Eine solche Struktur erlaubt die Auswahl eines bestimmten Vorgehens (gleichzusetzten mit Strategy) je nach gewünschtem Verhalten.

Die Klasse *TextUtilities* besitzt zwei Methoden (*findWordEnd*, *findWordStart*). Jede dieser Methoden implementiert ein Switch mit drei gleichen Case-Bedingungen. Diese verwenden *WHITE SPACE*, *WORD_CHAR* und *SYMBOL* in den Cases. Die Variable mit dem Namen „*type*“ wird in beiden Switches im Kopf verwendet.

Die Rückmeldungen über die Verwendung des Strategy Patterns bei diesem Kandidaten waren mit 41,1% grundlegend positiv. 7 Teilnehmer wussten keine Rückmeldung zu diesem Kandidaten zu geben. Im zuvor abgefragten Wissen über dieses Pattern kannten 10 Teilnehmer das Pattern nicht. Auf Grundlage dieser Aussage lässt sich schließen, dass die 7 Personen das Pattern nicht kannten und somit keine Wertung abgeben wollten.

Tabelle 84: Auswertung der Ergebnisse zum Strategy-Kandidaten

	1	2	3	4	5	6	
Ein Strategy Pattern erscheint sinnvoll.	9	12	10	3	10	7	Ein Strategy Pattern wäre nicht sinnvoll.
	17,6%	23,5%	19,6%	5,90%	19,6%	13,7%	

Die darauffolgende Frage beschäftigte sich mit einem Kandidaten für das Facade-Pattern. Ein Facade-Design Pattern-Kandidat kann unter der Annahme identifiziert werden, dass ein Package eine gewisse Unabhängigkeit vom Rest des Systems besitzt, d. h. ein Package sollte wenige Aufrufe zu anderen Bereichen des gleichen Programmes besitzen (Aufrufe zu Standardbibliotheken wie JAVA IO sind nicht mit einbezogen). Des Weiteren sollte das Package von einer gewissen Anzahl anderer Klassen außerhalb angesprochen werden. Wenige Aufrufe könnten auf das Vorhandensein einer Facade hindeuten. Letzte Annahme ist, dass das Package eine gewisse Anzahl an internen Verbindungen aufweist, d. h. Package-eigene Klassen rufen immer wieder andere Klasse im Package auf. Dies impliziert, dass das Package eine klare interne Aufgabenverteilung besitzt.

Der Kandidat lag im Package *recoder.util* mit folgenden Verbindungskennzahlen.

- In: In das Package hinein gab es 322 Verbindungen, die von 30 Klassen aus dem Programm stammten.

- Out: Von dem Package in andere Packages innerhalb des gleichen Programmes gab es nur 4 Verbindungen, was im Vergleich zu den Package-eigenen Klassen weniger als 0,14% entspricht.

- Intern: Innerhalb des Packages wurden 47 Verbindungen gefunden. Damit liegt das Kommunikationsniveau recht hoch bei 30 internen Klassen.

21,2% der Teilnehmer kannten das Facade-Pattern im Detail; viele hatten es auch schon angewandt (38,5%). Das schlägt sich im Vergleich zu anderen Fragen auch in den Rückmeldungen nieder. Die Anzahl der Rückmeldungen, die diesen Kandidaten als *sinnvoll* betrachteten, lieferte mit 54,9% eine klare Tendenz. Der Anteil von ausgelassenen Bewertungen liegt wieder bei 7 Teilnehmern. Mit 19,6% besitzt die Frage eine sehr geringe negative Rückmeldequote. Je mehr Teilnehmer das Pattern im Detail kennen, desto mehr können sie den Kandidaten bewerten. Wie zu Anfang schon beschreiben besitzen Entwurfsmuster viele menschliche Faktoren und Erfahrung hilft bei der Auswahl.

- *Tabelle 85: Auswertung der Ergebnisse zum Facade-Kandidaten*

	1	2	3	4	5	6	
Ein Facade-Pattern erscheint sinnvoll.	19	9	6	6	4	7	Ein Facade-Pattern wäre nicht sinnvoll.
	37,3%	17,6%	11,8%	11,8%	7,8%	13,7%	

Der Kandidat für das Mediator-Pattern wurde unter der Annahme identifiziert, dass ein Block von Klassen innerhalb eines Packages viel untereinander kommuniziert und jede Klasse zu jeder anderen innerhalb dieses Kommunikationsblocks eine Verbindung besitzt.

Im Package *org.gjt.sp.jedit.search* bilden die vier Klassen *SearchAndReplace*, *HyperSearchResults*, *SearchDialog* und *ReplacheActionHandler* einen Kommunikationsblock. Nahezu jede Klasse kommuniziert mit den anderen (siehe Kapitel 3.5).

Auch bei diesem Pattern gab es eine klare positive Rückmeldung mit 54%, wobei die Tendenz eher zu Wertung 2 (eher sinnvoll) geht. Interessanterweise war das von den Teilnehmern mitgebrachte Wissen bei diesem Entwurfsmuster mit nur 3,8% Detailwissen sehr gering. Angewandt hatten es nur 13,5%. Beide Werte sind die niedrigsten ermittelten von allen abgefragten Design Patterns. Die Definition des Mediator-Patterns und sein Anwendungsbereich sind einfach verständlich. Dieser Umstand erlaubt es, das Beispiel schnell zu bewerten, auch wenn noch kein Wissen vorhanden ist.

Tabelle 86: Auswertung der Ergebnisse zum Mediator-Kandidaten

	1	2	3	4	5	6	
Ein Mediator-Pattern erscheint sinnvoll.	8	19	8	5	4	6	Ein Mediator-Pattern wäre nicht sinnvoll.
	16,0%	38,0%	16,0%	10,0%	8,0%	12,0%	

Die Beispielerkennung für das Decorator-Pattern basiert auf der Annahme, dass eine Klasse existiert, von der eine gewisse Menge anderer Klassen erben. Bei jeder neuen Klasse werden neue Funktionalitäten hinzugefügt.

Die Klasse *RuntimeException* wird an 23 Klassen vererbt. Davon erben 6 Klassen direkt. 17 weitere erben von der Klasse *ModelException*, einer von einer der 6 Klassen auf Ebene 1.

In diesem Beispiel wurden *Exception*-Klassen angewandt, um einen Decorator-Kandidaten zu demonstrieren. Dies kann in der Theorie vielleicht möglich sein, in der Praxis ist so etwas aber schwierig zu implementieren, haben doch alle *Exception*-Klassen klar definierte Aufgaben, wodurch nur wenige Funktionen überschrieben oder durch Vererbung gemeinsam genutzt werden. Dies führte zu der Erkennung einer großen Klassenhierarchie. Die Teilnehmer haben in der Mehrzahl diese Konstellation bemerkt, was zu einer neutralen bis negativen Rückmeldung mit 41,1% führte. Außerdem war der Anteil der Teilnehmer, die keine Wertung hinterließen, mit 23,5% sehr hoch. Insgesamt 64,5% der Teilnehmer sahen das Beispiel als nicht *sinnvoll* an oder wollten es nicht bewerten.

Wie schon in Kapitel 5.2.3 diskutiert, gibt es bei der aktuellen Version der Decorator-Erkennungsregel Unschärfen, welche hier im Beispiel deutlicher zum Vorschein kommen. Das hier genannte Beispiel wurde von der Erkennungsregel ohne Erweiterung als Kandidat erkennt. Bei einer detaillierten Prüfung des Ergebnisses kamen jedoch Zweifel auf, ob es sich um einen brauchbaren Decorator-Kandidaten handelt, oder ob nicht auch ein anders Pattern eine Lösung wäre. Die Unklarheit über das Ergebnis liegt in der Struktur des Kandidaten, der ein Exception-Handling implementiert. Eine Exception stellt ein geplantes Ereignis / Fehler dar, der zur Laufzeit des Programmes auftritt und dessen normalen Ablauf stört. In Java ist jede Exception eine Subklasse von java.lang.Exception. Sollen weitere individuelle Exceptions für ein Programm implementiert werden, so geschieht dies über Vererbung. Je nach Anzahl und Art der Exceptions entstehen so viele Subklassen. Die Idee hinter dem Konzept ist die Fehler durch die Klassen zu markieren und später individuell zu bearbeiten. Die Exception beinhaltet dabei nicht die Lösung

für den auftretenden Fehler sondern nur die Informationen über den Fehler. Eine Verarbeitungslogik muss vom Entwickler der in entsprechenden Catch-Anweisung implementiert werden. Der Einsatz eines Dekorator macht somit wenig Sinn, da seine eigentliche Aufgabe ist, Verhalten zu orchestrieren. Exceptionklassen beinhalten wie gesagt nur wenig Verhalten. Die Stelle mag optimiert werden können, um die Anzahl an Vererbungen zu reduzieren, doch eher nicht mit einem Decorator-Pattern.

Tabelle 87: Auswertung der Ergebnisse zum Decorator-Kandidaten

	1	2	3	4	5	6	
Ein Decorator-Pattern erscheint sinnvoll.	8	10	9	7	5	12	Ein Decorator-Pattern wäre nicht sinnvoll.
	15,7%	19,6%	17,6%	13,7%	9,8%	23,5%	

Die letzte Frage beschäftigte sich mit dem State-Entwurfsmusterkandidaten. Wie beim Strategy Pattern werden Switch-Anweisungen gesucht, die eine gewisse Menge an Cases besitzen und in mehreren Methoden einer Klasse vorkommen. Zusätzlich muss die im Switch-Kopf verwendete Variable noch in jedem Case verwendet werden. Dies impliziert, dass im Switch-Kopf der Zustand erfasst und in jedem Case geändert wird.

Die Klasse *BeSLAStatsProcessor* besitzt 2 Methoden mit drei identischen Switches. Die Switches verwenden immer die 6 gleichen Bedingungen. Die im Switch-Kopf verwendete Variable stammt von der Klasse *BeTrapEvent* ab. Für die Case-Anweisungen wird ein *Enum* aus der Klasse *BeTrapEvent* verwendet. Die beiden Switches in der gleichen Methode sind noch von einem IF umgeben. Zur Verarbeitung der Variable wird in den Cases die Methode *putIfAbsent4APInterface* bzw. auch *removeIfExist4APInterface* mit der Variable als Übergabeparameter aufgerufen.

Trotz der Komplexität des State Patterns war das Wissen darüber bei den Teilnehmern breit gestreut. Mit nur 8 Teilnehmern, die das Pattern nicht kannten, war es das zweitbekannteste Entwurfsmuster innerhalb der Umfrage. Mit einer positiven Rückmeldequote von 51,2% gegenüber einer negativen Rückmeldequote von 15,6% gab es auch hier eine klare Tendenz. Es sollte jedoch beachtet werden, dass 35,5% der Rückmeldungen nur zu eher sinnvoll (Wertung 2) tendieren und sich damit nicht ganz sicher waren.

Tabelle 88: Auswertung der Ergebnisse zum State-Kandidaten

	1	2	3	4	5	6	
Ein State Pattern erscheint sinnvoll.	8	18	7	4	4	10	Ein State Pattern wäre nicht sinnvoll.
	15,7%	35,5%	13,7%	7,8%	7,8%	19,6%	

7.3.9 Validität der Ergebnisse

Abschließen sollte noch mögliche Gefahren in der Validität der Umfrageergebnisse und potenzielle Fehler (engl. Threats to validity) angesprochen werden. Trotz sorgfältiger Auswahl der Teilnehmer einerseits durch Selektion durch die IT-Abteilung der Allianz und durch Selektion von Personen, die in dem Umfeld arbeiten, liegt es nahe, dass weitere Personen durch schon teilnehmende Personen eingeladen wurden. Bei diesen kann der Wahrheitsgehalt der Antworten nicht bestätigt werden. Da die direkt eingeladenen Teilnehmer aber schon sorgfältig selektiert waren, ist davon auszugehen, dass auch diese eingeladenen Personen im Umfeld des Software Engineering arbeiten.

52 Teilnehmer sind im Vergleich zu Studien in anderen Bereichen außerhalb des Software Engineering wenige. Doch nach einer Studie von Sjorberg (Sjoberg, et al., 2005) liegt die Anzahl von Teilnehmern in Software Engineering bei ca. 49. Im Vergleich hatte diese Studie also einige Teilnehmer mehr. Zudem waren an dieser Studie 48 professionelle Software-Entwickler und 4 Studenten beteiligt. Laut Sjorberg lag bei anderen Studien die Studentenquote bei 87%. Der große Anteil an professionellen Entwicklern von 92,3%, zeigt dass der Teilnehmerkreis entsprechend den Vorgaben (vgl. Kapitel 1.2) ausgewählt wurden. Um Entwurfsmuster ordnungsgemäß einsetzen zu können ohne das hier beschrieben Verfahren ist viel Erfahrung notwendig. Dieser Teilnehmerkreis bringt dieser Erfahrung mit sich. Ein weitere Punkt der über die Jahre der Softwareentwicklung vermittelt wird, sind Probleme zu Analysieren und zu Verstehen. Entwurfsmuster sind Lösungen für wiederkehrende Probleme. Erfahrung in diesem Umfeld unterstützt bei der Bewertung der Kandidaten.

7.4 Evaluierung des Industrieprojekts

Neben den genannten Open Source-Projekten wurde noch ein Projekt untersucht, das von der Allianz Deutschland zur Verfügung gestellt wurde. Dieses Projekt wurde unabhängig evaluiert, da hier die Entwickler und Projektleiter direkt zum Audit zur Verfügung standen. Die aus dem Audit erarbeiteten Informationen sind in die nächsten Sektionen eingeflossen. Ein Vergleich mit Programmen anderer Versicherer war nicht möglich, da keine weiteren Firmen Quellcode bereitstellten.

7.4.1 Beschreibung Firmen-Individual-Tarifierung (FIT)

Das Programme Firmen-Individual-Tarifierung (FIT) ist eine Java-Anwendung zur Datenerfassung, Tarifierung und anschließender Angebotserstellung für Sachversicherungen (Versicherungssumme ab 10 Mio. €) im Firmen-Individualgeschäft. Es besteht die Möglichkeit, bis zu 13 Gefahren für die Positionen Gebäude, Inhalt, Betriebsunterbrechung oder auch Mietverlust zu versichern. Die relevanten Tarifdaten werden auf dem Host in Tabellen gepflegt. Die persistenten Daten von FIT werden in einer Server-Datenbank zentral abgelegt. Offline-Benutzer haben zusätzlich eine lokale Datenbank, die dann mit dem Server synchronisiert werden muss.

Es folgt die Beschreibung der Struktur des FIT-Quellcodes anhand der Metriken (Anzahl der Klassen etc.), die auch schon bei den Open Source-Projekten in Kapitel 7.2 Anwendung fanden.

Trotz seiner komplexen Aufgaben und seinem Alter ist der FIT-Quellcode mit 661 Klassen und 106.481 LoC schlank gehalten. Dies ist auf die Aufwände von ca. 90 Personentagen (PT) zurückzuführen, die in den letzten Jahren in das Refactoring und die Codequalität geflossen sind. Eine ständige Anpassung der ursprünglichen Codebasis durch gesetzliche Änderungen oder neue Features und zugleich ein sorgfältiges Refactoring über die Versionen hinweg hielten den Quellcode schlank. Hinzu kommt, dass das Projekt seit 9 Jahren von einem Key-Entwickler betreut wird, welcher auch beim Audit unterstützte. Durch diesen Umstand blieb die Erfahrung über die Änderungen und Jahre hinweg immer verfügbar. Tabelle 89 zeigt die wichtigsten Metriken des Quellcodes.

Tabelle 89: Zusammenfassung der wichtigsten Metriken des FIT-Quellcodes

Packages	LoC	Klassen	Methoden	Verbindungen
36	106.481	661	7034	37808

7.4.2 Audit des FIT

Das Vorgehen für das Audit des FIT-Quellcodes bestand aus drei Stufen. In der ersten Stufe wurde das DPCDT der Allianz IT übergeben und dort auf den Quellcode von FIT angewandt. Eine Übergabe des Quellcodes an Personen außerhalb der Allianz war aus Vertraulichkeitsgründen nicht möglich. Danach wurden die Ergebnisse zurück an den Autor dieser Arbeit übergeben und von ihm analysiert und mehrere Kandidaten für eine spätere Evaluation ausgewählt. In der dritten Stufe wurde ein mehrstündiger Workshop mit FIT-Experten von der Allianz durchgeführt. In diesem Workshop wurden die Experten zuerst unterrichtet, welche Idee hinter der Kandidaten-Identifikation steht und wie diese technisch abläuft. Dieses Wissen war nötig, damit sich die Experten ein besseres Urteil über die Ergebnisse bilden konnten. Im Anschluss wurden die ausgewählten Kandidaten gemeinsam analysiert und diskutiert. Die Experten der Allianz, die am Workshop teilnahmen, waren:

1. Herr Dipl.-Inf. Harald Hadwiger, der über 20 Jahre Programmiererfahrung verfügt, davon 9 Jahre an dem gewählten Projekt. Seit 2 Jahren ist er auch IT-Projektleiter für die Anwendung. Er arbeitet seit 14 Jahren bei der Allianz.

2. Herr Dr.-Ing. Thomas Schmidt, der 2 Jahre Entwickler und Leiter des FIT-Projekts (s. u.) war und in seinen 3 Jahren bei der Allianz verschiedenste IT-Projekte betreute. Außerdem besitzt der Partner Detailwissen im Bereich Design Patterns, auch aus seiner Zeit als Wissenschaftler an der Universität Stuttgart. Er arbeitet seit 15 Jahren bei der Allianz.

7.4.3 Ergebnisse der Analyse

Bei der Analyse von FIT wurden 16 Kandidaten für vier der sechs Erkennungsregeln gefunden. Dies ist ein Wert von einem Kandidaten pro 41,31 Klassen, was wesentlich geringer ist als der Durchschnitt von vier Kandidaten pro Klasse in den Open Source-Projekten. Dies kann auf das industrielle Umfeld oder auch auf den hohen Refactoring-Aufwand zurückgeführt werden. Es wurden für zwei Erkennungsregeln keine Kandidaten gefunden: State und Facade. Das Nichtauffinden von State-Kandidaten ist keine Seltenheit, ist der Einsatz doch eher spezifisch in Embedded Systemen. Dass keine Facade-Kandidaten gefunden wurden, ist ein Anzeichen dafür, dass die Komponenten klar voneinander getrennt sind und keines der untersuchten Packages übermäßig mit anderen kommuniziert. In der Tabelle 90 werden die einzelnen Kandidaten mit Erkennungsstufe aufgezeigt.

Tabelle 90: Übersicht über die gefundenen Kandidaten mit Erkennungsstufe

Anzahl	Erkennungsstufe	Erkennungsregel
3	Möglich	Builder
1	Sinnvoll	Builder
2	Möglich	Decorator
3	Empfohlen	Decorator
1	Sinnvoll	Decorator
1	Möglich	Mediator
3	Empfohlen	Mediator
2	Möglich	Strategy
16	Gesamt	

Wird die Verteilung der Kandidaten in FIT mit der ermittelten Verteilung in Kapitel 7.2.3 verglichen, so ergibt sich bei der industriellen Anwendung ein anders Verteilungsbild. Es stammen mit 46% die meisten Kandidaten von der Decorator-Erkennungsregel. Bei den anderen Projekten lagen diese auf Rang 2. Im Vergleich zu den anderen Ergebnissen wurden mit 8% nur wenige Builder-Kandidaten gefunden. Diese waren zuvor auf Rang 1. Ebenfalls überraschend ist die Anzahl an Mediatoren, mit 31% fast fünfmal so hoch wie der Durchschnitt in den anderen Projekten. Einzig die Verteilung der Strategy-Kandidaten liefert ein ähnliches Bild in beiden Fällen, mit 15% bei Fit zu 11% im Durchschnitt.

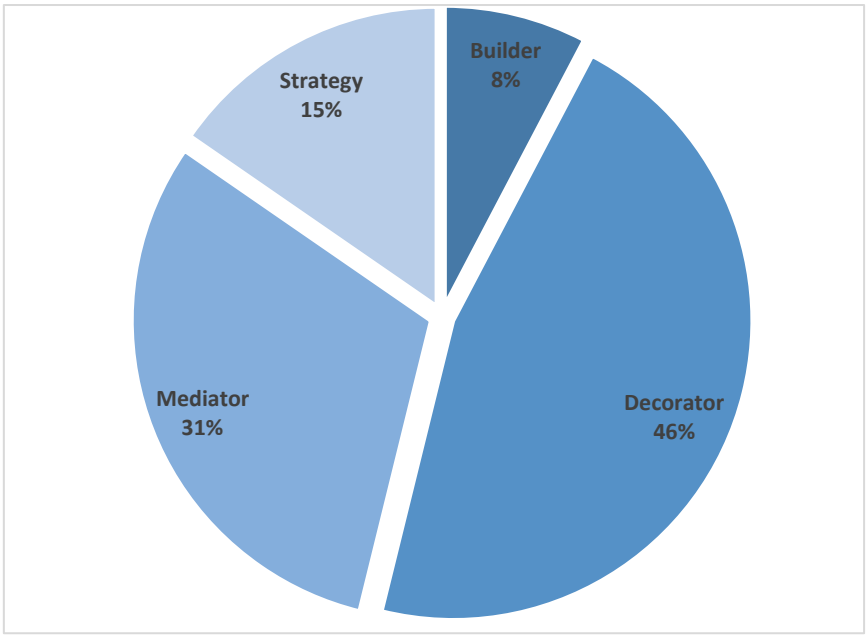


Abbildung 81: Verteilung der gefundenen Kandidaten in Prozent bei FIT

Nachfolgend werden einige Kandidaten beschrieben, die später in einem Workshop mit dem Experten diskutiert wurden.

Decorator-Beispiel in Klasse 1400

Das erste Beispiel war ein Decorator-Design Pattern-Kandidat in der Klasse 1400. Von dieser Klasse geht eine Vererbungshierarchie aus, die 23 weitere Klassen beinhaltet. Der Grenzwert für die Erkennungsstufe *Empfohlen* beim Decorator ist somit überschritten.

Decorator-Beispiel in Klasse 275

Im folgenden Beispiel handelt es sich wieder um einen Kandidaten für ein Decorator Design Pattern. Basierend auf der Analyse, besitzt dieser Kandidat eine Klassenhierarchie mit 230 Klassen auf der ersten Ebene.

Mediator-Beispiel in Package 1

Als nächstes wurde ein Mediator-Kandidat diskutiert, der sich im Package mit der Nr. 1 von FIT befindet. Hierbei handelt es sich um eine Gruppe von Controllern, die verschiedene Aktionen aus der Oberfläche ausführen.

Strategy-Beispiel in Klasse 2

Ebenfalls besprochen wurde ein Kandidat für ein Strategy-Design Pattern, welches mit der Erkennungsstufe *Möglich* eingestuft wurde. In der Klasse existieren zwei Methoden mit jeweils einem Switch, die über sieben gleiche Cases verfügen (VERTRAGSDATEN, ERGEBNIS, KUNDE, ANGEBOT, UEBERSICHT, DEFAULT, TARIFIERUNG).

Builder-Beispiel in Klasse 3883

Das letzte diskutierte Beispiel war ein Builder-Kandidat in der Klasse 3883, welcher 4 Konstruktoren besitzt.

7.4.4 Audit-Ergebnisse

Von jeder Erkennungsregel wurde ein Kandidat ausgewählt, um in einem Workshop mit den Partnern evaluiert zu werden. Es sei erwähnt, dass aus Datenschutzgründen kein Quellcode von FIT veröffentlicht werden darf. Deshalb wird die hier dargestellte Evaluierung anhand leicht abstrahierter Beispiele vorgenommen. Im Zuge des 2-stündigen Workshops wurden fünf Beispiele diskutiert.

Decorator-Beispiel in Klasse 1400

Die Stelle war den Experten bereits als verbesserungswürde Stelle bekannt. Es gab schon länger Diskussionen, dies zu verbessern. Beide bestätigten die Einschätzung des DPCDT-Ergebnisses: Eine Verbesserung durch das vorgeschlagene Entwurfsmuster würde bei weiteren Änderungen am Programm die Flexibilität verbessern.

Decorator-Beispiel in Klasse 275

Hier handelt es sich um eine versicherungsfachliche Besonderheit. Jede der einzelnen 230 Klassen bildet einen speziellen Tarifierungsfall ab (z. B. ob ein Gebäude eine bestimmte Pulverlöschanlage besitzt), der in seiner Art einzigartig ist. Den Experten war bewusst, dass dieser Umstand nicht optimal ist. Die fachliche Vorgabe kommt als komplexe Baumstruktur und bietet nur wenig Spielraum für eine Verbesserung. Möglich wäre ein neuer Ansatz im Design bei den Fachvorgaben, die aber im Rahmen des Workshops nicht weiter besprochen wurden, da dies nicht in den Möglichkeiten der IT liegt. Von der Grundidee her stimmen die Experten zu, dass eine problematische Stelle vorliegt, doch der Grund liegt nicht im Design des Quellcodes. Dieses Beispiel hingegen wurde von den Experten nicht als sinnvoll eingestuft. Zwar sind alle Grenzwerte der Erkennungsregel erreicht worden, doch in diesem Fall gibt es fachliche Gründe, die verantwortlich sind, ein Entwurfsmuster nicht zu verwenden. Es handelt sich um einen False-Positive-Kandidaten.

Mediator-Beispiel in Package 1

Als nächstes wurde ein Mediator-Kandidat diskutiert, der sich im Package mit der Nr. 1 von FIT befindet. Hierbei handelt es sich um eine Gruppe von Controllern, die verschiedene Aktionen aus der Oberfläche ausführen. Bei dieser Konstellation handelt es sich um eine komplizierte Anordnung von Kommunikationswegen. Ein Mediator wäre tatsächlich in der Lage, die Kommunikation zu verbessern. Aus Sicht der Experten müsste das Entwurfsmuster für die entsprechende Situation adaptiert werden. Diese Stelle soll im Zuge weiterer Wartungsarbeiten am Quellcode verbessert werden.

Strategy-Beispiel in Klasse 2

Ebenfalls besprochen wurde ein Kandidat für ein Strategy-Design Pattern, welches mit der Erkennungsstufe *Möglich* eingestuft wurde. In der Klasse existieren zwei Methoden mit jeweils einem Switch, die über sieben gleiche Cases verfügen (VERTRAGSDATEN, ERGEBNIS, KUNDE, ANGEBOT, UEBERSICHT, DEFAULT, TARIFIERUNG). Laut Aussage der Experten handelt es sich hierbei um eine Auswahl von verschiedenen User-Interaktionen. Je nach Auswahl wird entweder im Benutzer-Interface oder auf den Daten eine Aktion ausgeführt. Eine klare Trennung, wo die Aktion stattfinden soll, ist in der aktuellen Architektur zu diesem Zeitpunkt nicht möglich. Dies verhindert den Einsatz eines Strategy Patterns, da es durch diese komplexe Repräsentation unmöglich ist, die Aufgaben der einzelnen Aktionen in klare Strategien zusammenzufassen. Es muss immer noch unterschieden werden, ob Benutzer-Interface oder Daten angesprochen werden. So gibt es an dieser Stelle einen klaren Verbesserungsbedarf für das Design der Software, doch zuerst bedarf es einer Überarbeitung der architektonischen Vorgaben.

Diese Vorgaben sind laut Experten schon so zu erstellen, dass ein Strategy Pattern verwendet werden kann. Auf diese Weise wird es dann möglich sein, neue Aktionen schnell hinzuzufügen, was aktuell nicht gegeben ist.

Builder-Beispiel in Klasse 3883

Das letzte diskutierte Beispiel war ein Builder-Kandidat in der Klasse 3883, welcher 4 Konstruktoren besitzt. Im Zuge der Diskussion stellte sich heraus, dass hier auf jeden Fall Verbesserungsbedarf besteht. Nur 3 der 4 Konstruktoren sind aktuell im Einsatz. Für die restlichen 3 Konstruktoren wäre ein Builder-Pattern möglich. Dies entspricht auch der Empfehlung des DPCDT.

Allgemeine Einschätzung der Experten

Die Experten sahen die Vorschläge des Programms als gute Unterstützung für ihre Arbeiten an. Wie ich im Workshop zeigte, helfen die Vorschläge dabei, Stellen zu finden und diese dann zu diskutieren. Des Weiteren hilft das Vorgehen auch dabei, Stellen zu finden, die nicht im Blickpunkt stehen. Es sollte erwähnt werden, dass es Quellcodestellen gab, etwa das 1. Beispiel, die den Entwicklern hinlänglich als problematisch bekannt waren. Der Vorschlag eines Design Patterns kann helfen, diese Stellen in einem neuen Licht zu sehen.
Jedoch gab es auch Vorschläge die von den Experten abgelehnt wurden. Das DPCDT hat zwar einen Kandidaten erkannt, doch wegen externer Vorgaben, die nicht im Quelltext zu finden sind, kann das empfohlene Pattern nicht verwendet werden.

7.5 Auswertung der Evaluationsergebnisse

Es wurden Umfrageergebnisse von 50 Teilnehmern ausgewertet, die täglich mit der Entwicklung von Software zu tun haben. Die Auswertung ergab, dass in der Summe 22,62% die gezeigten Kandidaten für gute Vorschläge erachteten und 28,85% zumindest als sinnvoll. Nur 9,18% werden nicht von den Kandidaten überzeugt. Dies zeigt eine klare Tendenz für die vorgeschlagene Methodik. Auch in den Einzelwertungen für Entwurfsmusterkandidaten kommen vornehmlich positive Rückmeldungen.

Auch das Feedback der Experten im Audit war positiv gegenüber der Methode und den gezeigten Vorschlägen. Auch wenn nicht alle Vorschläge aus fachlicher Sicht umsetzbar waren, sehen sie schon allein die Möglichkeit der Unterstützung als sehr hilfreich und förderlich an.

Somit zeigen die Ergebnisse der Umfrage und die Rückmeldungen der Experten im Audit, dass die Methoden ein großes Potenzial besitzen.

7.6 Grenzen des Ansatzes

Der in dieser Arbeit vorgestellt Ansatz zur Erkennung von Entwurfsmusterkandidaten durch die Analyse von Quellcode besitzt ein paar Begrenzungen, welche im Folgenden kurz beschrieben sind.

So ist es aktuell nicht möglich Design Patterns zu erkennen wenn Dependency Injection Frameworks verwendet werden. In diesem Fall liegt der Quellcode nicht komplett vor bei der Untersuchung durch den AST, da erst während Laufzeit entschieden wird welche Teile eingebunden werden sollen.

Der Ansatz nimmt dem Entwickler nicht die Entscheidung ab, wo das Entwurfsmuster eingebaut werden soll. Er zeigt lediglich Möglichkeiten auf. Teilweise haben, wie beim Audit (vgl. Seite 164), fachliche Zusammenhänge Einfluss auf die Struktur des Quellcodes der nur von Menschen erkannt wird. So kann auch ein empfohlenes Pattern aus Entwicklersicht keinen Sinn machen.

Darüber hinaus ist der Ansatz nur in der Lage eine lokale verbesserte Quellcodestelle für den Einbau des Entwurfsmusters zu nennen. Er ist in der Lage den Standort des Smells zu bestimmen. Wie dieser dann in das Entwurfsmuster konvertiert werden soll, bleibt in der Hand des Entwicklers. Aus dem Smell ist nicht immer klar zu erkennen wie die Struktur des vorgeschlagen Design Patterns sein soll.

8 Verwandte und Zukünftige Arbeiten

„Die Zukunft kann man am besten voraussagen, wenn man sie selbst gestaltet."
- Alan Kay, amerikanischer Informatiker

In diesem Kapitel werden verwandte Arbeiten mit ähnlichem Inhalt kurz vorgestellt, die während der Entstehung dieser Arbeit veröffentlicht wurden. Außerdem wird im späteren Verlauf ein kurzer Ausblick über zukünftige Möglichkeiten gegeben, diese Arbeiten weiterzuführen.

8.1 Verwandte Arbeiten

Ein interessantes Verfahren, das bei der Dokumentation von Entscheidungen, die zur Verwendung eines Patterns führen, unterstützt, wurde von Durdik et al. (Durdik & Reussner, 2012) entwickelt. In ihrem Positionspapier skizzieren sie ein Verfahren das nach dem Ende der Entwicklung bei Wartungsarbeiten und beim Einbau von Änderungen unterstützen. Dabei werden wiederkehrende Design-Entscheidungen (z.B. Verwendung von Design Patterns) während der Design-Phase mit sogenannten pattern-specific-questions markiert. Eine Sammlung dieser Entscheidungen soll bei späteren Problemen / Entscheidungen helfen.

Ein Ansatz zu Empfehlung von Design Patterns wurde von Palma et al. (Palma, Farzin, Guéhéneuc, & Moha, 2012) publiziert. Diese favorisieren die Verwendung eines Expertensystems, basierend auf der Goal-Question-Metrik-Vorgehensweise (GQM). Dabei nutzten sie ein Frage-Antwort-Verfahren zur Bestimmung, ob ein Pattern geeignet ist. Ein Entwickler kann diese Fragen beantworten, um ein passendes Entwurfsmuster für sein aktuelles Problem zu finden. Das System bietet für jedes Entwurfsmuster Fragen an. Die Fragen zielen dabei auf das Design des Programmes und das Problem ab. Zur Erkennung des Patterns werden keine Informationen aus dem Quellcode verwendet. Dementsprechend handelt es sich hier um einen völlig manuellen Ansatz, der allein auf den Informationen des Entwicklers beruht.

In seiner Dissertation beschreibt Marinescu (Marinescu, 2002) eine Methode namens *strategy detection*. Diese basiert ebenfalls auf einem Regelsystem, welches den Entwickler unterstützen soll, den Quellcode zu verstehen und Entwurfsprobleme zu erkennen. Auf diese Weise lassen sich 15 verschiedene Smells erkennen, kategorisiert in vier Bereiche (Klassen, Methoden, Subsystem und Mirco-Design Pattern). Zu diesen Smells gehörten die sogenannte God-Klasse und ihre Varianten God-Methode und God-Package. Dabei vereint dieses Smell alle Aufgaben des Programmes in sich. Um diese Informationen bereitzustellen, verwendet der Ansatz verschiedene Kombinationen aus objektorientieren Metriken, wie z.B. Anzahl öffentlicher Attribute. Der Quellcode dient nicht als Basis.

Eine weitere Methode zum Erkennen von verbesserungswürdigen Strukturen in Quellcode stammt von Trifu (Trifu, 2008). Sein Ansatz beschäftig sich mit der Ermittlung von strukturellen Fehlern im Quellcode, um diese später mittels Refactoring verbessern zu können. Seine Arbeiten bauen auf denen von Marinescu (Marinescu, 2002) auf und erweitern diese. Zuerst wird das Problem analysiert und textuell beschrieben. Danach werden Indikatoren definiert, um vorhandene Entwurfsprobleme zu erkennen. Zum Schluss wird eine Restrukturierungsstrategie entwickelt, welche mit Refactorings das Problem löst.

Mit Hilfe dieser Methode ist es möglich neun verschiedene Entwurfsprobleme zu identifizieren und mit Refactoring-Lösungsansätzen zu verbessern. Die Verbesserung des Quellcodes geschieht nicht automatisch. Der Fokus liegt auf der Erkennung der Smells wie z.B. Schizophrene Klasse. Um diese zu identifizieren werden objekt-orientierte Softwaremetriken wie Weighted Method Count verwendet und nicht auf den Quellcode selbst zurückgegriffen.

In der Forschung gibt es schon seit längerem Ansätze, Design Patterns im Quellcode zu erkennen, aber noch keine, um mögliche Entwurfsmuster vorzuschlagen. Einen interessanten Ansatz zur Detektion von Design Patterns verfolgen Briand et al. (Briand, Labiche, & Sauve, 2006). Sie untersuchen UML-Diagramme nach Möglichkeiten, Design Patterns einzusetzen. Ihre Forschung basiert auf Entscheidungsbäumen, welche genutzt werden, um Einsatzgebiete für mögliche GoF-Patterns halbautomatisch in UML-Diagrammen zu identifizieren. Dazu wurden für verschiedene GoF-Pattens Entscheidungsbäume erstellt. Jede Entscheidung im Baum basiert auf der Analyse der Diagramme, welche unter Verwendung der Object Constraint Language (OCL) durchgeführt wurde. OCL ist eine formale Beschreibungssprache für Rahmenbedienungen in UML, wozu unter anderem Vor- und Nachbedienungen gehören. Das Ergebnis einer Analyse stellt fest, ob ein Entwurfsmuster möglich ist oder nicht. Es gibt aber keine Bewertung, wie sinnvoll der Einsatz eines Patterns an der gefundenen Stelle im UML ist. Aktuell unterstützt dieser Ansatz nur fünf von 23 der GoF-Patterns.

Issaoui et al. (Issaoui, Bouassida, & Ben-Abdallah, 2015) erarbeiteten eine interaktive Methode, basierend auf zwei Schritten, zur Unterstützung der Design Pattern-Auswahl. In Schritt 1 wird eine Matrix berechnet, die auf den Schlagwörtern der Beschreibungen und dem Aufbau einzelner Patterns basiert. Aus dieser Matrix wird eine Summe für jedes Pattern ermittelt: Je höher die Summe, desto eher ist ein Pattern für das zugrundliegende Problem geeignet. Im 2. Schritt werden dem Benutzer einige Fragen gestellt, um das Problem eingrenzen zu können.

In der jüngsten Vergangenheit gab es, wie in Kapitel 2.4 dargestellt, schon Ansätze zur Unterstützung von Entwicklern beim Einsatz von Design Patterns. Eine Unterstützung kann sowohl bei der Auswahl des Design Pattern-Typs als auch bei der Festlegung des Einsatzortes erfolgen. Jedoch waren alle Ansätze bisher immer ohne Bezug zum eigentlichen Quellcode und basierten meist nur auf Frage- und Antwort-Systemen.

Aktuell gibt es viele offene Fragen beim Einsatz von Design Patterns. Eine der wichtigsten Fragen in diesem Bereich ist, neben dem Identifizieren von Design Patterns in einem Quellcode, ob das Entwurfsmuster an dieser Stelle sinnvoll eingesetzt wurde. Im Forschungszweig der Entwurfsmuster-Erkennung sind Forscher wie Baranski et al. (Baranski & Voss, 2004) dieser Frage nachgegangen und haben erste Ergebnisse veröffentlicht. Unter anderem wurden Werkzeuge entwickelt, die verschiedene Technologien und Methoden nutzen.

Ein System zur Auswahl von Design Patterns stammt von Suresh et al. (Suresh, Naidu, Kiran, & Tathawade, 2011). Ihr Ansatz ruht auf der Sammlung von Entscheidungsinformationen wie Motivation, Konsequenzen etc. von unterschiedlichen Entwicklern. Aus den gesammelten Informationen wurden Fragen erstellt, die je nach Antwort ein Entwurfsmuster vorschlagen, ohne den Code zu kennen.

Ein weiteres Vorschlagssystem für Entwurfsmuster wurde von Guéhéneuc et al. (Guéhéneuc & Mustapha, 2007) entworfen. Dieser Entwurf wurde auf die 23 GoF-Patterns abgestimmt und basiert auf einer Analyse der textuellen Beschreibung von Design Patterns. Durch dieses Vorgehen entstand für jedes Entwurfsmuster ein Set an Schlagwörtern, aus denen der Benutzer die zutreffenden Wörter auswählen kann, um sein Problem zu beschreiben. Über diese Auswahl wird die Distanz zwischen den Sets und Wörtern berechnet und eine Rangliste mit Design Pattern-Vorschlägen aufgebaut.

Eine weitere Methode die Entwickler unterstützen soll Entwurfsmuster richtig einzusetzen nennt sich KARaCAs (*Knowledge Acquisition with Repertory Grids and Formal Concept Analysis for Dialog System Construction*) und basieren auf den Arbeiten von Garbe et al. (Garbe, Janssen, Möbus, Seebold, & de Vries, 2006). Wie andere Methoden ist es ein Frage-Antwort-System. Damit das richtige Pattern ausgewählt wird, wurde zuerst das Wissen verschiedener Experten zu Design Patterns gesammelt und über verschiedene Mechanismen zusammengefügt (formale Methoden und Ontologien). Aus den entstanden Merkmalen für verschiedene Pattern wurde ein bayes'sches Netz erzeugt, mit dessen Hilfe die Antworten des Entwicklers ausgewertet werden.

Der Automatisierungsgrad der vorgestellten Ansätze ist aktuell noch sehr niedrig. Einige vorgestellte Verfahren bauen auf Fragen, bei denen der Entwickler seine Situation darstellt und, ausgehend von seinen Antworten, ein entsprechendes Pattern vorgeschlagen bekommt. Dabei sind diese Verfahren unabhängig vom eigentlichen Design und vom Quellcode anzuwenden. Damit sind die genannten Verfahren nur halbautomatisch, da sie nur auf den Angaben des Entwicklers beruhen und die Implementierung auch von diesem manuell durchgeführt wird.

Wie die aufgeführten Referenzen zeigen, gibt es nach wie vor viele Forschungsfelder im Bereich Design-Pattern-Erkennung und im Bereich der Auswirkungen von Design Patterns. Ein weniger beachtetes Feld stellt das Empfehlen von Design Patterns bzw. die Unterstützung bei der Auswahl sowie die gezielte Erkennung von Quellcodezeilen dar, in denen Design Patterns angewandt werden sollten. Aktuell gibt es in dieser Hinsicht nur wenige Gruppen, die in diese Richtung forschen. Eine davon ist die um Christodoulou et al. (Christopoulou, Giakoumakis, Zafeiris, & Soukara, 2012). Ihre Arbeit konzentriert sich auf das Auffinden von möglichen Codestellen für ein Strategy Pattern. Zur Identifikation werden nur die Conditional Statements analysiert, alle anderen Informationen über den Quellcode werden nicht in Betracht gezogen. So liefert das Verfahren keine Informationen darüber, warum an einer Stelle dieses Pattern benötigt wird. Ein Vergleich der genannten Arbeiten mit der in dieser Arbeit beschrieben Methodik zeigt, dass ein Empfehlungssystem notwendig ist. Auf der einen Seite wird an Frage-Antwort-Systemen geforscht, die bei der Auswahl von Design Patterns während der Entwicklung von Programmen helfen sollen, bevor Quellcode geschrieben wird. Auf der anderen Seite werden Systeme entwickelt, um Design Patterns nach der Entwicklung im Quellcode wiederzuerkennen. Diese Arbeit füllt die Lücke dazwischen: neue Möglichkeiten, Design Patterns in schon vorhandenem Quellcode zu identifizieren und, mehr noch, den Entwickler in vielen Situationen dabei zu unterstützen, die Qualität seines Quellcodes durch den gezielten Einsatz von Design Patterns zu verbessern.

8.2 Künftige Arbeiten

Diese Methodik besitzt noch großes Potenzial um weitere Software Verbesserungsmöglichkeiten zu identifizieren, wie beispielsweise die Suche nach Refactoring Möglichkeiten, Dieser Abschnitt beschreibt die Richtungen für zukünftige Erweiterungen.

Zum einen sollten Regeln für andere GoF-Design Patterns in Erkennungsregeln umgesetzt werden (vgl. Kapitel 5.7), um die Unterstützung weiter auszubauen. Zur Erweiterung des vorgestellten Ansatzes liegt es nahe, das Vorgehen auch zur Erkennung von Code Smells (vgl. 2.3) oder dem Empfehlen von Refactorings nach Fowler einzusetzen. Bisherige Ansätze auf diesem Gebiet sind eher rudimentär, so dass ein automatisches Durchforsten von Code nach einer großen Zahl möglicher Refactorings interessante Anwendungsmöglichkeiten eröffnet. Aus der Sicht des Verfassers wäre es dadurch vorstellbar, eine völlig neuartige Metrik – nämlich die Anzahl der in einem Quellcode enthaltenen Smells – zu definieren, diese automatisiert zu erheben und so eine treffendere Aussage über die Codequalität zu erhalten, als dies mit „alleinstehenden" aktuellen Metriken möglich ist. Wie aussagekräftig dieses Verfahren sein kann, müssen zukünftige Untersuchungen. Dabei empfiehlt sich eine Fokussierung auf Vergleiche mit bereits existierenden Metriken.

Außerdem sollte die Möglichkeit untersucht werden, wie neben den GoF-Design Patterns auch Kandidaten für andere Patterns, wie Enterprise-Design Patterns (z. B. Service Layer), identifizieren zu können. Des Weiteren gibt es in der Software-Entwicklung noch andere Ebenen, wo Patterns oder Muster verkommen. So gruppieren Buschmann et al. (Buschmann, Henney, & Schimdt, 2007) Muster in der Software-Entwicklung in drei Ebenen, wie in Abbildung 82 dargestellt.

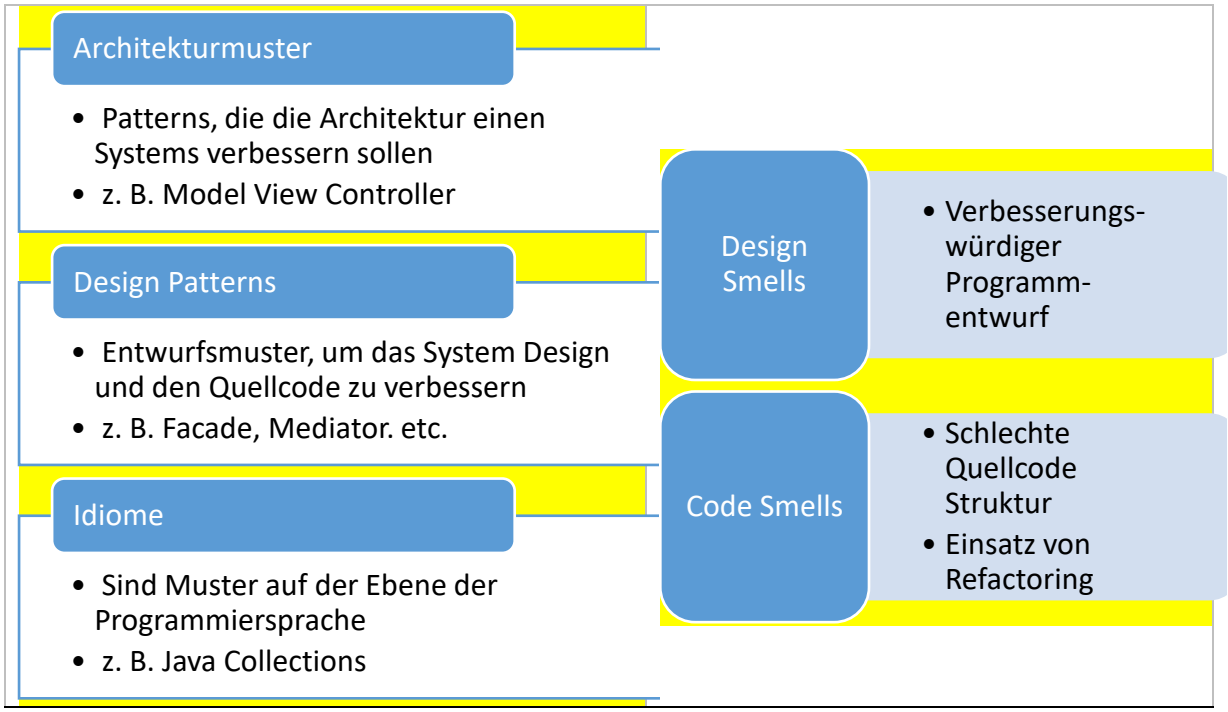


Abbildung 82 Auf der linken Seite Gruppierung von Muster nach Buschmann et al (Buschmann, Henney, & Schimdt, 2007) und rechten Seite die Zuordnung von Design und Code Smells zu den Gruppen

Die Detektion von Idiomen für Java und andere Programmiersprachen stellt eine zusätzliche Erweiterungsmöglichkeit dar, da diese Muster direkt mit der genutzten Programmiersprache zusammenhängen. Es sollte untersucht werden, inwieweit eine Detektion über den AST möglich ist. Bei der Ebene ‚Architektur-Pattern' stellt sich die Frage, ob und in welchem Maße die Architektur eines Programmes im Quellcode abgebildet ist. Eine Entscheidung, welches

Architektur-Pattern verwendet werden soll, wird immer von den Entwicklern getroffen. Eine nachträgliche Änderung ist nur mit sehr viel Aufwand durchführbar, wenn überhaupt. Die Möglichkeiten in dieser Richtung sind gering.

Architekturmuster und Design Patterns stehen Design Smells gegenüber. Diese Stellen haben negativen Einfluss auf die Qualität des Quellcodes. Durch den Einsatz von Mustern kann eine Verbesserung erfolgen. Zu hier gezeigt unterstütz die Identifikation von Entwurfsmuster-Kandidaten dabei. Auf einer Quellcode nahen Ebene sind es Code Smells die den Qualität verschlechtern. Auch hier können Entwurfsmuster unterstützen. Werden Programmiersprachmittel suboptimal eingesetzt helfen Refactorings. Gerade bei solchen Problemen wird gerne auf Hilfsmittel wie Stackoverflow zurückgegriffen um passende Idiome zu finden.

Im Zuge dessen sollte ebenfalls untersucht werden, inwieweit die schon bekannten Regeln auf andere Programmiersprachen angewendet werden können, sowohl auf andere objektorientierte als auch auf prozedurale Sprachen, etwa C. Generell sind Entwurfsmuster meist für objekt-orientieren Sprachen (Gamma, Helm, Johnson, & Vlissides, 1994) gedacht, doch einige Patterns wie State wurden auch für prozedurale Sprachen portiert.

Ob und wie sehr Software-Qualitätsprobleme durch Metriken, Patterns oder Refactoring besser unter Kontrolle gebracht werden können, ist nach wie vor ein offener Punkt (siehe 2.4.1). Daher ergibt sich eine weitere Aufgabe, Untersuchung der Auswirkungen anderer Ansätze, die ebenfalls die Codequalität erhöhen, wie z. B. Refactoring oder Architectural Patterns oder auch Coding Guidelines. Aus den insgesamt gewonnenen Erkenntnissen lassen sich letztlich neuartige Metriken (z. B. basierend auf einer Zählung von Code Smells) mit besserer Aussagekraft ableiten. Möglicherweise lassen sich sogar die erkannten Effekte aus bestehenden Metriken herausrechnen, um deren Aussagekraft zu verbessern, was zu verbesserten Vorschlägen für Patterns etc. führen würde.

9 Schlussfolgerungen

„Was ist denn das Erfinden? Es ist der Abschluß des Gesuchten."
- - Johann Wolfgang von Goethe (1749–1832), dt. Schriftsteller

In dieser Arbeit wurde eine neuartige Methodik vorgestellt, die Entwickler bei der komplexen Aufgabe der Identifikation und Auswahl von Design Patterns unterstützt. Der implementierte Prototyp ist in der Lage, automatisiert jede Art von Java-Quellcode zu analysieren, um Verbesserungspotenzial zu finden und direkt ein passendes Entwurfsmuster zu empfehlen. Auf dieser Grundlage ist der Entwickler in der Lage, schnell qualitativ hochwertigeren Quellcode zu erzeugen. Mit Stand heute gab es noch keinen Empfehlungsansatz (automatisiert oder manuell), der in der Lage war, solche Informationen direkt aus dem Quellcode zu extrahieren. Einzig einige Frage-und-Antwort-Systeme existieren, die vom Entwickler das Problem beschrieben haben wollen. Aktuell steht die Methodik noch am Anfang, doch sie bietet größeres Potenzial, noch weitere Smell- und Pattern-Möglichkeiten zu identifizieren, da das verwendete Regelsystem einfach erweitert werden kann.

Aus dieser Betrachtung wurden im Laufe dieser Arbeit die folgenden wissenschaftlichen Beiträge herausgearbeitet:

- Detaillierte Struktur- und Verhaltensanalyse, basierend auf der Definition nach Gamma et al. Basierend auf der Analyse wurden von sechs gängigen GoF-Patterns (Builder, Decorator, Facade, Mediator, State und Strategy) Praxisimplementierungen in Open Source-Projekten über die Merobase-Software gesucht und vermessen.

- Definition von Erkennungsregel der sechs genannten Design Patterns für Pattern-Kandidaten zur Identifikation und Bewertung von Quellcodestellen mit dem Ziel, die Qualität des Quellcodes zu verbessern. Dies beinhaltet auch die Festlegung von Grenzwerten zur Bestimmung des Verbesserungsbedarfs der gefundenen Kandidaten anhand von schon implementieren Design Patterns.

- Implementierung der festgelegten Regeln und Grenzwerte in einem Prototyp. Dieser ist in der Lage, Java-Quellcode zu analysieren, um Stellen zu identifizieren, die mit einem Entwurfsmuster verbessert werden können. Die Daten aus dem Prototyp sind später für die Evaluation genutzt worden.

- Nachweis über die Praxistauglichkeit der Entwickelten Konzepte und Erkennungsregeln durch:

- Analyse von 3,5 Millionen LoC aus 25 Open Source-Projekten mit der Prototypen-Software zur Sammlung von Evaluationsdaten, die in einer Studie mit 52 Teilnehmern evaluiert wurden. Der Teilnehmerkreis bestand zu über 92% nur aus professionellen Softwareentwicklern, der Rest waren Informatikstudenten.

- Durchführung einer Industriestudie in Zusammenarbeit mit der Allianz Deutschland, um die Qualität einer closed-Source-Software zu bestimmen und die Ergebnisse wie auch das Verfahren in einem Audit zu evaluieren.

Um eine Analyse des Quellcodes zu ermöglichen, wurden Erkennungsregeln für verschiedene Design Patterns festgelegt. Diese Erkennungsregeln basieren auf einem zuvor definierten, allgemeinen Vorgehen. Es beinhaltet die Analyse des Design Patterns, welche Charakteristika und welchen Einsatzzweck es besitzt, die Festlegung von Grenzwerten und die Auswahl von Metriken zum Bestimmen des Verbesserungsbedarfes. Bei der Auswahl der näher untersuchten Design Patterns wurde darauf Wert gelegt, von jedem Typus (Structural, Behavioural und Creational) ca. 25% der Gesamtmenge pro Typus zu untersuchen. So entstanden Regeln für die folgenden Design Patterns:

- Builder (Creational)
- Decorator (Structural)
- Facade (Structural)
- Mediator (Behavioural)
- State (Behavioural)
- Strategy (Behavioural)

Ein kritischer Punkt bei der Festlegung der einzelnen Regeln lag in der Festlegung der Grenzwerte. Ein Grenzwert ist nötig, um eine Charakterisierung dessen zu gewährleisten, wie hoch das mögliche Verbesserungspotenzial durch das Entwurfsmuster ist – und somit dem Nutzer zu zeigen, ob Handlungsbedarf besteht. Die Festlegung der Grenzwerte nutzt als Fundament den Index der Merobase Search Engine. Dieser Index umfasst mehrere Millionen LoC von verschiedenen Programmiersprachen, basierend auf verschiedenen Projekten, und damit auch viele bereits implementierte Design Patterns (siehe 4.2). Über die Merobase-eigene Suchsprache wurden mehrere Implementierungsbeispiele für jedes Entwurfsmuster gesucht und nach zuvor festgelegten Regeln vermessen. Aufbauend auf diesen Messergebnissen wurden Grenzwerte für verschiedene Software-Metriken festgelegt, die wiederum genutzt werden, um die Pattern-Kandidaten zu finden und zu charakterisieren.

Auf diesen Grundlagen wurde das DPCDT implementiert, das als Erweiterung für das Quellcode-Analysetool PMD konzipiert wurde. Dieses verwendet zur Quellcode-Analyse hauptsächlich den Abstract Syntax Tree der untersuchten Programmiersprache. Mit den definierten Erkennungsregeln und dem DPCDT wurden 25 Open Source-Projekte analysiert, um die Ergebnisse auszuwerten. Insgesamt wurden 3,6 Millionen LoC und 44.775 Klassen analysiert. Dabei wurden 1.913 Entwurfsmusterkandidaten identifiziert. Im Laufe der Analyse wurden für jedes Entwurfsmuster verschiedene Kandidaten mit unterschiedlichen Charakteristiken gefunden. Die Anzahl der gefundenen Kandidaten hing dabei sehr mit der Komplexität des Design Patterns zusammen. Für einfache Design Patterns wie den Builder sind Kandidaten einfacher zu identifizieren – einfach im Sinne, dass ihr Verwendungszweck und ihre Struktur wenig komplex sind und ihr Einsatzzweck vielseitig. Im Gegensatz dazu sind Kandidaten für hochkomplexe bzw. spezialisierte Patterns wie State, deren Einsatzgebiet sehr stark eingegrenzt ist (siehe 3.6), nur schwer zu finden.

Das Tool ist in der Lage, jede Art von Java-Quellcode zu untersuchen. Je höher die Anzahl an Klassen, desto mehr Zeit wird für die Untersuchung benötigt. Außerdem wird bei größeren Programmen immer mehr Speicher für die H2-Datenbank benötigt, die im Speicher gehalten wird. Abhängig vom System kann dies bei sehr großen Programmen zu Laufzeitproblemen führen. Eine

entsprechende technische Ausstattung wird dann benötigt, um die Analyse durchzuführen. Aktuell gibt es noch keine Mehrkernunterstützung, welche die Analyse beschleunigen könnte. Die Herausforderung hierbei liegt beim Datenbankzugriff, der mit mehreren Threads neu koordiniert werden muss.

Die so gewonnenen Ergebnisse der Open Source-Projekte wurden über eine Expertenbefragung evaluiert. An dieser Umfrage nahmen über 50 Experten, aus den verschiedensten Bereichen der Wirtschaft teil. Der Schwerpunkt des Teilnehmerkreises lag auf Entwicklern und IT-Architekten mit industriellem Hintergrund. Es konnten auch andere Teilnehmer, wie Projektleiter oder IT-Führungskräfte, für die Umfrage gewonnen werden. Im Durchschnitt kam jeder Teilnehmer auf mindestens 6 Jahre Programmiererfahrung bei einem Durchschnittsalter von ca. 32 Jahren. Bei dieser Befragung wurden ausgewählte Kandidaten-Beispiele den Teilnehmern zur Bewertung vorgelegt.

Eine Auswertung der Rückmeldungen ergab ein positives Bild über die gefundenen Entwurfsmusterkandidaten. 51,48% der Teilnehmer (siehe Tabelle 91) bewerteten die ihnen vorgelegten Beispiele, die auf der Analyse der Open Source-Programme basierten, als sinnvoll. Builder- und Facade-Entwurfsmusterkandidaten erhielten im Vergleich zu den anderen Kandidatentypen eine höhere Rückmeldequote. Durchweg wurden aber die meisten Kandidaten als sinnvoll bewertet. Ausnahme war der Kandidat für das Decorator-Design Pattern, welches von vornherein so gewählt war, dass es von seinem Aufbau her eher unsinnig wäre, einen Decorator zu verwenden. Dieses Beispiel diente als Kontrollfrage, denn wegen der Nutzung von Exception-Klassen in der Vererbungshierarchie mag theoretisch möglich sind aber praktisch nicht sinnvoll. Die meisten Teilnehmer merkten dies und gaben deshalb, wie zu erwarten war, keine positive Rückmeldung.

Tabelle 91: Kurzübersicht der Rückmeldungen

Design Pattern-Kandidat	Sinnvoll (1)	Eher sinnvoll (2)	Neutral	Eher nicht sinnvoll (4)	Nicht sinnvoll (5)	Enthaltung (6)
Anzahl Gesamt	69	88	44	28	28	48
Gesamt in%	22,62%	28,85%	14,43%	9,18%	9,18%	15,74%
	51,48%			18,36%		

Zudem wurde eine industrienahe Studie durchgeführt, in der das von der Allianz Deutschland bereitgestellte Tool „Firmen-Individual-Tarifierung“ (FIT) mit 661 Klassen analysiert wurde. Zu den Ergebnissen der industriellen Studie von FIT wurden der IT-Projektleiter des Programms und ein firmeninterner Experte interviewt. Dies ermöglichte es, die Ergebnisse direkt mit den verantwortlichen Stellen zu besprechen und in die Entwicklung einfließen zu lassen. Im Unterschied zu den zuvor evaluierten Open Source-Anwendungen zeigten sich schon in der Analyse andere Werte in der Erkennung von Entwurfsmusterkandidaten. Die zwei verfügbaren Experten der Allianz bewerteten die ausgewählten Kandidaten von FIT als positive Möglichkeit, eine andere Betrachtungsweise auf den Quellcode zu bekommen. Einige der vom DPCDT entdeckten Verbesserungsmöglichkeiten waren den Entwicklern schon als Problemstellen bekannt. Die vorgeschlagenen Design Patterns helfen dabei, eine andere Sichtweise auf den Quellcode zu erhalten. Basierend auf den Ergebnissen zeigen sich aber auch andere Quellcodestellen, die vorher nicht im Fokus von Verbesserung und Refactoring standen.

Aktuell konzentriert sich die Forschung auf zwei Bereiche: entweder das Erkennen von bereits implementierten Design Patterns oder das Anwenden von Frage-Antwort-Systemen zur Vorschlagsfindung, wie im Kapitel zuvor dargestellt.
Die hier vorgestellte Methodik verwendet ganz neue Ansätze:

- Erstens geht es, im Gegensatz zu Design Pattern-Erkennungswerkzeugen, um das Identifizieren und Bewerten von Entwurfsmusterkandidaten. Hierbei sind Kandidaten Codestellen, die durch ein Pattern verbessert werden können.

- Zweitens wird als Grundlage direkt der Quellcode verwendet. Damit entfällt eine zusätzliche Abstraktionsschicht wie bei Frage-Antwort-Systemen.

Dieser vollautomatische Ansatz erlaubt das schnelle und einfache Identifizieren von Kandidaten für den Entwickler. Auf Basis der sechs ausgewählten Design Patterns, die immer ca. 25% aus den verschiedenen Pattern-Typen (Structural, Behavioural und Creational) der GoF-Design Patterns abdecken, wurde durch die Analyse von 25 Softwareprojekten und die darauf aufbauende Evaluierung durch Experten der Nachweis erbracht, dass die Methodik die Erwartungen erfüllt. Eine Studie in dieser Größe, gemessen an der Anzahl an Projekten und LoC, wurde mit keinem Ansatz durchgeführt – ganz gleich, ob Expertensysteme oder Design Pattern-Erkennungswerkzeuge. Darüber hinaus ist die Evaluierung der Methodik durch über 50 Software-Engineering-Experten ein aufwendiger Evaluierungsansatz, der nur für wenige andere Forschungsansätze in der Software-Qualität genutzt wurde.

Wie im Kapitel 5.7 diskutiert, können für alle GOF-Patterns außer Composite Adapter und Interpreter entsprechende Erkennungsregeln definiert werden. Die drei genannten Patterns scheiden aus, da für ihren Einsatz Gegebenheiten außerhalb des untersuchten Codes berücksichtigt werden müssen, etwa beim Adapter die Inkompatibilität mit einer vorgegebenen Schnittstelle, für deren Verwendung sich ein Entwickler letztlich bewusst entscheiden muss.

Zusammenfassend kann gesagt werden, dass die Methodik und das DPCDT in der Lage sind, jede Art von Java-Quellcode zu untersuchen und dem Entwickler eine Rückmeldung über mögliche Verbesserungsmöglichkeiten mit den sechs ausgewählten Design Patterns zu geben.

10 Literaturverzeichnis

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools.

Alexander, C., Ishikawa, S., & Silverstein, M. (1977). Pattern languages. *Center for Environmental Structure*.

Alshayeb, M. (2009). Empirical investigation of refactoring effect on software quality. *Information and software technology, 51*(9), 1319-1326.

Alshayeb, M., & Li, W. (2003). *An Empirical Validation of Object-Oriented Metrics in Two Different Iterative Software Processes*. IEEE.

Alves, T. L., Ypma, C., & Visser, J. (2010). Deriving metric thresholds from benchmark data. *Software Maintenance (ICSM), 2010 IEEE International Conference on*, (S. 1-10).

Andrena Objects AG. (2015). *andrena objects Experts in Agile Software Engineering*. Abgerufen am 12. 04 2015 von https://www.andrena.de/

Apache MINA. (kein Datum). *Apache MINA Project*. Abgerufen am 12. 4 2015 von https://mina.apache.org/

Apache Wicket. (21. 10 2015). *Wicket 7*. Von apache.org: https://wicket.apache.org/ abgerufen

Athens Wireless Metropolitan Network. (2015). *wind-project/wind*. (github.com) Abgerufen am 12. 4 2015 von https://github.com/wind-project/wind

Aversano, L., Canfora, G., Cerulo, L., Del Grosso, C., & Di Penta, M. (2007). An empirical study on the evolution of design patterns. *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, (S. 385-394).

Baranski, M., & Voss, J. (2004). Detecting patterns of appliances from total load data using a dynamic programming approach. *Data Mining, 2004. ICDM'04. Fourth IEEE International Conference on*, (S. 327-330).

Bellman, R. (1956). *On a routing problem*. Tech. rep., DTIC Document.

Bieman, J. M., Straw, G., Wang, H., Munger, P. W., & Alexander, R. T. (2003). Design patterns and change proneness: An examination of five evolving systems. *Software metrics symposium, 2003. Proceedings. Ninth international*, (S. 40-49).

Binkley, A. B., & Schach, S. R. (1998). Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. *Proceedings of the 20th international conference on Software engineering*, (S. 452-455).

Briand, L. C., Labiche, Y., & Sauve, A. (2006). Guiding the application of design patterns based on uml models. *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*, (S. 234-243).

Burger, S., & Hummel, O. (2012). Applying maintainability oriented software metrics to cabin software of a commercial airliner. *16th European Conference on Software Maintenance and Reengineering (CSMR)*, (S. 457-460).

Burger, S., & Hummel, O. (2012). Lessons learnt from gauging software metrics of cabin software in a commercial airliner. *ISRN Software Engineering. 2012*. Hindawi Publishing Corporation.

Burger, S., & Hummel, O. (2013). Über die Auswirkungen von Refactoring auf Softwaremetriken. In G. f. eV (Hrsg.), *Software Engineering*, (S. 113-126).

Burger, S., Hummel, O., & Heinisch, M. (2013). Airbus Cabin Software. *Software, IEEE, 30*(1), 21-25.

Buschmann, F., Henney, K., & Schimdt, D. (2007). *Pattern-oriented Software Architecture: On Patterns and Pattern Language*. John Wily & Sons.

Castells, M. (1996). *The information age: economy, society and culture. Vol. 1, The rise of the network society* (Bd. 1). Blackwell Oxford.

Chatzigeorgiou, A., Tsantalis, N., & Stephanides, G. (2006). Application of graph theory to OO software engineering. *international workshop on Workshop on interdisciplinary software engineering research* (S. 29-36). ACM.

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on, 20*(6), 476-493.

Christopoulou, A., Giakoumakis, E. A., Zafeiris, V. E., & Soukara, V. (2012). Automated refactoring to the Strategy design pattern. *Information and Software Technology, 54*(11), 1202-1214.

Coleman, D., Ash, D., Lowther, B., & Oman, P. (1994). Using metrics to evaluate software system maintainability. *Computer, 27*(8), 44-49.

CollabNet. (2015). *Tigris.org Open Source Software Engineering Tools*. Abgerufen am 12. 4 2015 von http://www.tigris.org/

CollabNet, Inc. (2015). *ArgoUML*. Abgerufen am 12. 4 2015 von http://argouml.tigris.org/

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C., & others. (2001). *Introduction to algorithms* (Bd. 2). MIT press Cambridge.

Dice Holdings, Inc. (2015). *SourceForge - Download, Develop and Publish Free Open Source Software*. Abgerufen am 12. 4 2015 von http://sourceforge.net/

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik, 1*(1), 269-271.

Dong, J., Zhao, Y., & Peng, T. (2009). A Review of Design Pattern Mining Techniques., (S. 823-855).

Du Bois, B., & Mens, T. (2003). Describing the impact of refactoring on internal program quality. *International Workshop on Evolution of Large-scale Industrial Software Applications*, (S. 37-48).

Durdik, Z., & Reussner, R. (2012). Position paper: approach for architectural design and modelling with documented design decisions (ADMD3). *Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures*, (S. 49-54).

Fabry, J., & Mens, T. (2004). Language-independent detection of object-oriented design patterns. *Computer Languages, Systems & Structures*, S. 21-33.

Fenton, N. E., Pfleeger, & Lawrence, S. (1998). *Software Metrics: A Rigorous and Practical Approach.* Boston: PWS Publishing Co.

Floyd, R. W. (#jun# 1962). Algorithm 97: Shortest Path. *Commun. ACM, 5*(6), 345--. Von http://doi.acm.org/10.1145/367766.368168 abgerufen

Fowler, M. (2002). *Refactoring: improving the design of existing code.* Pearson Education India.

Freeman, A. (2015). The Builder Pattern. In *Pro Design Patterns in Swift* (S. 233-250). Springer.

Freeman, E., Robson, E., Bates, B., & Sierra, K. (2004). *Head first design patterns.* " O'Reilly Media, Inc.".

Gamma, E., & Eggenschwiler, T. (2015). *JHotDraw as Open-Source Project.* Abgerufen am 12. 4 2015 von SourceForge: http://jhotdraw.org/

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software.* Pearson Education.

Garbe, H., Janssen, C., Möbus, C., Seebold, H., & de Vries, H. (2006). KARaCAs: knowledge acquisition with repertory grids and formal concept analysis for dialog system construction. *International Conference on Knowledge Engineering and Knowledge Management* (S. 3-18). Springer.

Gillies, A. C. (1992). *Software Quality: Theory and Management.* London: Chapman & Hall, Ltd.

GitHub, Inc. (2015). *Github - Build software better, together.* Abgerufen am 12. 4 2015 von https://github.com/

Guéhéneuc, Y.-G., & Antoniol, G. (2008). DeMIMA: A Multilayered Approach for Design Pattern Identification. (S. Engineering, Hrsg.) *IEEE Transactions*, S. 667-684.

Guéhéneuc, Y.-G., & Mustapha, R. (2007). A Simple Recommender System for Design Patterns. *1st EuroPLoP Focus Group on Pattern Repositories.*

Guéhéneuc, Y.-G., Sahraoui, H., & Zaidi, F. (2004). Fingerprinting design patterns. *11th Working Conference on Reverse Engineering, 2004*, (S. 172-181).

Hahsler, M. (2003). *A quantitative study of the application of design patterns in java.* WU Vienna University of Economics and Business: Institut für Informationsverarbeitung und Informationswirtschaft.

Halstead, M. H. (1977). *Elements of software science.* (E. S. Inc, Hrsg.) North-Holland.

Hauer, P. (2. Februar 2016). *Das Strategy Design Pattern.* Von Homepage Philipp Hauer: http://www.philipphauer.de/study/se/design-pattern/strategy.php abgerufen

Hegedűs, P., Bán, D., Ferenc, R., & Gyimóthy, T. (2012). Myth or reality? analyzing the effect of design patterns on software maintainability. In *Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity* (S. 138-145). Springer.

Heitlager, I., Kuipers, T., & Visser, J. (2007). A practical model for measuring maintainability. *6th International Conference on the Quality of Information and Communications Technology*, (S. 30-39).

Heuzeroth, D., Holl, T., Hogstrom, G., & Löwe, W. (2003). Automatic design pattern detection. *Program Comprehension, 2003. 11th IEEE International Workshop on*, (S. 94-103).

Heuzeroth, D., Trifu, M., & Gutzmann, T. (2015). *RECODER*. (sourceforge.net) Abgerufen am 12. 4 2015 von http://sourceforge.net/projects/recoder/

Hoffmann, D. W. (2013). *Software-Qualität*. Springer-Verlag.

Holzner, S. (2006). *Design patterns for dummies*. John Wiley & Sons.

HTTP Server Apache. (21. 10 2015). *HTTP Server Project*. Von Apache.org: http://httpd.apache.org/ abgerufen

Huston, B. (2001). The effects of design pattern application on metric scores. *Journal of Systems and Software, 58*(3), S. 261-269.

IBM Software United States. (2009). *Rational Logiscope V6.6 helps deliver greater*. Tech. rep., IBM. Von http://www-01.ibm.com/common/ssi/rep_ca/3/897/ENUS209-173/ENUS209-173.PDF abgerufen

ISO. (2005). Software engineering--software product quality requirements and evaluation (square)--guide to square. *ISO Standard, 25000*, 2005.

ISO, I. (2001). Software Engineering-Product Quality-Part I: Quality Model. *ISO/IEC*, 9126-1.

Issaoui, I., Bouassida, N., & Ben-Abdallah, H. (2015). A New Approach for Interactive Design Pattern. *Lecture Notes on Software Engineering*. IACSIT Press.

Jacobson, I., Booch, G., & Rumbaugh, J. (1999). *The unified software development process* (Bd. 1). Addison-Wesley Reading.

Janjic, W., Hummel, O., Schumacher, M., & Atkinson, C. (2013). An unabridged source code dataset for research in software reuse. *Proceedings of the 10th Working Conference on Mining Software Repositories*, (S. 339-342).

Jones, C. (1994). Software metrics: good, bad and missing. *Computer, 27*(9), 98-100.

Kaur, K., Minhas, K., Mehan, N., & Kakkar, N. (2009). Static and Dynamic Complexity Analysis of Software Metrics. *World Academy of Science, Engineering and Technology, 56*, 2009.

Khomh, F., & Guéhéneuc, Y.-G. (2008). Do design patterns impact software quality positively? *12th European Conference on Software Maintenance and Reengineering*, (S. 274-278).

Khomh, F., Guéhéneuc, Y.-G., & Team, P. (2008). *An empirical study of design patterns and software quality*. GEODES–Research Group on Open, Distributed Systems. University of Montreal.

Kitchenham, B., & Pfleeger, S. L. (1996). Software quality: The elusive target. *IEEE software, 13*(1), 12-21.

Klein, H. K., & Myers, M. D. (1999). A set of principles for conducting and evaluating interpretive field studies in information systems. *MIS quarterly*, S. 67-93.

Lions, J.-L., & others. (1996). *Ariane 5 flight 501 failure*. Report by the enquiry board. University of California Berkeley.

Louridas, P. (2006). Static code analysis. *Software, IEEE, 23*(4), 58-61.

Marinescu, R. (2002). *Measurement and quality in object-oriented design*. Timişoara, Romania: Politehnica University of Timisoara.

Martin, R. C. (2008). *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

McCabe, T. J. (1976). A complexity measure. *Software Engineering, IEEE Transactions on*(4), 308-320.

McConnell, S. (2004). *Code complete*. Microsoft press.

Meng, X.-L., Rosenthal, R., & Rubin, D. B. (1992). Comparing correlated correlation coefficients. *Psychological bulletin*.

Mens, T., & Tourwé, T. (2004). A survey of software refactoring. *Software Engineering, IEEE Transactions on, 30*(2), 126-139.

Misa, T. J. (2007). Understanding'how computing has changed the world'. *IEEE Annals of the History of Computing*(4), 52-63.

Mueller, T. (2015). *H2 Database Engine*. Abgerufen am 12. 4 2015 von http://h2database.com/

Munro, M. J. (2005). Product metrics for automatic identification of" bad smell" design problems in java source-code. *11th IEEE International Symposium on Software Metrics*, (S. 15-15).

Nagappan, N., Ball, T., & Zeller, A. (2006). Mining metrics to predict component failures. *Proceedings of the 28th international conference on Software engineering*, (S. 452-461).

Naveh, B. (2015). *Welcome to JGraphT - a free Java Graph Library*. Abgerufen am 12. 4 2015 von http://jgrapht.org/

Nestler, F. (2015). Electronic Arts schließt SimCity- und Sims-Entwicklerstudio. *Frankfurter Allgemeine Zeitung*.

Opdyke, W. F. (1992). *Refactoring object-oriented frameworks*. Ph.D. dissertation, University of Illinois, University of Illinois at Urbana-Champaign, Urbana-Champaign.

Oracle. (13. 11 2015). *Java Language and Virtual Machien Speciifcations*. Von Oracle.com: https://docs.oracle.com/javase/specs/ abgerufen

Oracle. (2. Februar 2016). *A Visual Guide to Layout Manager*. Von Java Documentation: http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html abgerufen

Palma, F., Farzin, H., Guéhéneuc, Y.-G., & Moha, N. (2012). Recommendation system for design patterns in software development: An dpr overview. *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering*, (S. 1-5).

Parnas, D. L., & Clements, P. C. (02. Februar 1986). A Rational Design Process: How and Why to Fake it. *IEEE Ransactions on Software Engineering*.

PMD. (2015). *PMD Don't shoot the messenger*. Abgerufen am 12. 4 2015 von http://pmd.sourceforge.net/

Prechelt, L., Unger-Lamprecht, B., Philippsen, M., & Tichy, W. F. (2002). Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *IEEE Transactions on Software Engineering*, S. 595-606.

Rasool, G., & Mäder, P. (2011). Flexible design pattern detection based on feature types. *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (S. 243-252). IEEE.

Rauch, N., Kuhn, E., & Friedrich, H. (2008). Index-based Process and Software Quality Control in Agile Development Projects. *CompArch2008*.

RedMonk. (2015). *The RedMonk Programming Language Rankings: January 2014*. Abgerufen am 12. 4 2015 von http://redmonk.com/sogrady/2014/01/22/language-rankings-1-14/

Rich, A., Alexandron, G., & Naveh, R. (2009). An Explanation-Based Constraint Debugger. *Haifa Verification Conference*, (S. 52-56).

Rising, L. (2000). *The pattern almanac.* Addison-Wesley Longman Publishing Co., Inc.}.

Roberts, F. (1979). Measurement theory, encyclopedia of mathematics and its applications (Vol. 7). *Massachusetts: Addison-Wesley Publishing Company*.

Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, S. 131-164.

Samoladas, I., Gousios, G., Spinellis, D., & Stamelos, I. (2008). The SQO-OSS quality model: measurement based open source software evaluation. In *Open source development, communities and quality* (S. 237-248). Springer.

Schmidt, D. C., Stal, M., Rohnert, H., & Buschmann, F. (2013). *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects.* John Wiley & Sons.

Schröter, A., Zimmermann, T., & Zeller, A. (2006). Predicting component failures at design time. *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, (S. 18-27).

Scientific Toolworks, Inc. (2015). *scitools Understand.* Abgerufen am 12. 4 2015 von https://scitools.com/

Seiffert, D., & Hummel, O. (2015). Adapting Collections and Arrays: Another Step towards the Automated Adaptation of Object Ensembles. *International Conference on Software Reuse* (S. 348-363). Springer.

Seng, O. (2008). *Suchbasierte Strukturverbesserung objektorientierter Systeme.* Karlsruhe: Universitätsverlag Karlsruhe.

Sjoberg, D. I., Hannay, J. E., Hansen, O., Kampenes, V. B., Karahasanovic, A., Liborg, N.-K., & Rekdal, A. C. (2005). A survey of controlled experiments in software engineering. *IEEE transactions on software* , S. 733-753.

Slaughter, S. A., Harter, D. E., & Krishnan, M. S. (1998). Evaluating the cost of software quality. *Commications of the ACM*, 67-73.

Smith, C. U., & Williams, L. G. (2000). Software performance antipatterns. *Workshop on Software and Performance*, (S. 127-136).

Sneed, H. M., Seidl, R., & Baumgartner, M. (2010). *Software in Zahlen.* München: Carl Hanser Verlag GmbH & Co. KG.

Sommerlad, P., & Noble, J. (2007). point / counterpoint. *IEEE Software*, S. 68 - 71.

Stamelos, I., Angelis, L., Oikonomou, A., & Bleris, G. L. (2002). Code quality analysis in open source software development. *Information Systems Journal, 12*(1), 43-60.

Stavrinoudis, D., & Xenos, M. N. (2008). Comparing internal and external software quality measurements. *JCKBSE*, (S. 115-124).

Stöcker, C. (2013). Server-Ausfall: Wütende Fans verfluchen "SimCity". *Spiegel Online*.

Stroggylos, K., & Spinellis, D. (2007). Refactoring--Does It Improve Software Quality? *Proceedings of the 5th International Workshop on Software Quality*, (S. 10).

Suresh, S., Naidu, M., Kiran, S. A., & Tathawade, P. (2011). Design pattern recommendation system: a methodology, data model and algorithms. *ICCTAI'2011*.

Survio s.r.o. (2015). *Kostenlos Umfrage erstellen*. Abgerufen am 15. August 2015 von http://www.survio.com/de/

Suryanarayana, G., Samarthyam, G., & Sharma, T. (2014). *Refactoring for software design smells: Managing technical debt*. Morgen Kaufmann.

The Apache Software Foundation. (2015). *The Apache Software Foundation*. Abgerufen am 12. 4 2015 von http://apache.org/

Tourwé, T., & Mens, T. (2003). Identifying refactoring opportunities using logic meta programming. *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*, (S. 91-100).

Trifu, A. (2008). *Towards automated restructuring of object oriented systems*. Karlsruhe: Unversitätsverlag Karlsruhe.

Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., & Halkidis, S. T. (2006). Design pattern detection using similarity scoring. *Software Engineering, IEEE Transactions on, 32*(11), 896-909.

Van Emden, E., & Moonen, L. (2002). Java quality assurance by detecting code smells. *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, (S. 97-106).

Van Solingen, R., Basili, V., Caldiera, G., & Rombach, H. D. (2002). Goal question metric (GQM) approach. *Encyclopedia of Software Engineering*.

Vokac, M. (2004). Defect frequency and design patterns: An empirical study of industrial code. *IEEE Transactions on Software Engineering, 30*(12), 904-917.

W3C. (14. 11 2015). *XML Path Language (XPATH)*. Von W3C Rocommendation: http://www.w3.org/TR/xpath/ abgerufen

Welker, K. D. (2001). The software maintainability index revisited. *Crosstalk*, S. 12 - 21.

Wendorff, P. (2001). Assessment of design patterns during software reengineering: Lessons learned from a large commercial project. *Fifth European Conference on Software Maintenance and Reengineering*, (S. 77-84).

Wilking, D., Kahn, U. F., & Kowalewski, S. (2007). An Empirical Evaluation of Refactoring. *e-Informatica, 1*(1), 27-42.

Willnauer, S. (21. 10 2015). *Elasticsearch*. Von Github / Elasticsearch: https://github.com/elastic/elasticsearch abgerufen

Wolsey, L. A., & Nemhauser, G. L. (2014). *Integer and combinatorial optimization*. John Wiley & Sons.

Wydaeghe, B., Verschaeve, K., Michiels, B., Van Bamme, I., Arckens, E., & Jonckers, V. (1998). Building an OMT-editor using design patterns: An experience report. *Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings*, (S. 20-32).

Yin, R. K. (1981). The case study as a serious research strategy. *Science communication*, S. 97-114.

Zar, J. H. (1998). Spearman rank correlation. *Encyclopedia of Biostatistics*(7).

11 Abbildungsverzeichnis

12 Tabellenverzeichnis

13 Anhang

13.1Software Projekte in der Evaluation

Projekt	Version	Link
ArgoUML	0.34	http://argouml-downloads.tigris.org/argouml-0.34/
Columba	1.4	http://sourceforge.net/projects/columba/
JEdit	5.2	http://sourceforge.net/projects/jedit/
Apache Lucene	4.10.3	http://lucene.apache.org/
JHotDraw	7.6	http://sourceforge.net/projects/jhotdraw/
Apache Ant	1.9.4	https://ant.apache.org/
Apache Wicket	6.18.0	http://wicket.apache.org/
Ganttproject	2-6-1-r1499	http://sourceforge.net/projects/ganttproject/
Jrefactory	2.9.19	http://sourceforge.net/projects/jrefactory/
OpenHab	1.6	http://sourceforge.net/projects/openhab/
Freedomotic	5.5.0	http://sourceforge.net/projects/freedomotic/
Jfreechart	1.0.19+	http://sourceforge.net/projects/jfreechart/
Junit	r4.12	https://github.com/junit-team/junit/
Recoder	0.97	http://sourceforge.net/projects/recoder/
Jenkins	1.598	https://github.com/jenkinsci
Wind	1.0.1	http://sourceforge.net/projects/wind/
Derby	10.11.1.1	http://db.apache.org/derby/
Elasticsearch	1.4.4	https://github.com/elastic/elasticsearch
Freemind	1.0.1	http://sourceforge.net/projects/freemind/
Hibernate	4.5.2	http://sourceforge.net/projects/hibernate/
Jabref	2.10	http://sourceforge.net/projects/jabref/
Megamek	0.40.1	http://sourceforge.net/projects/megamek/
Mina	2.0.9	https://mina.apache.org/
spring-core	4.1.5	http://sourceforge.net/projects/springframework/
Triplea	1.8.0.5	http://sourceforge.net/projects/triplea/

13.2 Liste der verwenden Projekt für die Façade Grenzwerte

Projekt
Columba (ContactFacacde und IContactFacade)
proud (ProudMaschineFacade
Log4JDBC (Slf4jSpyLogDelegator verwendet SLF4j)
Apache solr (Mehrere Klassen nutzen SLF4J)
Apache Batik I18N (css.parser)
Apache Batik I18N (SVGDOMImplementation)
able (Strips ActionBeanContext)
Jboss Controller (using XMLElementWriter from Staxmapper)
Jboss Controller (using XMLElementReader from Staxmapper)
Storymaker using Soundcloud Java-API

13.3 Gefundene Kandidaten nach Projekten

Projekt	RULENAME	SMELLLEVEL
ANT	Builder Count Constructor	Possible
ANT	Builder Count Constructor	Recommended
ANT	Builder Count Constructor	Useful
ANT	Builder Count Parameter	Useful
ANT	Builder Count Parameter	Possible
ANT	Builder Count Parameter	Recommended
ANT	Decorator	Useful
ANT	Decorator	Recommended
ANT	Decorator	Possible
ANT	Facade	Recommended
ANT	Facade	Useful
ANT	Facade	Possible
ANT	Mediator	Possible
ANT	Mediator	Recommended
ANT	Strategy	Possible
ArgoUML	Builder Count Constructor	Possible
ArgoUML	Builder Count Constructor	Recommended
ArgoUML	Builder Count Constructor	Useful
ArgoUML	Builder Count Parameter	Recommended
ArgoUML	Builder Count Parameter	Possible
ArgoUML	Builder Count Parameter	Useful
ArgoUML	Decorator	Useful
ArgoUML	Decorator	Possible
ArgoUML	Decorator	Recommended
ArgoUML	Facade	Useful
ArgoUML	Facade	Possible
ArgoUML	Mediator	Recommended
ArgoUML	Mediator	Possible
ArgoUML	Strategy	Possible
ArgoUML	Strategy	Useful
Columba	Builder Count Constructor	Useful
Columba	Builder Count Constructor	Possible

Columba	Builder Count Constructor	Recommended
Columba	Builder Count Parameter	Useful
Columba	Builder Count Parameter	Possible
Columba	Builder Count Parameter	Recommended
Columba	Decorator	Possible
Columba	Decorator	Useful
Columba	Decorator	Recommended
Columba	Facade	Useful
Columba	Facade	Recommended
Columba	Facade	Possible
Columba	Mediator	Recommended
Columba	Mediator	Possible
Columba	Strategy	Possible
freedomotic	Builder Count Constructor	Possible
freedomotic	Builder Count Constructor	Recommended
freedomotic	Builder Count Constructor	Useful
freedomotic	Builder Count Parameter	Recommended
freedomotic	Builder Count Parameter	Useful
freedomotic	Builder Count Parameter	Possible
freedomotic	Decorator	Useful
freedomotic	Decorator	Recommended
freedomotic	Decorator	Possible
freedomotic	Facade	Possible
freedomotic	Facade	Recommended
freedomotic	Mediator	Possible
Ganttprojekt	Builder Count Constructor	Useful
Ganttprojekt	Builder Count Constructor	Possible
Ganttprojekt	Builder Count Parameter	Recommended
Ganttprojekt	Builder Count Parameter	Useful
Ganttprojekt	Builder Count Parameter	Possible
Ganttprojekt	Decorator	Possible
Ganttprojekt	Decorator	Recommended
Ganttprojekt	Decorator	Useful
Ganttprojekt	Facade	Recommended
Ganttprojekt	Facade	Useful
Ganttprojekt	Facade	Possible
Ganttprojekt	Mediator	Recommended
Ganttprojekt	Mediator	Possible
Ganttprojekt	Strategy	Possible
Jedit	Builder Count Constructor	Possible
Jedit	Builder Count Constructor	Useful
Jedit	Builder Count Constructor	Recommended
Jedit	Builder Count Parameter	Recommended
Jedit	Builder Count Parameter	Possible
Jedit	Builder Count Parameter	Useful
Jedit	Decorator	Recommended
Jedit	Decorator	Possible
Jedit	Decorator	Useful
Jedit	Facade	Possible
Jedit	Facade	Recommended
Jedit	Mediator	Recommended

Jedit	Mediator	Possible
jedit	Strategy	Possible
jedit	Strategy	Useful
Jenkins	Builder Count Constructor	Possible
Jenkins	Builder Count Constructor	Useful
Jenkins	Builder Count Constructor	Recommended
Jenkins	Builder Count Parameter	Recommended
Jenkins	Builder Count Parameter	Possible
Jenkins	Builder Count Parameter	Useful
Jenkins	Decorator	Recommended
Jenkins	Decorator	Useful
Jenkins	Decorator	Possible
Jenkins	Mediator	Possible
Jenkins	Mediator	Recommended
jfreechart	Builder Count Constructor	Useful
jfreechart	Builder Count Constructor	Possible
jfreechart	Builder Count Parameter	Useful
jfreechart	Builder Count Parameter	Recommended
jfreechart	Builder Count Parameter	Possible
jfreechart	Decorator	Useful
jfreechart	Decorator	Possible
jfreechart	Decorator	Recommended
jfreechart	Facade	Recommended
jfreechart	Mediator	Recommended
jfreechart	Strategy	Possible
jhotdraw	Builder Count Constructor	Useful
jhotdraw	Builder Count Constructor	Possible
jhotdraw	Builder Count Parameter	Possible
jhotdraw	Builder Count Parameter	Useful
jhotdraw	Builder Count Parameter	Recommended
jhotdraw	Decorator	Useful
jhotdraw	Decorator	Possible
jhotdraw	Decorator	Recommended
jhotdraw	Facade	Useful
jhotdraw	Facade	Recommended
jhotdraw	Facade	Possible
jhotdraw	Mediator	Possible
jhotdraw	Strategy	Useful
jhotdraw	Strategy	Possible
jrefactory	Builder Count Constructor	Possible
jrefactory	Builder Count Constructor	Useful
jrefactory	Builder Count Constructor	Recommended
jrefactory	Builder Count Parameter	Useful
jrefactory	Builder Count Parameter	Possible
jrefactory	Builder Count Parameter	Recommended
jrefactory	Decorator	Useful
jrefactory	Decorator	Recommended
jrefactory	Decorator	Possible
jrefactory	Facade	Useful
jrefactory	Facade	Recommended
jrefactory	Facade	Possible

jrefactory	Mediator	Recommended
jrefactory	Mediator	Possible
jrefactory	Strategy	Possible
junit	Builder Count Constructor	Possible
junit	Builder Count Constructor	Recommended
junit	Builder Count Parameter	Useful
junit	Builder Count Parameter	Possible
junit	Decorator	Useful
junit	Decorator	Possible
junit	Decorator	Recommended
junit	Facade	Useful
junit	Facade	Possible
lucene	Builder Count Constructor	Useful
lucene	Builder Count Constructor	Possible
lucene	Builder Count Constructor	Recommended
lucene	Builder Count Parameter	Possible
lucene	Builder Count Parameter	Useful
lucene	Builder Count Parameter	Recommended
lucene	Decorator	Possible
lucene	Decorator	Useful
lucene	Decorator	Recommended
lucene	Facade	Possible
lucene	Facade	Recommended
lucene	Mediator	Recommended
lucene	Mediator	Possible
lucene	Strategy	Possible
lucene	Strategy	Useful
openhab	Builder Count Constructor	Useful
openhab	Builder Count Constructor	Possible
openhab	Builder Count Parameter	Useful
openhab	Builder Count Parameter	Recommended
openhab	Builder Count Parameter	Possible
openhab	Decorator	Possible
openhab	Decorator	Useful
openhab	Decorator	Recommended
openhab	Facade	Useful
openhab	Facade	Possible
openhab	Facade	Recommended
openhab	Mediator	Possible
openhab	Mediator	Recommended
openhab	Strategy	Possible
openhab	Strategy	Useful
recorder	Builder Count Constructor	Useful
recorder	Builder Count Constructor	Recommended
recorder	Builder Count Constructor	Possible
recorder	Builder Count Parameter	Useful
recorder	Builder Count Parameter	Possible
recorder	Builder Count Parameter	Recommended
recorder	Decorator	Recommended
recorder	Decorator	Possible
recorder	Decorator	Useful

recorder	Facade	Possible
recorder	Mediator	Possible
recorder	Mediator	Recommended
recorder	Strategy	Useful
recorder	Strategy	Possible
wicket	Builder Count Constructor	Possible
wicket	Builder Count Constructor	Useful
wicket	Builder Count Constructor	Recommended
wicket	Builder Count Parameter	Recommended
wicket	Builder Count Parameter	Useful
wicket	Builder Count Parameter	Possible
wicket	Decorator	Possible
wicket	Decorator	Recommended
wicket	Decorator	Useful
wicket	Facade	Recommended
wicket	Facade	Useful
wicket	Mediator	Possible
wicket	Strategy	Possible
wind	Builder Count Constructor	Useful
wind	Builder Count Constructor	Possible
wind	Builder Count Constructor	Recommended
wind	Builder Count Parameter	Possible
wind	Builder Count Parameter	Useful
wind	Builder Count Parameter	Recommended
wind	Decorator	Useful
wind	Decorator	Recommended
wind	Decorator	Possible
wind	Facade	Useful
wind	Facade	Possible
wind	Facade	Recommended
wind	Mediator	Possible
wind	Mediator	Recommended
derby	Builder Count Constructor	Possible
derby	Builder Count Constructor	Useful
derby	Builder Count Parameter	Possible
derby	Builder Count Parameter	Useful
derby	Builder Count Parameter	Recommended
derby	Decorator	Recommended
derby	Decorator	Useful
derby	Decorator	Possible
derby	Facade	Possible
derby	Facade	Useful
derby	Facade	Recommended
derby	Mediator	Recommended
derby	Mediator	Possible
derby	Strategy	Possible
derby	Strategy	Recommended
derby	Strategy	Useful
wind	Strategy	Possible
wind	Strategy	Recommended
wind	Strategy	Useful

13.4Liste von Software-Metriken

Name	Untertyp
McCabe Cyclomatic Complexity	
Halstead	Volumen
	Effort
	Level
	Difficulty
	Potential
	Length
	Vocabulary
	N1/N2 (Unique)
	n1/n2 (sum)
	Time
Code Clones	
Lines of Code	Blank
	Statements
	Declarations
	Comment
Maintainability Index	
McClures Control Flow Metric	
Henry / Kafura's Information Flow Metric	Fan-in/Fan-out
Woodfield's Syntatic	Interconnection Measurcement
Code Size	Number of Files
	Number of Functions
	Number of Modules
Characters	Number of Characters
	Number of Comments
	Number of Comment Characters
	Number of Code Characters
Belady's Bandwith Metric	
Number of Parameters	
Ratio of Total number of Comments to LoC	
Syntactic Length of Function Names	
Number of Möglich Pathes	
Number of Binary Decisions	
Yau/Collefello's Logical Stability Metrics	
Semantic Metrics	Logical Relatedness of Methods
	Class Domain Complexity
	Relative Class Domains Complexity
	Keyclass Identify
	Class Overlap
OO-Metrics	Couping Between Objects
	Number of Services
	Lack of Cohesion
	Tight Class Cohesion
	Depth of Inheritance
	Weighted Methods per Class

13.5 Umfrage Bogen