Masterarbeit
zur Erlangung des akademischen Grades
Master of Science (M. Sc.)

an der

Freien Universität Berlin
Fachbereich Mathematik und Informatik
Studiengang Informatik

Entwicklung einer Methode und Werkzeugunterstützung für die Analyse, Bewertung und Optimierung von Vererbungsstrukturen in objektorientierten Softwaresystemen

1. Betreuer: Prof. Dr. Elfriede Fehr
2. Betreuer: Dr. Joachim Wegener

Eingereicht von: Christopher Zell
Matrikelnummer: 4749860
E-Mail: zelldon91@gmail.com
Datum: 30. September 2015

Freie Universität Berlin

BERNER & MATTNER
AN ASSYSTEM COMPANY

Danksagung

Ich danke meinen Eltern Annette und Karsten Zell, die mir dieses Studium ermöglichten und mich stetig in meinen Vorhaben bestärken. Meiner geliebten Partnerin Nancy Kujawa, die mich in allen Lebenslagen unterstützt und mir fortlaufend meinen Rücken frei hält sowie meinen Schwiegereltern Marlis und Ralf Kujawa, die mich tatkräftig unterstützten. Mein Dank gilt außerdem Gisela Balzer, Hans Richter und Stephan Eckart, die diese Arbeit zur Korrektur lasen. Des Weiteren danke ich Christopher Kruczek, der mir immer ein treuer Freund ist und mich bei dieser Arbeit mit seinem Rat unterstützte.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 1
- 1.2 Ziel und Aufgabenstellung . . . 1
- 1.3 Berner & Mattner . . . 1

2 Grundlagen **2**
- 2.1 Objektorientierte Programmierung . . . 2
- 2.2 Softwareentwicklungsprinzipien . . . 6
- 2.3 Typunterscheidungen in Programmiersprachen . . . 8
- 2.4 Restrukturierung . . . 11
- 2.5 Werkzeuge . . . 12
- 2.6 Testprojekte . . . 14

3 Indikatoren für Vererbungsprobleme **15**
- 3.1 Superklasse kennt Subklasse . . . 15
- 3.2 Bedingte Anweisungen mit Typüberprüfungen . . . 17
- 3.3 Klassen mit identischer Funktionalität . . . 20
- 3.4 Partiell Analoge Klassenschnittstellen . . . 22
- 3.5 Parallele Vererbungshierarchien . . . 24
- 3.6 Delegation Überbeanspruchung . . . 25
- 3.7 Subklassen variieren nur in Konstanten . . . 27
- 3.8 Downcasting . . . 28
- 3.9 Mehrfachvererbung . . . 31
- 3.10 Gewichtung . . . 33

4 Werkzeuganalyse **34**
- 4.1 Como . . . 34
- 4.2 Sotograph . . . 34
- 4.3 Klocwork . . . 35
- 4.4 CCFinder . . . 36

5 Entwurf und Implementierung **37**
- 5.1 Core . . . 37
- 5.2 Database . . . 38
- 5.3 IO . . . 40
- 5.4 Graphics . . . 41
- 5.5 Detection . . . 43
- 5.6 ICCore . . . 52

6 Evaluation **54**
- 6.1 BAT . . . 54
- 6.2 DC . . . 57
- 6.3 DÜ . . . 59
- 6.4 KIF . . . 60
- 6.5 MV . . . 62
- 6.6 PAKS . . . 64
- 6.7 PVH . . . 66
- 6.8 SKS . . . 68

7 Ergebnis **70**
- 7.1 Kritische Würdigung . . . 71
- 7.2 Ausblick . . . 72

8 Abkürzungsverzeichnis **74**

9 Glossar **75**

Abbildungsverzeichnis **75**

Tabellenverzeichnis **78**

Beispielverzeichnis **79**

Literatur **80**

Zusammenfassung

Code Quality Analyses (CQAs) ermöglichen die Überprüfung und Bewertung von Systemen auf deren Codequalität und vorhandenen Problematiken. Die Firma Berner & Mattner (B. M.) führt CQAs für die Analyse, Bewertung und Optimierung von Softwaresystemen durch. Diese Arbeit soll die Entwicklung von Anwendungen sowie die von B. M. durchgeführten CQAs unterstützen.

Es wurde innerhalb dieser Arbeit eine Methode und Werkzeugunterstützung zur Analyse, Bewertung und Optimierung von objektorientierten Vererbungsstrukturen entwickelt. Für die entwickelte Methode wurden Problematiken ausgearbeitet und Indikatoren definiert, die auf diese Problematiken hinweisen. Für jeden dieser definierten Indikatoren wurde eine Methode entwickelt. Diese ermöglicht die Analyse und Bewertung eines Indikators. Mithilfe der Analyse und Bewertung wird die Auswahl einer geeigneten Restrukturierungsmethode ermöglicht. Eine Anwendung der ausgewählten Restrukturierungsmethode löst einen Indikator und die damit verbundene Problematik auf. Daraus folgt eine Optimierung der vorhandenen Vererbungsstruktur. Zur Erkennung der definierten Indikatoren wurde eine Werkzeugunterstützung entwickelt, welche mithilfe von Werkzeugen wie Sotograph, Klocwork und CCfinder, die beschriebenen Indikatoren erkennt. Anhand der Open Source Projekte XBMC und HtmlUnit wurden die entwickelten Methoden und die Werkzeugunterstützung evaluiert.

Einleitung
1

1.1 Motivation

> » Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. « [vgl. Gam+94, S. 11]

Laut [Gam+94] ist das Erstellen einer objektorientierten Software schwer. Die Wiederverwendbarkeit erschwert die Erstellung. Wart- und Erweiterbarkeit sind wichtige Ziele in der Erstellung von Anwendungen [vgl. Gam+94, S. 39]. Diese Ziele können mithilfe von Design Pattern und einem gut durchdachten Design gewährleistet werden. Code Quality Analyses (CQAs) ermöglichen die Überprüfung und Bewertung der vorhandenen Systeme auf Wart- und Erweiterbarkeit. Diese Arbeit soll die Entwicklung von Anwendungen sowie die CQA unterstützen. Mit dem zu erstellenden Werkzeug soll es möglich sein, Probleme in Vererbungsstrukturen innerhalb objektorientierter Software aufzudecken und durch Auflösung dieser gefundenen Problematiken die Wart- und Erweiterbarkeit der Software zu erhöhen.

1.2 Ziel und Aufgabenstellung

Ziel dieser Masterarbeit ist die Entwicklung einer Methode und einer geeigneten Werkzeugunterstützung, welche für die systematische Analyse, Bewertung und Optimierung von Vererbungsstrukturen in objektorientierten Softwaresystemen genutzt werden können. Im theoretischen Teil dieser Arbeit werden aus der Literatur bekannte Anomalien zusammengetragen, Probleme in Vererbungsstrukturen analysiert und wenn möglich geeignete Indikatoren definiert. Die Indikatoren dienen zur Erkennung von Problemen in Vererbungsstrukturen. Innerhalb der Analyse wird erläutert, welche Probleme diese Indikatoren verursachen und welche Restrukturierungsmethoden existieren. Die Indikatoren werden anhand der verursachten Probleme gewichtet. Mit dieser Gewichtung soll verdeutlicht werden, auf welche Indikatoren bei der Behebung besonders geachtet werden muss. Die praktische Aufgabe in dieser Masterarbeit befasst sich mit der Entwicklung eines Analysewerkzeug, welches die im theoretischen Teil identifizierten Indikatoren in objektorientierten Softwaresystemen erkennt und auf diese hinweist. Die zu analysierenden Softwaresysteme sind dabei in Programmiersprachen wie Java, C++ oder C# implementiert. Die erkannten Indikatoren werden mittels Text sowie Grafiken angezeigt, wobei eine grafische Darstellungen bevorzugt wird. Die Implementierung erfolgt in C# und wird so anpassbar sein, dass das Werkzug in ein bereits existierendes Analysewerkzeug der Firma Berner & Mattner (B. M.) eingebunden werden kann.

1.3 Berner & Mattner

Die Firma B. M. ist spezialisiert auf System Engineering, Entwicklung und Tests leistungsfähiger elektronischer und mechanischer Systeme. Ihr branchenübergreifendes Leistungsspektrum reicht von der Beratung, Konzeption, Software- und Systementwicklung bis hin zum Aufbau und Betrieb kompletter Test- und Integrationssysteme. Seit 2011 ist B. M. Mitglied der Assystem Group. Assystem ist ein internationales Engineering und Consulting Unternehmen mit über 11.000 Mitarbeitern und einer Präsenz in 19 Ländern [BM15].

Grundlagen
2

2.1

Objektorientierte Programmierung

Die in diesem Abschnitt beschriebenen Object-oriented programming (OOP) Konzepte werden zur Verständlichkeit dieser Arbeit kurz vorgestellt und beschrieben. Es wird dabei nicht auf alle Details bzw. alle Konzepte des OOP eingegangen. Wissen über Klassen, Objekte bzw. Instanzen wird dabei vorausgesetzt. Die objektorientierte Programmierung konzentriert sich auf das Design, Implementieren und Verwenden von Klassenhierarchien. Die Klassenhierarchien liefern sogenannten run-time Polymorphismus und Verkapselung [vgl. Str13, S. 11]. Die Schlüsselideen hinter der Objektorientierung sind laut [Aho+14] Datenabstraktion und die Vererbung von Eigenschaften. Diese Konzepte ermöglichen die modularere Gestaltung und vereinfachen die Wartbarkeit von Programmen [vgl. Aho+14, S. 18].

2.1.1

Vererbung

Vererbung oder auch Ableitung beschreibt in der objektorientierten Programmierung ein Konzept zur Wiederverwendbarkeit von Quellcode. Es erlaubt die Wiederverwendung von Quellcode durch das Erstellen von neuen abgeleiteten Klassen, die die Eigenschaften der Basisklassen erben. Eine abgeleitete Klasse erbt die Eigenschaften der Basisklasse, das schließt Daten und Methoden mit ein. Es können neue Daten und Methoden zu den abgeleiteten Klassen hinzugefügt werden. Zudem können die Implementierungen der Methoden der Basisklassen innerhalb der abgeleiteten Klassen überschrieben werden [15c]. Eine Klasse kann von einer anderen Klasse deren Funktionalitäten sowie Attribute erben und sie somit wiederverwenden und erweitern. Die Vererbung wird verwendet, um hierarchische Beziehungen und Gemeinsamkeiten zwischen Klassen anzuzeigen [vgl. Str13, S. 578]. Vom Design Standpunkt ist die Vererbung das Ausdrücken einer Generalisierung oder Spezialisierung. Die Vererbung beschreibt eine `ist-ein` Beziehung zwischen den Super- und Subklassen. Ein Objekt einer abgeleiteten Klasse kann überall verwendet werden, wo auch immer ein Objekt der Superklasse benötigt wird. Die Superklasse agiert als Abstraktion für die abgeleitete Klasse.

Die Sprachen C++, C# und Java haben die Idee der Klassen sowie die der Klassenhierarchien von Simula übernommen [vgl. Str13, S. 577], [Aho+14]. Die Programmiersprache Simula war eine der ersten Sprachen, die dieses Konzept einsetzte. Die Implementierung von Vererbung und Schnittstellen bildet die Basis von dem sogenannten OOP [vgl. Aho+14, S. 18]. Die Schnittstellenimplementierung wird auch oft als Laufzeit-Polymorphismus oder dynamischer Polymorphismus bezeichnet. Im Gegensatz dazu steht der Übersetzungszeit-Polymorphismus oder statischer Polymorphismus [vgl. Str13, S. 578]. Die statische Polymorphie beschreibt die Überladung von Methoden [LR06a].

2.1.2

Abstrakte Klassen und Schnittstellen

Laut [Str13] liefert eine abstrakte Klasse eine Schnittstelle ohne Implementationsdetails. Abstrakte Klassen beinhalten abstrakte Methoden. Diese Methoden besitzen keine Implementierung, sie sind eine reine Deklaration der Methodensignatur. Eine abstrakte Klasse kann nur als Schnittstelle für eine andere Klasse verwendet werden [vgl. Str13, S. 598]. Dies gilt jedoch nur für C++. Denn Java und C# unterscheiden in abstrakte Klassen und Schnittstellen. Schnittstellen stellen im Grunde Verträge dar, die erfüllt werden müssen, wenn eine Klasse diese Schnittstelle implementieren soll. Wird eine Schnittstelle verwendet, ist die Verwendung der Schnittstelle von der Schnittstellenimplementierung unabhängig. Die Klasse, die diese Schnittstelle implementiert, muss sich an die vorgeschriebenen Methodensignaturen halten [Ora15c]. Es ist nicht möglich Instanzen von Schnittstellen oder abstrakten Klassen anzulegen. Der Designstil, der durch abstrakte Klassen und Schnittstellen unterstützt wird, wird `interface inheritance` genannt. Er ermöglicht es, einen gewissen Grad an Abstraktion zu schaffen. Dieser steht im Gegensatz zur `implementation inheritance`. Hier liefert eine Superklasse Funktionalität oder Daten, die von der Subklasse wiederverwendet werden können. Laut [vgl. Str13, S. 70] vereinfacht dies das Implementieren der abgeleiteten Klasse.

Bei Superklassen mit Zustand und/oder definierten Methoden ist eine Kombination von beiden Herangehensweisen möglich. D. h. es wird eine Superklasse mit Zustand und abstrakten Methoden definiert(in C++ pur virtuellen Methoden genannt). Solch eine Mixtur von Herangehensweisen kann laut [Str13] verwirrend sein und erhöhte Aufmerksamkeit fordern. [vgl. Str13, S. 599]. Diese Mixtur stellt die abstrakten Klassen in Java und C# dar. Laut [15e] wird die Phrase `interface inheritance` häufig verwendet, aber korrekt wäre eher `interface implementation`, da eine Schnittstelle von einer Klasse implementiert wird.

Es besteht die Möglichkeit der Schnittstellenvererbung, d. h. dass eine Schnittstelle von einer anderen erbt, sodass die Sammlung von Methoden erweitert wird. In Java und C# ist es nur möglich, dass eine Klasse von einer Klasse erbt. Wogegen eine Schnittstelle von mehreren Schnittstellen erben kann und eine Klasse auch mehrere Schnittstellen implementieren kann [Ora15a]. C++ dagegen erlaubt die sogenannte Mehrfachvererbung. Eine Subklasse darf direkt von mehreren Klassen erben, siehe dazu Abschnitt 2.1.3.

2.1.3

Mehrfachvererbung

Das direkte Erben von mehr als einer Klasse wird Mehrfachvererbung genannt [vgl. Str13, S. 618]. Die Mehrfachvererbung wird in C++ unterstützt. In Java und C# ist diese nicht verfügbar. Die Unterstützung von Vererbung und zur Übersetzungszeit geprüften Schnittstellen bzw. die Benutzung einer Superklasse für die Implementierungsdetails und eine andere abstrakte Klasse für die Schnittstelle ist in allen Programmiersprachen verbreitet. Speziell in C++ ist die Benutzung einer abstrakten Klasse fast eindeutig das gleiche wie die Benutzung von Schnittstellen in C# und Java [vgl. Str13, S. 619]. In C++ kann jede Klasse, die keinen veränderbaren Zustand besitzt, als Schnittstelle in einer Mehrfachvererbung ohne signifikante Komplikationen und Mehraufwand benutzt werden. Die Verwendung von mehrfachen Schnittstellen ist in Object-oriented designs (OODs) beinahe einheitlich [vgl. Str13, S. 624].

[Str13] bevorzugt eine einfache Vererbungshierarchie und wo es nötig ist, einige abstrakte Klassen, die Schnittstellen liefern, denn dies ist typischerweise flexibler und führt zu Systemen, die einfacher zu entwickeln sind [vgl. Str13, S. 626]. Durch die Mehrfachvererbung von Klassen können Mehrdeutigkeiten entstehen.

Beispielsweise wenn zwei Superklassen ein Element mit identischem Namen besitzen, kann die Subklasse nicht implizit zwischen den Elementen unterscheiden [15c]. Eine Verwendung der Subklasse und eine dieser mehrdeutigen Elemente wird vom Compiler erkannt und somit auch nicht übersetzt. Diese Mehrdeutigkeit kann mithilfe von Überschreibung aufgelöst werden. Eine Methode, die in einer abgeleiteten Klasse definiert ist, überschreibt alle Methoden mit gleicher Signatur in den Superklassen [vgl. Str13, S. 627]. Sind in der Vererbungshierarchie nur pur virtuelle bzw. abstrakte Methoden ohne Implementierung vorhanden, dann führt die Mehrfachvererbung zu keinerlei Problemen. Dergleichen wird durch Schnittstellen ermöglicht, wie z. B. in den Sprachen C# und Java. Es besteht kein Problem so vielen verschiedenen Schnittstellen wie gewünscht zu implementieren, dies wird als Mehrfachimplementierung bezeichnet. [15e].

2.1.4 Polymorphie

Das OXFORD-Dictionary definiert Polymorphie (engl. polymorphism) als die Voraussetzung des Erscheinens in verschiedenen Formen. Es wird auch als Vielgestaltigkeit bzw. Verschiedengestaltigkeit definiert [de]. In der Informatik ist Polymorphie ein besonderes Konzept des OOP. Es ermöglicht, dass ein Objekt für verschiedene Klassen stehen kann [Spo07]. Laut Kemper und Eickler kann eine Instanz einer Subklasse überall dort verwendet werden, wo eigentlich eine Instanz einer Superklasse gefordert ist [vgl. KE09, S. 391]. Polymorphismus ist die Möglichkeit einer Methode verschiedene Dinge abhängig vom aktuellen Objekt zu tun. Polymorphismus erlaubt es, Schnittstellen zu definieren und verschiedene Implementierungen zu besitzen [Sin14; Dow12].

2.1.5 Assoziation

Die Unified Modeling Language (UML) Spezifikation beschreibt die Assoziation als eine Beziehung zwischen zwei Klassen, die benutzt wird, um zu zeigen, dass Instanzen von Klassen als logische oder physikalische Aggregation verbunden oder kombiniert werden können. Die Spezifikation kategorisiert die Assoziation als semantische Beziehung.

Aggregation

Eine Aggregation ist eine binäre Assoziation, die eine ”ist-teil-von” Beziehung darstellt. Jede Aggregation ist eine Assoziation, jedoch ist nicht jede Assoziation eine Aggregation. Es existieren zwei Arten von Aggregationen: die geteilte Aggregation und die Komposition. Die erste ist die ”schwache” Form der Aggregation. Dies ist der Fall, wenn die Teilinstanz unabhängig von der Zusammensetzung (Kompositum) ist. Dasselbe Objekt kann von mehreren verschiedenen Zusammensetzungen inkludiert werden. Wird das Kompositum gelöscht, können die geteilten Objekte bzw. Teile weiter existieren. Ein Beispiel für diese geteilte Aggregation ist die Verwendung eines Parameters oder einer Variable vom Typ der Teilklasse innerhalb einer Methode. Die Komposition ist die ”starke” Form der Aggregation. Der Unterschied zur ”schwachen” Aggregation besteht in der Beziehung zu den Teilobjekten. Ein Teil bzw. ein Objekt, welches zur Zusammensetzung gehört, gehört nur zu einem Kompositum. Wird das Kompositum gelöscht, werden alle Teile mit gelöscht [vgl. 15i, S. 110].

Komposition

Komposition und die Vererbung sind in objektorientierten Systemen die zwei Techniken, um Wiederverwendung zu ermöglichen. Vererbung wird dabei oft als White-Box-Wiederverwendung bezeichnet, da die internen Strukturen der Elternklasse einer Subklasse offen liegen. Dies ist oft auch ein Kritikpunkt der Vererbung, da diese die Verkapselung aufhebt. Die Komposition bzw. Objektkomposition ist eine Alternative. Dieser Stil der Wiederverwendung wird auch als Black-Box-Wiederverwendung bezeichnet, da keine internen Strukturen offen liegen. Deshalb sind gut definierte Schnittstellen notwendig. Laut [Gam+94] wird die Vererbung als Wiederverwendungstechnik von Designern überbeansprucht. Entwürfe sind laut den Autoren oft wiederverwendbarer und einfacher durch die Verwendung von Objektkomposition. Aus diesem Grund sollte die Objektkomposition der Klassenvererbung vorgezogen werden [vgl. Gam+94, S. 32]. Das Vorziehen der Komposition vor der Vererbung hilft einer Klasse, die Kapselung aufrecht zu erhalten und sich auf eine Aufgabe zu konzentrieren. Die Klassen sowie die Hierarchien bleiben klein. Aber dadurch wächst die Verwendung von Objekten und das Verhalten des Systems ist abhängig von deren Wechselbeziehungen. Idealerweise müssen keine neuen Komponenten erstellt werden, um eine Wiederverwendung zu ermöglichen. Dies ist nicht immer der Fall, da die verfügbaren Komponenten in der Praxis oft nicht breit genug gefächert sind. Wiederverwendung durch Vererbung macht es leichter neue Komponenten zu erstellen, die dann komponiert werden können. Vererbung und die beschriebene Objektkomposition arbeiten zusammen [vgl. Gam+94, S. 31 f.].

Delegation

Laut [Gam+94] ist die Delegation ein extremes Beispiel der Objektkomposition. Es zeigt, dass die Vererbung immer mit Objektkomposition als Wiederverwendungsmechanismus ersetzt werden kann [vgl. Gam+94, S. 32 f.]. In der Delegation werden Anfragen oder Methodenaufrufe delegiert, zwei Objekte sind dabei involviert. Zum einen das Objekt, das die Anfragen erhält, und ein delegiertes Objekt, zu dem die Anfragen weitergeleitet werden. Anstatt von einer Klasse deren Funktionalitäten zu erben, wird ein Objekt dieser Klasse als Instanzvariable verwendet und Teile der benötigten Funktionalitäten aufgerufen und infolgedessen Anfragen delegiert [vgl. Gam+94, S. 33]. Der Vorteil dieser Methode ist es, dass das zu delegierende Objekt während der Laufzeit ersetzt werden kann und die Datenkapselung erhalten bleibt. Jedoch ist die Delegation nur eine gute Wahl, wenn sie mehr vereinfacht als erschwert, denn dynamisch hoch parametrisierte Software ist schwerer zu verstehen als statische [vgl. Gam+94, S. 32 f.] [vgl. KS08, S. 440]. [KS08] zeigt, dass in Programmiersprachen, in denen nur Einzelvererbung erlaubt ist, durch die Delegation eine neue Vererbung ermöglicht wird. Wurde z. B. eine Vererbung durch Restrukturierung (siehe dazu Abschnitt 2.4) in eine Delegation umgewandelt, dann kann die Klasse von einer anderen Klasse erben.

2.1.6

Casting

Das Konvertieren eines Typs in einen anderen wird als casting bezeichnet. Es wird zwischen Konvertierungen von primitiven Typen und Klassentypen unterschieden. Konvertierungen von primitiven Typen sind castings von einfachen Typen in andere einfache. Ein Beispiel ist eine Konvertierung von `int` nach `double`. Diese können bereits zur Übersetzungszeit überprüft werden. Der Übersetzer erkennt, ob dieses casting valide ist. Bei Konvertierungen von Klassentypen wird zwischen explizitem und implizitem casting unterschieden. Laut [Ora15b] zeigt ein casting die Verwendung eines Objekts eines Typs an Stelle eines anderen Typs an. Unter der Bedingung, dass diese in einer Vererbungsbeziehung oder Implementierungsbeziehung stehen. In dem Beispiel 2.1 sind Konvertierungen dargestellt.

Beispiel 2.1: Castings

```
Parent parent1 = new Child();
Child child1 = (Child) partent1; //success

Parent parent2 = new Parent();
Child child2 = (Child) parent2; //runtime error
```

Die erste Anweisung stellt ein implizites casting auch `upcast` genannt dar. Diese Konvertierung kann bereits zur Übersetzungszeit überprüft werden. Die zweite Anweisung stellt ein explizites casting auch `downcast` genannt dar. Ein `downcast` stellt die Konvertierung eines Objekts von einem Superklassentyp in einen Subklassentyp dar. Diese Konvertierung ist nicht immer möglich, da eine Subklasse eine Spezialisierung einer Superklasse ist. Die Konvertierung wird zur Laufzeit überprüft. Die Variable `parent1` ist vom Typ `Parent`, aber sie enthält zur Laufzeit eine Objektreferenz von der Subklasse `Child`. Aus diesem Grund kann der Typ ohne Laufzeitfehler konvertiert werden. Die vierte Anweisung aus dem Beispiel 2.1 schlägt fehl. Das casting führt zu einem Laufzeitfehler, da das Objekt vom Typ `Parent` ist und keine Informationen vom Typ `Child` enthält. Durch `downcasts` kann es zu Laufzeitfehlern kommen, oft ist auch ein anderes Vorgehen möglich. Ein `upcast` ist immer unter der Bedingung möglich, dass sich beide Typen in derselben Vererbungshierarchie befinden [Ora15b].

2.2 Softwareentwicklungsprinzipien

In den folgenden Abschnitten werden wichtige Softwareentwicklungsprinzipien, die in dieser Arbeit verwendet werden, vorgestellt und beschrieben. Einige dieser Prinzipien wurden durch [Mar12] definiert und sind als Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion (SOLID) Prinzipien zusammengefasst. Hauptsächlich finden diese Prinzipien im OOD Anwendung.

2.2.1 Don't Repeat Yourself

Das Don't Repeat Yourself (DRY)-Prinzip stammt aus der Softwareentwicklung und wurde von Hunt und Thomas definiert. DRY besagt, dass jedes Wissensstück eine einzelne, eindeutige und autoritative Repräsentation im System haben sollte [HT99].

2.2.2 Single Responsibility Principle

Single Responsibility Principle (SRP) ist ein OOD Konzept. Die Grundaussage von SRP ist dabei, dass eine Klasse nur eine Ursache zur Veränderung besitzen sollte [vgl. Mar03, S. 110]. Jede Verantwortlichkeit wird als eine Achse der Veränderung betrachtet. Wenn die Anforderungen sich ändern, werden diese Veränderungen auch die Verantwortlichkeiten der jeweiligen Klassen verändern. Nimmt eine Klasse mehr als eine Verantwortlichkeit an, dann besitzt diese Klasse mehr als einen Grund, um verändert zu werden. Dies sollte verhindert werden [vgl. Mar09, S. 138 ff.]. Wichtig ist dabei, um das System nicht unnötig zu verkomplizieren, SRP nur anzuwenden, wenn auch Grund zur Annahme besteht, dass Veränderungen erfolgen können [Mar03].

2.2.3

Open Close Principle

Der Grundgedanke des Open Close Principle (OCP) kann wie folgt zusammengefasst werden: Software Entitäten (Klassen, Funktionen etc.) sollten offen für Erweiterungen, aber geschlossen für Veränderungen sein [Mar96d]. Klassen, die mit diesem Prinzip übereinstimmen, haben zwei primäre Attribute [Mar96d]:

1. Sie sind offen für Erweiterungen.
   Das Verhalten der Klassen kann erweitert werden. Es ist vor allem notwendig, wenn sich Anforderungen ändern oder die Klassen in neuen Anwendungen eingesetzt werden sollen.
2. Sie sind verschlossen für Modifizierungen.
   Der Code einer solchen Klasse ist unberührt. Niemandem ist es erlaubt den Code der Klasse zu ändern.

Der Schlüssel dieses Konzepts ist die Abstraktion. In vielerlei Hinsicht ist dieses Prinzip das Herz des OOD [vgl. Mar96d, S. 13]. Programme die nach diesem Prinzip entwickelt wurden, sind robust und müssen bei Veränderungen nicht mehrfach geändert werden. Anders die Programme, die nicht nach diesem Prinzip entwickelt wurden. Dort hat eine Änderung meist eine Kaskade von Veränderungen zur Folge, denn OCP valide Programme werden verändert durch Hinzufügen von neuem Quellcode und nicht durch Modifizieren von altem [Mar96d].

2.2.4

Liskov Substitution Principle

Das Liskov Substitution Principle (LSP) wurde erstmalig von Liskov und Wing in [LW93] beschrieben. In diesem heißt es, dass Objekte eines Subtyps die an Stelle von Supertypobjekten verwendet werden, sich gleich verhalten sollten [vgl. LW93, S. 1]. Diese Anforderung wurde in [LW99] weiter eingeschränkt.

Ist `q(x)` eine beweisbare Eigenschaft für Objekte `x` eines Typs `T`, dann sollte `q(y)` genauso für Objekte `y` vom Typ `S` gelten. `S` stellt dabei ein Subtyp von `T` dar [vgl. LW99, S. 1]. D. h. das es möglich sein muss Funktionen oder Methoden denen Objekte von Basisklassen übergeben werden, Objekte von Subklassen ohne Veränderung bzw. Folgen zu übergeben [vgl. Mar96c, S. 2]. Invarianten bzw. Eigenschaften, die für diese Methoden und den Supertypen definierten wurden, müssen dabei auch für die Subtypen gelten.

LSP wird als eine Erweiterung des zuvor beschrieben OCP angesehen. Das bedeutet, dass eine Verletzung gegen das LSP eine Verletzung gegen das OCP darstellt. Die Bedeutsamkeit dieses Prinzips wird durch die Folgen einer Verletzung deutlich. Ein nicht beachten des LSP Prinzips kann zu undefiniertem Verhalten und Laufzeitfehlern führen.

2.2.5

Interface Segregation Principle

Das Interface Segregation Principle (ISP) behandelt die Nachteile von sogenannten `fat interfaces`. Klassen die `fat interfaces` besitzen sind Klassen deren Schnittstellen nicht einheitlich geschlossen sind. Die Schnittstelle kann in Methodengruppen für bestimmte Funktionalitäten unterteilt werden. Verschiedene Klassen bzw. Clients nutzen die einzelnen Gruppen von Funktionalitäten [vgl. Mar96b, S. 1 f.]. Diese `fat interfaces` führen zu einer Kopplung von Klassen. Klassen die normalerweise keine Verbindung zueinander haben [vgl. Mar96b, S. 13]. [Mar96b] definiert das ISP wie folgt: Clients sollten nicht dazu gezwungen werden von Schnittstellen abzuhängen, die sie nicht nutzen. Durch die Kopplung und Abhängigkeit von anderen Klassen bzw. des `fat interface` kann es zu einer Kaskade von Änderung bei Veränderungen kommen.

2.2.6

Dependency Inversion Principle

Dependency Inversion Principle (DIP) ist die strukturelle Implikation der Prinzipien OCP und LSP. Es wird dabei definiert, dass Abstraktionen nicht von Details, sondern Details von Abstraktionen abhängig sein sollen [vgl. Mar96a, S. 6]. Die Grundaussage dieses Prinzips ist somit, dass Klassen von Abstraktionen (abstrakte Klassen oder Schnittstellen) und nicht von konkreten Details (Implementierungen) abhängig sein sollen [vgl. Mar09, S. 150]. Dieses Prinzip bzw. Vorgehen ermöglicht es Schnittstellenimplementierungen auszutauschen, ohne dass die Klassen, die diese Schnittstelle verwenden, angepasst werden müssen. Bspw. kann eine Schnittstelle zum Speichern von Daten erstellt werden, welche von Klassen genutzt wird. Dabei sind die Klassen, die diese Schnittstelle nutzen, unabhängig von der Implementierung der Schnittstelle. Die Daten können als XML oder normaler Text gespeichert werden, ohne das die Klasse, die diese Schnittstelle verwendet, angepasst werden muss.

2.3

Typunterscheidungen in Programmiersprachen

In den objektorientierten Programmiersprachen C++, Java und C# existieren Operatoren zur Typenunterscheidung. In den folgenden Abschnitten werden die existierenden Operatoren für die jeweiligen Sprachen genannt und näher erläutert. Des Weiteren wird auf Probleme der Verwendung dieser Operatoren eingegangen.

2.3.1

Instanceof Operator

Der `instanceof` Operator ist ein Java spezifischer Operator zur Überprüfung des Objekttyps.

```
Object instanceof Class
```

Er erhält als erstes Argument ein Objekt oder einen Ausdruck und als zweites Argument eine Klasse oder einen Ausdruck. Als Ergebnis liefert er einen booleschen Wert, der `true` ist, wenn das Objekt bzw. der Ausdruck eine Instanz der Klasse ist [Ora08]. Ein Objekt, welches eine `null`-Referenz ist, ergibt immer `false`. Der Test von `instanceof` testet das Objekt auf die Hierarchie. Der Test für ein Objekt auf die Klasse `Object` ist immer wahr [Ull11].

2.3.2

Dynamic_cast Operator

Der `dynamic_cast` Operator ist ein C++ spezifischer Operator zur Typenumwandlung.

```
type t = dynamic_cast<type>(obj_ptr);
```

Er kann nur mit Zeigern, Referenzen auf Klassen oder mit `void` Zeigern verwendet werden. Er stellt sicher, dass der Zielzeiger als Ergebnis der Typkonvertierung auf ein valides fertiges Objekt zeigt. Hat das Objekt bzw. der Zeiger `obj_ptr` keinen vollständigen Typ, um zu Typ `Type` konvertiert zu werden, wird ein `null_ptr` zurückgegeben. Kein vollständiger Typ bedeutet, dass eine Superklasse A nicht zu einer Subklasse B konvertiert werden kann, da diese unvollständig in Bezug auf B ist. Denn B beinhaltet mehr Informationen als die Elternklasse A [14]. Dies ermöglicht (neben der Typkonvertierung) das Verwenden des Operators, ähnlich wie den `instanceof` Operator, in bedingten Anweisungen und somit typspezifische Entscheidungen zu treffen [14; 15a; 15f].

2.3.3

Typeid Operator

Der `typeid` Operator ist ein C++ spezifischer Operator zur Typenerkennung.

```
const type_infp& info = typeid(obj)
```

Er erlaubt es, den Typ eines Objekts zur Laufzeit zu erkennen. Das Ergebnis des Operators ist eine Referenz auf ein `type_info` Objekt (`const type_infp&`). Das Objekt repräsentiert entweder die Typ ID oder den Typ des Ausdrucks, abhängig davon, wie der Operator benutzt wurde [15b; 15g]. Er kann ähnlich wie der `dynamic_cast` Operator in bedingten Anweisungen verwendet werden.

Anders als der zuvor beschriebene `dynamic_cast` Operator erkennt der `typeid` Operator nur den Typ des Objekts, auf welches der Zeiger zeigt, unter der Bedingung, dass die Klassen Polymorphie unterstützen.

Der Operator erkennt nicht die weiteren Basisklassen des Objekts. Beinhalten die Superklassen keine virtuellen Methoden und unterstützen somit keine Polymorphie, so wird nur die Klasse des Zeigers erkannt [15d].

2.3.4

Is Operator

Der `is` Operator ist ein C# spezifischer Operator zur Typenunterscheidung.

```
bool isType = obj is Class
```

Er ermöglicht das Überprüfen, ob ein Objekt mit einem gegebenen Typ kompatibel ist, ähnlich wie der `instanceof` in Java. Das obige Beispiel prüft, ob das Objekt `obj` eine Instanz vom Typ `Class` oder einem abgeleiteten Typ dieser Klasse ist. Das Ergebnis dieser Operation ist ein boolescher Wert. Der Wert ist `true`, wenn der Ausdruck nicht `null` ist und das Konvertieren des Objekts zum gegebenen Typ, ohne das eine *Exception* ausgelöst wird, erfolgreich durchgeführt werden kann [15h].

2.3.5

Typcode

Typcode (engl. type codes) sind spezielle Werte, die auf besondere Typen einer Instanz hinweisen. Es existieren verschiedene Möglichkeiten für Typcodes. Sie sind oft durch Aufzählungen, bspw. durch konstante Zahlen, aber auch durch Zeichenketten dargestellt [vgl. Fow98, S. 138].

2.3.6

Probleme Typisierung

Bedingte Anweisungen mit Typüberprüfungen stellen ein Problem dar. Sie erschweren die Wart- und Erweiterbarkeit. Kommen bedingte Anweisungen und Typüberprüfungen mehr als einmal vor, führen Veränderungen bzw. Erweiterungen zu einer Kaskade von Veränderungen. In der Regel zeigt die Kontrolllogik dieser Art tendenziell ein Designproblem an und kann oft anders gelöst werden [Ull11]. Es existiert dazu folgende Restrukturierungsmethode: `Prefer Polymorphism to If/Else or Switch/Case` [vgl. Mar09, S. 299]. Auf diese Heuristik und deren Restrukturierung wird im nächsten Abschnitt 2.4 näher eingegangen.

2.4 Restrukturierung

Softwarerestrukturierung ist eine Form der Codemodifizierung. Sie stellt eine durchaus effektive Methode dar, um Softwarestrukturen in Hinsicht auf Wart- und Erweiterbarkeit zu verbessern [vgl. Bro+98, S. 68]. Die Restrukturierung liefert eine Technik, die das Umstrukturieren effizienter und kontrollierter gestaltet. Restrukturierungen beschreiben Veränderungen, die Software verständlicher machen [vgl. Fow98, S. 47]. Eine Restrukturierung verändert das Verhalten eines Programms nicht. Bei mehrmaliger Ausführung des Programms vor und nach der Restrukturierung, mit den gleichen Eingaben, ist das Ergebnis identisch. Dadurch das kein Verhalten verändert wird, kann es die Weiterentwicklung eines Programms unterstützen. In den folgenden Teilabschnitten werden Restrukturierungsmethoden vorgestellt, die im Laufe dieser Arbeit verwendet werden.

Replace conditional with Polymorphism

Sind bedingte Anweisungen innerhalb des Codes vorhanden, die abhängig vom Typ eines Objekts anderes Verhalten ausführen, wird diese Restrukturierungsmethode angewendet. Jeder Zweig dieser bedingten Anweisung wird in eine überschreibende Methode einer Subklasse verschoben. Die vorherige Methode wird nach der Restrukturierung als abstrakt deklariert [vgl. Fow98, S. 205].

Replace type code with sub classes

Besitzen Klassen unveränderbaren Typcode, der deren Verhalten beeinflusst, wird diese Restrukturierungsmethode angewendet. Der Typcode wird durch Subklassen ersetzt [vgl. Fow98, S. 181].

Replace type code with state strategy

Besitzen Klassen sogenannten Typcode, der deren Verhalten beeinflusst, aber es können keine neuen Subklassen eingeführt werden, kann die zuvor beschriebe Restrukturierungsmethode nicht angewendet werden. Es wird daher auf diese Restrukturierungsmethode bzw. auf das `State` oder `Strategy` Pattern [vgl. Gam+94, S. 338 ff.] zurückgegriffen [vgl. Fow98, S. 184 ff.].

Extract Method

Diese Restrukturierungsmethode findet Anwendung, wenn ein Codefragment, welches zusammengefasst werden kann, existiert. Dieses Codefragment wird in eine eigene Methode verschoben, wobei die Benennung den Zweck der Methode beschreibt [vgl. Fow98, S. 89 f.].

Extract Class

Wenn in einer Klasse Funktionalität die auf zwei Klassen aufgeteilt werden kann existiert, findet diese Restrukturierungsmethode Anwendung. Es wird eine neue Klasse erstellt und die relevanten Elemente bzw. Teile von der Alten in die neue Klasse verschoben [vgl. Fow98, S. 122 f.].

Extract Superclass

Existieren zwei Klassen mit vergleichbaren Funktionalitäten, wird diese Restrukturierungsmethode angewendet. Es wird eine Superklasse für diese Klassen erstellt und die gemeinsamen Funktionalitäten in diese verschoben [vgl. Fow98, S. 272 f.].

Extract Interface

Diese Restrukturierung findet Anwendung, wenn zwei Klassen Gemeinsamkeiten in ihren Schnittstellen aufweisen. Die Teilmenge wird extrahiert und in einer neuen Schnittstelle deklariert [vgl. Fow98, S. 277 f.].

Move Method/Field

Diese Restrukturierungsmethoden werden angewendet, wenn ein Element, eine Methode oder ein Attribut, mehr in einer anderen Klasse verwendet wird, als in der Klasse in der es definiert ist. Handelt es sich dabei um ein Attribut wird ein neues Attribut in der anderen Klasse erstellt und jegliche Benutzung angepasst [vgl. Fow98, S. 119 f.]. Ist das Element eine Methode wird eine neue Methode in der anderen Klasse erzeugt. Diese erhält den gleichen Methodenkörper. Die alte Methode wird in eine Delegation geändert oder gänzlich entfernt [vgl. Fow98, S. 115 f.].

Pull Up Method/Field

Besitzen Subklassen gleiche Attribute oder Methoden, die die gleichen Resultate liefern, werden diese Restrukturierungsmethoden angewendet. Es werden die gleichen Elemente in die Superklasse verschoben [vgl. Fow98, S. 259 f.].

Replace Delegation with Inheritance

Delegiert eine Klasse an die komplette Schnittstelle einer anderen Klasse, findet diese Restrukturierungsmethode Anwendung. Die delegierende Klasse erbt von der Klasse, an die zuvor delegiert wurde [vgl. Fow98, S. 289 f.].

Remove Middle Man

Verwendet eine Klasse zu viele einfache Delegationen, wird diese Restrukturierungsmethode angewendet. Der Client, der diese Klasse verwendet, verwendet nun direkt die delegierten Methoden [vgl. Fow98, S. 130 f.].

Collapse Hierarchy

Besitzt eine Subklasse keine weiteren Funktionalitäten als die Superklasse, d. h. sie unterscheiden sich nicht, kann diese Restrukturierungsmethode angewendet werden. Die Klassen werden zu einer Klasse zusammengeführt [vgl. Fow98, S. 279 f.].

2.5 Werkzeuge

2.5.1 Como

Como bezeichnet das Analysewerkzeug von B. M., welches in der Automotive Abteilung, innerhalb der durchgeführten CQAs, verwendet wird. Es ist in C# geschrieben und verwendet verschiedene Werkzeuge, um die Analyse so genau wie möglich zu gestalten. Como ermöglicht das darstellen von doppeltem Code, Anti-Pattern und einigen weiteren Problemen. Diese gefundenen Probleme können mithilfe des Werkzeuges grafisch dargestellt werden, um den Kunden die Problematik zu verdeutlichen. Das in dieser Arbeit zu erstellende Werkzeug sollte in Como integrierbar sein, weshalb die Implementierungssprache C# ist.

2.5.2

Sotograph

Sotograph ist eine Erweiterung des statischen Analysewerkzeugs Sotoarc, welches von der Firma hello2morrow GmbH entwickelt wird. Die hello2morrow GmbH wurde als unabhängiger Softwareanbieter im Jahre 2005 gegründet und besitzt zurzeit mehr als 200 Kunden in Europa, Amerika, Asien und Australien [hel15a]. Sotoarc ermöglicht das Modellieren der Architektur und des Softwaresystems. Mithilfe von Sotograph kann C++, C# und Java Code analysiert werden. Ein Softwaresystem wird initial analysiert und die Daten werden in einer Datenbank gespeichert [hel15b]. Diese Datenbank wird von B. M. in Como verwendet, um Probleme und Metriken anzuzeigen. Die Sotograph Datenbank wird für das zu erstellende Werkzeug verwendet, sodass der Code nicht erneut statisch analysiert werden muss und die Integration in Como erleichtert wird. Die Sotograph-Datenbank, welche zur Evaluation verwendet wird, wurde mithilfe der Sotograph Version 4.2 und den, in Abschnitt 2.6, beschriebenen Testprojekten erzeugt.

2.5.3

CCFinder

Der CCfinder wurde von National Institute of Advanced Industrial Science and Technology (AIST) bis zum Jahre 2010 entwickelt. AIST ist eine der größten Forschungsorganisationen in Japan. Sie sind fokussiert auf die Erstellung und praktische Realisierung von Technologien, die für die japanische Industrie und Gesellschaft von Nutzen sind [AT15a]. Der CCfinder ist ein Codeklon-Detektor. Er erkennt Codeklone in Quelldateien, die in Java, C++, C# und einige weiteren Sprachen geschrieben sind [AT15b]. CCfinder wird bereits von B. M. verwendet, um Codeklone zu erkennen und in Como darzustellen. Um die Kompatibilität mit Como zu gewährleisten, wird CCfinder zum Erkennen von Codeklonen innerhalb dieser Arbeit verwendet. Es wird dabei mit der Version 10.2.7.4 gearbeitet. Die Erkennung von doppeltem Code spielt in der Erkennung von Vererbungsproblemindikatoren eine wichtige Rolle, siehe dazu Kapitel 3.

2.5.4

Klocwork

Klocwork ist ein Rogue Wave Softwareprodukt und stellt ein statisches Codeanalysewerkzeug dar, welches mithilfe von sogenannten Checkern angepasst werden kann. Rogue Wave liefert Softwareentwicklungswerkzeuge für missionskritische Anwendungen [Klo15a]. Como nutzt Klocwork, um Defekte zu erkennen und darzustellen. Die gefundenen Defekte sind abhängig von den jeweilig benutzten Checkern. Como nutzt einige built-in Checker von Klocwork, aber auch benutzerdefinierte Checker, um die Nutzung von Anti-Pattern zu erkennen. Für die Erkennung einiger Indikatoren werden eigene Checker erzeugt. Innerhalb dieser Arbeit wird die Klocwork Version 10.2 verwendet.

2.5.5 Graphviz

Graphviz stellt eine Open Source Graph-Visualisierungssoftware dar. Eine Graph-Visualisierung ist ein Weg der Darstellung von strukturellen Informationen. Die abstrakten Graphen und Netzwerke werden in Form von Diagrammen dargestellt. Die Darstellung spielt eine wichtige Rolle in der Bioinformatik, im Software Engineering, im Web und Datenbank Design und anderen Bereichen [Gra15b].

Den Graphviz Layout-Werkzeugen werden Beschreibungen von Graphen übergeben. Graphbeschreibungen werden in der sogenannten DOT-Language verfasst [Gra15a]. In dieser Arbeit werden Dateien, die diese Beschreibung enthalten, als dot-Dateien bezeichnet. Die Werkzeuge erzeugen aus der übergebenen Beschreibung Diagramme in nützlichen Formaten, wie PNG oder PDF. Graphviz besitzt einige nützliche Features für die Erstellungen von Diagrammen. Es können optional Farben, Fonts, tabellarische Layouts usw. genutzt werden. Das Werkzeug dot wird zur hierarchischen Darstellung von direkten Graphen genutzt. Dies ist das vorgegebene Werkzeug für die Erstellung, wenn die Kanten eine Richtung besitzen. [Gra15b] Innerhalb dieser Arbeit wird diese Software mit der Version 2.38.0, für die Erzeugung von Ausgaben und Darstellung von Vererbungsstrukturen, verwendet.

2.6 Testprojekte

Zur Evaluation, der im Abschnitt 3 entwickelten Methode sowie zum Testen des zu erstellenden Werkzeugs, werden die Projekte XBMC und HtmlUnit verwendet. Bei diesen Projekten handelt es sich um Open Source Projekte. In der folgenden Tabelle 2.1 werden die genannten Projekte und deren Umfang aufgelistet. Das XBMC-Projekt ist am umfangreichsten und besitzt 2204 Klassen, 1415 Dateien und 582142 Codezeilen. Das HtmlUnit-Projekt dagegen besitzt 1048 Klassen, 859 Dateien und 133038 Codezeilen. Diese Daten wurden mithilfe von Sotograph ermittelt.

Projekt	**LOC**	**Klassen**	**Dateien**
XBMC	582142	2204	1415
HtmlUnit	133038	1048	859

Tabelle 2.1: Projektgrößen

XBMC, heute als Kodi bekannt, ist eine freie und Open Source Media Center Lösung. XBMC ist plattformunabhängig und erlaubt Nutzern das Abspielen von Videos, Musik, Podcasts und anderen digitalen Medien von lokalen oder Netzwerkspeichermedien und vom Internet [Kod15]. Die verwendete Version bzw. der verwendete Datensatz, wurde bereits von der Firma B. M. zu Testzwecken verwendet. Die git-commit-Version des Projekts lautet `6d20d5cd3d91a337ebc778616e12fd7b407a885d` und ist vom 04.12.2013.

Bei dem HtmlUnit-Projekt handelt es sich um einen Webbrowser ohne grafische Nutzeroberfläche. Er modelliert Hypertext Markup Language (HTML)-Dokumente und liefert Schnittstellen, welche das Aufrufen von Seiten, Befüllen von HTML-Forms, das Klicken von Links usw. ermöglicht. Das HtmlUnit-Projekt wird typischerweise zu Testzwecken verwendet, wobei es beabsichtigt ist, dieses Projekt innerhalb anderer Testumgebungen wie JUnit etc. zu nutzen [Inc15]. Die in der Arbeit zu Test- und Evaluationszwecken verwendete Version lautet 2.16.

Indikatoren für Vererbungsprobleme 3

In der Abbildung 3.1 ist die Methode zur Analyse, Bewertung und Optimierung von Vererbungsstrukturen in objektorientierten Softwaresystemen dargestellt. Diese Methode beinhaltet eine statische Quellcodeanalyse, die als Basis zur Erkennung der, in diesem Kapitel beschriebenen, Indikatoren genutzt wird. Indikatoren sind dabei Hinweise auf bestimmte Problematiken in Vererbungsstrukturen. Nach der Ausführung der statischen Quellcodeanalyse wird die Indikatorenerkennung ausgeführt. Die Indikatoren können mithilfe des in dieser Arbeit entwickelten Werkzeugs, siehe Kapitel 5, erkannt werden. Nach der Indikatorenerkennung müssen die Methoden zur Analyse und Bewertung auf die einzelnen erkannten Indikatoren angewendet werden. Es existiert spezifisch für jeden Indikator eine Methode die das Auswählen einer geeigneten Restrukturierungsmethode, mithilfe der Analyse und Bewertung des erkannten Indikators, ermöglicht. Die einzelnen Indikatoren werden nach einem ausgearbeiteten Schema gewichtet. Die Gewichtung ermöglicht die Auswahl der Indikatoren, welche nach der Erkennung als erstes behoben werden müssen. Für die erkannten Indikatoren können die damit verbundenen Problematiken, mithilfe der ausgewählten Restrukturierungsmethode, aufgelöst und die Vererbungsstruktur optimiert werden.

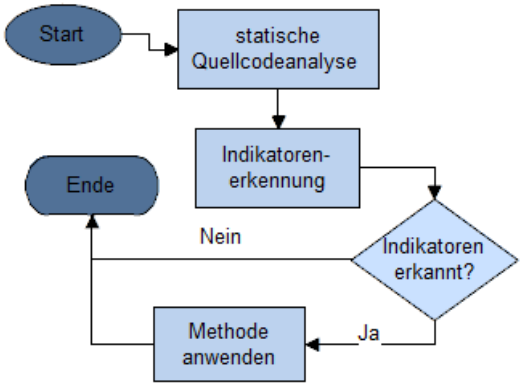


Abbildung 3.1: Methode

In den nächsten Abschnitten werden die besagten Indikatoren definiert. Es wird jeder Indikatoren kurz beschrieben und auf deren Erkennung hingewiesen. Die mit den Indikator verbundenen Problematiken werden erläutert und es wird eine Methode zur Restrukturierungsmethodenfindung vorgestellt. Im Anschluss werden die definierten Indikatoren gewichtet, sodass deren Problematiken noch einmal aufgezeigt und die Dringlichkeit der Behebung einiger Indikatoren hervorgehoben werden können.

3.1 Superklasse kennt Subklasse

Der Indikator kennzeichnet den Fall, dass eine Superklasse innerhalb ihrer Definition/Deklaration eine Referenz bzw. ein Objekt einer Subklasse verwendet. Dieser Indikator wird mit `Superklasse kennt Subklasse` (SKS) bezeichnet.

3.1.1

Problematik

Die Superklasse ist die generalisierte Version und die Subklasse eine speziellere Version. Der Indikator SKS kennzeichnet ein grundlegendes Vererbungsproblem. Kennt die Superklasse eine Subklasse, hängt diese von Details ab. Dies stellt eine Verletzung des DIP dar und erhöht die Komplexität des Systems. Es kann zu unerwartetem Verhalten innerhalb eines Subsystems führen. Durch Hinzufügen einer weiteren Subklasse muss höchstwahrscheinlich die Superklasse angepasst werden. Daraus folgt eine Kaskade von Änderungen. Das beschriebene Verhalten stellt einen Verstoß gegen das OCP dar, die Wart- und Erweiterbarkeit wird dadurch vermindert. Die Klassen, die diesem Indikator entsprechen, brechen die Verkapselung und den Polymorphismus, welche Grundprinzipien der Objektorientierung darstellen [vgl. Gar06, S. 320].

3.1.2

Methode

In der Abbildung 3.2 ist das Verfahren zur Analyse und Bewertung eines erkannten SKS-Indikators dargestellt. Mithilfe dieses Verfahrens kann eine geeignete Restrukturierungsmethode ausgewählt werden, die die Optimierung der vorhandenen Vererbungsstruktur ermöglicht.

Ein wichtiger Aspekt bei der Bewertung der geeigneten Restrukturierungsmethode ist, ob die Superklasse anhand von Parametern verschiedene Subklassenobjekte erstellt und als Rückgabewert liefert. Ist dies der Fall handelt es sich um eine Art des `Parameterized Factory Method` Pattern, welches nicht korrekt angewendet wurde. Diese Methode bzw. Funktionalität muss mithilfe der Restrukturierungsmethode `Extract Method` ausgelagert und das `Factory Pattern` muss dahingehend noch einmal angepasst werden. Ist dieses Muster nicht vorhanden, werden jedoch anhand von Parametern oder des derzeitigen Klassenobjekts verschiedene Funktionalitäten ausgeführt, handelt es sich dabei um bedingte Anweisungen mit Typüberprüfung. Diese beschreiben einen parallelen Indikator, siehe Abschnitt 3.2. Zur Lösung sollte der von den objektorientierten Sprachen unterstützte Polymorphismus angewendet werden. D. h. die Subklassen überschreiben die Methode, in der der SKS-Indikator erkannt wurde. In allen anderen Fällen muss die Restrukturierungsmethode `Extract Class` angewendet werden. Mithilfe dieser Restrukturierungsmethode nutzt die neu extrahierte Klasse die Funktionalitäten der Subklassen, die zuvor von der Superklasse genutzt wurden. Die Verwendung der Subklassen wird vollends aus der Superklasse entfernt. Innerhalb der Anwendung muss die Verwendung der Superklasse, mit der neu erstellten Klasse, ausgetauscht werden.

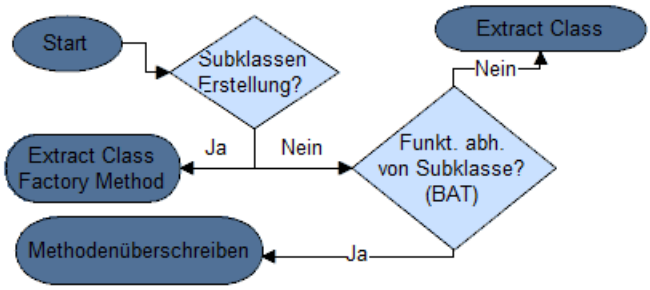


Abbildung 3.2: SKS-Methode

Nach Anwendung der beschriebenen und in der Abbildung 3.2 dargestellten Methode kann der erkannte SKS Indikator analysiert, bewertet und eine geeignete Restrukturierungsmethode ausgewählt werden. Mithilfe dieser Restrukturierungsmethode kann der SKS Indikator aufgelöst und die vorhandene Vererbungsstruktur optimiert werden.

3.2 Bedingte Anweisungen mit Typüberprüfungen

Dieser Indikator zeichnet sich durch die Überprüfung eines Objekttyps innerhalb einer bedingten Anweisung aus. Abhängig vom Typ des Objektes wird eine Entscheidung getroffen bzw. eine Aktion ausgeführt. Der Indikator wird aus diesem Grund als `Bedingte Anweisungen mit Typüberprüfungen` (BAT) bezeichnet. Im folgenden Beispiel wird eine Typüberprüfung mit dem `instanceof`-Operator dargestellt. Zur Erklärung des Operators siehe Abschnitt 2.3.1.

Beispiel 3.1: BAT-Indikator

```
public static void moveAnimal(Animal animal) {
  if (animal instanceof Bird)
    //move bird
  if (animal instanceof Cat)
    //move cat
}
```

In dem Beispiel 3.1 ist eine Methode dargestellt, die abhängig vom Objekttyp des Parameters eine andere Aktion ausführt. Der Parameter besitzt den Typ `Animal`, wobei dies die Superklasse beschreibt. Die Klassen `Bird` und `Cat` erben von dieser Klasse und können somit auch dieser Methode übergeben werden. Ist das übergebene Objekt eine Instanz der Klasse `Cat` werden Funktionalitäten der `Cat`-Klasse zur Bewegung ausgeführt. Bei einer Instanz der Klasse `Bird` werden die Funktionalitäten der `Bird`-Klasse ausgeführt.

3.2.1 Problematik

Im Abschnitt 2.3.6 wurden bereits Problematiken des Indikators beschrieben. Der BAT Indikator verletzt unter anderem das SRP, OCP und DIP. Laut [Mar09] verletzen bedingte Anweisungen per Definition durch ihre Eigenschaften das SRP. Ein Beispiel dafür wären `switch-case`-Anweisungen. Diese definieren **N** verschiedene Verzweigungen. D. h. es existieren **N** Abängigkeiten und **N** Gründe, diesen Code zu ändern. Somit wird das SRP verletzt [vgl. Mar09, S. 37 f.]. Durch die Typüberprüfung ist es nicht möglich, weitere Typen ohne Änderungen hinzuzufügen.

In dem Beispiel 3.1 ist zu erkennen, dass bei jeder weiteren Subklasse, der Superklasse `Animal`, diese Methode angepasst werden muss. Werden die Typen der Objekte andernorts überprüft, folgt aus dem Hinzufügen von neuen Subklassen eine Kaskade von Änderungen an existierenden Code. Dies stellt einen Verstoß gegen das OCP dar.

Die bedingten Anweisungen führen zu einer erhöhten Komplexität des Systems. Werden Typunterscheidungen in einer Klasse getätigt, folgt daraus, dass höchstwahrscheinlich in anderen Klassen diese Unterscheidung erneut vorkommt. Durch das Fehlen von Polymorphismus ist die Unterscheidung notwendig, um den Programmablauf zu definieren.

Dies hat doppelten Code zur Folge. Wird ein weiterer Fall der bedingten Anweisung hinzugefügt, müssen alle Anweisungen gefunden und geändert werden [vgl. Fow98, S. 82]. Die bedingten Anweisungen mit Typüberprüfungen bringen Abhängigkeiten mit sich, d. h. die Methode, die diese Anweisung enthält, ist von Details abhängig und nicht von Abstraktionen. Dies verletzt das DIP [vgl. Mar96a, S. 4].

Die Typüberprüfungen sind vor allem durch die Benutzung von sprachspezifischen Operatoren wie: `instanceof`, `dynamic_cast` und `is` erkennbar. Der BAT Indikator ist jedoch auch bei benutzerdefinierten Typüberprüfungen z. B. Enums oder Strings, genannt Typcode, vorhanden. Zur Erklärung der Operatoren und Typcode siehe Abschnitt 2.3.

3.2.2

Methode

Zur Analyse, Bewertung und Auswahl der Restrukturierungsmethode kann das, in der Abbildung 3.3, dargestellte Verfahren angewendet werden. Ist parallel zum BAT-Indikator der SKS-Indikator vorhanden muss die Methode des SKS-Indikators angewendet werden. Anderenfalls wird der BAT-Indikator mit den nachfolgenden Kriterien analysiert und bewertet, sodass eine geeignete Restrukturierungsmethode angewendet werden kann.

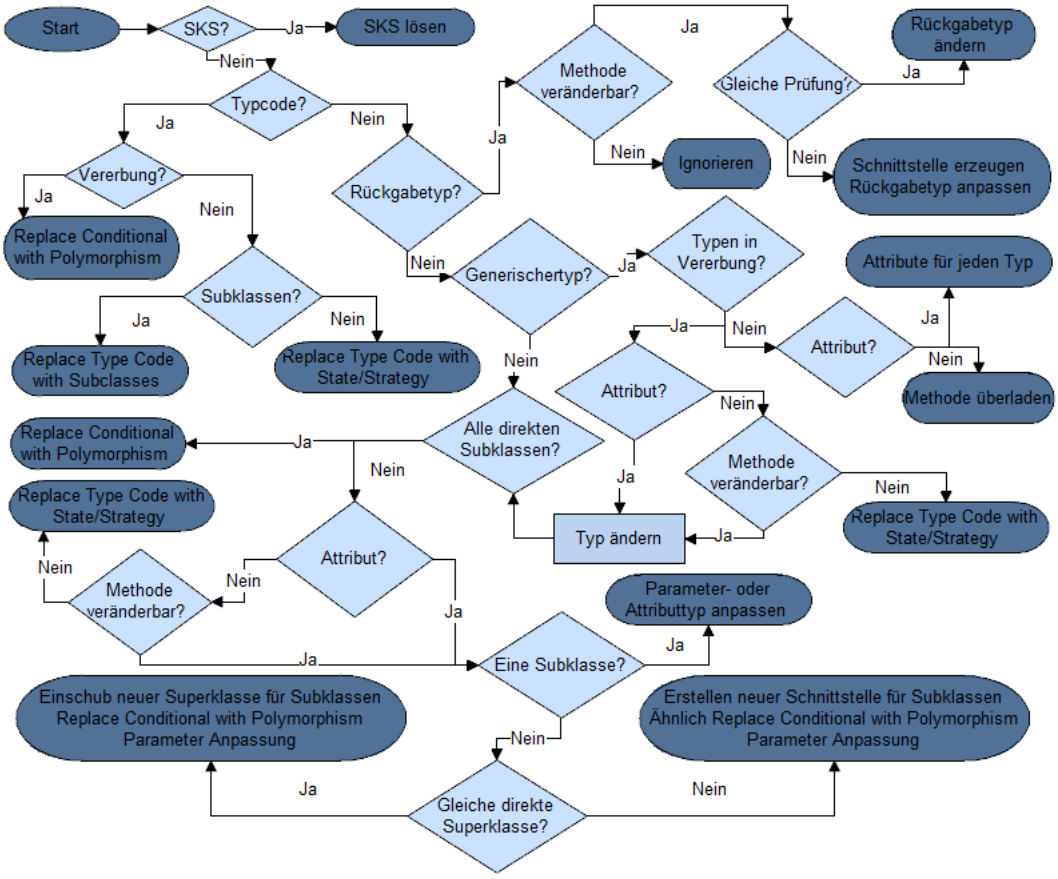


Abbildung 3.3: BAT-Methode

Bei einem nicht vorhandenen SKS-Indikator muss die Art des BAT-Indikators analysiert werden. Handelt es sich bei dem erkannten Indikator um Typcode muss geprüft werden, ob bereits eine Vererbungsstruktur für diese geprüften Typen existiert. Ist dies der Fall muss die Restrukturierungsmethode `Replace Conditional with Polymorphism` angewendet werden. Besteht keine Vererbungsstruktur muss als nächstes geprüft werden, ob das Erstellen von Subklassen möglich ist. Ist die Vererbungsstruktur erweiterbar, muss die Restrukturierungsmethode `Replace Conditional with Subclasses` angewendet werden. Anderenfalls muss die Restrukturierungsmethode `Replace type code with State/Strategy` eingesetzt werden.

Handelt es sich bei dem erkannten Indikator um kein Typcode, sondern um die Verwendung eines Operators zur Typüberprüfung (bspw. `instanceof`), muss dieser Indikator weiter analysiert und bewertet werden. Bei der weiteren Bewertung spielt das Objekt, welches geprüft wird, eine entscheidende Rolle. Findet die Überprüfung eines Methodenrückgabewerts statt, müssen die folgenden Fälle betrachtet werden: Veränderbarkeit der Methode, die den Rückgabewert liefert und wiederholte Prüfung des Rückgabewerts auf den gleichen Typ. Ist die Methode nicht veränderbar, da bspw. eine Methode einer Sprachschnittstelle verwendet wird, kann dieser Indikator nicht aufgelöst werden. Ist der Rückgabetyp veränderbar und das zurückgegebene Objekt wird immer auf den gleichen Typ geprüft, kann der Rückgabetyp auf den überprüften Typ angepasst werden. Bei nicht gleicher Typüberprüfung ist das ein Hinweis darauf, dass die Methode selbst restrukturiert werden sollte. Um diesen BAT Indikator aufzulösen, muss eine Schnittstelle erzeugt werden. Diese Schnittstelle vereint die Methodenrückgabetypen und deren Funktionalitäten. Der Methodenrückgabetyp wird auf diese Schnittstelle angepasst. Die Klassen bzw. Typen implementieren die neue Schnittstelle mit den deklarierten Funktionalitäten, die nach Aufruf der Methode und Typüberprüfung ausgeführt wurden.

Bei der Typüberprüfung eines Parameters oder Attributs ergeben sich andere Restrukturierungsmethoden. Es muss analysiert werden, ob der Parameter oder das Attribut, welches überprüft wird, ein generischen Typ besitzt (bpsw. `Object` in Java oder `void*` C++). Ist dies nicht der Fall, wird ein Superklassentyp verwendet. Der BAT Indikator weist in diesem Fall auf eine Subklassenprüfung hin. Bei der Überprüfung aller vorhandenen Subklassen muss die Restrukturierungsmethode `Replace Conditional with Polymorphism` eingesetzt werden. Wird nur ein Teil der Subklassen geprüft, muss ermittelt werden, ob es sich bei den geprüften Objekten um Parameter oder Attribute handelt. Handelt es sich um einen Parameter, ist es notwendig die Methode auf Veränderbarkeit zu analysieren. Ist die Methode nicht veränderbar, muss die Restrukturierungsmethode `Replace Type Code with State/Strategy` angewendet werden. Bei einer veränderbaren Methode oder einem Attribut, welches geprüft wird, muss analysiert werden, ob nur eine Subklasse geprüft wird. Ist dies der Fall, kann der Typ des geprüften Objekts in den geprüften Typ geändert werden. Eine Überprüfung mehrerer Subklassen, die die gleiche direkte Superklasse besitzen, kann durch folgendes Vorgehen aufgelöst werden: es wird eine Superklasse für diese Subklassen erstellt, die neue Superklasse erbt von der vorherigen Superklasse. Der Typ des geprüften Objekts wird in den Klassentyp der neuen Superklasse geändert, des Weiteren wird die Restrukturierungsmethode `Replace conditional with Polymorphism` angewendet. Besitzen die Subklassen nicht die gleiche direkte Superklasse, sind sie in der Vererbungsstruktur verstreut. Es wird für diese Klassen, anstatt einer Superklasse, eine neue Schnittstelle erzeugt. Die weiteren Schritte gleichen sich mit der vorherigen Methodik für Subklassen mit gleicher direkter Superklasse.

Bei einem generischen Parameter- oder Attributtyp müssen weitere Kriterien zur Lösung bewertet werden. Es muss analysiert werden, ob die überprüften Klassen innerhalb der gleichen Vererbungsstruktur existieren. Beschreiben die geprüften Klassen keine Vererbungsbeziehung, muss bei einem überprüften Attribut für jede geprüfte Klasse ein neues Attribut angelegt werden. Für jedes dieser Attribute wird eine eigene Methode erzeugt, die die vorherigen Funktionalitäten des bedingten Anweisungszweiges für diesen Typ beinhaltet. Handelte es sich um eine Überprüfung von Parametern, muss für jeden Typ eine neue Methode, mit dem dazugehörigen Parametertyp, erzeugt werden. D. h. die Methode wird für diese Parametertypen überladen. Existieren die geprüften Klassen innerhalb der gleichen Vererbungsstruktur, muss das geprüfte Objekt analysiert werden. Handelt es sich dabei um ein Attribut, muss der Typ des Attributs in eine gemeinsame Superklasse oder Schnittstelle der geprüften Klassen geändert werden. Danach wird verfahren als wenn der Attributtyp nicht generischer Natur war.

Gleiches gilt für einen Parameter, wenn die Methode, in welcher der BAT-Indiaktor erkannt wurde, veränderbar ist. Der Parametertyp kann gleichermaßen, wie dem Attributtyp, angepasst werden und die Verfahrensweise folgt dem, als wenn der Parametertyp nicht generischer Natur war. Ist die Methode nicht veränderbar, muss die Restrukturierungsmethode `Replace Type Code with State/Strategy` angewendet werden.

Mit dieser beschriebenen und in der Abbildung 3.3 dargestellten Methode zur Analyse und Bewertung eines BAT-Indikators, ist es möglich eine geeignete Restrukturierungsmethode auszuwählen. Die ausgewählte Restrukturierungsmethode ermöglicht die Auflösung des BAT-Indikators und die Optimierung der Vererbungsstruktur.

3.3 Klassen mit identischer Funktionalität

Der Indikator beschreibt Attribute und/oder Methoden, die in verschiedenen Klassen mehrmals vorkommen. Die Methoden besitzen identische Logik. Der Indikator wird grundsätzlich durch doppelten Quellcode erkannt, was laut [Fow98] einen Code Smell darstellt. Dieser Code Smell wird mit `Duplicated Code` bezeichnet. Da der Indikator sich auf Codeklone in verschiedenen Klassen beschränkt, wird dieser mit `Klassen mit Identischer Funktionalität` (KIF) bezeichnet. Die Methodenköpfe in den jeweiligen Klassen müssen sich nicht gleichen. In der Abbildung 3.4 sind zwei Klassen dargestellt, die diesem Indikator entsprechen. `ClassA` und `ClassB` besitzen jeweils mehrere Funktionalitäten. Es wird jedoch jeweils nur eine Methode der Übersichtlichkeit halber dargestellt. In den Methoden `methodA` und `otherMethod` wurde doppelter Quellcode erkannt. Die Methoden besitzen aber nicht unbedingt den gleichen Namen. Es ist außerdem möglich, dass die Klassen gleiche Attribute besitzen, die in diesen Methoden benutzt werden.

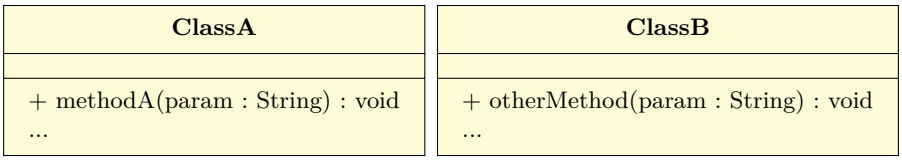


Abbildung 3.4: KIF-Indikator

3.3.1 Problematik

Der KIF-Indikator verletzt grundsätzlich kein SOLID Prinzip, dennoch weist dieser auf einige Probleme hin. Doppelter Quellcode bzw. identische Funktionalitäten in unterschiedlichen Klassen sollte laut [Fow98] auf jeden Fall restrukturiert werden [vgl. Fow98, S. 63]. Des Weiteren stellt diese Problematik eine Verletzung gegen das DRY-Prinzip dar, siehe zur Erklärung Abschnitt 2.2.1. Bei Änderungen muss nicht nur eine Klasse, sondern mehrere modifiziert werden. Veränderungen haben kaskadierende Änderungen zufolge. Dies schränkt die Wart- und Erweiterbarkeit ein. Der KIF-Indikator weist auf eine fehlende Abstraktion hin, dies ist durch das Fehlen einer Vererbung bzw. Wiederverwendung erkennbar.

3.3.2

Methode

Nach der Erkennung wird der Indikator und deren Umfeld analysiert und bewertet, sodass eine geeignete Restrukturierungsmethode angewendet werden kann. Zur Analyse, Bewertung und Auswahl der Restrukturierungsmethode wird das, in der Abbildung 3.5, dargestellte Verfahren angewendet.

Als erstes muss geprüft werden, ob ganze Klassen mit dem KIF-Indikator gekennzeichnet sind. Ist dies der Fall, muss geprüft werden, ob diese Klassen sich nur in Konstanten unterscheiden. Besitzen die gekennzeichneten Klassen keine Konstanten und somit keine Unterschiede, können diese Klassen bis auf eine entfernt werden. Bei vorhandenen Konstanten werden die Klassen zusammengeführt und für die Konstanten ein Attribut angelegt. Die Benutzung der Klassen muss auf die verbleibende Klasse angepasst werden.

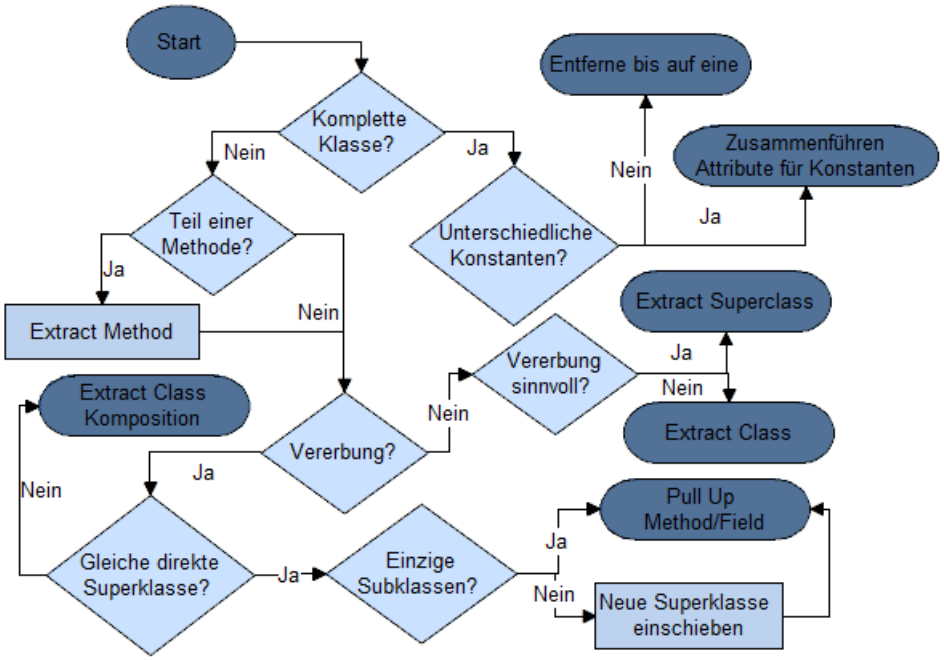


Abbildung 3.5: KIF-Methode

Ist nur ein Teil der Klasse als Klon gekennzeichnet, müssen weitere Kriterien zur Auswahl der geeigneten Restrukturierung, bewertet werden. Ist nur ein Teil einer Methode gekennzeichnet, muss für das weitere Vorgehen die Restrukturierungsmethode `Extract Method` angewendet werden [vgl. Fow98, S. 63]. Nachfolgend muss analysiert werden, ob die gekennzeichneten Klassen innerhalb der gleichen Vererbungsstruktur existieren. Ist dies nicht der Fall, muss die Bewertung einer möglichen Vererbung folgen. Ist eine Vererbung sinnvoll, muss die Restrukturierungsmethode `Extract Superclass` angewendet werden. Anderenfalls sollte die Restrukturierungsmethode `Extract Class` Anwendung finden. Die neu erstellte Klasse wird von den gekennzeichneten Klassen via Komposition eingebunden [vgl. Fow98, S. 64]. Ist bereits eine Vererbung vorhanden muss analysiert werden, ob diese Subklassen die gleiche direkte Superklasse besitzen. Ist dies der Fall muss bewertet werden, ob diese Klassen die einzigen Subklassen der Superklasse darstellen. Bei einem wahren Ergebnis muss die Restrukturierungsmethode `Pull Up Method` bzw. `Pull Up Field` angewendet werden. Existieren mehr Subklassen, als mit dem KIF-Indikator gekennzeichnet wurden, muss eine neue Superklasse erstellt werden. Diese Klasse erbt von der vorherigen Superklasse und die Subklassen erben von der neuen Klasse. Es wird somit eine neue Superklasse eingeschoben.

Nach dieser Restrukturierung können die Methoden `Pull Up Method` bzw. `Pull Up Field` angewendet werden. Sind die Subklassen innerhalb der Vererbungsstruktur verstreut, d. h. sie besitzen nicht die gleiche direkte Superklasse, muss die Restrukturierungsmethode `Extract Class` angewendet werden. Die erzeugte Klasse wird von den Subklassen via Komposition eingebunden.

Durch die Analyse und Bewertung, nachdem beschriebenen und in der Abbildung 3.5 dargestelltem Verfahren, kann eine geeignete Restrukturierungsmethode ausgewählt werden. Mithilfe dieser Restrukturierungsmethode ist es möglich den erkannten KIF-Indikator korrekt aufzulösen. Die Auflösung des Indikators führt zu einer Optimierung der Vererbungsstruktur.

3.4 Partiell Analoge Klassenschnittstellen

Dieser Indikator ist dem KIF-Indikator ähnlich. Er beschreibt Klassen, die teilweise mit Schnittstellen anderer Klassen übereinstimmen. Die Methodensignaturen gleichen sich in verschiedenen Klassen. Der Indikator wird daher mit `Partiell Analoge Klassenschnittstellen` (PAKS) bezeichnet. Anders als beim KIF-Indikator, wird beim PAKS nicht auf doppelten Quellcode geachtet. Ein Beispiel ist eine Methode mit dem Namen `getNoise`. Diese soll einen Laut liefern. Die Methode wird von verschiedenen Tierklassen implementiert. Diese Methode würde verschiedene Laute in verschiedenen Tierklassen zurückgeben, jedoch sind Bedeutung und Methodenkopf gleich. In der Abbildung 3.6 sind zwei Klassen für den PAKS-Indikator dargestellt. Die Klassen `Cat` und `Dog` besitzen jeweils die Methode `getNoise`. Ein Objekt der Klasse `Dog` würde als Laut ein Bellen zurückgeben und ein Objekt der Klasse `Cat` ein Miauen. Die Schnittstellen der Klassen gleichen sich in diesem Beispiel vollends, da sie jeweils nur eine Methode mit dem gleichen Methodenkopf besitzen. Gleichen sich nicht nur die Methodenköpfe sondern auch die Methodenkörper, beschreibt dies den KIF-Indikator.

Abbildung 3.6: PAKS-Indikator

3.4.1 Problematik

Das Fehlen einer Schnittstelle für die Klassen, die durch diesen PAKS-Indikator beschrieben werden, stellt das Hauptproblem dar. Die Klasse `Cat` und `Dog` aus der Abbildung 3.6 implementieren keine Schnittstelle. Es ist also keine Abstraktion vorhanden. Die Endanwendung ist somit von Details abhängig. Dies widerspricht dem DIP.

Ein Beispiel für eine Endanwendung wäre, dass die Laute der Tiere auf die Kommandozeile gedruckt werden sollen. Dafür wird eine Klasse erzeugt, die die Methode `getNoise` der Klassen aufruft. In dem Beispiel 3.2 ist die beschriebene Klasse dargestellt. Die Klasse wird mit `AnimalNoisePrinter` bezeichnet. Durch das Fehlen einer gemeinsamen Schnittstelle und somit einer Abstraktionsschicht, muss für jede Klasse eine Methode erzeugt werden. Die Folge ist, dass für jeden weiteren Typ eine neue Methode erstellt werden muss, was das OCP verletzt und eine Kaskade von Änderungen zufolge hat. Diese Problematik schränkt die Erweiterbarkeit ein und steigert den Aufwand zur Implementierung.

Beispiel 3.2: AnimalNoisePrinter

```
public AnimalNoisePrinter {
  public void printDogNoise(Dog dog) {
    System.out.println(dog.getNoise());
  }
  public void printCatNoise(Cat cat) {
    System.out.println(cat.getNoise());
  }
}
```

3.4.2

Methode

In der Abbildung 3.7 ist das Verfahren zur Analyse und Bewertung eines PAKS-Indikators dargestellt. Diese Methode ermöglicht das Auswählen einer geeigneten Restrukturierungsmethode. Mithilfe der ausgewählten Restrukturierungsmethode kann der erkannte Indikator aufgelöst und die vorhandene Vererbungsstruktur optimiert werden.

Es muss nach der Erkennung des PAKS-Indikators geprüft werden, ob für diese Klassen bereits ein KIF-Indikator besteht. Ist dies der Fall, muss als erstes dieser aufgelöst werden. Nach der Auflösung des KIF-Indikators ist zu prüfen, ob weiterhin der PAKS-Indikator besteht. Dies ist möglich, da bspw. die KIF-Restrukturierung eine Komposition beinhaltete, die Methodennamen in den Klassen verblieben und somit nicht geändert wurden. Infolgedessen muss der PAKS Indikator aufgelöst werden.

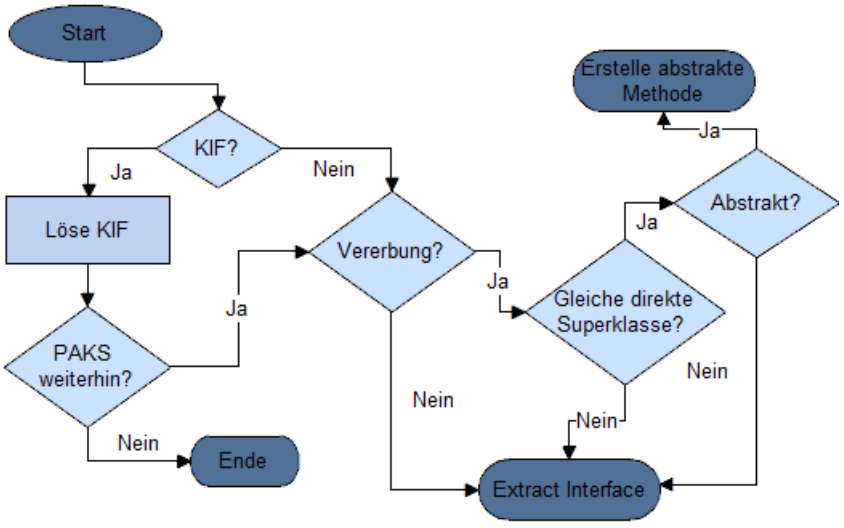


Abbildung 3.7: PAKS-Methode

Ist kein KIF-Indikator vorhanden oder wurde dieser bereits aufgelöst, muss analysiert werden, ob die Klassen innerhalb der gleichen Vererbungsstruktur existieren. Dabei sind auch gemeinsame Schnittstellenimplementierungen zu betrachten. Besteht keine gemeinsame Vererbungsbeziehung, muss die Restrukturierungsmethode `Extract Interface` angewendet werden. Bei einer vorhanden Vererbung oder Implementierung einer Schnittstelle muss geprüft werden, ob die Klassen eine gemeinsame direkte Superklasse bzw. Schnittstelle besitzen. Ist diese abstrakt bzw. eine Schnittstelle, muss eine abstrakte Methode, die diese Methodensignatur deklariert, in der abstrakten Superklasse oder Schnittstelle erzeugt werden. In allen anderen Fällen, muss die Restrukturierungsmethode `Extract Interface` angewendet werden.

Die erzeugte Schnittstelle deklariert die Methodensignatur und die gekennzeichneten Klassen implementieren die neue Schnittstelle.

Durch Analyse und Bewertung der erkannten PAKS-Indikatoren, mithilfe der in Abbildung 3.7 dargestellten Methode, kann eine geeignete Restrukturierungsmethode ausgewählt werden. Diese Restrukturierungsmethode ermöglicht das Auflösen des PAKS Indikators und Optimieren der vorhandenen Vererbungsstruktur.

3.5 Parallele Vererbungshierarchien

Dieser Indikator wird mit `Parallele Vererbungshierarchien` (PVH) bezeichnet und beschreibt das gleichnamige Problem und Code Smell. PVH bestehen, wenn zwei Vererbungsbäume existieren und einer von dem anderen abhängig ist. In der Abbildung 3.8 ist eine parallele Hierarchie dargestellt. Eine der Vererbungshierarchien beinhaltet Klassen für verschiedenartige Dateitypen. Die andere Hierarchie beinhaltet Klassen, um Daten verschiedener Dateitypen auf die Festplatte zu schreiben.

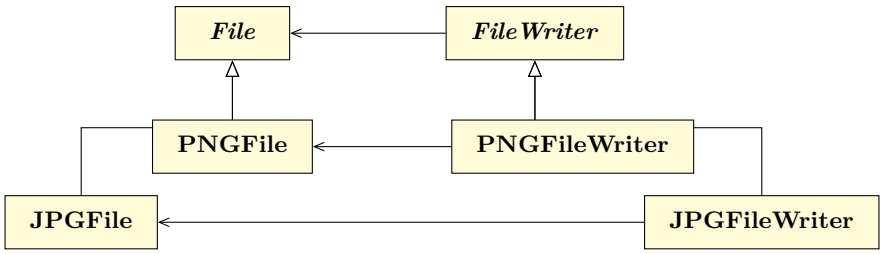


Abbildung 3.8: PVH-Indikator

Laut [Fow98] und [MLC05] können parallele Hierarchien durch die Verwendung gleicher Präfixe innerhalb der Klassennamen erkannt werden [vgl. Fow98, S. 68], [vgl. MLC05, S. 2]. Bspw. `JPGFile` und `JPGFileWriter`, aus dem Beispiel in Abbildung 3.8, beschreiben diese Anomalie. `JPGFileWriter` ermöglicht das Schreiben der Daten einer Datei vom Typ `JPGFile`.

3.5.1 Problematik

Die Problematik des PVH-Indikators besteht darin, dass bei jeder neu erstellten Subklasse eine weitere Subklasse im parallelen Vererbungsbaum erstellt werden muss. Bei Änderungen an der Hierarchie, kann es zu kaskadierenden Veränderungen kommen. Der PVH-Indikator beschreibt einen bekannten Code Smell [vgl. Fow98, S. 68], [vgl. LR06b, S. 46]. Die parallele Vererbungsstruktur und die vorhandenen Abhängigkeiten führen zu einer erhöhten Komplexität des Softwaresystems. Für das Beispiel aus der Abbildung 3.8 bedeutet das, dass für jeden neuen Dateityp eine neue Klasse zum Schreiben der Daten dieses Dateityps in der parallelen Hierarchie erstellt werden muss.

3.5.2

Methode

In der Abbildung 3.9 ist das Verfahren zur Analyse, Bewertung und Optimierung eines PVH-Indikators dargestellt. Zu Beginn der Analyse sollte bewertete werden, ob die erkannten Vererbungsstrukturen ein MVC Pattern darstellen, welches in der GUI Entwicklung häufiger verwendet wird. Ist dies der Fall, sollte das MVC Pattern mit Adaptern verbessert werden. D. h. es wird das MVA Pattern angewendet.

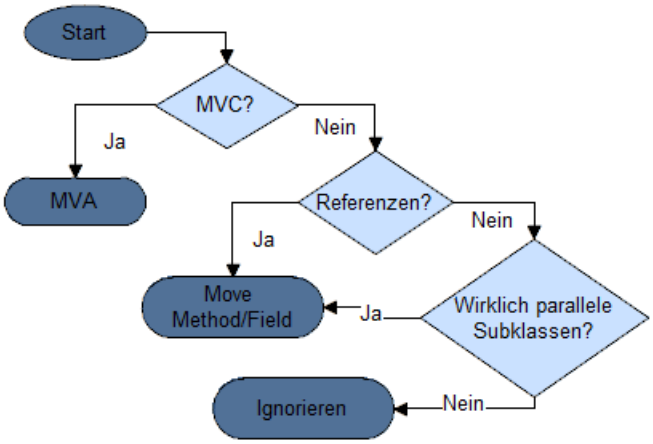


Abbildung 3.9: PVH-Methode

Ist dieses Pattern nicht vorhanden, muss geprüft werden, ob die Subklassen der einen Hierarchie auf die Subklassen der anderen Hierarchie referenzieren. Ist dies nicht der Fall, muss analysiert werden, ob die Subklassen wirklich parallel in beiden Hierarchien erstellt werden müssen. Der Indikator kann ignoriert werden, falls dies nicht der Fall ist. In allen anderen Fällen muss die Restrukturierungsmethode `Move Method` bzw. `Move Field` angewendet werden, sodass die parallelen Hierarchien zusammengeführt werden [vgl. Fow98, S. 68]. Durch diese Restrukturierung bleibt eine Hierarchie übrig.

Nach Anwendung der beschriebenen und in der Abbildung 3.9 dargestellten Methode, kann der erkannte PVH-Indikator analysiert, bewertet und eine geeignete Restrukturierungsmethode ausgewählt werden. Mithilfe dieser ausgewählten Restrukturierungsmethode, kann der PVH-Indikator aufgelöst und die vorhandene Vererbungsstruktur optimiert werden.

3.6 Delegation Überbeanspruchung

Dieser Indikator beschreibt eine übermäßige Nutzung der Delegation und gleicht dem Code Smell `Middle Man` [vgl. Fow98, S. 69]. D. h. eine Klasse `A` delegiert an eine Klasse `B`. Die Klasse `A` nutzt jegliche Funktionalität von `B` per Delegation. Dieser Indikator wird mit `Delegation Überbeanspruchung` (DÜ) bezeichnet. Die Klasse `A` kann weitere eigene Methoden bzw. Funktionalitäten besitzen. In der Abbildung 3.10 sind die Klassen `A` und `B` dargestellt.

Die Klasse `A` besitzt die Methoden `methodZ` und `methodX`. Mit der Methode `methodX` wird die Methode `methodX` des Attributs `mB` vom Typ `B` aufgerufen. Dies stellt eine Delegation von `A` nach `B` dar. `B` besitzt keine weiteren Methoden, weshalb ein DÜ Indikator vorliegt.

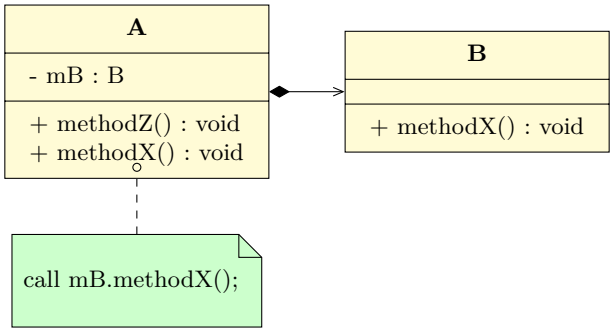


Abbildung 3.10: DÜ-Indikator

3.6.1

Problematik

Grundlegend verletzt dieser Indikator kein OOD-Prinzip. Durch die komplette Delegation an die Schnittstelle einer anderen Klasse entsteht jedoch eine erhöhte Komplexität. Es existieren viele Methoden, die keine eigenen Funktionalitäten besitzen. Die Übersichtlichkeit und somit Wartbarkeit wird dadurch vermindert.

3.6.2

Methode

In der Abbildung 3.11 ist das Verfahren zur Analyse, Bewertung und Optimierung eines DÜ-Indikators dargestellt. Ein wichtiges Kriterium, bei der Analyse des erkannten DÜ Indikators, zur Bewertung einer geeigneten Restrukturierung, stellt die Frage der Funktionalität der delegierten Klasse dar. Besitzt die delegierende Klasse keine eigene weitere Funktionalität, muss die Restrukturierungsmethode `Remove Middle Man` angewendet werden [vgl. Fow98, S. 130].

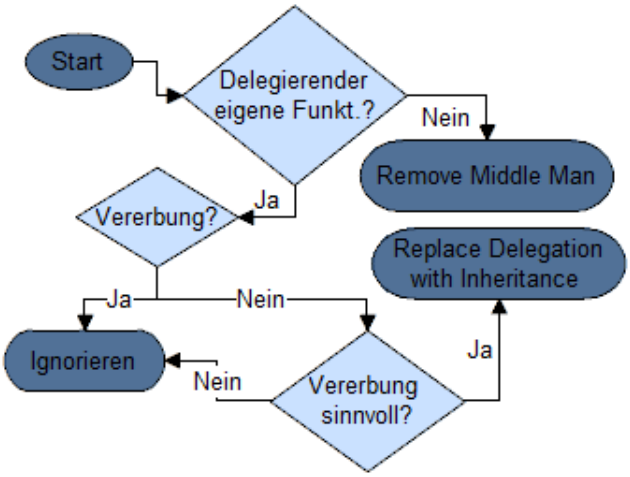


Abbildung 3.11: DÜ-Methode

Ist weitere Funktionalität vorhanden, muss analysiert werden, ob die delegierende Klasse bereits Teil einer Vererbungsbeziehung ist. Ist dies nicht der Fall und eine Vererbung wird als sinnvoll bewertet, muss die Restrukturierungsmethode `Replace Delegation with Inheritance` angewendet werden [vgl Fow98, S. 289]. Durch diese Restrukturierung kann neue Funktionalität von bereits existierenden Funktionalitäten separiert werden. Die Subklasse stellt durch weitere Funktionalitäten eine Spezialisierung dar. In allen anderen Fällen muss dieser Indikator ignoriert werden. Eine Restrukturierung würde in diesem Fall das System verkomplizieren.

Mit Zuhilfenahme der beschriebenen und in der Abbildung 3.11 dargestellten Methode, zur Analyse und Bewertung einer geeigneten Restrukturierungsmethode eines DÜ-Indikators, kann der erkannte DÜ-Indikator aufgelöst und die vorhandene Vererbungsstruktur optimiert werden.

3.7

Subklassen variieren nur in Konstanten

Dieser Indikator beschreibt Subklassen, die sich nur durch die Rückgabe von Konstanten in einer Methode unterscheiden. Der Indikator wird mit `Subklassen variieren nur in Konstanten` (SVK) bezeichnet und beschreibt die Code Smells Lazy Class und Data class [vgl. Fow98, S. 68 ff.].

[Fow98] gibt als Beispiel die in Abbildung 3.12 dargestellte Vererbungshierarchie an. Die Vererbungshierarchie besteht aus der Superklasse `Person` und den Subklassen `Male` und `Female`. Die Methode `getCode` liefert den Typ des verwendeten Objekts, den Geschlechtscode. D.h. entweder `M` oder `F` [vgl. Fow98, S. 188].

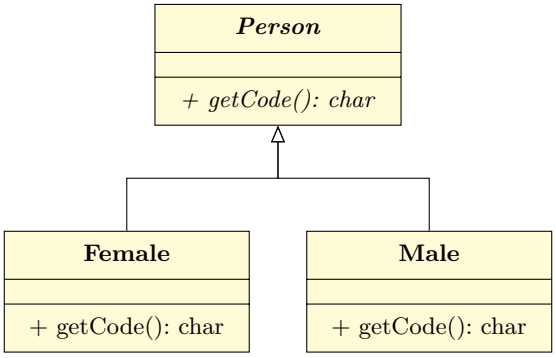


Abbildung 3.12: SVK-Indikator

3.7.1

Problematik

Laut [Fow98] haben Subklassen keine Existenzberechtigung, wenn sie nur Methoden, die Konstanten zurückliefern, besitzen. Diese Klassen können komplett durch Attribute in den Superklassen ersetzt werden. [vgl. Fow98, S. 188]. Sie entsprechen den Code Smell Lazy Class bzw. Data Class. Es wird durch die Subklassen eine erhöhte Komplexität geschaffen, welches die Wartbarkeit vermindert.

3.7.2

Methode

In der Abbildung 3.13 ist das Verfahren zur Analyse und Bewertung eines SVK Indikators dargestellt. Anhand der Methode kann eine geeigneten Restrukturierungsmethode ausgewählt werden. Diese ermöglicht die Optimierung der vorhandenen Vererbungsstruktur. Das Verfahren zur Auswahl einer geeigneten Restrukturierungsmethode ist dabei nicht umfangreich. Es muss lediglich analysiert werden, ob die gekennzeichneten Subklassen die einzigen Subklassen einer Superklasse darstellen. Ist dies der Fall, wird der Superklasse ein Attribut hinzugefügt, welches die verschiedenen Werte der Subklassen annehmen kann. Sind mehr Subklassen, als die gekennzeichneten, vorhanden, muss eine neue Subklasse erstellt werden. Diese Subklasse ersetzt die vorherigen und besitzt ein Attribut, für die Konstanten der Subklassen. Durch diese Vorgehensweise kann die Komplexität vermindert und die Vererbungsstruktur optimiert werden.

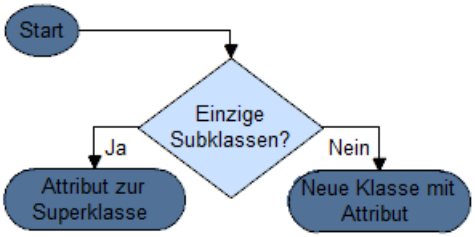


Abbildung 3.13: SVK-Methode

3.8

Downcasting

Im Abschnitt 2.1.6 wurde bereits das Konvertieren von Typen beschrieben. Beim `downcast` handelt es sich um eine Konvertierung von einem Superklassentyp in einen Subklassentyp. Der Indikator kennzeichnet die Parametertypkonvertierung vom Superklassentyp zum Subklassentyp innerhalb von Methoden und wird mit `Downcasting` (DC) bezeichnet. Bei Aufruf und Konvertieren des Rückgabetyps einer Methode wird vorausgesetzt, dass die Umsetzung bzw. das Verhalten der Methode bekannt ist. Daher wird, die Konvertierung von Rückgabewerten als weniger problematisch angesehen und innerhalb dieser Arbeit nicht behandelt. Der Aufruf einer Methode und die damit verbundene Konvertierung von Parametertypen kann nicht vorhergesehen werden, weshalb der Indikator DC diese Problematik kennzeichnet. Im Beispiel 3.3 wird die Methode `makeNoise` dargestellt. Diese Methode besitzt einen Parameter vom Typ `Animal`. Die Methode soll für eine Subklasse der Klasse `Animal` einen Laut erzeugen. Das übergebene Objekt wird nach Aufruf der Methode in den Typ `Dog` gewandelt. `Dog` stellt dabei eine Subklasse der Klasse `Animal` dar. Nach der Konvertierung wird die Methode `whoof` der Klasse `Dog` aufgerufen. Diese Methode soll ein Bellen erzeugen.

Beispiel 3.3: DC-Indikator

```
void makeNoise(Animal animal) {
  Dog dog = (Dog) animal;
  dog.whoof();
}
```

3.8.1

Problematik

Ist das übergebene Objekt keine Instanz des zu wandelnden Typs, kommt es zu einer Laufzeitausnahme, auch als `Runtime Exception` bezeichnet. In dem Beispiel 3.3 würde die Übergabe von einem Objekt der Super- oder einer anderen Subklasse zu einer Ausnahme führen. Dieses Verhalten verletzt das LSP, da die Methode sich abhängig vom Typ anders verhält. Die Problematik besteht darin, dass bei Unkenntnis die Anwendung unerwartet beendet, bzw. es zu anderen Laufzeitfehlern oder undefinierten Verhalten führen kann.

3.8.2

Methode

In der Abbildung 3.14 ist das Verfahren zur Analyse und Bewertung eines erkannten DC-Indikators dargestellt. Mithilfe dieser Methode kann eine geeignete Restrukturierungsmethode ausgewählt werden, die es ermöglicht, die vorhandene Vererbungsstruktur zu optimieren. Ein wichtiges Kriterium bei der Analyse und Bewertung des erkannten DC-Indikators ist die Existenz eines BAT-Indikators. Ist parallel ein BAT-Indikator vorhanden, muss dieser aufgelöst werden. Dies führt simultan zur Auflösung des DC-Indikators.

Wurde kein BAT-Indikator erkannt, folgt daraus das ein Parameter ohne Überprüfung umgewandelt wird. Der Parametertyp kann direkt in den konvertierten Typ angepasst werden, wenn die Methode, in der der DC-Indikator erkannt, nicht geerbt ist. Bei der Implementierung einer Sprachschnittstelle oder Bibliotheksschnittstelle, muss dieser DC-Indikator ignoriert werden. Es existiert keine Restrukturierungsmöglichkeit, da die Schnittstelle bzw. Methodensignatur nicht veränderbar ist.

Bei einer geerbten Methode oder Schnittstellenimplementierung, welche keine Sprach- oder Bibliotheksschnittstelle darstellt, müssen die Implementierungen der verwandten Subklassen analysiert werden. Handelt es sich immer um die gleiche Typkonvertierung, in den Subklassenimplementierungen, muss der Parametertyp innerhalb der Superklasse oder Schnittstelle auf diesen Typ angepasst werden. Sind verschiedene Typkonvertierungen in den Subklassen vorhanden, muss analysiert und bewertet werden, ob die gewandelten Typen in der gleichen Vererbungsstruktur existieren. Ist dies nicht der Fall, ist dies ein starker Verstoß gegen das LSP. Bei Verwendung der verschiedenen Subklassen kann dies zu Laufzeitfehlern führen. Es muss die vorhandene Vererbung der Methode aufgelöst werden. Die Methode verbleibt in den einzelnen Subklassen, die Parametertypen werden in die jeweilig konvertierten Typen angepasst. Existieren die gewandelten Typen innerhalb der gleichen Vererbungsstruktur, muss analysiert werden, ob diese die gleiche direkte Superklasse besitzen. Ist dies nicht der Fall, sind sie innerhalb der Vererbungsstruktur verstreut. Es muss die Restrukturierungsmethode `Extract Interface` für diese Subklassen angewendet werden. Infolgedessen wird der Parametertyp der geerbten Methode auf die neu erzeugte Schnittstelle angepasst. Die nach der Typkonvertierung aufgerufenen Methoden, werden innerhalb der Schnittstelle deklariert. Bei einer Konvertierung in allen vorhandenen Subklassen, kann eine abstrakte Methode in der Superklasse erzeugt werden. Diese kann in der Methode, in der der DC Indikator erkannt wurde, aufgerufen werden. Ist der Parametertyp der konvertiert wird generischer Natur, z. B. eine Wurzelklasse (`Object`) in Java oder ein `void*` in C++, muss dieser Parametertyp auf die Superklasse angepasst werden. Bei einem Parametertyp der Superklasse ist nichts weiter zu tun. Besitzen die Subklassen eine gemeinsame direkte Superklasse, aber es handelt sich nicht um alle existierenden Subklassen muss die Restrukturierungsmethode `Extract Superclass` angewendet werden. Es wird eine neue Superklasse in die Vererbungsstruktur eingeschoben sie enthält eine abstrakte Methode. Diese Superklasse erbt von der vorherigen Superklasse und die Subklassen erben von der neuen Klasse.

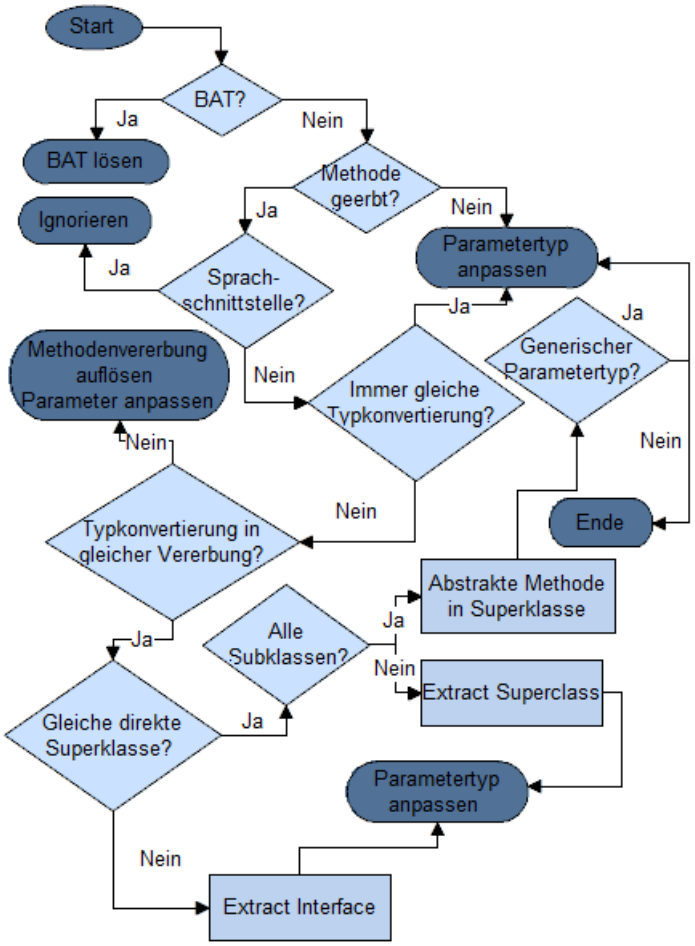


Abbildung 3.14: DC-Methode

Der Parametertyp der Methode muss auf die neue Superklasse angepasst und die abstrakte Methode muss innerhalb der Methode, in der der DC-Indikator erkannt wurde, aufgerufen werden.

Nach Anwendung der beschriebenen und in der Abbildung 3.14 dargestellten Methode, kann der erkannte DC-Indikator analysiert, bewertet und eine geeignete Restrukturierungsmethode ausgewählt werden. Mithilfe dieser Restrukturierungsmethode kann der DC-Indikator aufgelöst und die vorhandene Vererbungsstruktur optimiert werden.

3.9 Mehrfachvererbung

Im Abschnitt 2.1.3 wurde bereits beschrieben, dass die Mehrfachvererbung von Klassen nur in C++ erlaubt ist. In Java und C# ist diese für Klassen nicht möglich, aber für Schnittstellen. Schnittstellen können von mehreren verschiedenen anderen Schnittstellen erben. Die Möglichkeit, mehrere Schnittstellen zu implementieren, besteht in allen objektorientierten Sprachen. Der Indikator, der als `Mehrfachvererbung` (MV) bezeichnet wird, zeigt an, dass eine Mehrfachvererbung oder Mehrfachimplementierung existiert, in der Methodensignaturen mehrmals in verschiedenen Superklassen oder Schnittstellen vorkommen. In der Abbildung 3.15 wird der MV Indikator mit einer Mehrfachimplementierung dargestellt. Die Schnittstellen `Walking` und `Flying` deklarieren die Methode `move`. Diese Methode besitzt in den Schnittstellen die gleiche Signatur. Die Klasse `Bird` implementiert diese Schnittstellen.

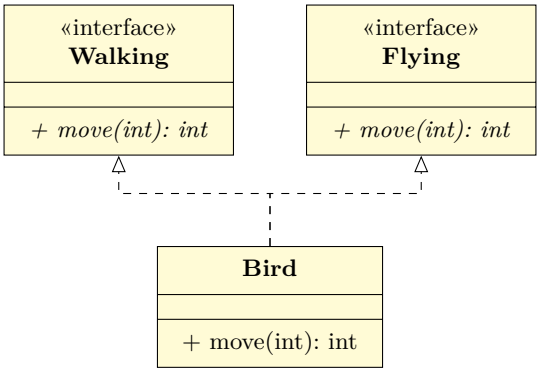


Abbildung 3.15: MV-Indikator

3.9.1 Problematik

Der Indikator beschreibt ein logisches Problem, welches nicht leicht erkennbar ist. Die Mehrfachvererbung bzw. Implementierung wird vom Compiler übersetzt, da keine Mehrdeutigkeiten existieren. Die Subklasse überschreibt die Methoden, die in den Schnittstellen oder Superklassen deklariert wurden. Durch diese Überschreibung kann die Mehrdeutigkeit aufgelöst werden. D. h. es wird die Implementierung der Subklasse verwendet. Das Problem ist, dass nur eine Implementierung für wahrscheinlich zwei semantisch verschiedene Methoden existiert. Es wird somit mindestens ein Schnittstellenvertrag gebrochen. Bei der Mehrfachvererbung bzw. Implementierung von gleichnamigen Methoden kann nicht garantiert werden, für welchen Anwendungsfall die Methode überschrieben wurde. Infolgedessen kann dies zu unerwartetem Verhalten führen.

Im Beispiel 3.15 ist eine Mehrfachimplementierung von zwei Schnittstellen vorhanden. Eine Instanz der Klasse `Bird` soll fliegen und laufen können, weshalb diese Schnittstellen implementiert werden. Die Schnittstellen deklarieren eine Methode mit gleicher Signatur `+ move(int): int`. Implementiert die Klasse `Bird` nun die Methode `move`, muss entschieden werden, welche Implementierung gewählt wird. Durch die Entscheidung für eine Implementierung wird der Vertrag der anderen Schnittstelle gebrochen. Wird für die Klasse `Bird` die Methode zum Laufen implementiert, kann diese nicht fliegen und andersherum.

Die Verwendung in anderen Programmabschnitten, in denen die Implementierung der Schnittstellen erwartet wird, ist nicht mehr vorhersehbar, was somit das LSP verletzt. Dies kann zu unerwarteten Fehlern bei der Verwendung der Implementierung führen.

3.9.2

Methode

In der Abbildung 3.16 ist das Verfahren zur Analyse, Bewertung und Optimierung eines MV Indikators dargestellt. Mithilfe dieser Methode kann eine geeignete Restrukturierungsmethode ausgewählt werden, mit der die vorhandene Vererbungsstruktur optimiert werden kann.

Es muss zu erst analysiert werden, ob die, mit dem MV-Indikator, gekennzeichneten Klassen bzw. Schnittstellen Codeklone darstellen. D. h. das parallel bereits der KIF-Indikator erkannt wurde, dieser muss gelöst werden. Die Auflösung des KIF-Indikators löst simultan den vorhandenen MV-Indikator. Ist kein KIF Indikator vorhanden, die Methoden besitzen aber die gleiche Intention, muss die Restrukturierungsmethode `Extract Interface` angewendet werden. Die neu erzeugte Schnittstelle wird von den Superklassen implementiert bzw. von den Schnittstellen geerbt. Besitzen die Methoden, die mit dem MV-Indikator gekennzeichnet wurden, nicht die gleiche Intention, muss analysiert und bewertet werden, ob diese veränderbar sind. Ist dies der Fall, müssen die Methodennamen geeignet angepasst werden. Bei nicht veränderbaren Methoden ist zu analysieren, ob eine Implementierung durch eine weitere neue Klasse möglich ist. Die neue Klasse implementiert eine der Schnittstellen bzw. erbt von einer der Superklassen und wird als Komposition von der Subklasse verwendet. Ist diese Auslagerung keine Option, muss dieser MV-Indikator ignoriert werden.

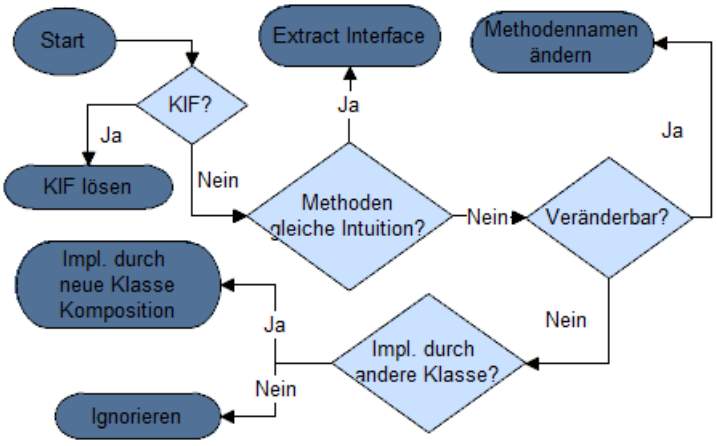


Abbildung 3.16: MV-Methode

Nach Anwendung der beschriebenen und in der Abbildung 3.16 dargestellten Methode kann der erkannte MV-Indikator analysiert, bewertet und eine geeignete Restrukturierungsmethode ausgewählt werden. Mithilfe dieser Restrukturierungsmethode kann der MV-Indikator aufgelöst und die vorhandene Vererbungsstruktur optimiert werden.

3.10 Gewichtung

Die Gewichtung ermöglicht das Eingliedern der einzelnen Indikatoren. Mithilfe dieser Gewichtung ist erkennbar, welche Indikatoren als erstes aufgelöst werden müssen. Je höher die Gewichtung, umso mehr Problematiken werden dem definierten Indikator zugeschrieben. Die Tabelle 3.1 beinhaltet die Gewichtung der zuvor definierten Indikatoren. Sie besitzt bestimmte Kritik- und Problempunkte, welche mit "Ja" oder "Nein" beantwortet werden können. Besteht eine bestimmte Problematik für diesen Indikator, so muss ein Punkt vergeben werden, wobei 1 für "Ja" und 0 für "Nein" steht. Es werden zum Ende die Wertungen zusammengerechnet und diese ergeben eine Gesamtbewertung und somit eine Gewichtung des Indikators.

	SKS	BAT	KIF	PAKS	PVH	DÜ	SVK	DC	MV
Kaskadierende Änderungen	1	1	1	0	1	0	0	0	0
Fehlende Abstraktion	0	1	1	1	0	0	0	0	0
Unerwartetes Verhalten	1	0	0	0	0	0	0	1	1
Erhöhte Komplexität	1	1	0	0	1	1	1	0	0
Gesamtpunkte	3	3	2	1	2	1	1	1	1

Tabelle 3.1: Gewichtung der Indikatoren

Der SKS-Indikator weist auf kaskadierenden Änderungen, erhöhte Komplexität sowie auf undefiniertes/unerwartetes Verhalten hin. Durch diese Problematiken ergibt sich für den SKS-Indikator eine Gewichtung von drei. Die Indikatoren BAT und SKS stellen die Indikatoren mit der höchsten Gewichtung dar. Ein BAT-Indikator weist kaskadierende Änderungen, fehlende Abstraktion und erhöhte Komplexität auf. Eine Gewichtung von zwei besitzen die Indikatoren KIF und PVH. Der KIF-Indikator weist auf eine fehlende Abstraktionsschicht und kaskadierende Änderungen hin. Ein erkannter PVH-Indikator weist kaskadierende Änderungen und erhöhte Komplexität auf. Die Indikatoren PAKS, DÜ, SVK, DC und MV sind mit einem Punkt gewichtet und weisen demzufolge auf eine, der in Tabelle 3.1 dargestellten, Problematik hin. Der Indikator PAKS weist auf eine fehlende Abstraktion hin. Die Indikatoren DÜ und SVK weisen auf erhöhte Komplexität und die Indikatoren DC und MV auf undefiniertes/unerwartetes Verhalten hin.

Werkzeuganalyse
4

Im folgendem Kapitel wird eine Werkzeuganalyse durchgeführt, um festzustellen, welche Methoden und Daten bereits existieren und zur Verfügung stehen. Es werden dabei die Werkzeuge Klocwork, Sotograph und Como betrachtet, da diese von der Firma B. M. innerhalb einer CQA verwendet werden. Diese Analyse soll abgrenzen, welche Funktionalitäten genutzt und welche entwickelt werden müssen.

4.1 Como

Das von der Firma B. M. hauseigene Werkzeug Como benutzt, wie bereits im Abschnitt 2.5.1 beschrieben, die Werkzeuge Klocwork, Sotograph und CCfinder zur Unterstützung der von B. M. durchgeführten CQAs. Die Daten, die das Werkzeug Sotograph bereitstellt, werden analysiert, ausgelesen und angemessen dargestellt. Für Como wurden eigene Checker erzeugt, die nach bestimmten eigenen Anti-Pattern suchen. Diese und die allgemeinen Klocwork-Checker werden mit Klocwork ausgeführt. Die Daten von Klocwork werden nach der Analyse in eine MySQL- und Lucene- Datenbank gespeichert. Sotograph speichert die Analysedaten innerhalb einer PostgreSQL-Datenbank. Zur Verbindung zu einer dieser Datenbanken stellt Como Klassen bzw. Schnittstellen zur Verfügung. Die Funktionalitäten zur Verbindung mit einer Datenbank werden vom erstellten Werkzeug wiederverwendet. Es werden Klassen und Schnittstellen erzeugt, sodass die Verwendung geeignet gekapselt und später ersetzt werden kann.

4.2 Sotograph

Im Abschnitt 2.5.2 wurde bereits beschrieben, dass das Sotograph-Werkzeug den gegebenen Quellcode statisch analysiert. Aus dieser statischen Quellcodeanalyse wird eine PostgreSQL-Datenbank pro Projekt und Version angelegt. Diese Datenbank beinhaltet unter anderem Informationen zu Klassen, Methoden, Dateien und viele weitere Bestandteile des analysierten Softwareprojekts. Die Sotograph-Datenbank besitzt eine große und komplexe Datenbankstruktur, die aus Platzgründen innerhalb dieser Arbeit nicht dargestellt werden kann. Die verwendeten Tabellen werden jedoch in diesem Abschnitt kurz benannt und erläutert. Die Sotograph-Datenbankstruktur ist für jedes Projekt identisch und kann somit zur Analyse gleichbleibend verwendet werden. Es besteht kein Unterschied in der Datenbank, wenn ein C++ oder Java Projekt analysiert wurde. Es wird nur innerhalb der Datensätze vermerkt, um welche Sprache es sich beim analysierten Projekt handelt. Somit sind einige Tabellen oder Spalten von Tabellen bei verschiedenen Sprachen gefüllt oder leer. Das hat den Vorteil, dass die Analysewerkzeuge nicht grundlegend auf die einzelnen Sprachen angepasst werden müssen. Es können somit die gleichen Structured Query Language (SQL)-Abfragen verwendet werden, um die Daten aus der Sotograph-Datenbank zu erhalten.

Sotograph erstellt zu den analysierten Daten eines Softwareprojekts verschiedene Metriken, diese werden von Como eingelesen und dargestellt. Die Sotograph-Metriken finden innerhalb des erstellten Werkzeugs keine Anwendung. Es wird auf die Repräsentationsdaten des Quellcodes zurückgegriffen. Eine wichtige Rolle spielen dabei die Tabellen `classes`, `sourcefiles`, `files`, `positions`, `methods`, `methodreferencesflat`, `inheritings` und `inheritancenestings`. Jeder Datensatz innerhalb einer Tabelle stellt ein Symbol dar und besitzt eine eigene Symbol-ID, welche in der `symbols`-Tabelle gespeichert ist. Demzufolge referenzieren alle Tabellen mit einer Symbol-ID auf diese Tabelle. Die Tabelle `classes` beinhaltet die Informationen über die Klassen innerhalb des analysierten Projekts. Die dazu gehörige Quelldatei wird mittels eines Fremdschlüssels referenziert. Dieser Fremdschlüssel verweist auf die Tabelle `files`. Die Tabelle `files` enthält Informationen über die Quelldateien, sowie eine weitere Referenz auf die Tabelle `sourcefiles`. Die Tabelle `sourcefiles` beinhaltet die Pfadangaben der Quelldateien. Methodeninformationen sind innerhalb der Tabelle `methods` gespeichert. Sie verweisen auf die dazu gehörige Klasse, Datei sowie auf eine bestimmte Position innerhalb einer Quelldatei.

Positionsinformationen wie z. B. Zeilen- oder Spaltennummer werden in der Tabelle `positions` gespeichert. Die Tabelle beinhaltet für verschiedene Symbole (Klassen, Methoden, Attribute etc.) die Positionsinformationen. Sie referenziert auf die Tabelle `sourcefiles`, welche die Quelldateiinformation beinhaltet, und auf die `symbols`-Tabelle. Die `methods`-Tabelle besitzt eine Beziehung mit sich selbst, diese Beziehung wird durch die Tabelle `methodreferencesflat` dargestellt. Diese Tabelle speichert die Methodenaufrufe innerhalb von Methoden. Die Spalte `refingsymbolid` referenziert die Methode, die delegiert, und die Spalte `refedsymbolid` referenziert auf die delegierte Methode (auf die aufgerufene Methode). Vererbungsinformationen von Klassen und Schnittstellen werden innerhalb der Tabelle `inheritings` gespeichert. In dieser sind nur direkte Vererbungsbeziehungen gespeichert. Tiefere Vererbungsbeziehungen, nicht nur direkte sondern über Kind- Elternbeziehungen hinaus, werden in der Tabelle `inheritancenestings` gespeichert. Die Tiefe der Vererbung wird mithilfe der Spalte `distance` angeben. Jede Klasse ist Superklasse von sich selbst und wird mit `distance` gleich 0 angegeben. Handelt es sich bei dem analysierten Projekt um ein C++ Projekt, wird die Tabelle `typedefs` mit vorhandenen `typedef` Definitionen befüllt. Die Tabelle beinhaltet die Informationen zu den Namen und den Ursprungstypen der `typedefs`.

Mithilfe von Sotograph, den beschriebenen Tabellen und deren Daten ist es möglich, durch Verwendung von SQL-Abfragen, die Indikatorerkennung zu realisieren. Der Hauptbestandteil der Indikatorerkennung wird durch Abfragen an die Sotograph-Datenbank umgesetzt. Die PostgreSQL-Datenbank bzw. deren Daten werden in Como und innerhalb der CQA verwendet. Die gewünschten Daten liegen bereits vor, wenn diese mit dem zu erstellenden Werkzeug analysiert werden sollen, weshalb auf die statische Quellcodeanalyse von Sotograph nicht weiter eingegangen wird.

4.3 Klocwork

Mithilfe von Klocwork ist es möglich, auf Quellcode-Ebene Indikatoren wie BAT, DC oder SVK zu erkennen. Für diese Erkennung müssen benutzerdefinierte Checker erstellt und in Klocwork eingebunden werden. Klocwork unterstützt zwei Arten von benutzerdefinierten Checkern. Die einen ermöglichen das Überprüfen des Klocwork abstract syntax tree (KAST). Sie werden daher KAST-Checker genannt. Die anderen ermöglichen das Überprüfen der `intermediate representation` (engl. Zwischendarstellung) und werden Path-Checker genannt. Die KAST-Checker sind nützlich, um syntaxverwandte Probleme oder Eigenschaften aufzudecken. Sie operieren gegen den von Klocwork generierten abstract syntax tree (AST). KAST ist eine XPath ähnliche domänenspezifische Sprache, die Operatoren und Funktionen liefert, um die Konstruktion von deklarativen Anweisungen zu unterstützen. So kann ein gesuchter und entsprechender Teil des AST identifiziert werden. Klocwork beinhaltet eine Anwendung zum Erstellen und Testen von KAST-Checkern, das sogenannte Checker Studio. Die Path-Checker suchen Kontrollfluss- und Datenflussprobleme. Diese sind für die Erstellung dieser Arbeit uninteressant, weshalb auf diese nicht näher eingegangen wird [Klo12].

Klocwork ermöglicht das Überprüfen von Quellcode und Schreiben von Checkern in den Sprachen C++, Java und C. Die Erstellung von benutzerdefinierten C#-Checker ist in der Klocwork-Version 10.2 nicht möglich, welche zur Erstellung dieser Arbeit benutzt wird. In der Version 10.3 ist es möglich, benutzerdefinierte Checker für C# zu erstellen [Klo15c]. Es werden jedoch nur die Sprachspezifikationen 1.0, 2.0, 3.0 und 4.0 unterstützt [Klo15b]. Die aktuelle Version für C# ist 6.0 [Net15], weshalb in dieser Arbeit die Unterstützung für C# zurückgestellt wird. Im weiteren Verlauf der Arbeit werden zur Erkennung der Indikatoren die Sprachen C++ und Java betrachtet. Die Erkennung für die Sprache C# steht damit in Ausblick. Das gilt nur für die Indikatoren, die mithilfe von Klocwork erkannt werden müssen. Die Indikatoren, die durch Sotograph oder CCfinder erkannt werden, sind für C# erkennbar.

4.4 CCFinder

Das Werkzeug CCfinder ermöglicht, wie bereits im Abschnitt 2.5.3 beschrieben, das Erkennen von Codeklonen. Diese Klone können sich entweder innerhalb einer Datei oder in verschiedenen Dateien befinden. CCfinder erzeugt Ausgabedateien, die Referenzen auf die gefundenen Codeklone aus den durchsuchten Dateien enthalten. Como nutzt dieses Werkzeug bereits um Codeklone zu erkennen und anzuzeigen. Zum Auslesen dieser Ergebnisse bzw. der Klone kann die Funktionalität von Como wiederverwendet werden. Innerhalb dieser Arbeit werden nur die Klone betrachtet, die in verschiedenen Klassen auftauchen. Diese Klone stellen den KIF-Indikator dar und können durch Vererbung aufgelöst werden. Klone innerhalb einer Datei bzw. einer Klasse können durch einfaches extrahieren einer neuen Methode aufgelöst werden und stellen kein Problem dar, welches in dieser Arbeit betrachtet wird.

Entwurf und Implementierung
5

Mithilfe der zuvor definierten und analysierten Indikatoren und der Analyse der zur Verfügung stehenden Werkzeuge, wird im folgendem Kapitel der Entwurf und die Implementierung für das erstellte Werkzeug beschrieben. Das Kapitel wird nach den entworfenen Namensräumen gegliedert. Die einzelnen Abschnitte für die Namensräume enthalten kurze Erläuterungen zu deren Implementierung. Es wird genauer auf die Implementierung der Detektoren eingegangen.

Das Werkzeug wird IChecker genannt. Es erkennt und stellt die in Kapitel 3 definierten Indikatoren, innerhalb von objektorientierten Softwaresystemen, geeignet dar. Die Implementierung des Werkzeugs erfolgt in C#. Bei der Erkennung wird auf bereits existierende Funktionalitäten von Como sowie auf die zuvor beschriebenen Werkzeuge zurückgegriffen. Es wird für das Werkzeug eine gleichnamige C#-`Solution` erzeugt. Eine `Solution` beinhaltet eine oder mehrere C# Projekte. Dies ermöglicht es, verschiedene Projekte zu definieren und einzubinden. Bspw. können Teilprojekte von Como eingebunden und genutzt werden. Sollte im Ausblick eine Graphical User Interface (GUI) erstellt werden, kann diese separiert vom Kern des Werkzeugs erstellt werden.

Die notwendigen Informationen zur Verbindung mit der Datenbank sowie Pfadangaben werden in einer Einstellungsdatei gespeichert. Die Daten werden innerhalb dieser Datei als Schlüssel-Wert Paare gespeichert. Diese Einstellungen sind innerhalb des Quellcodes über den jeweiligen Schlüssel erreichbar. `Properties.Settings.Default.[KEY]` ermöglicht den Zugriff auf einen Wert, wobei `KEY` dabei mit dem entsprechenden Schlüssel ersetzt wird. Mithilfe dieser Einstellungsdatei können Konstanten gespeichert werden. Die Konstanten können später geändert werden, ohne das Projekt neu zu übersetzen. Die Einstellungen werden als Datei mit der Dateiendung `.config` mit der ausführbaren Datei ausgeliefert. Im Ausblick wäre es möglich, dass die Parameter zur Pfad- und Datenbankangabe nicht nur über die Einstellungen, sondern auch über die Kommandozeile übergeben und somit geändert werden können.

Für den Kern des Werkzeugs wird ein Projekt namens `ICCore` innerhalb der `Solution` erstellt. Der Hauptnamensraum ist gleichnamig zum Projektnamen. Dieser beinhaltet die Namensräume `Core`, `Database`, `Graphics`, `Detection` und `IO`. Die nachfolgenden Abbildungen beinhalten UML-Klassendiagramme zum Darstellen der einzelnen Projektnamensräume. Es wird für die C# spezifische Eigenschaft (engl. property) ein neuer Stereotyp eingeführt. `<<C# Property>>` stellt diesen Stereotypen dar. Er ermöglicht es aus Platzgründen auf `getter` und `setter` zu verzichten.

5.1
Core

Der Namensraum `Core` ist in der Abbildung 5.1 dargestellt. Er beinhaltet die Klassen, mit denen die erkannten Indikatoren als Objekte abgebildet werden können. D. h. die Klasse `Indicator`, die Klasse `IndicatorList`, die Schnittstelle `IDescriptor` sowie Klassen, die diese Schnittstelle implementieren. Die Klasse `Indikator` beinhaltet eine Eigenschaft namens `IndicatorKey`, welche nähere Informationen zum Indikator beinhaltet, eine Liste von Objekten die die Schnittstelle `IDescriptor` implementieren und eine Eigenschaft für den Namen des Indikators. Die Liste aus `IDescriptor`-Objekten soll den Indikator soweit wie möglich beschreiben. Es können verschiedene Objekte vom Typ `IDescriptor` dem `Indicator`-Objekt hinzugefügt werden, die dessen Auftreten beschreiben. Die Klasse `FileDetail` beinhaltet Informationen zur Datei, in der der Indikator aufgetreten ist. `ClassDetail` besitzt Informationen zur Klasse, `MethodDetail` zur Methode und `MethodLineDetail` besitzt Informationen zur genauen Zeile innerhalb des Quellcodes sowie den Zeileninhalt. Die Klasse `CloneDetail` ist speziell für den KIF-Indikator vorgesehen. Sie beinhaltet Informationen zur Klasse, in der der Klon gefunden wurde, sowie Start- und Endzeilennummer des Klons, innerhalb einer Datei. Die einzelnen Klassen zur Beschreibung der Details bauen aufeinander auf, ein `ClassDetail` Objekt benötigt ein `FileDetail`-Objekt etc. Die Klasse `IndicatorList` stellt eine Liste von Indikatoren dar und besitzt als einzige Eigenschaft eine Liste von Objekten vom Typ `Indicator`.

Die Wrapperklasse `IndicatorList` war notwendig, um die Liste von Indikatoren erfolgreich mithilfe der C# `XmlWriter`-Klasse zu serialisieren. Den einzelnen Klasseneigenschaften wurden `XmlElement`-Attribute hinzugefügt, sodass diese Eigenschaften korrekt mit dem geeigneten Namen und deren Werten serialisiert werden.

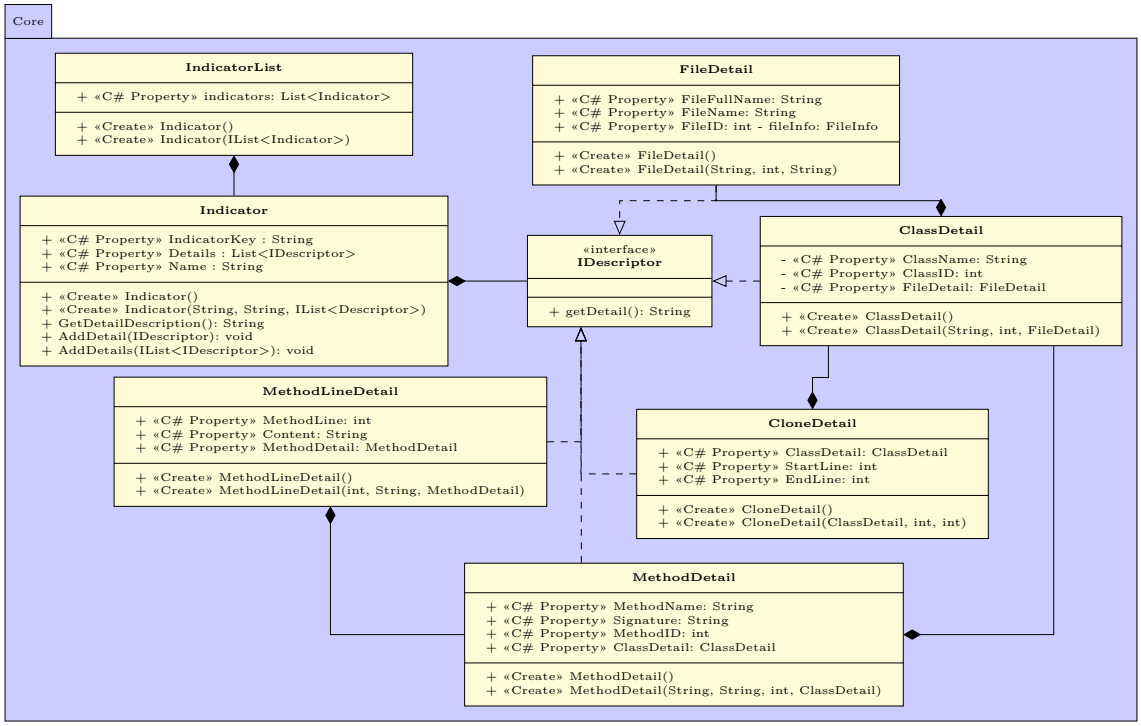


Abbildung 5.1: Core-Namensraum

5.2 Database

Der Namensraum `Database` beinhaltet die Schnittstelle `ISQLQueryExecutor` zum Kommunizieren mit der Datenbank, siehe Abbildung 5.2. Diese Schnittstelle wird von den erstellten Detektoren verwendet. Die Klasse `SQLQueryExecutorImpl` stellt die Implementierung der Schnittstelle `ISQLQueryExecutor` dar. Diese verwendet das `DatabaseCore`-Como-Projekt, um mit der Sotograph-Datenbank zu kommunizieren. Durch diese Kapselung ist es möglich, die Implementierung der Kommunikation einfach auszutauschen, ohne dass die Detektoren angepasst werden müssen.

Die erstellten SQL-Abfragen wurden als Ressourcen zum C# Projekt hinzugefügt. Durch die Nutzung von Ressourcen für die Abfragen sowie für Konstanten können diese direkt angesprochen werden. Sie verhalten sich ähnlich wie Einstellungsdateien. `Properties.Resources.[NAME]` ermöglicht den direkten Zugriff auf die Ressource innerhalb des Quellcodes, wobei `[NAME]` mit dem Namen der Ressource ersetzt wird. Größere Abfragen, vor allem Abfragen für die Detektoren, werden in einzelnen Dateien hinterlegt und geeignet benannt. Dateien mit detektorspezifischen Abfragen besitzen als Präfix den Indikatornamen. Durch die Verwendung der Ressourcendateien befinden sich alle verwendeten Konstanten an einem zentralen Punkt. Das hat den Vorteil, dass bei Änderungen nur eine Datei geändert werden muss und die Suche sich auf diese Datei beschränkt. Die Ressourcen werden mit in den Assembler Code integriert, sodass bei Auslieferung des Werkzeugs diese nicht verändert oder entwendet werden können, anders als die Einstellungsdateien. Veränderungen könnten zu undefiniertem oder gar schädlichem Verhalten führen. Der `Database`-Namensraum beinhaltet eine weitere Schnittstelle namens `IKlocworkRequester`.

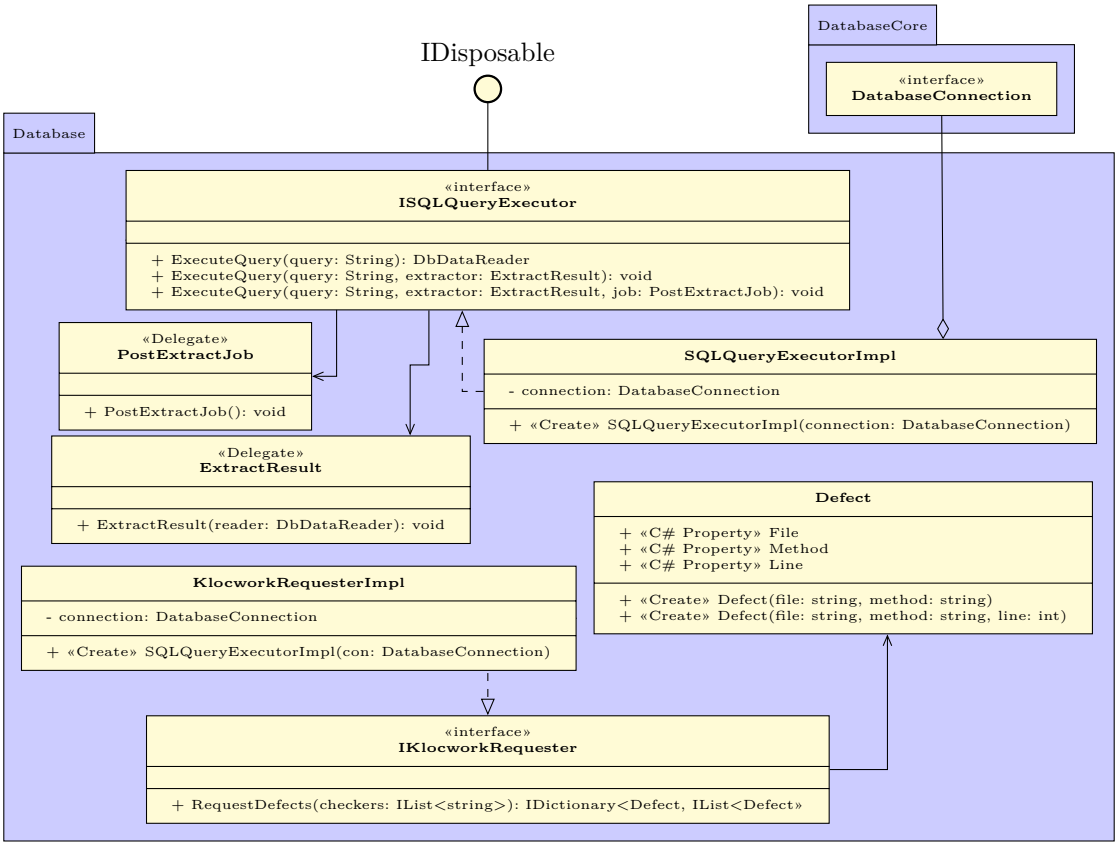


Abbildung 5.2: Database-Namensraum

`IKlocworkRequester` ermöglicht es, dass sogenannte Klocwork-Defekte abgefragt werden können. Die Defekte werden unter der Verwendung der erstellten Klocwork-Checkern von Klocwork erkannt und in einer Lucene-Datenbank gespeichert. Jeder Defekt innerhalb der Lucene-Datenbank beinhaltet als Eintrag den Namen des Checkers, durch den er erkannt wurde. Die Schnittstelle `IKlocworkRequester` besitzt eine Methode `IDictionary<Defect, IList<Defect>> RequestDefects(IList<string> checkers)`, diese fragt für die gegebenen Checkernamen die dazugehörigen Defekte aus der Lucene-Datenbank ab und gibt diese zurück. Für die Darstellung und das Speichern der Informationen der Defekte wurde eine Klasse `Defect` erstellt, welche den Methodennamen, die Zeilennummer sowie den Dateipfad beinhaltet. Die Defekte werden pro Methode und Datei zusammengefasst. Es wird somit ein `Dictionary` erstellt, welches als Schlüssel einen `Defect`-Objekt und als Wert eine Liste von `Defect`-Objekten besitzt. Die einzelnen Schlüssel werden nach Datei und Methode unterschieden. Ist ein `Defect`-Objekt mit der gleichen Methode und Datei bereits im `Dictionary` enthalten, wird das `Defect`-Objekt der Liste hinzugefügt. Anderenfalls wird ein neuer Eintrag im `Dictionary` erstellt und eine neue Liste mit diesem Objekt hinzugefügt. Die Klasse `KlocworkRequesterImpl` setzt dies um und implementiert die Schnittstelle `IKlocworkRequester`. Mithilfe der Lucene-Bibliothek werden die Daten ausgelesen. Die notwendigen Abfragen und Konstanten für die Lucene-Datenbank befinden sich in der Ressourcendatei und der Pfad der Datenbank in der Einstellungsdatei. Durch die Verwendung und Einführung einer Schnittstelle für das Auslesen der Defekte, sind die Detektoren nicht von Details abhängig. Die Implementierung kann später ersetzt werden, ohne dass die Detektoren angepasst werden müssen.

`PostExtractJob` und `ExtractResultSetFromReader` stellen C# spezifische `delegates` dar. Sie definieren das Aussehen einer Funktions-/Methodensignatur ähnlich Typedefs von Funktionszeigern in C++. Diese `delegates` werden von den Methoden der `ISQLQueryExecutor`-Schnittstelle erwartet. Die Methode mit der entsprechenden Signatur von `ExtractResultSetFromReader` soll dabei die Funktionalität zum Extrahieren der Daten aus einem gegebenen `DbDataReader`-Objekt beinhalten.

Das `DbDataReader`-Objekt wird durch Abfragen an die Datenbank erstellt und dann der entsprechenden Methode vom Typ `ExtractResultSetFromReader` übergeben. `PostExtractJob` ist optional und kann ausgeführt werden, um nach der Extraktion noch weitere Funktionalitäten auszuführen.

5.3 IO

In der Abbildung 5.3 wird der Namensraum `IO` dargestellt. Der Namensraum `IO` beinhaltet Klassen, die es ermöglichen Indikatoren in Dateien zu speichern.

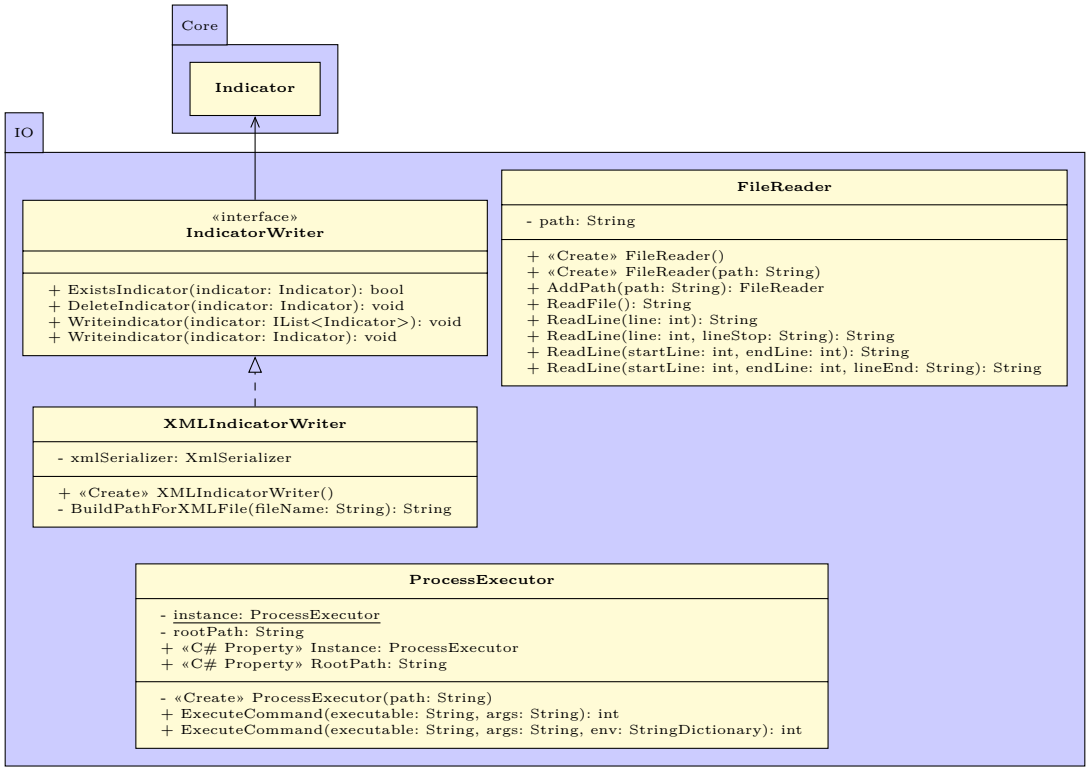


Abbildung 5.3: IO-Namesensraum

`IndicatorWriter` stellt die Schnittstelle dar, die zum Speichern der erkannten Indikatoren verwendet wird. Es wird eine Schnittstelle verwendet, sodass das Schreiben und Speichern angepasst bzw. verändert werden kann. Die Klasse `XMLIndicatorWriter` stellt eine Implementierung der Schnittstelle dar und ermöglicht es, die Indikatoren in Extensible Markup Language (XML)-Format zu serialisieren. Durch diese Serialisierung soll es möglich sein, dass die Indikatoren ausgetauscht und mithilfe von XML-Parsern wieder eingelesen werden können. Durch Extensible Stylesheet Language Transformations (XSLT) kann das generierte XML in HTML transferiert werden, sodass die gefunden Indikatoren besser dargestellt werden können. Indikatoren können Details zu bestimmten Zeilen enthalten (Zeilennummer und Zeileninhalt). Mit diesen Details kann das Auftreten des Indikators detailliert beschrieben werden. Die `FileReader`-Klasse stellt Funktionalitäten zum Einlesen von Dateien zur Verfügung. Mithilfe dieser Klasse ist es vor allem möglich, einzelne oder mehrere bestimmte Zeilen aus einer Datei einzulesen.

Die Klasse `ProcessExecutor` wurde nach dem `Singleton` Pattern [vgl. Gam+94, S. 144] implementiert und ermöglicht das Ausführen einer externen ausführbaren Datei innerhalb eines eigenen Prozesses. Für die Erkennung der Codeklone wird das Werkzeug CCfinder verwendet.

Um dieses und einige weitere Werkzeuge wie Python und Graphviz auszuführen, wurde die besagte Klasse erstellt und in der Implementierung genutzt. Python wird in Verbindung mit dem Werkzeug CCfinder ausgeführt. Dieses benötigt die Ausführung eines Python-Scripts, um die Daten der Codeklone korrekt zu speichern. Die zuvor erwähnten Werkzeuge werden mit dem erstellten Werkzeug IChecker ausgeliefert, sodass dieses korrekt funktioniert und verwendet werden kann.

Innerhalb der Implementierung wurde eine XSLT-Datei erstellt, die den Ressourcen hinzugefügt wurde. Werden die Indikatoren serialisiert und in eine XML-Datei gespeichert, wird gleichzeitig die XSLT-Ressource in eine Datei geschrieben, falls sie nicht bereits existiert. Durch die XSLT-Datei und einer Referenz innerhalb der XML-Datei können die Indikatoren und deren Details innerhalb eines Browsers dargestellt werden. Der XSLT-Prozessor des Browsers transferiert die XML-Datei in valides HTML und ermöglicht somit das Darstellen der Indikatoren und deren Details. Der Browser stellt mithilfe des HTMLs eine Liste von Indikatoren, welche pro Indikator den Indikatornamen, die Schlüsselbeschreibung sowie eine Tabelle von Details enthält, dar. Die Indikatoren besitzen verschiedene Arten von Details, weshalb die Tabelle mit den Indikatordetails je nach Indikator verschieden Spalten besitzt. In der Abbildung 5.4 ist ein Ausgabebeispiel für den PAKS-Indikator dargestellt. Das Beispiel zeigt, dass in den Klassen `CWIN32Util`, `CPowerManager` und `IPowerSyscall` eine Methode mit der Signatur `BatteryLevel()` existiert.

PAKS indicator for method signature BatteryLevel()

File	Class	Method
WIN32Util.h	CWIN32Util	BatteryLevel()
PowerManager.h	CPowerManager	BatteryLevel()
IPowerSyscall.h	IPowerSyscall	BatteryLevel()

Abbildung 5.4: Ausgabebeispiel

Die XSLT-Datei fügt zu den Indikatordetails des Weiteren noch die zugehörigen Indikatorabbildungen hinzu. Die Abbildungen werden nur für Indikatoren hinzugefügt für die PNG Dateien generiert werden. Die Beschreibung zu den generierten Indikatorabbildungen erfolgt im nächsten Abschnitt 5.4. Es wird außerdem dem HTML JavaScript hinzugefügt, um ein Entfernen-Button jedem Indikator hinzuzufügen. Der Button entfernt nach der Betätigung den zugehörigen Indikator aus der dargestellten Indikatorenliste.

5.4 Graphics

Für die Darstellung der Indikatoren BAT, MV, PVH und SKS werden mithilfe von Graphviz UML-Klassendiagramme als PNG-Dateien erzeugt, welche die Vererbungsstruktur darstellen. Für die Erstellung dieser Darstellungen wurde der Namensraum `Graphics` erzeugt. Dieser ist in der Abbildung 5.5 als UML-Klassendiagramm dargestellt. Der Namensraum `Graphics` beinhaltet zum einen die Klasse `ClassHierarchyDotFileCreator`, um die Vererbungsstruktur zu ermitteln und in eine dot-Datei zu speichern, und zum anderen die Klasse `DotExecutor`, um Graphviz auszuführen bzw. die erzeugte dot-Datei in eine PNG-Datei umzuwandeln.

Ein Objekt der Klasse `ClassHierarchyDotFileCreator` erhält bei der Erstellung eine Liste von Klassennamen. Für diese werden die Vererbungsstrukturen, in denen sich diese Klassen befinden, mithilfe der Sotograph-Datenbank ermittelt. Es wird für die ermittelten Vererbungsstrukturen eine dot-Datei erzeugt. Die übergebenen Klassennamen werden innerhalb der Vererbungsstruktur markiert. Bei der Übergabe einer Liste von `ClassDetail`-Objekten werden nur Teilvererbungshierarchien ermittelt. Die übergebenen Klassen werden als Superklassen angenommen und es werden nur deren Subklassen ermittelt, dies verhindert die Erstellung von sehr großen Hierarchien. Eine weitere Klasse in diesem Namensraum ist die Klasse `GraphicConstants`. Diese beinhaltet Konstanten wie Pfadangaben, die erst zur Laufzeit bestimmt werden können. Alle anderen Konstanten und Pfadangaben werden in den Ressourcen gespeichert.

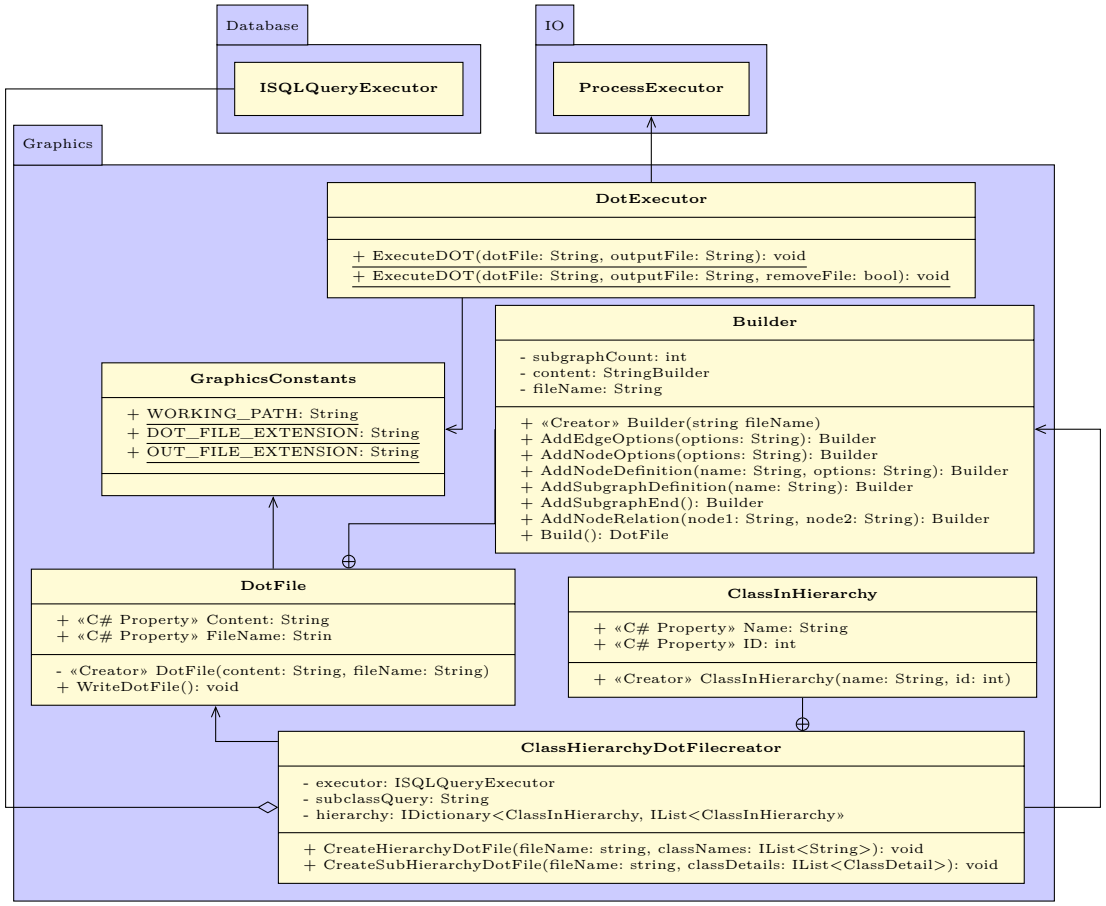


Abbildung 5.5: Graphics-Namensraum

Ein Objekt der Klasse `DotFile` enthält den Inhalt einer dot-Datei und deren Pfad. Des Weiteren beinhaltet diese Klasse eine innere Klasse namens `Builder`. Die `Builder` Klasse wurde nach dem gleichnamigen Pattern erstellt [vgl. Gam+94, S. 120]. Sie ermöglicht das schrittweise Erstellen eines `DotFile`-Objekts. Die erzeugten PNG-Dateien werden innerhalb des Ausgabeordners `graphics` gespeichert. Dieser Speicherort kann jedoch innerhalb der Einstellungsdatei geändert werden. Die PNG-Dateien werden vom erzeugten XSLT referenziert und bei der Darstellung mit den Indikatordetails angezeigt. In der Abbildung 5.6 ist ein Ausgabebeispielbild für den Indikator SKS dargestellt. Die Abbildungen der Indikatoren werden über der Tabelle mit den Indikatordetails angezeigt.

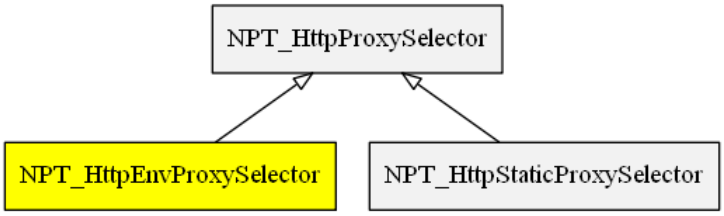


Abbildung 5.6: Ausgabebeispiel Graphviz

In dem Beispiel ist ein SKS-Indikator dargestellt, der aufzeigt, dass die Superklasse `NPT_HttpProxySelector` die gelb markierte Subklasse `NPT_HttpEnvProxySelector` kennt bzw. im Quellcode referenziert. In der Tabelle mit den Indikatordetails wird genauer beschrieben bzw. darauf hingewiesen, wo und wie genau die Superklasse die Subklasse referenziert bzw. die Superklasse die Subklasse nutzt.

5.5 Detection

Der Namensraum `Detection` wird in der Abbildung 5.7 dargestellt und besteht aus den Klassen, die zur Erkennung der Indikatoren dienen, einen Detektor für jeden Indikatorentyp. Für die Detektoren existiert eine abstrakte Klasse `Detector` von der jeder spezifische Detektor erbt. Die Klasse `Detector` implementiert die Grundfunktionalitäten, wie das Verwenden der Datenbankschnittstelle. Jede `Detector`-Subklasse wird in einem eigenen Thread ausgeführt, sodass die Erkennung parallel erfolgt. Für diese Anwendung existiert die Klasse `DetectorThread`. Durch die abstrakte Klasse `Detector` muss die Implementierung der `DetectorThread`-Klasse nicht auf die einzelnen `Detector`-Subklassen angepasst werden und ist somit nicht von Details abhängig. Innerhalb des Threads wird die abstrakte Methode `Detect()` der Klasse `Detector` aufgerufen, welche von jeder Subklasse geeignet implementiert ist. Die Methode `Detect()` liefert eine Liste von Indikatoren, die bei der Erkennung gefunden wurden. Nachdem diese von der jeweiligen `Detector`-Subklasse geliefert wurden, werden sie vom `DetectorThread` an den `IndikatorWriter` übergeben. Die Implementierung des `IndikatorWriter` ermöglicht das Speichern der gefundenen Indikatoren.

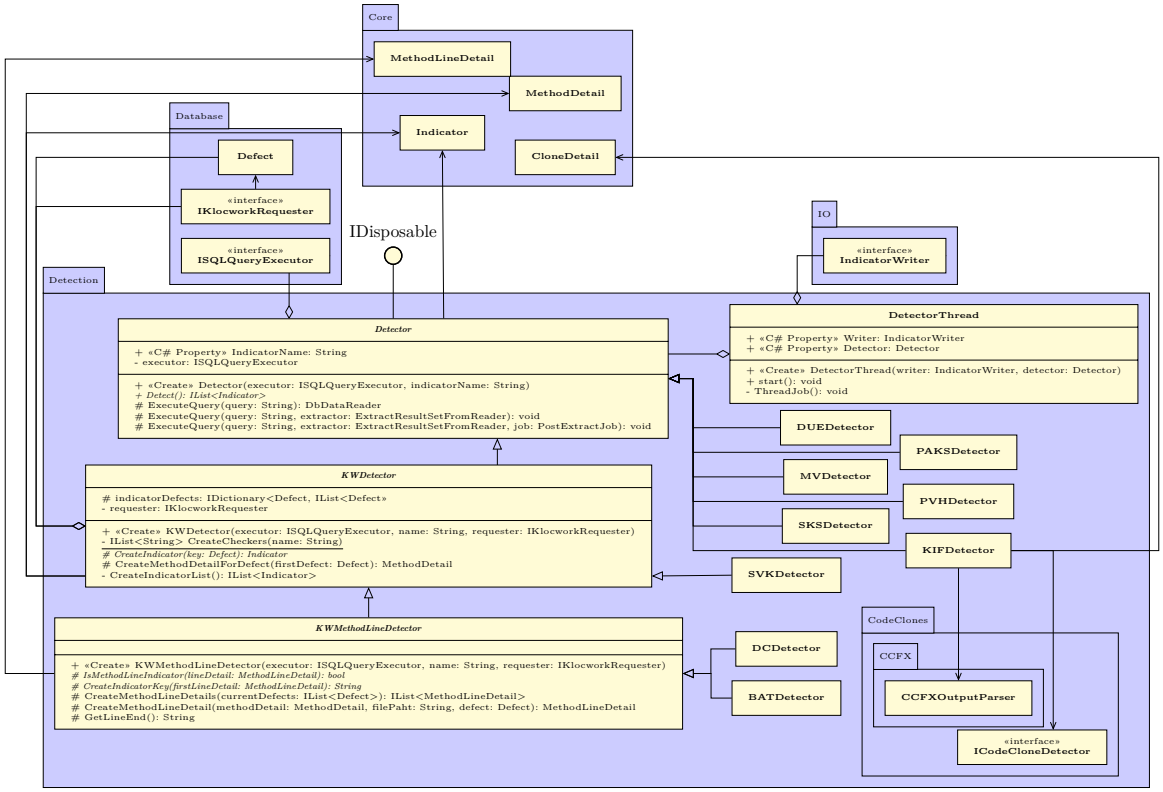


Abbildung 5.7: Detection-Namensraum

Durch die Menge an `Detector`-Subklassen wird für jede Subklasse ein Unterabschnitt erstellt, in dem die Implementierung beschrieben wird. Aus Platzgründen wurden einige Details der `Detector`-Subklassen nicht in der Abbildung 5.7 dargestellt.

5.5.1 KWDetector

Klocwork wird zur Erkennung einiger Indikatoren verwendet. Für diese Indikatoren wurden Klocwork-Checker erstellt, die mithilfe von Klocwork ausgeführt werden. Die Klasse `KWDetector` stellt eine abstrakte Superklasse für alle Detektoren dar, welche die Indikatoren anhand der Klocwork-Defekte bestimmen. `KWDetector` verwendet die `IKlocworkRequester`-Schnittstelle, um die Defekte abzufragen, und erbt von der abstrakten Klasse `Detector`. Die Klasse `KWDetector` implementiert die Methode `Detect` und deklariert eine neue abstrakte Methode `CreateIndicator` zum Erstellen eines Indikators, für einen gegebenen Defekt der von Klocwork erkannt wurde. Die Verwendung der `IKlocworkRequester`-Schnittstelle schafft eine Abstraktionsschicht. Die Klasse `KWDetector` und deren Subklassen sind unabhängig von der Umsetzung der Klocwork-Defekt-Abfragen.

Es wurden Checker für die Indikatoren BAT, DC und SVK erstellt. Abhängig vom Indikatornamen, den eine `Detector`-Subklasse beinhaltet, kann der dazugehörige Checker bestimmt werden. Die von der Schnittstelle zurückgegebenen `Defect`-Objekte werden einzeln, innerhalb der `Detect` Methodenimplementierung, der abstrakten Methode `CreateIndicator` übergeben. Die Subklassen implementieren diese abstrakte Methode und analysieren das übergebene `Defect`-Objekt. Mithilfe dieser Analyse werden die spezifischen Indikatoren erkannt. Beschreibt ein erkannter Defekt einen Indikator, wird mithilfe der Informationen des `Defect`-Objekts ein `Indicator`-Objekt erstellt und zurückgegeben.

5.5.2 KWMethodLineDetector

Wie in der Abbildung 5.7 dargestellt ist die Klasse `KWMethodLineDetector` eine abstrakte Subklasse der Klasse `KWDetector`. Sie wurde für die weitere Analyse der Klocwork-Defekte erstellt. Die Klasse `KWMethodLineDetector` ermöglicht das Erstellen von `Indicator`-Objekten, welche `MethodLineDetail`-Objekte besitzen. Diese sind notwendig, um auf bestimmte Zeilen und Zeileninhalte zu verweisen.

Die Subklassen `BATDetector` und `DCDetector` besaßen einige gemeinsame Funktionalitäten, weshalb die Erstellung der Klasse `KWMethodLineDetector`, als neue Superklasse dieser Klassen, notwendig war. Durch diese Umsetzung wurde doppelter Quellcode verhindert. Die abstrakte Klasse `KWMethodLineDetector` deklariert die abstrakten Methoden `IsMethodLineIndicator` und `CreateIndicatorKey`. Die genannten Methoden werden von den Subklassen `BATDetector` und `DCDetector` zur korrekten Erkennung und Erstellung der jeweiligen Indikatoren implementiert. In der Implementierung der Methode `CreateIndicator` werden die zuvor genannten abstrakten Methoden aufgerufen. Die Methode `IsMethodLineIndicator` erhält als Parameter ein `MethodLineDetail`-Objekt. Die Subklassenimplementierung dieser Methode gibt den Wert `true` zurück, wenn anhand der gegebenen Informationen ein Indikator erkannt wurde. In allen anderen Fällen wird `false` zurückgegeben. Die Kurzbeschreibung eines erkannten Indikators wird mithilfe der Subklassenimplementierung der Methode `CreateIndicatorKey` erstellt. Der Methode `CreateIndicatorKey` wird das erste `MethodLineDetail`-Objekt, welches zu dem Indikator gehört, übergeben.

5.5.3

BATDetector

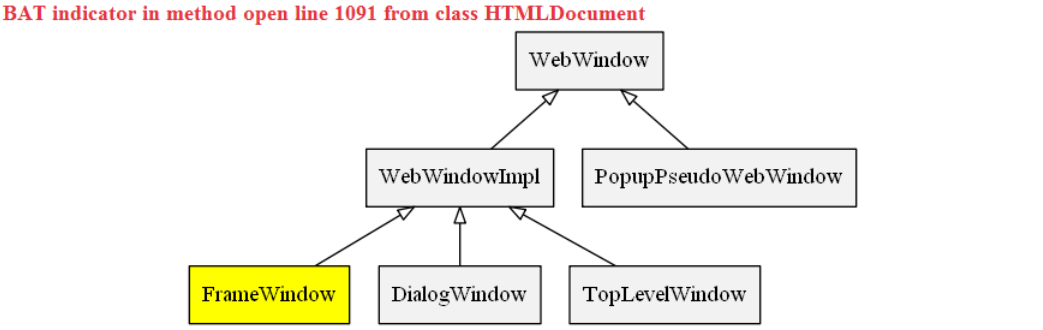
Zur Erkennung der BAT-Indikatoren wurden zwei Klocwork-Checker erstellt. Diese erkennen die Verwendung von `instanceof` und `dynamic_cast` Operatoren. Zur Erklärung dieser Operatoren siehe Abschnitt 2.3. Bei der Verwendung der besagten Operatoren ist die Feststellung eines BAT-Indikator zu meist eindeutig und kann ohne viel Aufwand durchgeführt werden. Die Erkennung von Typcode und deren Bestimmung, ob es sich dabei um einen BAT-Indikator handelt, ist durch Mangel an Zeit und Erkennungsleistung innerhalb dieser Arbeit nicht möglich. Aus diesen Gründen muss die Erkennung von Typcode in den Ausblick gestellt werden.

Die Klasse `BATDetector` stellt, wie in Abbildung 5.7 dargestellt, eine Subklasse der Klasse `KWMethodLineDetector` dar. Die abstrakten Methoden werden implementiert und die Methode `CreateMethodLineDetails` überschrieben. Die Überschreibung der Methode `CreateMethodLineDetails` ermöglicht das Speichern der Klassennamen, die innerhalb eines BAT-Indikators geprüft wurden. In dem Beispiel 5.1 ist ein regulärer Ausdruck dargestellt. Dieser wurde entwickelt, um die geprüften Klassennamen aus einer Codezeile zu extrahieren.

Beispiel 5.1: Klassennamen-Regex

```
((instanceof)|(dynamic_cast)|(DYNAMIC_CAST))\s*[\(<]?\s*(\w+)
```

Mithilfe der extrahierten Klassennamen kann für den jeweiligen BAT-Indikator ein geeignetes UML-Klassendiagramm für die Ausgabe erstellt werden. Die Klasse `ClassHierarchyDotFileCreator` wird zur Ermittlung der Vererbungsstruktur und Erzeugung des Klassendiagramms verwendet. Zur Erklärung dieser Klasse siehe Abschnitt 5.4. In der Abbildung 5.8 ist die Ausgabe für einen BAT Indikator dargestellt, sie wurde bereits teilweise in den Abschnitten 5.3 und 5.4 beschrieben. Aus diesem Grund wird diese nicht detailliert erläutert, sondern nur auf die Beispielausgabe eingegangen.



File	Class	Method	Line	LineContent
HTMLDocument.java	HTMLDocument	open (java.lang.String, java.lang.Object, java.lang.Object, java.lang.Object)	1091	`if (getWindow().getWebWindow() instanceof FrameWindow\n`

Abbildung 5.8: BAT-Indikator Ausgabe

Der in der Abbildung 5.8 dargestellte BAT-Indikator wurde innerhalb der Methode `open` in der Klasse `HTMLDocument` erkannt. Ein BAT-Indikator wird pro Methode innerhalb einer Klasse erstellt. Würden mehrere Typüberprüfungen in einer Methode vorkommen, würden diese in der Tabelle jeweils eine Zeile erhalten. Die Tabelle beinhaltet die Angaben zur Datei, Klasse, Methode, Zeile und zu dem Zeileninhalt. Auf die Dateien wird mithilfe einer File-URI verwiesen.

Über der beschriebenen Tabelle befindet sich das generierte UML-Klassendiagramm. Die geprüften Klassen innerhalb der Vererbungshierarchie werden gelb markiert. Diese Markierungen können mit den Zeileninhalten verglichen werden. In dem UML-Klassendiagramm ist zu erkennen, dass ein Objekt auf die Klasse `FrameWindow` geprüft wird. Es findet keine weitere Typüberprüfung statt. Die Methode `open` führt weitere Funktionalität bei einem Objekt der Instanz `FrameWindow` aus. Die Methode ist abhängig von Details und muss gegebenenfalls bei weiteren Subklassen angepasst werden.

5.5.4

DCDetector

Ein DC-Indikator kann mithilfe der Klasse `DCDetector` erkannt werden. Diese Klasse stellt eine weitere Subklasse der abstrakten Klasse `KWMethodLineDetail` dar. Es wurde jeweils ein Klocwork-Checker für Java und C++ verfasst. Der DC-Klocwork-Checker für Java erkennt die casting-Operationen, durch den Klammeroperator sowie die Verwendung der `cast`-Methode der Klasse `Class`. Konvertierungen von primitiven Datentypen können bereits durch den Klocwork-Checker ausgeschlossen werden. Der C++-Klammeroperator sowie die C++ spezifischen casting-Operatoren (`static_cast, dynamic_cast` etc.) können mithilfe des erstellten C++ Klocwork-Checker für DC-Indikatoren erkannt werden. Ähnlich dem Java-Checker können primitive Konvertierungen ausgeschlossen werden. Typen die durch `typedefs` definiert wurden können bei der Erkennung nicht durch den Klocwork-Checker ausgeschlossen werden. Sie werden als Defekte erkannt.

In dem Beispiel 5.2 ist ein weiterer regulärer Ausdruck dargestellt. Dieser Ausdruck ermöglicht, innerhalb einer Codezeile, das Erkennen des Typs in den gewandelt werden soll, vorausgesetzt es ist ein DC-Indikator vorhanden. Dieser extrahierte Typ wird zur weiteren Analyse verwendet. Bspw. wird geprüft, ob der Typ in den gewandelt werden soll ein Synonym für ein Basistyp in C++ darstellt. Ist dies der Fall, wird diese Konvertierung ignoriert und es wird kein Indikator erkannt bzw. angezeigt.

Beispiel 5.2: DC-Regex

```
([(<])([\w*\.]+[<>?\w\s]*)([>)])([\s(]*)([\w*&]+)(([)]*[\.])([\w]+))?
```

[1]

Die Sotograph-Datenbank beinhaltet eine Tabelle `typedefs`, welche alle `typedefs` des Softwaresystems enthält. Es wurde eine Abfrage erzeugt, die alle Namen der `typedefs` liefert, die Synonyme für Basistypen in C++ sind. Mithilfe dieser Abfrage können die Namen gespeichert und die Typkonvertierungen in diese Typen ausgeklammert werden. Vordefinierte `typedefs` sind nicht innerhalb der Sotograph-Datenbank vorhanden. Aus diesem Grund wurde eine weitere Ressourcendatei erzeugt, die bekannte `typedefs` beinhaltet. Diese Liste von `typedefs` wird außerdem verwendet, um bestimmte falsch-Erkennungen auszuschließen. Für jede Methode, in denen Defekte erkannt wurden, wird ein DC-`Indicator`-Objekt erstellt. Die Informationen der Klocwork-Defekte werden geeignet in Indikatordetails umgewandelt. Dabei wird jeder Defekt betrachtet.

Für die Erkennung der Indikatoren wurde folgende Heuristik definiert: jede explizite casting-Operation, die keinen primitiven Typ wandelt, stellt entweder einen `downcast` oder `sidecast` dar. Das Vorkommen eines `upcast` wird ausgeschlossen, da diese Konvertierung jede betrachtete Programmiersprache implizit beherrscht, sodass keine explizite Konvertierung notwendig ist. Mithilfe dieser definierten Heuristik werden die erkannten Klocwork-Defekte, nach Aussortierung der `typedefs`-Konvertierungen, direkt als DC-Indikator erkannt. `Sidecasts` werden dabei genauso als DC-Indikator erkannt, da sie die gleichen Problematiken beinhalten. Durch diese definierte Heuristik kann einiges an Arbeits- und Rechenaufwand eingespart werden. Anderenfalls müsste geprüft werden, welchen Typ das Objekt besitzt, welches umgewandelt werden soll und in welchen Typ gewandelt werden soll. Nach Feststellung dieser Typen muss geprüft werden, in welcher Relation diese zu einander stehen.

In der Abbildung 5.9 ist die Ausgabe eines erkannten DC-Indikators dargestellt. Dort ist zu erkennen, dass in der Zeile 109 der Methode `staticThread` der Klasse `CThread` ein DC-Indikator erkannt wurde. Der Zeiger `data` wird in den Zeigertyp `CThread*` mithilfe des Klammeroperators konvertiert. Ein casting mittels des Klammeroperators, auch C-Style casting genannt, kann einen beliebigen Zeiger in einen beliebigen Zeigertyp konvertieren. Ist dieser Zeiger nicht mit dem Zeigertyp kompatibel kann dies zu Laufzeitfehlern oder undefinierten Verhalten führen [vgl. Str13, S. 302].

DC indicator in method staticThread from class CThread

File	Class	Method	Line	LineContent
Thread.cpp	CThread	staticThread(void *)	109	`CThread* pThread = (CThread*)(data);\n`

Abbildung 5.9: DC-Indikator Ausgabe

5.5.5 SVKDetector

Die Klasse `SVKDetector` erbt, wie in der Abbildung 5.7 dargestellt, von der Klasse `KWDetector` und sollte die gleichnamigen SVK-Indikatoren erkennen. Es wurden zur Erkennung dieser Indikatoren zwei Klocwork-Checker verfasst. Jeweils einer für die Sprachen Java und C++. Klocwork erkennt mithilfe dieser Checker Defekte, die Methoden beschreiben, die als erste Anweisung eine Rückanweisung besitzen. Rückanweisungen mit Methodenaufrufen werden dabei ignoriert. Die erkannten Defekte werden analysiert, sodass erkannte `Getter`-Methoden ausgeschlossen werden können. Infolgedessen müsste geprüft werden, ob das zurückgegebene Objekt eine Konstante darstellt. Außerdem, ob Geschwisterklassen existieren, die eine gleiche Methode besitzen, die nur eine Konstante liefert. Eine weitere Bedingung des SKS-Indikators ist, dass die Klasse keine weitere Funktionalität besitzt, nur diese Methode zum Liefern der Konstanten. Die Überprüfung der vorhandenen Methoden, des zurückgegebenen Objekts und der Existenz von Geschwisterklassen übersteigt den Aufwand-Nutzen-Faktor für diesen Indikator. Der SVK Indikator beschreibt einen Code Smell, stellt jedoch keine weiteren bedenklichen Problematiken dar. Aus diesem und aus Grund des zeitlichen Mangels, wird die Erkennung des SVK-Indikators in den Ausblick gestellt.

5.5.6 DUEDetector

Die Klasse `DUEDetector` erbt von der abstrakten Klasse `Detector` und ermöglicht das Erkennen von DÜ-Indikatoren. Die Erkennung wird mithilfe der Sotograph-Datenbank und SQL-Abfragen durchgeführt. Der DÜ-Indikator beschriebt die Delegation von einer Klasse an die komplette Schnittstelle einer anderen Klasse. Die formulierte SQL-Abfrage liefert die `symbol`-ID's, der delegierenden und delegierten Klassen. Die Anfrage betrachtet dabei, dass die delegierende Klasse an alle Methoden der delegierten Klasse delegiert bzw. referenziert. Die Methodenreferenzen sind in der Tabelle `methodreferencesflat` der Sotograph-Datenbank gespeichert. Klassen, die bereits innerhalb einer Vererbungshierarchie existieren, werden bei der Abfrage ausgeschlossen. Vererbungshierarchieinformationen sind in der Tabelle `inheritings` gespeichert. Die ermittelten und zurückgegebenen Daten der beschriebenen SQL-Abfrage stellen DÜ-Indikatoren, innerhalb des analysierten Softwaresystems, dar. Aus den Daten werden `Indicator`-Objekte erzeugt, die Informationen zu den delegierten und delegierenden Klassen beinhalten. Ein Indikator beschreibt jeweils zwei Klassen, die delegierende und die delegierte. Die Details des Indikators beinhalten jede Methode, die in der Delegation beteiligt ist.

In der Abbildung 5.10 wird die Ausgabe eines DÜ-Indikators dargestellt. In dieser ist zu erkennen, dass die Klasse `UniversalTersePrinter<char*>` mit der Methode `Print` auf die einzige Methode `Print` der Klasse `UniversalTersePrinter<char const *>` delegiert. Die Methoden unterscheiden sich lediglich in den Parametertypen. Die Ausgabe stellt alle notwendigen Details des DÜ-Indikators dar.

Delegator: 1310201 delegates to 1310199

File	Class	Method
gtest-printers.h	UniversalTersePrinter<char*>	Print(char*, std::basic_ostream<char, std::char_traits<char>> *)
gtest-printers.h	UniversalTersePrinter<char const *>	Print(char const *, std::basic_ostream<char, std::char_traits<char>> *)

Abbildung 5.10: DÜ-Indikator Ausgabe

5.5.7

KIFDetector

Für die Erkennung der KIF-Indikatoren müssen Codeklone erkannt und in verschiedenen Klassen identifiziert werden. Für diese Erkennung wird das Werkzeug CCfinder verwendet. Um das Werkzeug auszuführen, die Daten auszulesen und weiter zu verarbeiten, wurde ein neuer Unternamensraum `CodeClones` im `Detection`-Namensraum erstellt. Dieser Namensraum ist in der Abbildung 5.11 dargestellt und beinhaltet die Schnittstelle `ICodeCloneDetector`, die Implementierung der Schnittstelle `CodeCloneDetector` und die `Exception`-Subklasse `CCException`.

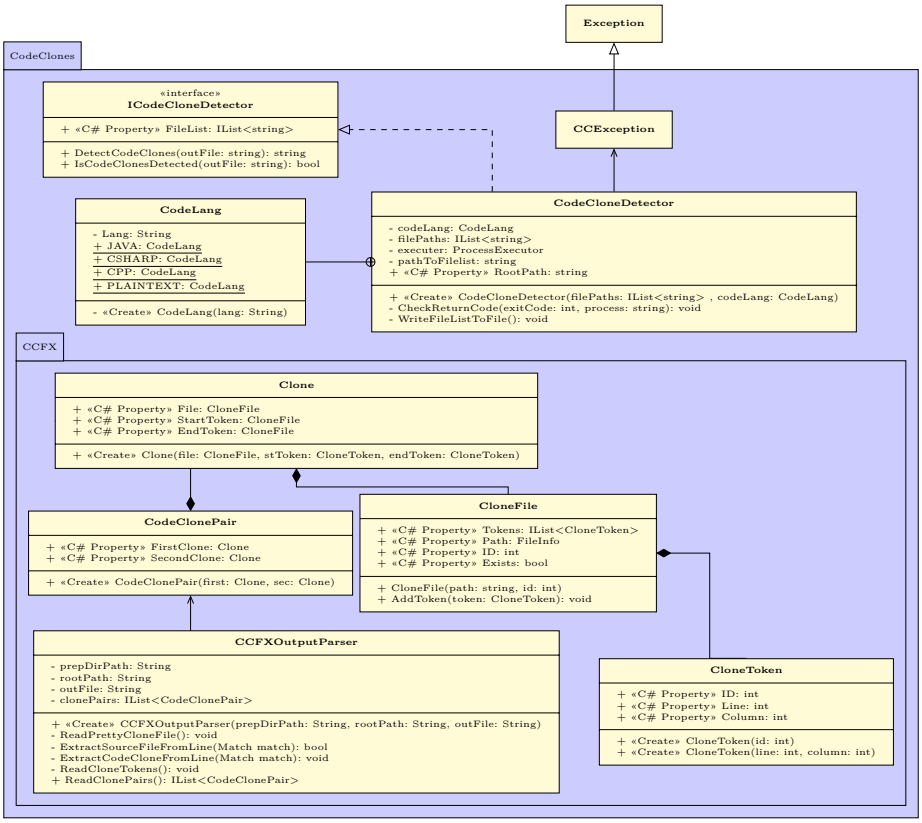


Abbildung 5.11: CodeClones-Namesensraum

Die Klasse `KIFDetector` stellt eine weitere Subklasse der abstrakten Klasse `Detector` dar und ermöglicht das Erkennen der KIF-Indikatoren. Die Klasse verwendetet, zum bestimmten der Codeklone, die `ICodeCloneDetector`-Schnittstelle und eine SQL-Abfrage an die Sotograph-Datenbank, um die jeweiligen Klassen korrekt zu bestimmen. Die Klasse `CodeCloneDetector` führt das Werkzeug CCfinder aus, um die besagten Codeklone zu erkennen. Erkannte Klone werden in einer Ausgabedatei gespeichert. Mithilfe eines von CCfinder mitgelieferten Python-Scripts, kann die Ausgabedatei verfeinert und in ein von CCfinder definiertes verarbeitbares Format konvertiert werden. Der Namensraum `CodeClones` besitzt einen weiteren Unternamensraum `CCFX`. Dieser beinhaltet Klassen, die zum Auslesen der Ausgabedaten verwendet werden. Codeklone werden durch Objekte der Klasse `Clone` dargestellt und Klonpaare durch ein Objekt der Klasse `CodeClonePair`. Die Objekte beinhalteten Informationen zu den jeweiligen Klonen. Es wurde teilweise Quellcode aus dem Como-Werkzeug wiederverwendet. Der größte Teil wurde angepasst und verbessert. Bspw. wurden alle Konstanten in Ressourcen ausgelagert. Pfadangaben wurden in die Einstellungsdatei ausgelagert, sodass diese besser angepasst werden können. Die Klasse `KIFDetector` ermittelt nach der Ausführung des CCfinder-Werkzeugs die Klassen, in denen die Klone erkannt wurden. Die Ermittlung erfolgt mithilfe einer SQL-Abfrage an die Sotograph-Datenbank. Die Klone werden durch Dateinamen und Zeilennummern von-bis identifiziert. Mit diesen Angaben können die dazugehörigen Klassen bestimmt werden.

In der Abbildung 5.12 wird die Ausgabe für einen KIF-Indikator dargestellt. In dieser Darstellung ist zu erkennen, dass ein Klon innerhalb der Klasse `NPT_XmlTagFinder` und `PLT_XmlHelper` existiert. Zur Analyse und Optimierung muss die Methode aus Abschnitt 3.3 angewendet werden. Als Ausblick ist es möglich den Codeklon direkt anzuzeigen, innerhalb der HTML-Ausgabe und ein Klassendiagramm zu erstellen, in welchem die Beziehungen der Klassen zu einander dargestellt werden.

Clone class: NPT_XmlTagFinder line 145-162 with class: PLT_XmlHelper line 208-226

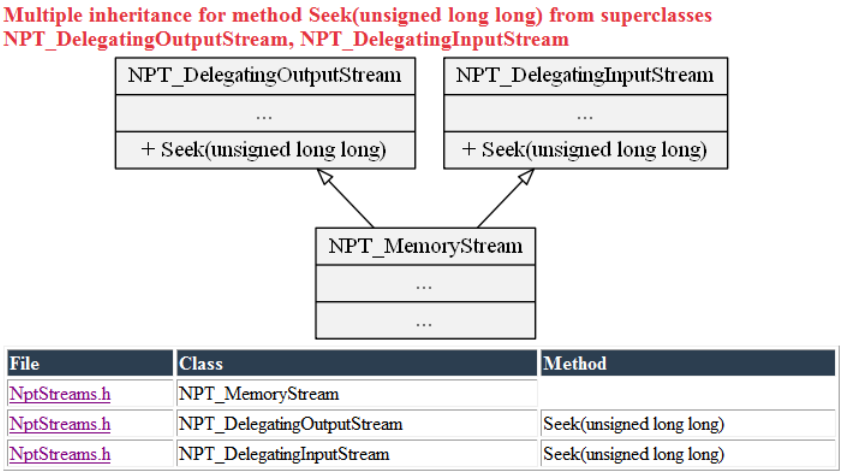
File	Class	StartLine	EndLine
NptXml.cpp	NPT_XmlTagFinder	145	162
PltUtilities.h	PLT_XmlHelper	208	226

Abbildung 5.12: KIF-Indikator Ausgabe

5.5.8 MVDetector

Zur Erkennung der MV-Indikatoren wird die Klasse `MVDetector` verwendet. Diese stellt, wie in der Abbildung 5.7 dargestellt, eine direkte Subklasse der Klasse `Detector` dar. Es werden zur Erkennung der MV-Indikatoren zwei SQL-Abfragen, an die Sotograph-Datenbank, verwendet. Die erste Abfrage liefert Daten von Subklassen, die mehr als eine Superklasse oder Schnittstelle besitzen. Diese Subklassen besitzen somit eine Mehrfachvererbung bzw. Mehrfachimplementierung. Zu den Informationen der Subklassen werden die Daten der dazugehörigen Superklassen geliefert. Die Superklassen, der ermittelten Subklassen, werden auf Methoden mit gleicher Signatur überprüft. Die Überprüfung wird mithilfe der zweiten SQL-Abfrage durchgeführt. Dabei werden jeweils zwei Superklassen verglichen. Die SQL-Abfrage liefert für die Superklassen mit gleicher Methodensignatur die spezifische Methodensignatur. Mithilfe dieser Informationen wird ein `Indicator`-Objekt erzeugt, welches Informationen zu der Subklassen, zu den Superklassen sowie zu der Methodensignatur enthält.

In der Abbildung 5.13 ist die Ausgabe eines MV-Indikators dargestellt. Ähnlich dem BAT-Indikator wird die Vererbung mithilfe eines UML-Klassendiagramms dargestellt. Die Darstellung der Vererbungsstruktur beschränkt sich dabei nur auf die Subklasse und die direkten Superklassen. Des Weiteren wird die gleiche Methodensignatur in den Superklassen angezeigt.



File	Class	Method
NptStreams.h	NPT_MemoryStream	
NptStreams.h	NPT_DelegatingOutputStream	Seek(unsigned long long)
NptStreams.h	NPT_DelegatingInputStream	Seek(unsigned long long)

Abbildung 5.13: MV-Indikator Ausgabe

In dem Beispiel, aus der Abbildung 5.13, ist zu erkennen, dass die Subklasse `NPT_MemoryStream` von den Klassen `NPT_DelegatingOutputStream` und `NPT_DelegatingInputStream` erbt. Die Superklassen besitzen eine Methode mit gleicher Signatur `Seek(unsigned long long)`. Diese Konstellation führt zu einem Compilerfehler, falls die Methode an einer Subklasseninstanz aufgerufen wird und diese Methode nicht von der Subklasse überschrieben wurde. Überschreibt die Subklasse diese Methode, stellt das ein logisches Problem dar. Bei der Subklassenimplementierung wird eines der beiden Schnittstellenziele verfolgt. Durch diese Implementierung kann jedoch die Funktionalität für die Schnittstelle der anderen Superklasse nicht mehr gewährleistet werden. Die Subklasse verhält sich anders als die Superklasse. Dies ist vor allem problematisch, wenn eine Instanz der Subklasse übergeben wird und die Funktionalität der Superklasse erwartet wird. Das gleiche gilt für mehrfache Schnittstellenimplementierung.

5.5.9 PAKSDetector

Die Klasse `PAKSDetector` bestimmt mithilfe einer SQL-Abfrage an die Sotograph-Datenbank die PAKS-Indikatoren eines objektorientierten Softwareprojekts. Sie stellt eine weitere Subklasse der Klasse `Detector` dar. Bei der Bestimmung werden alle Methodensignaturen betrachtet, die ein Vorkommen ≥ 2 besitzen. Geerbte Methoden, Konstruktoren sowie anonyme Klassen und Methoden werden dabei ausgeschlossen. Für die erkannten Methodensignaturen werden die dazugehörigen Klassen bestimmt und ein `Indicator`-Objekt erzeugt. Das Objekt besitzt Informationen zu der mehrmals vorkommenden Methodensignatur sowie zu den dazugehörigen Klassen. In der Abbildung 5.14 ist die Ausgabe für einen PAKS-Indikator dargestellt. Das Beispiel zeigt die Klassen `CGUIWindowManager` und `CGUIWindow`, die eine Methode mit gleicher Signatur besitzen. Die Methode `AfterRender` besitzt, innerhalb der dargestellten Klassen, den gleichen Verwendungszweck. Es fehlt in diesem Fall an einer Abstraktionsschicht, die das Verwenden dieser Methode durch eine Schnittstelle ermöglicht.

PAKS indicator for method signature AfterRender()

File	Class	Method
GUIWindowManager.h	CGUIWindowManager	AfterRender()
GUIWindow.h	CGUIWindow	AfterRender()

Abbildung 5.14: PAKS-Indikator Ausgabe

5.5.10

PVHDetector

Zur Erkennung der PVH-Indikatoren wurde eine weitere Subklasse der Klasse `Detector` erstellt. Die Subklasse `PVHDetector` verwendet, zur Erkennung der PVH-Indikatoren, eine SQL-Abfrage an die Sotograph-Datenbank. Die Ergebnisse dieser Abfrage stellen valide erkannte PVH-Indikatoren dar. Aus den ermittelten Daten werden direkt `Indicator`-Objekte erzeugt. Die Informationen eines dieser Objekte beinhaltet die zwei Klassen, die eine parallele Vererbungshierarchie beschreiben.

Die SQL-Abfrage verwendet die in Abschnitt 3.5 beschriebene Erkennung. Es werden die Klassennamen überprüft, ob diese in anderen Klassennamen als Präfixe auftauchen. Bei der Erkennung zweier Klassen, in der ein Klassenname im anderen vorkommt, werden die Namen der Subklassen dieser Klassen geprüft. Die Überprüfung gleicht sich mit der zuvor beschriebenen. Besitzen die Subklassen genauso Namen die in der anderen Vererbungshierarchie vorkommen werden diese Vererbungshierarchien als PVH-Indikator erkannt. In der Abbildung 5.15 ist die Ausgabe für einen PVH-Indikator dargestellt. Für die PVH-Indikatoren werden die erkannten parallelen Vererbungshierarchien dargestellt. Die Superklassen werden dabei gelb markiert. Die Darstellung ermöglicht das Nachvollziehen des erkannten PVH-Indikators und somit der vorhandenen parallelen Vererbungsstruktur.

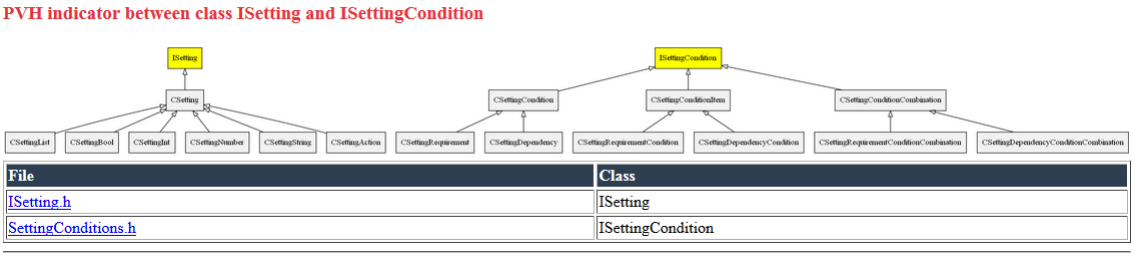


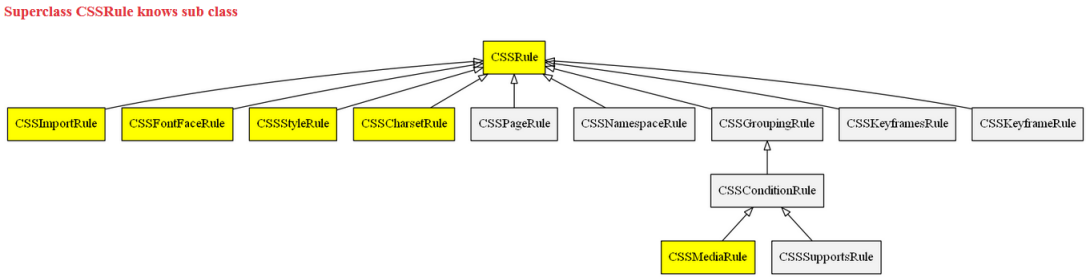
Abbildung 5.15: PVH-Indikator Ausgabe

5.5.11

SKSDetector

Innerhalb der Sotograph-Datenbank existiert eine Metrik namens `PckgClassKnowingDerievedRule`, welche die Anzahl an Superklassenreferenzen auf Subklassen beinhaltet. Diese Metrik wird bei der Analyse des Softwareprojekts durch Sotograph erzeugt. Die Anzahl der Referenzen bietet jedoch keine genaue Angabe der Klassen und Zeilen, in denen eine Subklasse verwendet oder referenziert wird. Aus diesem Grund wurde eine weitere Subklasse der Klasse `Detector` erzeugt. Diese Subklasse, genannt `SKSDetector`, ermöglicht das Erkennen von SKS-Indikatoren und erzeugen von `Indicator`-Objekten mit allen notwendigen Informationen. Die erwähnte Metrik wurde zur Abgrenzung und Überprüfung der Korrektheit, der erstellten `SKSDetector`-Klasse, verwendet. Für die Erkennung durch die `SKSDetector`-Klasse wurde eine weitere SQL-Abfrage verfasst, die an der Sotograph-Datenbank ausgeführt wird. Die Abfrage prüft, ob Einträge in der Tabelle `symbolreferences` existieren, die darauf hinweisen, dass eine Superklasse auf eine Subklasse referenziert. Es wird ein `Indicator`-Objekt pro erkannter Superklasse, die auf eine oder mehreren ihrer Subklassen verweist, erzeugt. Das erzeugt Objekt beinhaltet Informationen zur Superklasse und allen getätigten Subklassenreferenzen, insbesondere Methodenname, Zeilennummer und entsprechender Zeileninhalt der erkannten Referenz.

In der Abbildung 5.16 ist die Ausgabe für einen erkannten SKS-Indikator dargestellt. In der Detailstabelle werden die Methoden sowie die Codezeilen angezeigt, die auf Subklassen referenzieren. Es wird außerdem die vorhandene Vererbungshierarchie dargestellt. Dies verdeutlicht die existierenden Referenzen der Superklasse auf die Subklassen. Die Superklasse und deren referenzierten Klassen sind in der Darstellung gelb markiert. In der Abbildung 5.16 ist zu erkennen, dass die Superklasse `CSSRule` einige ihrer Subklassen referenziert bzw. verwendet. In diesem Fall ist eine statische Methode `Create` implementiert, die abhängig vom übergebenen Parameter ein Objekt einer Subklasse zurückgeben soll.



File	Class	Method	Line	LineContent
CSSRule.java	CSSRule	create(com.gargoylesoftware.htmlunit.javascript.host.css.CSSStyleSheet, org.w3c.dom.css.CSSRule)	97	`return new CSSImportRule(stylesheet, (org.w3c.dom.css.CSSImportRule) rule);\n`
CSSRule.java	CSSRule	create(com.gargoylesoftware.htmlunit.javascript.host.css.CSSStyleSheet, org.w3c.dom.css.CSSRule)	101	`return new CSSMediaRule(stylesheet, (org.w3c.dom.css.CSSMediaRule) rule);\n`
CSSRule.java	CSSRule	create(com.gargoylesoftware.htmlunit.javascript.host.css.CSSStyleSheet, org.w3c.dom.css.CSSRule)	103	`return new CSSFontFaceRule(stylesheet, (org.w3c.dom.css.CSSFontFaceRule) rule);\n`
CSSRule.java	CSSRule	create(com.gargoylesoftware.htmlunit.javascript.host.css.CSSStyleSheet, org.w3c.dom.css.CSSRule)	95	`return new CSSStyleRule(stylesheet, (org.w3c.dom.css.CSSStyleRule) rule);\n`
CSSRule.java	CSSRule	create(com.gargoylesoftware.htmlunit.javascript.host.css.CSSStyleSheet, org.w3c.dom.css.CSSRule)	99	`return new CSSCharsetRule(stylesheet, (org.w3c.dom.css.CSSCharsetRule) rule);\n`

Abbildung 5.16: SKS-Indikator Ausgabe

5.6 ICCore

Der Namensraum `ICCore` stellt den Hauptnamensraum dar. Er beinhaltet alle bereits beschriebenen Namensräume und die Hauptklassen zum Ausführen des Werkzeugs IChecker . In der Abbildung 5.17 ist der Namensraum `ICCore` dargestellt. Aus Platzgründen und der Übersichtlichkeit halber werden nur, die von der Klasse `MainIndicatorDetector` referenzierten, Klassen und Schnittstellen dargestellt. Die Klasse `MainIndicatorDetector` verbindet bzw. bringt alle Funktionalitäten zusammen. Sie besitzt eine einzige öffentliche Methode `StartDetection`. Diese Methode erzeugt Objekte aller vorhandenen `Detector`-Subklassen und führt diese innerhalb eines eigenen Thread, mithilfe der Klasse `DetectorThread`, aus. Den Objekten der `Detector`-Subklassen werden notwendige Objekte zur Kommunikation mit der Sotograph- oder Klocwork-Datenbank im Konstruktor übergeben. Die übergebenen Objekte zur Kommunikation sind vom Typ `ISQLQueryExecutorImpl` bzw. für Klocwork vom Typ `IKlocworkRequester`. Die Erzeugung dieser Objekte erfolgt mithilfe der Datenbankinformationen, die in der Einstellungsdatei gespeichert sind. Die Klasse `PostgreDatabaseConnection`, aus dem `DatabaseCore`-Namensraum, stammt aus einem Como-Teilprojekt und wird zur Verbindung mit einer PostgreSQL-Datenbank wiederverwendet. Es wird jeweils ein Objekt der `PostgreDatabaseConnection`-Klasse einem Objekt der Klasse `SQLQueryExecutorImpl` bei der Erstellung übergeben. Die Schnittstellenimplementierungen `SQLQueryExecutorImpl` sowie `KlocworkRequesterImpl` können beliebig durch andere Implementierungen der Schnittstellen ersetzt werden, dies steigert die Erweiterbarkeit des Systems. Die `DetectorThread`-Objekte erhalten die erstellten `Detector`-Subklassenobjekte und ein Objekt der Klasse `XMLIndicatorWriter`, zum Serialisieren der erkannten Indikatoren. Die Implementierung der `IndicatorWriter`-Schnittstelle kann gleichwohl ersetzt werden. Eine Ersetzung der Klasse `XMLIndicatorWriter` ermöglicht die Veränderung der Abspeicherung von Indikatorinformationen.

Die Klasse `Program` beinhaltet die `Main`-Methode der Anwendung. Innerhalb dieser wird ein `MainIndicatorDetector`-Objekt erstellt und die Erkennung, durch Aufruf der Methode `StartDetection`, gestartet.

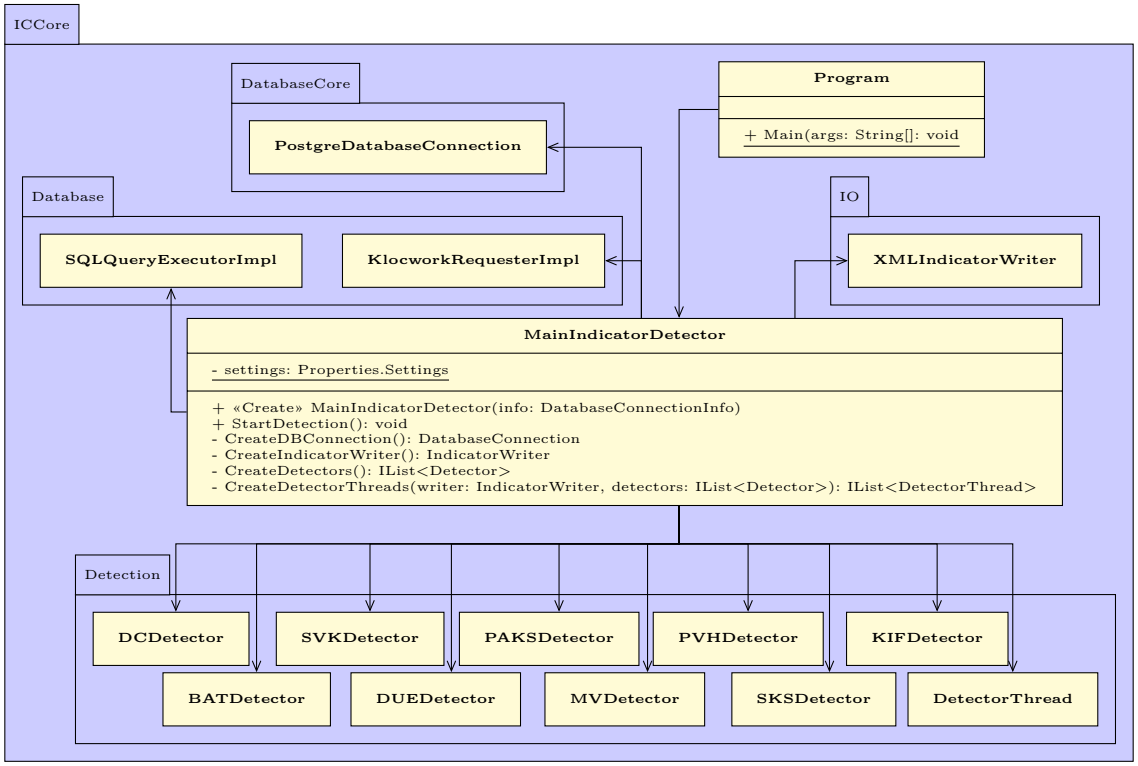


Abbildung 5.17: ICCore-Namensraum

Evaluation
6

Im folgenden Kapitel wird eine Evaluation der in Kapitel 3 entwickelten Methode und definierten Indikatoren durchgeführt. Mithilfe dieser Evaluation kann das Praxisvorkommen dieser Indikatoren und die Methodenanwendung eingeschätzt werden. Die Evaluation wird anhand der Open Source Projekte XBMC und HtmlUnit durchgeführt. Das Werkzeug IChecker wird durch Ausführung dieser Evaluation eingehend getestet und verbessert.

Es wurden für das Projekt XBMC 512 und für das Projekt HtmlUnit 4066 Indikatoren erkannt. Diese große Anzahl an Indikatoren kann innerhalb dieser Arbeit nur teilweise im Detail evaluiert und beschrieben werden. Es wird pro Projekt ein Indikator stichprobenartig ausgewählt und im Detail evaluiert und beschrieben. Die weiteren Indikatoren werden weitestgehend geprüft und evaluiert. Es kann anhand der überprüften Indikatoren eine eingehend kritische Würdigung der entwickelten Methode und des Praxisvorkommen dieser Indikatoren verfasst werden.

6.1

BAT

6.1.1

XBMC

Im Projekt `XBMC` wurde ein BAT-Indikator erkannt. Dieser BAT Indikator wurde innerhalb des Konstruktors der Klasse `NPT_SimpleMessageCapsule` erkannt und ist in dem Beispiel 6.1 dargestellt. Mithilfe des `dynamic_cast` wird das Attribut `m_Handler` vom Typ `NPT_MessageHandler` in den Typ `NPT_MessageHandlerProxy` konvertiert. Nach der Konvertierung folgt eine Überprüfung, ob die Konvertierung erfolgreich war. Ist dies der Fall, wird die Methode `AddReference` der Klasse `NPT_MessageHandlerProxy` aufgerufen. Die Klasse `NPT_MessageHandlerProxy` erbt von der Klasse `NPT_MessageHandler`. Laut der Dokumentation der Subklasse stellt diese einen Proxy dar. Die Subklasse besitzt ein Objekt der Superklasse und delegiert alle Methodenaufrufe an dieses Objekt. Eingehende Nachrichten werden von der Klasse `NPT_MessageHandler` behandelt. Es soll von der Subklasse ermöglicht werden, dass während der Nachrichtenübertragung keine weiteren Übertragungen stattfinden. Dies wurde mithilfe von `locks` umgesetzt. Die Methode `AddReference` wird verwendet, um die Referenzzählung eines Objekts durchzuführen. Diese Funktionalität ist in der Superklasse auch als sinnvoll zu erachten, da mithilfe der Referenzzählung nicht verwendeter Speicher freigegeben werden kann.

Beispiel 6.1: XBMC BAT-Indikator

```
NPT_SimpleMessageCapsule(NPT_Message* message, NPT_MessageHandler* handler) :                1
        m_Message(message), m_Handler(handler), m_Proxy(NULL) {                              2
  m_Proxy = NPT_DYNAMIC_CAST(NPT_MessageHandlerProxy, m_Handler);                            3
  if (m_Proxy) m_Proxy->AddReference();                                                      4
}                                                                                            5
```

Nach Anwendung der entwickelten Methode, für den BAT-Indikator, kann der erkannte Indikator aufgelöst und die Vererbungsstruktur optimiert werden. Die Methode ermöglicht das eingehende Analysieren und Bewerten des vorhandenen Indikators, um eine geeignete Restrukturierungsmethode auszuwählen. Zu Beginn der Analyse wird geprüft, ob für diesen Fall ein SKS-Indikator besteht. Dies ist nicht der Fall, da die Superklasse `NPT_MessageHandler` keine Kenntnis über deren Subklasse besitzt. Es wird analysiert, ob es sich um Typcode oder um die Verwendung eines Typüberprüfungsoperators handelt. Durch die fehlende Erkennung von Typcode, kann per Default mit "Nein" für den Typcode geantwortet werden. Bei der Überprüfung des Objekts handelt es sich um keinen Rückgabetyp und dieser ist nicht generischer Natur, was die Bewertung der nächsten Fragen ermöglicht.

Die Subklasse `NPT_MessageHandlerProxy` stellt die einzige Subklasse dar. Aus diesem Grund kann die Frage, ob alle Subklassen geprüft wurden, mit "Ja" beantwortet werden. Die Beantwortung dieser Frage führt zur Restrukturierungsmethode `Replace Conditional with Polymorphism`. Nach Anwendung dieser Restrukturierungsmethode besitzt die Klasse `NPT_MessageHandler` eine abstrakte bzw. pur virtuelle Methode `AddReference`. Diese Methode wird von der Subklasse implementiert. Dies ermöglicht das Auflösen des BAT-Indikators und somit der vorhandenen Typüberprüfung. Die Klasse `NPT_MessageHandlerProxy` kann zudem weiter restrukturiert werden, in dem die Komposition aufgelöst wird und die Superklassenmethoden direkt, ohne Delegation an ein extra Objekt, aufgerufen werden.

6.1.2

HtmlUnit

Von den erkannten 4066 Indikatoren stellen 336 BAT-Indikatoren dar. Innerhalb des HtmlUnit-Projekts liegen sehr starke Vererbungsstrukturen vor, mit mehr als hundert Klassen in einer Vererbungshierarchie. Durch diese großen Vererbungsstrukturen ist eine Veränderung kompliziert, da viele Abhängigkeiten existieren. Eine große Anzahl an Klassen sind voneinander abhängig. Die erkannten Typüberprüfungen tauchen wiederholt über verschiedene Klassen verteilt auf. Dies führt zu der in Abschnitt 3.2.1 angesprochenen Problematik der kaskadierenden Änderungen. Der BAT-Indikator ist, wie an den Funden aus dem Projekt HtmlUnit zu erkennen, ein starker Hinweis auf degenerierte Vererbungsstrukturen.

In dem Beispiel 6.2 ist einer der erkannten BAT Indikatoren dargestellt. Die Typüberprüfung findet innerhalb der Methode `getCalculatedWidth()` in der Klasse `ComputedCSSStyleDeclaration` statt. Diese Methode soll, abhängig vom Typ eines Elements, die berechnete Breite zurückgeben. Die überprüften Klassen sind teilweise in der gleichen großen Vererbungsstruktur. Sie existieren jedoch in verschiedenen Teilbäumen.

Beispiel 6.2: HtmlUnit BAT-Indikator

```
private int getCalculatedWidth() {
   [...]
  if (StringUtils.isEmpty(styleWidth) && parent instanceof HtmlElement) {
    // hack: TODO find a way to specify default values for different tags
    if (getElement() instanceof HTMLCanvasElement) {
        return 300;
    }
    if ("right".equals(cssFloat) || "left".equals(cssFloat)) {
      [...]
    }
    else if ("block".equals(display)) {
        [...]
    }
    else if (node instanceof HtmlSubmitInput || node instanceof HtmlResetInput
                || node instanceof HtmlButtonInput || node instanceof HtmlButton
                || node instanceof HtmlFileInput) {
        final String text = node.asText();
        width = 10 + (text.length() * PIXELS_PER_CHAR);
    }
    else if (node instanceof HtmlTextInput || node instanceof HtmlPasswordInput) {
        width = 50; // wild guess
    }
    else if (node instanceof HtmlRadioButtonInput || node instanceof
          HtmlCheckBoxInput) {
        width = 20; // wild guess
    }
    else if (node instanceof HtmlTextArea) {
        width = 100; // wild guess
    }
    else {
      [...]
    }
  }
  [...]
  return width;
}
```

In der Abbildung 6.1 ist die Ausgabe der Vererbungsstruktur des IChecker Werkzeugs für den beschriebenen BAT-Indikator dargestellt. Es werden die kompletten Vererbungsstrukturen abgebildet und die geprüften Klassen gelb markiert. Die Abbildung beinhaltet zwei verschiedene Vererbungsstrukturen mit mehreren hundert Klassen. Die überprüften Klassen sind innerhalb dieser Strukturen stark verstreut. Aus diesem Grund werden die Vererbungsstrukturen stark vereinfacht, in der Abbildung 6.2 dargestellt. Es werden nur die geprüften Klassen und deren direkten Superklassen abgebildet.

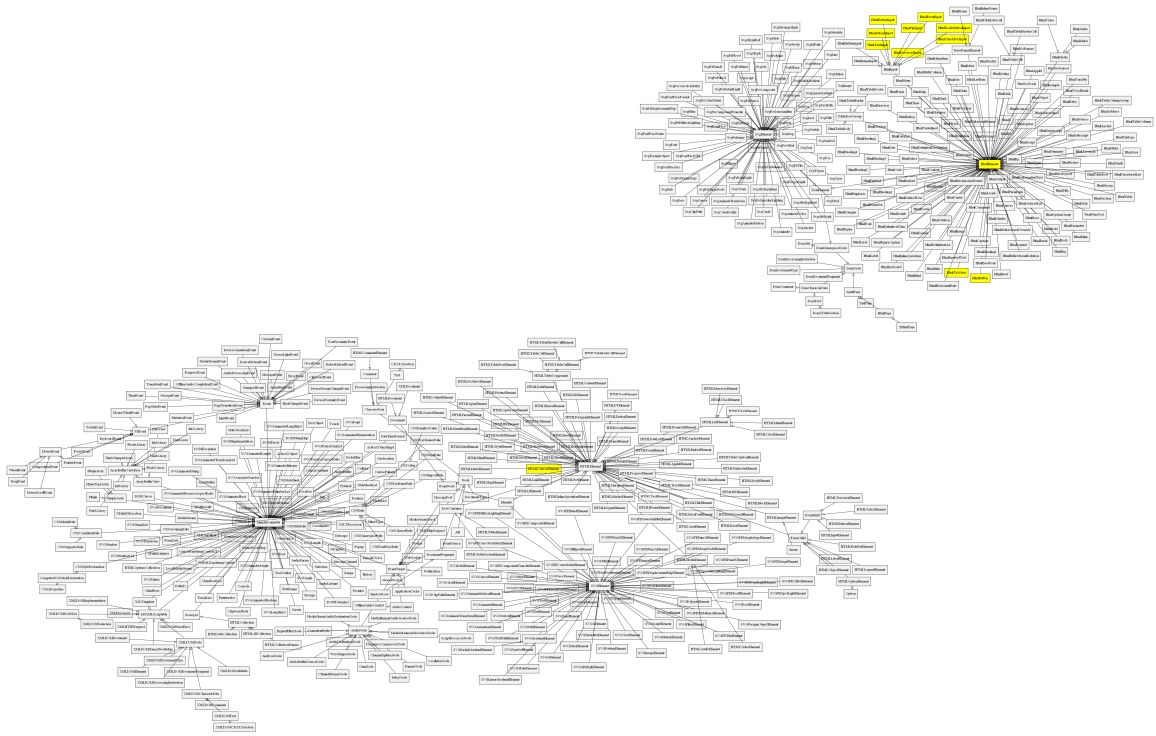


Abbildung 6.1: HtmlUnit Vererbungsstrukturen

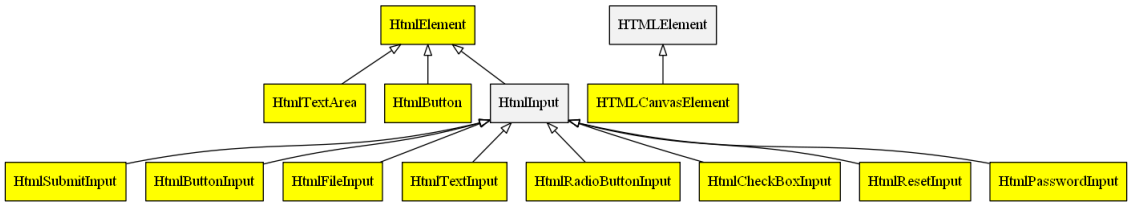


Abbildung 6.2: HtmlUnit BAT-Indikator

Es müssen für die erkannten BAT-Indikatoren, mithilfe der entwickelten Methode, geeignete Restrukturierungsmethoden ausgewählt werden. Erst nach Anwendung dieser Methoden und Auflösung aller existierender BAT-Indikatoren kann die Vererbungsstruktur komplett optimiert werden. Nach der Auflösung aller BAT-Indikatoren sind keine Problematiken, wie kaskadierende Änderungen, extra Abhängigkeiten etc. vorhanden. Die Wart- sowie Erweiterbarkeit wird durch die Optimierung der Vererbungsstruktur immens erhöht.

Nach Anwendung der entwickelten BAT-Methode auf den erkannten BAT-Indikator, aus dem Beispiel 6.2, kann eine geeignete Restrukturierungsmethode ausgewählt werden. Die Restrukturierungsmethode besteht aus der Erstellung einer Schnittstelle, die die geprüften Klassen implementieren und der Anpassung des Rückgabetyps der Methode, dessen Rückgabewert überprüft wird. Diese Restrukturierungsmethode wird ausgewählt, da kein SKS-Indikator und kein Typcode existiert. Es wird ein Rückgabewert überprüft und die Methode die diesen liefert ist veränderbar, da sie nicht von einer Sprachschnittstelle oder Bibliothek implementiert bzw. vererbt wird. Die Anwendung der Restrukturierungsmethode zieht einige weitere Anpassungen mit sich, diese müssen genauso wie die Restrukturierungen der anderen BAT-Indikatoren angewendet werden. Nach den weiteren Restrukturierungen ist die Verwendung der Schnittstelle und eines Methodenaufrufs, ohne Typüberprüfung und Typkonvertierung, möglich.

6.2 DC

Die falsch-positiv Rate der DC-Indikatorerkennung beläuft sich auf rund 23%. Es wurden im Projekt XBMC 36 Indikatoren erkannt, davon sind 16 falsch-positiv. Für das HtmlUnit-Projekt wurden 157 Indikatoren erkannt aus denen 28 Indikatoren falsch-positiv sind. Aus 193 erkannten Indikatoren sind 44 falsch-positiv, dies ergibt eine falsch-positiv Rate von 22.79 %. Für den DC-Indikator sollen nur Konvertierungen von Parametertypen betrachtet werden. Eine Erweiterung und Verbesserung des Detektors wäre die Methodensignatur aus der Quelldatei zu lesen, anstatt aus der Sotograph-Datenbank. Durch diese Veränderung könnten die Parameternamen identifiziert und somit Attribute und Variablen ausgefiltert werden, was die falsch-positiv Rate verringert. Es wird deutlich, dass die beschriebenen `typedefs` einen Teil zur falsch-positiv Rate beitragen. In Java existiert diese Funktionalität nicht, weshalb die falsch-positiv Rate niedriger ist. Durch die externe Ressourcendatei, die die ausgefilterten `typedefs` beinhaltet, kann dieser Filtermechanismus stetig verbessert werden.

6.2.1 XBMC

Einige der evaluierten Indikatoren aus dem Projekt XBMC zeigten Konvertierungen von Parametern ohne vorherige Typprüfung. Gerade diese Indikatoren zeigen schwere Problematiken und müssen auf jeden Fall behoben werden. Durch die Konvertierung der Parameter ohne jegliche vorherige Prüfung kann die Übergabe einer Instanz eines anderen Typs zu einem Laufzeitfehler oder undefinierten Verhalten führen. Diese Problematik besteht nur, wenn der angegebene Parametertyp in der Methodensignatur andere Typen zu lässt. Bspw. der Typ `void*` oder ein Typ einer Superklasse und ein nachfolgende Konvertierung in einen Subklassentyp.

Das Beispiel 6.3 stellt die Methode `OnBrowseResult`, welche unter anderem den Parameter `userdata` vom Typ `void*` erhält, dar. Dieser Parameter wird nach Eintritt in die Methode in den Zeigertyp `PLT_BrowseDataReference` konvertiert. Nach dieser Konvertierung wird ein Wert in das Attribut `res` des `PLT_BrowseDataReference`-Zeigers geschrieben. Für den Parameter `userData` kann jeglicher Zeigertyp übergeben werden. Zum Schreiben des Wertes in das Attribut wird ein offset an den `PLT_BrowseDataReference`-Zeiger verwendet. Wird dieses offset an einem übergebenen Zeiger eines anderen Typs verwendet, führt dies zu undefinierten Verhalten, da der Speicher von anderen Variablen oder Attributen überschrieben wird. Exploits bzw. Buffer Overflows seien dabei nicht ausgeschlossen. Ist diese Funktionalität unbekannt oder wird die Methode von Entwicklern verwendet, die das System nicht kennen, kann dies zu unbekannten Fehlern oder Verhalten führen.

Beispiel 6.3: XBMC DC-Indikator

```
void PLT_SyncMediaBrowser::OnBrowseResult(NPT_Result res,
                PLT_DeviceDataReference& device, PLT_BrowseInfo* info,
                void* userdata)
{
  NPT_COMPILER_UNUSED(device);
  if (!userdata) return;

  PLT_BrowseDataReference* data = (PLT_BrowseDataReference*) userdata;
  (*data)->res = res;
```

Bei der Analyse und Bewertung des erkannten Indikators ist als erstes zu prüfen, ob parallel ein BAT-Indikator existiert. Dies ist nicht der Fall, da der Parameter ohne Typüberprüfung konvertiert wird. Die Klasse `PLT_SyncMediaBrowser` erbt die Methode `OnBrowseResult` von der Klasse `PLT_MediaBrowserDelegate`. Es existiert immer die gleiche Typkonvertierung, da die Klasse `PLT_SyncMediaBrowser` die einzige Subklasse der Klasse `PLT_MediaBrowserDelegate` darstellt. Aus dieser Analyse und Bewertung kann die Restrukturierungsmethode `Parameter Typ anpassen` ausgewählt werden. Durch Anwendung der Restrukturierungsmethode wird der DC-Indikator und deren Problematiken aufgelöst.

6.2.2

HtmlUnit

Die aus dem Projekt HtmlUnit erkannten DC-Indikatoren zeigen ähnliche Problematiken, wie aus dem Projekt XBMC. Ein erkannter DC-Indikator ist in dem Beispiel 6.4 dargestellt. Das Beispiel zeigt einen Teil der Methode `drawImage` aus der Klasse `CanvasRenderingContext2D`. Einige der Parameter sind vom Typ `Objekt`, welche die Superklasse aller Klassen in Java darstellt. Eine Instanz jeden Typs kann diesem Methodenparameter übergeben werden. In der Methode werden die übergebenen Objekte mit einem Objekt vom Typ `Undefined` verglichen. Handelt es sich bei dem Übergebenen Objekt nicht um das Objekt vom Typ `Undefined` ist dieser Vergleich wahr und es wird eine Konvertierung in den Typ `Number` durchgeführt. Dies kann bei der Übergabe von Objekten anderen Typs zu einem Laufzeitfehler führen. Es existiert kein BAT-Indikator und die Methode ist nicht vererbt. Aus dieser Analyse und Bewertung des DC-Indikators kann, mithilfe der DC-Methode, die Restrukturierungsmethode `Parameter anpassen` ausgewählt werden. Der Parametertyp `Object` wird mit dem Typ `Number` ersetzt.

Beispiel 6.4: HtmlUnit DC-Indikator

```
public void drawImage(final Object image, final int sx, final int sy, final Object
      sWidth, final Object sHeight,
final Object dx, final Object dy, final Object dWidth, final Object dHeight) {
  Integer dxI;
  Integer dyI;
  Integer dWidthI = null;
  Integer dHeightI = null;
  Integer sWidthI = null;
  Integer sHeightI = null;
  if (dx != Undefined.instance) {
       dxI = ((Number) dx).intValue();
       dyI = ((Number) dy).intValue();
       dWidthI = ((Number) dWidth).intValue();
       dHeightI = ((Number) dHeight).intValue();
  }
  else {
       dxI = sx;
       dyI = sy;
  }
  if (sWidth != Undefined.instance) {
       sWidthI = ((Number) sWidth).intValue();
       sHeightI = ((Number) sHeight).intValue();
  }
```

Gleiches gilt für das Beispiel 6.5. Dort wird der Parameter vom Typ `Throwable` in eine `XNIException` gewandelt. In diesem Beispiel ist zum Auflösen des Indikators erneut die Anpassung des Parametertyps notwendig.

Beispiel 6.5: Weiterer HtmlUnit DC-Indikator

```
static Throwable extractNestedException(final Throwable e) {
  Throwable originalException = e;
  Throwable cause = ((XNIException) e).getException();
```

6.3 DÜ

Der DÜ-Indikator wurde in den evaluierten Projekten achtmal erkannt. Aus diesem Grund konnte der DÜ Indikator nicht eingehend evaluiert werden. Es wird davon ausgegangen, dass dieser Indikator kein häufiges Vorkommen in der Praxis besitzt. Die Erkennung wird dabei als korrekt angenommen, da das Werkzeug Sotograph zum Erkennen von Methodenaufrufen und vorhanden Methoden genutzt wurde.

6.3.1 XBMC

Innerhalb des XBMC-Projekts wurden sechs DÜ-Indikatoren erkannt. Ein DÜ-Indikator ist in dem Beispiel 6.6 dargestellt. In diesem delegiert die Klasse `UniversalTersePrinter<const char*>` an die Klasse `UniversalTersePrinter<char*>`. Sie besitzen den gleichen Klassennamen und eine Methode mit gleichem Namen. Sie unterscheiden sich in deren Klassentemplate. Die Methode `Print` besitzt abhängig vom Klassentemplate einen anderen Parametertypen für den Parameter `str`.

Beispiel 6.6: XBMC DÜ-Indikator

```
template <>
class UniversalTersePrinter<const char*> {
 public:
  static void Print(const char* str, ::std::ostream* os) {
    if (str == NULL) {
      *os << "NULL";
    } else {
      UniversalPrint(string(str), os);
    }
  }
};
template <>
class UniversalTersePrinter<char*> {
 public:
  static void Print(char* str, ::std::ostream* os) {
    UniversalTersePrinter<const char*>::Print(str, os);
  }
};
```

Die Klasse `UniversalTersePrinter<char*>` delegiert mit der Methode `Print` an die Methode `Print` der Klasse `UniversalTersePrinter<const char*>`. Durch diese Delegation und das komplette Verwenden einer Klassenschnittstelle entsteht der DÜ-Indikator. Ein weiterer Indikator kennzeichnet die Klassen `UniversalTersePrinter<unsigned short*>` und `UniversalTersePrinter<unsigned short const *>`. Durch diese Indikatoren entsteht erhöhte Komplexität. Es führt zu einer großen Anzahl an Klassen, die nicht notwendig sind. Die Anwendung der DÜ-Methode führt zur Restrukturierungsmethode `Remove Middle Man`. Die Klassen `UniversalTersePrinter<char*>` und `UniversalTersePrinter<unsigned short*>` besitzen keine eigene Funktionalität, weshalb sie entfernt werden können.

Anwendungen dieser Klassen können durch die delegierten Klassen ersetzt werden. Mithilfe dieser Restrukturierungsmethode kann ein Teil der vorhandenen Komplexität des Softwareprojekts entfernt und somit optimiert werden.

6.3.2

HtmlUnit

Es wurden zwei DÜ-Indikatoren innerhalb des HtmlUnit-Projekts erkannt. Die erkannten DÜ-Indikatoren stellen keine Problematiken dar. Ein Indikator beschreibt die Nutzung einer anonymen Klasse, die durch Ihre Verwendung eine einzelne existierende Methode aufruft. Des Weiteren wird eine Schnittstelle mit einer Methode als Parametertyp genutzt, wodurch der DÜ-Indikator erkannt wird. Diese Nutzung ist vollkommen legitim und stellt kein Problem dar. Anhand der geringen Anzahl von erkannten Indikatoren konnte der DÜ-Indikator nicht weiter detailliert evaluiert werden. Es liegt nahe, dass dieser Indikator eine geringe Häufigkeit in der Praxis besitzt.

6.4

KIF

6.4.1

XBMC

Innerhalb des XBMC-Projekts wurden 84 KIF-Indikatoren erkannt. Bei diesen Indikatoren handelt es sich um echte Codeklone. Der CCfinder analysiert und erkennt Klone korrekt. Innerhalb der Klone müssen zur Erkennung die Variablennamen nicht übereinstimmen. In der Abbildung 6.3 wird die Ausgabe für einen erkannten KIF-Indikator dargestellt. Die Klassen `PLT_XmlHelper` und `NPT_XmlTagFinder` besitzen einen gemeinsamen Codeklon. Durch Analyse dieses KIF-Indikators wurde erkannt, dass diese Klassen in keinerlei Beziehung zueinander stehen. Die Methode `IsMatch` aus der Klasse `PLT_XmlHelper` und die Operator `bool operator()` Überladung der Klasse `NPT_XmlTagFinder` ist komplett identisch, von der Variablenbezeichnungen abgesehen. Die Methode `IsMatch` prüft, ob ein Objekt vom Typ `NPT_XmlNode` mit einem gegebenen Tag und Namensraum übereinstimmt. Diese Parameter werden einem `NPT_XmlTagFinder`-Objekt bei der Erstellung übergeben. Der Operator `bool operator()` erhält bei Aufruf das Objekt vom Typ `NPT_XmlNode`, welches geprüft werden soll. Der KIF-Indikator umfasst nicht die ganzen Klassen und nicht nur einen Teil einer Methode. Des Weiteren besteht keine Vererbung, diese ist auch nicht sinnvoll, weshalb durch Analyse und Bewertung des KIF-Indikators die Restrukturierungsmethode `Extract Class` ausgewählt wird. Nach Anwendung dieser Restrukturierungsmethode wird eine neue Klasse erzeugt, die diesen Codeklon als eigene Methode beinhaltet. Die zuvor genannten Klassen delegieren auf diese Methode und nutzen ein Objekt der Klasse als Komposition. Durch diese Restrukturierung wird der Quellcode wiederverwendet, der Codeklon und der KIF-Indikator aufgelöst. Die Nutzung der Klassen muss nicht weiter angepasst werden, da die Methoden weiterhin bestehen. Die Problematiken der kaskadierenden Änderungen des KIF-Indikators konnten vollständig beseitigt werden. Der KIF-Indikator kennzeichnet einen wichtigen Teil, eine fehlende Wiederverwendung von Quellcode bspw. durch Vererbung oder Komposition.

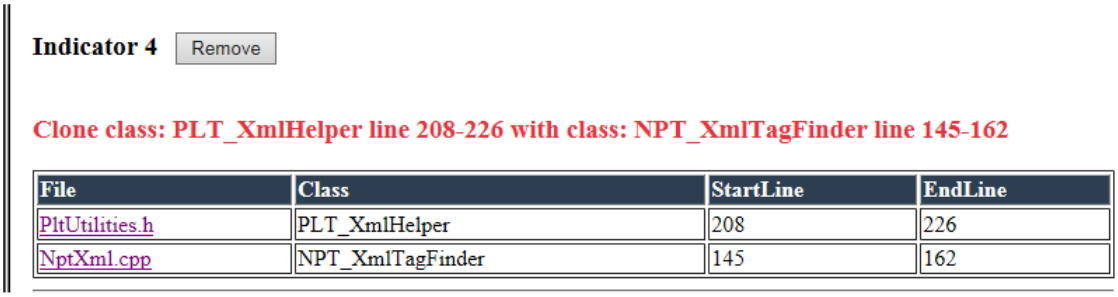


Abbildung 6.3: XBMC KIF-Indikator

Das Fehlen von Wiederverwendung zeigt sich in 61 weiteren erkannten Indikatoren. In diesen wurden Klone in verschiedenen Subklassen der Klasse `CxImage` erkannt. Die einzelnen Subklassen definieren die gleichen Methoden `bool Decode(FILE *hFile)` und `bool Encode(FILE *hFile)`, mit gleicher Implementierung. Es wird innerhalb dieser Methoden mit dem Parameter `hFile` ein Objekt vom Typ `CxIOFile` erstellt. Danach wird an die jeweilige Methode `bool Encode(CxFile * hFile)` bzw. `bool Decode(CxFile * hFile)` delegiert. Die Implementierung dieser Methoden unterscheidet sich in den beschriebenen bzw. erkannten Subklassen. Durch Erstellung einer neuen abstrakten Klasse, die von der Klasse `CxImage` erbt und die als neue Superklasse für die gekennzeichneten Klassen fungiert, können die KIF-Indikatoren aufgelöst werden. Die abstrakte Klasse deklariert als pur virtuelle Methoden (abstrakt) die Methoden `bool Encode(CxFile * hFile)` und `bool Decode(CxFile * hFile)`. Die Methoden `bool Encode(FILE *hFile)` und `bool Decode(FILE *hFile)` werden von dieser Klasse implementiert. Die Subklassen erben somit die Implementierung und implementieren jeweils die abstrakten Methoden. Nicht alle Subklassen der Klasse `CxImage` besitzen die beschriebenen Klone, weshalb eine neue Superklasse zur Lösung dieser Problematik erstellt werden muss. Es wird somit im Vererbungsbaum ein neuer Knoten eingefügt und die Klassen umgehangen. Die 61 Indikatoren können durch diese Vorgehensweise aufgelöst werden. Doppelter Quellcode und kaskadierenden Änderungen werden dadurch verhindert. Die Restrukturierungsmethode konnte mithilfe der erstellten KIF-Methode zur Analyse und Bewertung der Indikatoren ausgewählt werden.

6.4.2

HtmlUnit

In dem Projekt `HtmlUnit` erkennt das erstellte Werkzeug IChecker 2922 KIF-Indikatoren. Einige dieser Indikatoren kennzeichnen die Klassen `HtmlHeading1-6`, welche von eins bis sechs durchnummeriert sind. Diese Klassen stellen die Abbildung für den HTML-Tag `h1` bis `h6` dar. Diese sechs Klassen sind identische Klone mit Ausnahme einer Konstanten, die den Namen des Tags liefert. Sie erben von der Klasse `HtmlElement` und stellen somit Geschwisterklassen dar.

Die Analyse und Bewertung mit der erstellten KIF-Methode führt zur Auswahl der Restrukturierungsmethode, diese besteht aus dem Zusammenführen der Klassen und das Erzeugen eines Attributs für die Konstanten. Denn die Codeklone beschreiben die kompletten Klassen und diese unterschiedenen sich nur in Konstanten. Die zusammengeführte Klasse erhält ein Attribut, welches den derzeitigen HTML-Tag besitzt. Das Attribut kann per Konstruktor gesetzt werden. Die Methodenimplementierung ist in diesen sechs Klassen identisch, weshalb diese in die zusammengeführte Klasse übernommen werden kann. Es verbleibt nach Auflösung des KIF-Indikators eine Klasse `HtmlHeading`, mit den gleichen Methoden, den Konstanten für die einzelnen Tags und einem Attribut `tagName`. Innerhalb der Klasse `DefaultElementFactory` im HtmlUnit-Projekt werden Objekte von Subklassen der Klasse `HtmlElement` bezüglich des gegebenen HTML-Tags erstellt und zurückgegeben. Es wird für den Tag `h1` ein Objekt der Klasse `HtmlHeading1` erstellt und zurückgegeben. Die Erstellung der Objekte innerhalb der Klasse

`DefaultElementFactory` kann dahin angepasst werden, dass für die Tags `h1` bis `h6` jeweils ein Objekt der selben Klasse erstellt wird. Der derzeitige Tag kann dem Objekt bei Erstellung im Konstruktor übergeben werden. Dies vereinfacht die Klassenstruktur, verhindert doppelten Quellcode und löst demzufolge den KIF-Indikator auf. Die Verwendung der einzelnen Klassen kann ohne Probleme auf die zusammengefasste Klasse angepasst werden.

6.5

MV

6.5.1

XBMC

Für das XBMC-Projekt wurden sieben MV-Indikatoren erkannt. In der Abbildung 6.4 ist eine Vererbungshierarchie dargestellt, die durch zwei der erkannten Indikatoren identifiziert wurde. Die Klasse `NPT_MemoryStream` erbt von den Klassen `NPT_DelegatingOutputStream` und `NPT_DelegatingInputStream`. `NPT_DelegatingOutputStream` ist eine Subklasse der Klasse `NPT_OutputStream` und `NPT_DelegatingInputStream` der Klasse `NPT_InputStream`.

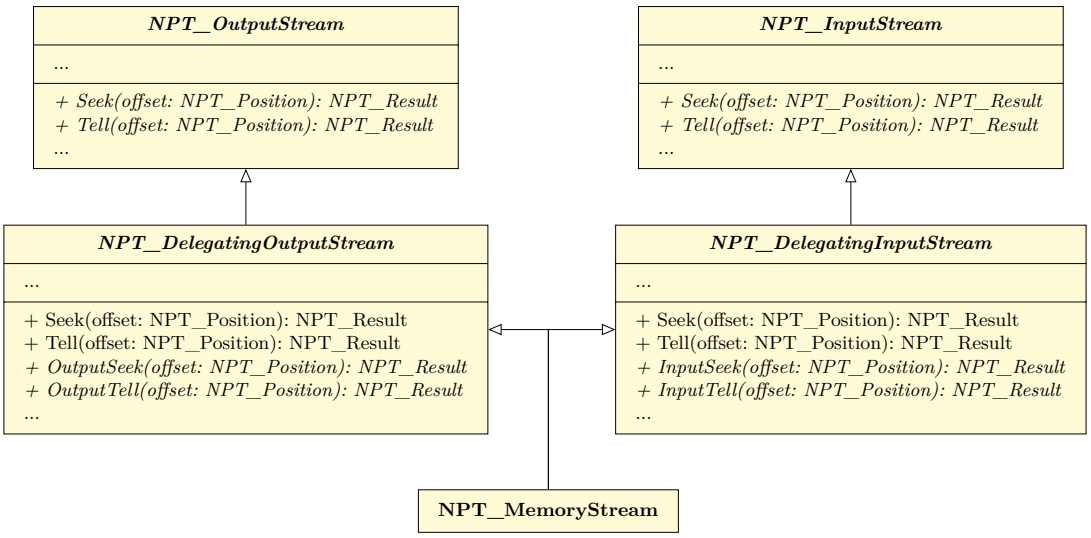


Abbildung 6.4: XBMC MV-Indikator

Die in der Abbildung dargestellte Vererbungsstruktur stellt ein Versuch dar, die Problematik der Mehrfachimplementierung aufzulösen. `NPT_DelegatingOutputStream` implementiert die `Seek` und `Tell` Methode der Klasse `NPT_OutputStream` und delegiert an die abstrakten Methoden `OutputTell` und `OutputSeek`. Gleiches gilt für die Klasse `NPT_DelegatingInputStream` und den Input Methoden. Die Klasse `NPT_MemoryStream` erbt von diesen Subklassen. Diese Struktur soll es ermöglichen, dass ein Objekt vom Typ `NPT_MemoryStream` als `NPT_OutputStream` und `NPT_InputStream` verwendet werden kann. Dies ist durch die Delegation und hinzugefügten Methoden möglich, jedoch existiert weiterhin das Problem der Mehrdeutigkeit. Der Aufruf der `Seek` oder `Tell` Methode an einem Objekt vom Typ `NPT_MemoryStream` führt zu einem Übersetzungsfehler. Bei Erweiterungen von Entwicklern die diese Struktur nicht kennen kann dies zu Missverständnissen, Verwirrung und Umsetzungsfehlern führen.

Es existiert kein KIF-Indikator für diese Klassen und Methoden. Die Intention der Methoden ist nicht gleich und der Methodenname ist veränderbar, da es sich nicht um eine Sprach- oder Bibliotheksschnittstelle oder Klasse handelt. Die Bewertung des MV-Indikators, mithilfe der entwickelten MV-Methode, ermöglicht die Auswahl der Restrukturierungsmethode. Die Restrukturierung durch Umbenennung der Methodennamen der Superklassen wird die Indikatoren für die Methoden `Tell` und `Seek` vollständig auflösen. Es kann durch die Umbenennung auf die Klassen `NPT_DelegatingOutputStream` und `NPT_DelegatingInputStream` komplett verzichtet werden, was die Komplexität des Systems weiter verringert. Nach der Restrukturierung existieren keine mehrdeutigen Methoden die zu Übersetzungsfehlern führen können. Dieses Beispiel zeigt, dass die Problematik des MV-Indikators in der Praxis nicht unbekannt ist, Handlungsbedarf besteht und der MV-Indikator aufgelöst werden muss.

6.5.2

HtmlUnit

Innerhalb des HtmlUnit-Projekts wurden drei MV-Indikatoren erkannt. Jeder dieser Indikatoren kennzeichnet die Schnittstelle `SelectableTextInput` und weitere Klassen die als Superklassen fungierten. Die Methode `focus()` wird von den Superklassen und der Schnittstelle deklariert. In der Abbildung 6.5 ist die Ausgabe für einen MV-Indikator aus dem Projekt HtmlUnit dargestellt.

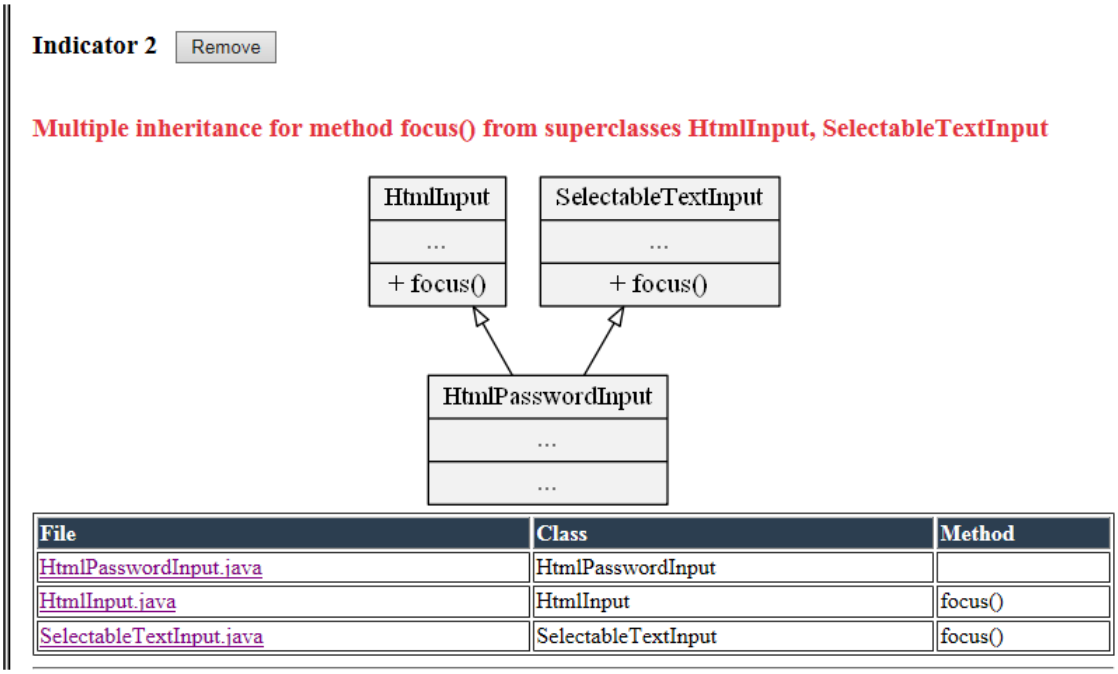


Abbildung 6.5: HtmlUnit MV-Indikator

Die Klasse `HtmlTextInput` implementiert die Schnittstelle `SelectableTextInput` und erbt von der Klasse `HtmlInput`. Die Klasse `HtmlInput` erbt die Methode `focus` von der Klasse `HtmlElement`. Die Methoden der Klassen besitzen die gleiche Intention, weshalb bei der Überschreibung kein logisches Problem entsteht. In diesem Fall fehlt es an der Implementierung einer gemeinsamen Schnittstelle. Die Anwendung der MV-Methode zur Analyse und Bewertung eines MV-Indikators liefert eine geeignete Restrukturierungsmethode. Die Restrukturierungsmethode `Extract Interface` wird ausgewählt, da kein KIF-Indikator existiert und die Methoden gleiche Intentionen besitzen. Es wird eine neue Schnittstelle `Focusable` erzeugt, welche die Methode `focus` deklariert. Die Schnittstelle `SelectableTextInput` erbt von dieser und die Superklasse `HtmlElement` implementiert diese neue Schnittstelle.

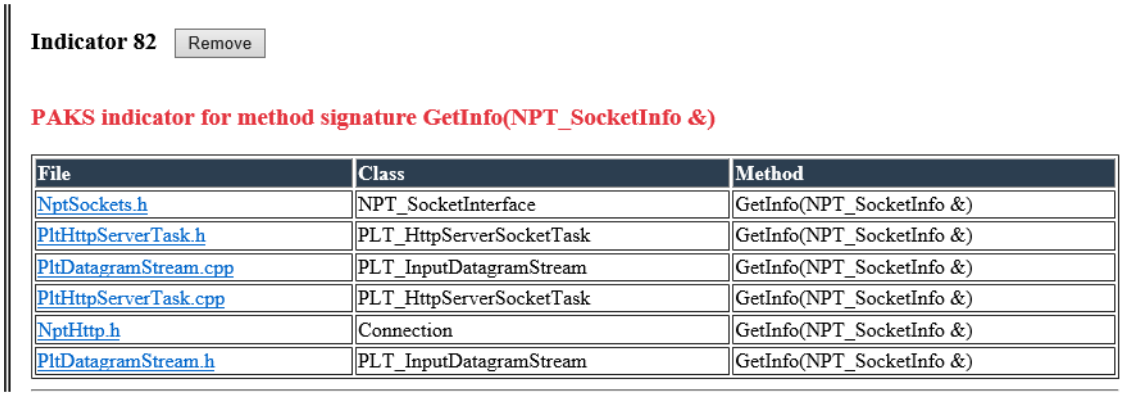
Durch die neu erzeugte Schnittstelle ist ein neues Level der Abstraktion geschaffen. Es ermöglicht die Schnittstelle `Focusable` als Parametertyp einer Methode zu verwenden. Daraufhin kann die Methode `focus` an dem übergebenen Objekt aufgerufen werden. Objekte von Klassen die diese Schnittstelle implementieren können der Methode übergeben werden.

6.6 PAKS

6.6.1 XBMC

Innerhalb des XBMC-Projekts wurden 371 PAKS-Indikatoren erkannt. Einige dieser Indikatoren gleichen sich mit den KIF-Indikatoren. Die Subklassen der Klasse `CxImage` sind mit dem PAKS-Indikator gekennzeichnet, da sie die gleiche Methode `Encode(FILE *hFile)` deklarieren. Durch das Verschieben der Methoden in eine neue Superklasse und somit Auflösung der KIF-Indikatoren werden diese PAKS-Indikatoren aufgelöst. Dieses Restrukturierungsverfahren kann mithilfe der Analyse und Bewertung des PAKS-Indikators durch die erstellte PAKS-Methode erkannt bzw. ausgewählt werden. Dort ist festgelegt, dass wenn ein KIF-Indikator existiert die KIF-Methode angewendet werden muss. Existiert der PAKS-Indikator weiterhin muss die PAKS-Methode fortgeführt werden. In dem beschrieben Beispiel führt die Auflösung der KIF-Indikatoren simultan zur Auflösung der PAKS-Indikatoren.

In der Abbildung 6.6 wird die Ausgabe für einen weiteren erkannten PAKS-Indikator aus dem XBMC-Projekt dargestellt. Die gekennzeichneten Klassen deklarieren jeweils eine Methode mit der Signatur `GetInfo(NPT_SocketInfo &)`. Die Methode erhält als Parameter eine Referenz auf ein Objekt vom Typ `NPT_SocketInfo`. Die Analyse des PAKS-Indikators ergab, dass die Methode das referenzierte Objekt mit Informationen zu einer derzeitigen Socket-Verbindung befüllt. Nach Aufruf dieser Methode wird das Objekt vom Typ `NPT_SocketInfo` ausgewertet. Für diesen Indikator besteht kein KIF-Indikator und die Klassen stehen in keiner gemeinsamen Vererbungsbeziehung, was die Auswahl der Restrukturierungsmethode `Extract Interface`, mithilfe der PAKS-Methode, ermöglicht. Die erstellte Schnittstelle deklariert die zuvor genannte Methode und wird von allen Klassen implementiert bzw. von Schnittstellen geerbt. Das Auflösen dieses Indikators ermöglicht die gemeinsame Nutzung dieser Schnittstelle, verhindert Wiederholung von Methodendeklarationen und fügt eine neue Abstraktionsschicht ein. Methoden können nun diese Schnittstellen nutzen, ohne von Details abzuhängen. Demzufolge kann die Implementierung ausgetauscht werden.



Indicator 82 Remove

PAKS indicator for method signature GetInfo(NPT_SocketInfo &)

File	Class	Method
NptSockets.h	NPT_SocketInterface	GetInfo(NPT_SocketInfo &)
PltHttpServerTask.h	PLT_HttpServerSocketTask	GetInfo(NPT_SocketInfo &)
PltDatagramStream.cpp	PLT_InputDatagramStream	GetInfo(NPT_SocketInfo &)
PltHttpServerTask.cpp	PLT_HttpServerSocketTask	GetInfo(NPT_SocketInfo &)
NptHttp.h	Connection	GetInfo(NPT_SocketInfo &)
PltDatagramStream.h	PLT_InputDatagramStream	GetInfo(NPT_SocketInfo &)

Abbildung 6.6: XBMC PAKS-Indikator

6.6.2

HtmlUnit

Es wurden innerhalb des HtmlUnit-Projekts 591 PAKS-Indikatoren erkannt. Einige der erkannten PAKS-Indikatoren wurden, ähnlich wie in dem XBMC-Projekt, bereits durch KIF-Indikatoren gekennzeichnet. Durch Restrukturierung der KIF-Indikatoren werden bereits einige PAKS-Indikatoren simultan aufgelöst.

Sotographs statische Quellcodeanalyse erfolgt anhand der gegebenen Quelldateien. Klassen wie `Object` oder andere Superklassen, die aus der Standardbibliothek stammen, werden nur als Referenz innerhalb der Sotograph-Datenbank gespeichert. Es ist nicht möglich zu erkennen, ob eine Methode von diesen Superklassen vererbt wurde. Aus diesem Grund werden Methodenüberschreibungen wie `equals, clone` etc. als PAKS-Indikatoren erkannt. Diese Methoden werden von `Object` vererbt und stellen daher kein Problem dar. Im Ausblick könnten Methoden von `Object` gefiltert werden. Die Filterung von anderen Methoden anderer Superklassen aus der Standardbibliothek ist dabei nicht möglich.

In der Abbildung 6.7 ist ein erkannter PAKS-Indikator dargestellt. Dieser PAKS-Indikator kennzeichnet mehrere Klassen, die die Methode `getName()` deklarieren. Diese Methode soll einen Bezeichner liefern bspw. den Inhalt eines Name-Attributs eines HTML-Tags. Mithilfe der PAKS-Methode konnte eine geeignete Restrukturierungsmethode ausgewählt werden. Es existiert für diesen Indikator kein KIF-Indikator und die gekennzeichneten Klassen stehen in keiner gemeinsamen Vererbungsbeziehung. Anhand dieser Bewertungen wird die Restrukturierungsmethode `Extract Interface` ausgewählt.

Indicator 244 Remove

PAKS indicator for method signature getName()

File	Class	Method
XMLDOMAttribute.java	XMLDOMAttribute	getName()
XMLDOMDocumentType.java	XMLDOMDocumentType	getName()
FormEncodingType.java	FormEncodingType	getName()
DomAttr.java	DomAttr	getName()
DomDocumentType.java	DomDocumentType	getName()
HtmlAttributeChangeEvent.java	HtmlAttributeChangeEvent	getName()
Attr.java	Attr	getName()
DocumentType.java	DocumentType	getName()
FormField.java	FormField	getName()
HTMLAnchorElement.java	HTMLAnchorElement	getName()
HTMLFormElement.java	HTMLFormElement	getName()
HTMLFrameElement.java	HTMLFrameElement	getName()
HTMLIFrameElement.java	HTMLIFrameElement	getName()
HTMLMetaElement.java	HTMLMetaElement	getName()
Namespace.java	Namespace	getName()
Plugin.java	Plugin	getName()
Window.java	Window	getName()
PluginConfiguration.java	PluginConfiguration	getName()
Cookie.java	Cookie	getName()
NameValuePair.java	NameValuePair	getName()
WebWindow.java	WebWindow	getName()
WebWindowNotFoundException.java	WebWindowNotFoundException	getName()

Abbildung 6.7: HtmlUnit PAKS-Indikator

Die neue Schnittstelle, bspw. `Nameable` genannt, deklariert die Methode `getName()`. Die gekennzeichneten Klassen implementieren die neu erstellte Schnittstelle. Methoden, die diese Klassen als Parametertyp nutzen, um die Methode `getName()` aufzurufen, können nun diese Schnittstelle verwenden. Infolgedessen sind diese Methoden nicht weiter von Details abhängig und die Implementierung der aufgerufenen Methode kann ausgetauscht werden.

Der PAKS-Indikator ermöglicht das Erkennen von fehlenden Abstraktionsschichten und das fehlende Wiederverwenden von Schnittstellen. Des Weiteren können die Indikatoren auf Methoden hinweisen, die von Details abhängig sind. Dies ist durch die Verwendung der gekennzeichneten Klassen und Methoden erkennbar.

6.7

PVH

6.7.1

XBMC

Innerhalb des XBMC-Projekts wurden zwei PVH-Indikatoren erkannt. In der Abbildung 6.8 sind zwei Vererbungshierarchien dargestellt. Die Klassen `ISetting` und `ISettingCondition` sind durch den PVH-Indikator gekennzeichnet und erkannt. Diese Vererbungsstrukturen stellen zwei parallele Vererbungshierarchien dar. Die Namen der Superklasse `ISettingCondition` und deren Subklassen besitzen Präfixe aus der parallelen Vererbungshierarchie. Die Superklasse `ISetting` steht in einer Aggregationsbeziehung mit der Subklasse `CSettingRequirement` und die Subklasse `CSetting` in einer unidirektionale Assoziationsbeziehung mit der Klasse `CSettingDependency`. Diese parallelen Vererbungsstrukturen hängen voneinander ab. Diese Abhängigkeiten stellen jedoch kein größeres Problem dar. Es können weiterhin Klassen hinzugefügt und verändert werden, ohne dass dies Änderungen in der anderen Struktur zufolge hat.

Das größere Problem stellt die zweite parallele Vererbungshierarchie in dem Vererbungsbaum der Superklasse `ISettingCondition` dar. Die Subklassen der Klasse `ISettingCondition` und deren Teilbäume sind voneinander abhängig. Es existieren unidirektionale Assoziationsbeziehungen zwischen den Superklassen und Subklassen der Klassen `CSettingConditionCombination` und `CSettingConditionItem` sowie den Klassen `CSettingCondition` und `CSettingConditionCombination`.

Wird eine Subklasse in einen der Teilbäume hinzugefügt, müssen die anderen Teilbäumen mit einer weiteren Subklasse ergänzt werden. Die Vererbungsstruktur besitzt starke Abhängigkeiten und ist daher sehr komplex. Mit Anwendung der erstellten PVH-Methode zur Analyse und Bewertung eines PVH-Indikators ist die Auswahl einer geeigneten Restrukturierungsmethode möglich. Die Klassen beschreiben kein MVC Pattern und besitzen untereinander Abhängigkeiten, weshalb die Restrukturierungsmethoden `Move Methode` und `Move Field` angewendet werden müssen. Das Anwenden dieser Restrukturierungsmethoden ermöglicht das schrittweise Zusammenführen der vorhandenen Subklassen und parallelen Vererbungshierarchien. Nach erfolgreicher Zusammenführung der parallelen Vererbungshierarchie und somit Auflösung des PVH-Indikators wurde das System stark optimiert. Die Wartbarkeit und Erweiterbarkeit wurde dahingehend immens erhöht. Die resultierende Vererbungsstruktur ist nachvollziehbar und kann ohne Folgeänderungen erweitert werden. Der PVH-Indikator ermöglicht das Erkennen von stark degenerierten und voneinander abhängigen Vererbungsstrukturen, deren Optimierung notwendig ist.

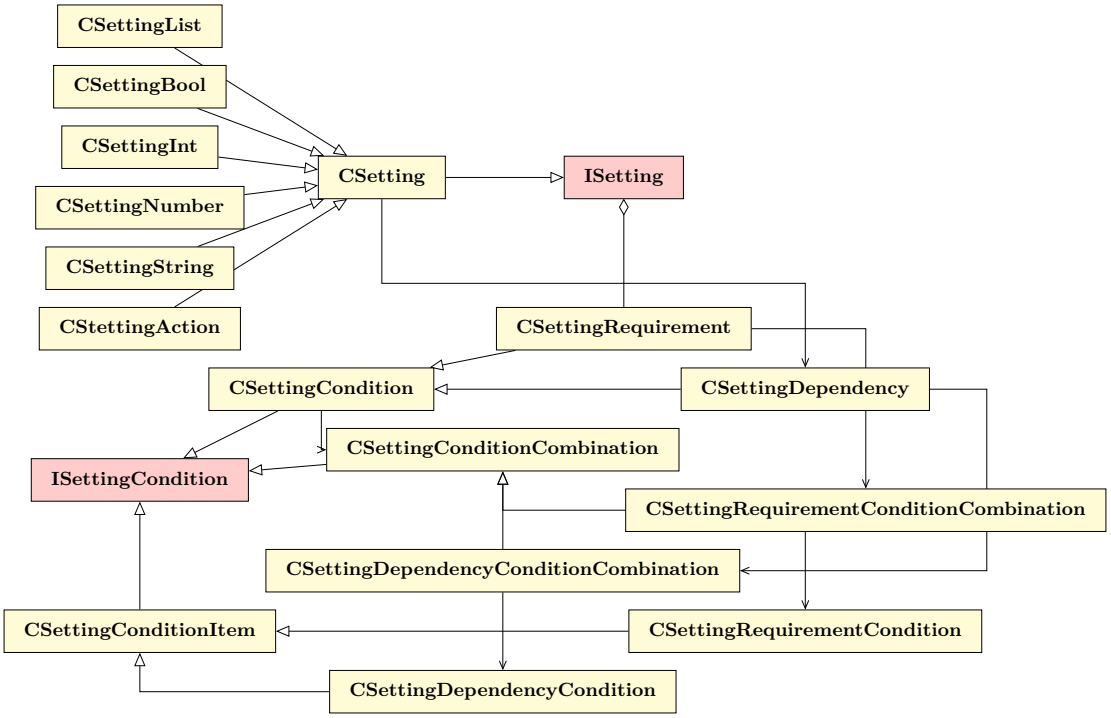


Abbildung 6.8: XBMC PVH-Indikator

6.7.2

HtmlUnit

Innerhalb des HtmlUnit-Projekts wurden neun PVH-Indikatoren erkannt. Einer dieser PVH-Indikatoren ist in der Abbildung 6.9 dargestellt. Dieser ist einer der Ursachen für die bereits erwähnten übermäßig großen und degenerierten Vererbungsstrukturen. Die Ursache liegt in den Superklassen `HTMLElement` und `HtmlElement`. Diese beschreiben mit ihren Subklassen eine parallele Vererbungshierarchie mit mehr als hundert Klassen. Die Vererbungsstruktur der Superklasse `HtmlElement` bildet HTML-Elemente bzw. HTML-Tags ab. `HTMLElement` und deren Vererbungsstruktur bildet die JavaScript-Objekte der HTML-Elemente bzw. HTML-Tags ab. Die Subklassen der Klasse `HTMLElement` besitzen jeweils eine parallele Klasse im Vererbungsbaum der Superklasse `HtmlElement` und eine unidirektionale Assoziationsbeziehung zu dieser. Infolgedessen muss beim Hinzufügen eines HTML-Elements bzw. HTML-Tags eine Subklasse in beiden Vererbungsstrukturen hinzugefügt werden. Durch diese Abhängigkeit kommt es zu kaskadierenden Änderungen, nicht nur beim Hinzufügen von Subklassen.

Durch die Anwendung der PVH-Methode kann eine geeignete Restrukturierungsmethode ausgewählt werden. Die Vererbungsstrukturen bilden kein MVC-Pattern und besitzen Referenzen untereinander, sodass die Restrukturierungsmethoden `Move Methode` und `Move Field` angewendet werden müssen. Die Anwendung dieser Restrukturierungsmethoden ermöglicht das Zusammenführen der erkannten parallelen Vererbungsstruktur. Die Klassen innerhalb der parallelen Vererbungsstruktur beschreiben jeweils die gleichen Elemente. Die Subklassen der Klasse `HTMLElement` referenzieren an die Subklassen der Klasse `HtmlElement`, weshalb diese Restrukturierung nachvollziehbar und möglich ist. Nach der Restrukturierung der Vererbungsstruktur werden die parallelen Vererbungsstrukturen und die damit verbundenen PVH-Indikatoren aufgelöst. Die vorhandene Vererbungsstruktur wird optimiert, sodass die Veränderung bzw. das Hinzufügen und das Entfernen von Klassen keine kaskadierenden Änderungen nach sich zieht.

6.8

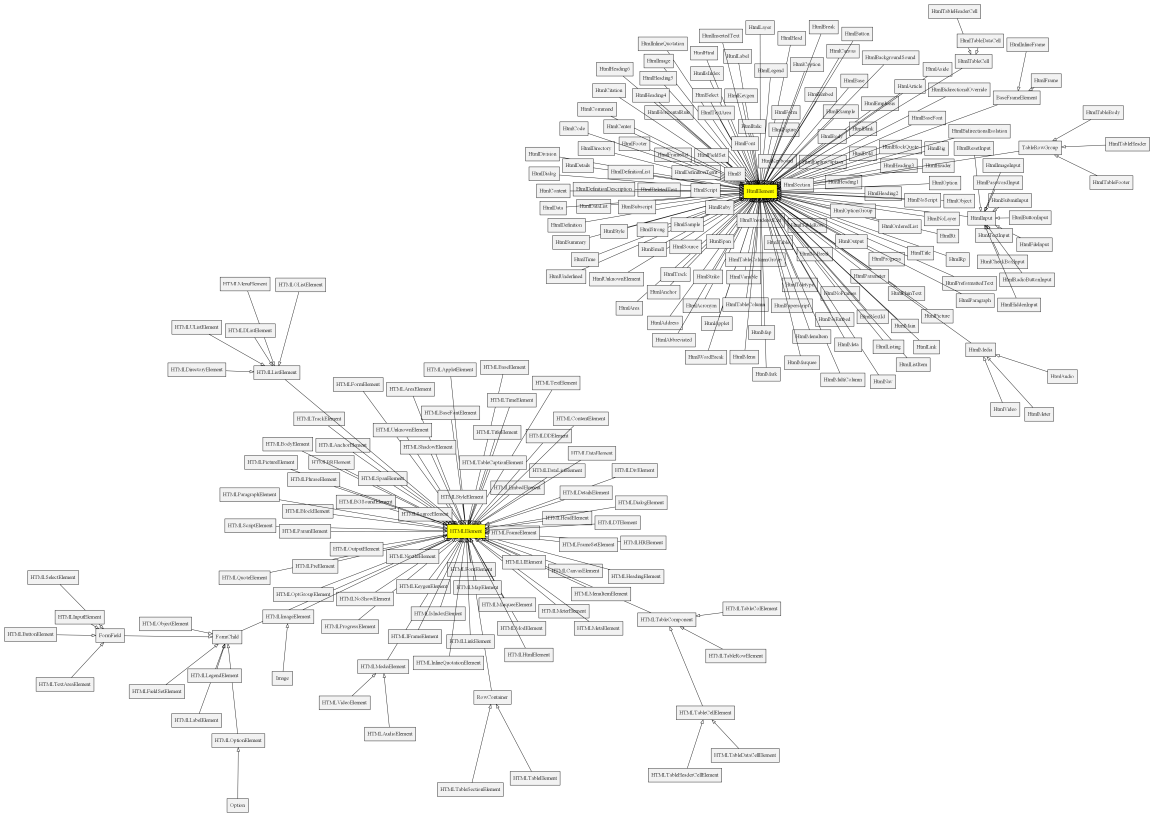


Abbildung 6.9: HtmlUnit PVH-Indikator

SKS

6.8.1

XBMC

Die Sotograph-Metrik `PckgClassKnowingDerievedRule` konnte mit den erkannten Indikatoren und Subklassen verglichen werden. Die Anzahl der Subklassen stimmte bei der Überprüfung überein. Innerhalb des XBMC-Projekts wurden fünf SKS-Indikatoren erkannt. Die Indikatoren beinhalten je eine Superklasse die eine oder mehrere ihrer Subklassen kennt. In der Abbildung 6.10 ist ein erkannter SKS-Indikator dargestellt. Die Superklasse `NPT_LogHandler` referenziert auf all ihre Subklassen. Innerhalb der statischen Methode `Create` wird ein Subklassenobjekt erstellt und zurückgegeben. Anhand einer gegebenen Zeichenkette wird unterschieden, welche Subklasse zur Erstellung eines Objekts verwendet werden soll. Diese Vorgehensweise ähnelt stark dem `Parameterized factory methods` Pattern, eine Variation des `Factory Method` Pattern [vgl. Gam+94, S. 125].

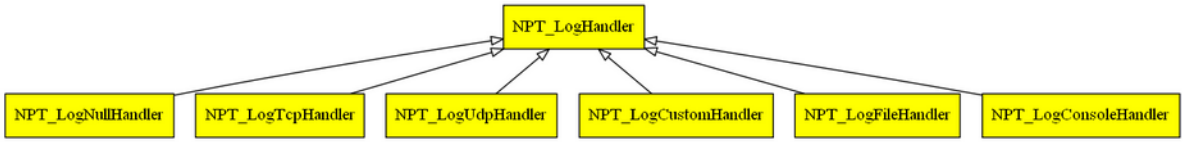


Abbildung 6.10: XBMC SKS-Indikator

Dieses Pattern wurde jedoch nicht korrekt umgesetzt, weshalb eine Restrukturierung erforderlich ist. Die SKS-Methode zur Analyse und Bewertung eines SKS-Indikators ermöglicht die Auswahl einer geeigneten Restrukturierungsmethode. Der erkannte SKS-Indikator kennzeichnet eine Erstellung von Subklassenobjekten, weshalb die Restrukturierungsmethode `Extract Method` und das `Factory Method` Pattern angewendet werden muss. Durch Anwendung der Restrukturierungsmethode wird eine neue Klasse `NPT_LogHandlerFactory` erstellt. Das angewendete `Factory Method` Pattern bzw. in diesem Fall `Parameterized Factory Method` Pattern wird in die Methode `Create` der neu erstellen Klasse verschoben. Durch diese Restrukturierung ist die Superklasse unabhängig von deren Subklassen. Es existiert weiterhin die Möglichkeit der Erstellung der einzelnen speziellen `NPT_LogHandler`-Subklassen, anhand einer gegebenen Zeichenkette.

6.8.2

HtmlUnit

Innerhalb des HtmlUnit-Projekts wurden 11 SKS-Indikatoren erkannt. Der in der Abbildung 6.11 dargestellte SKS-Indikator verhält sich ähnlich zu dem im XBMC-Projekt erkannten SKS-Indikator. Die Superklasse `CSSRule` besitzt eine `Create` Methode mit der verschiedene Subklassenobjekte erstellt werden können. Die Methode erzeugt anhand übergebener Parameter verschiedene Subklassenobjekte. Dies ähnelt stark dem `Parameterized Factory Method` Pattern. Die Umsetzung folgt jedoch nicht komplett den Mustervorschriften, weshalb eine Restrukturierung vorgenommen werden muss.

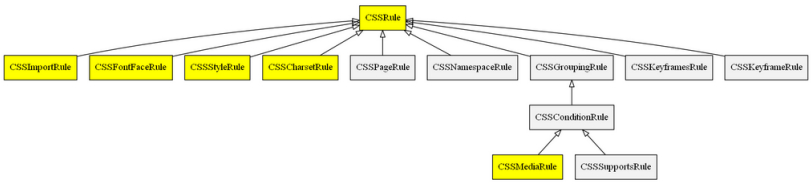


Abbildung 6.11: HtmlUnit SKS-Indikator

Die Anwendung der SKS-Methode führt zur Auswahl der Restrukturierungsmethode `Extract Class` und dem `Factory Method` Pattern. Durch Anwendung der Restrukturierungsmethode wird eine Klasse `CSSRuleFactory` erzeugt. Diese erhält die beschriebene `Create`-Methode und kann mithilfe der übergebenen Parameter die Subklassen der Klasse `CSSRule` erstellen. Die Verwendung muss lediglich an diese neue Klasse angepasst werden, weitere Veränderungen sind nicht notwendig. Die Restrukturierung ermöglicht die Unabhängigkeit der Superklasse von deren Subklassen.

Mithilfe des SKS-Indikators können Superklassenabhängigkeiten von deren Subklassen sowie fehlerhafte Umsetzungen von bspw. dem `Factory Method` Pattern erkannt werden.

Ergebnis
7

Das Ziel dieser Arbeit war es, eine Methode und eine geeignete Werkzeugunterstützung für die systematische Analyse, Bewertung und Optimierung von Vererbungsstrukturen in objektorientierten Softwaresystemen zu entwickeln.

Innerhalb des Kapitels 3 wurde eine Methode zur systematischen Analyse, Bewertung und Optimierung von Vererbungsstrukturen in objektorientierten Softwaresystemen entwickelt. Diese Methode ist in der Abbildung 3.1 dargestellt. Mithilfe der Literatur, vorhandenen Softwareentwicklungsprinzipien, Restrukturierungsmethoden und Design Pattern wurden Vererbungsstrukturproblematiken identifiziert. Für diese Problematiken wurden Indikatoren definiert, die auf diese Problematiken hinweisen. Die dabei definierten Indikatoren lauten BAT, DC, DÜ, KIF, MV, PAKS, PVH, SKS, SVK. Es wurde für jeden Indikator eine Methode zur Analyse und Bewertung, um eine geeignete Restrukturierungsmethode auszuwählen, entwickelt. Die Indikatoren wurden nach bestimmten Problematiken gewichtet. Je höher die Gewichtung, umso mehr Problematiken werden dem definierten Indikator zugeschrieben. Die Gewichtung zeigt an, auf welche Indikatoren besonders bei der Analyse sowie deren Restrukturierung geachtet werden sollte. Die Methoden zur Analyse und Bewertung der Indikatoren gehen auf diese Gewichtung ein und verlangen die Restrukturierung von vorhandenen parallelen Indikatoren, da diese simultan den Indikator mit auflösen können. Die in Kapitel 3 entwickelte Methode umfasst eine statische Quellcodeanalyse, eine Indikatorerkennung, die Methodenanwendung zur Analyse und Bewertungen der jeweiligen erkannten Indikatoren, um eine geeignete Restrukturierungsmethode auszuwählen, und die Anwendung der ausgewählten Restrukturierungsmethode. Eine Anwendung dieser Methode auf ein objektorientiertes Softwaresystem ermöglicht somit die Analyse und Bewertung der Vererbungsstruktur, sodass diese gegebenenfalls geeignet optimiert werden kann.

Mithilfe der theoretischen Vorarbeit konnte in dem Kapitel 5 ein Werkzeug zur Unterstützung der Indikatorerkennung entwickelt werden. Das entwickelte Werkzeug IChecker erkennt die definierten Indikatoren anhand der in Kapitel 3 beschriebenen Merkmale. Die Erkennung wird mithilfe sogenannter Detektoren bzw. Subklassen der Klasse `Detector` durchgeführt. Diese können jeweils in einem eigenen Thread ausgeführt werden, was die parallele Ausführung der Erkennung ermöglicht. Die Werkzeugumsetzung verwendet Abstraktionen und Schnittstellen, sodass die Implementierungen ohne kaskadierenden Änderungen ersetzt oder erweitert werden können. Zurzeit werden die erkannten Indikatoren und deren Informationen in XML serialisiert, dies kann aber durch die Verwendung von Schnittstellen mit dem Speichern in normalen Text oder durch eine JavaScript Object Notation (JSON) Serialisierung ersetzt werden. Dies gilt auch für die Verwendung der Como-Schnittstellen. Bspw. die Implementierung zur Verbindung mit der Datenbank kann ohne Folgeänderungen ausgetauscht werden. Das Werkzeug IChecker setzt auf einige externe Werkzeuge, mit denen die Indikatoren erkannt werden können. Sotograph, Klocwork und CCfinder werden zur Erkennung der beschriebenen Indikatoren verwendet.

In dem Kapitel 4 wurden diese Werkzeuge eingehend analysiert und anhand dieser Analyse wurden die notwendigen Funktionalitäten für das Werkzeug IChecker entworfen und implementiert. Die Werkzeuge Sotograph und Klocwork stellen statische Codeanalysewerkzeuge dar, welche vor der Erkennung der Indikatoren ausgeführt werden müssen. Innerhalb der entwickelten Methode stellt das den ersten Punkt dar, danach kann die Indikatorerkennung mithilfe des erstellten Werkzeugs durchgeführt werden. Die Werkzeuge erzeugen die Datenbasis für die gegebenen objektorientierten Softwareprojekte. Die Datenbasis wird vom IChecker genutzt, um die Indikatoren korrekt zu erkennen. Im Kapitel 5 wurde der Entwurf und die Implementierung des Werkzeugs IChecker beschrieben. Es wurden detailliert die erstellten Namensräume und die darin enthaltenen Klassen erläutert und durch UML-Klassendiagramme dargestellt. Die Ausgaben für die erkannten Indikatoren wurden innerhalb dieses Kapitels vorgestellt und erklärt. Für die Indikatoren BAT, MV, PVH und SKS werden nach der Erkennung für die existierende Vererbungsstruktur PNG-Grafiken erzeugt, in der die erkannten Indikatoren gekennzeichnet werden. Die Darstellung ermöglicht das Nachvollziehen der erkannten Indikatoren und die damit verbundenen Problematiken. Die Erzeugung der Grafiken erfolgt mit dem Werkzeug Graphviz. Es wird mithilfe der Sotograph-Datenbank die existierende Vererbungsstruktur ermittelt und eine dot-Datei erzeugt. Die betreffenden Klassen die durch ein Indikator gekennzeichnet sind werden innerhalb dieser Datei markiert. Mithilfe von Graphviz erfolgt die Erzeugung der PNG-Datei, die bei der Darstellung der Indikatoren mit angezeigt wird.

Bei Ausführung des IChecker wird nach der Erkennung für jeden Indikatortyp eine XML-Datei erzeugt, die die erkannten Indikatoren beinhaltet. Es wird außerdem eine XSLT-Datei erzeugt, die das Darstellen der Indikatordetails innerhalb eines Browsers ermöglicht. Es werden für jeden Indikator spezifische Informationen bereitgestellt, die auf bestimmte Problematiken hinweisen.

Innerhalb des Kapitels 6 wurde eine Teilevaluation durchgeführt, um die definierten Indikatoren und die erstellte Methode einschätzen zu können. Die Evaluation wurde anhand der Open Source Projekte XBMC und HtmlUnit durchgeführt. Für jedes dieser genannten Projekte wurde die erstellte Methode schrittweise durchgeführt. Es erfolgte die Durchführung der statischen Analyse mit Klocwork und Sotograph, die Indikatorerkennung mithilfe des erstellten Werkzeugs IChecker , die Anwendung der Indikatormethoden für die erkannten Indikatoren sowie die Anwendung der ausgewählten Restrukturierungsmethoden, um diese aufzulösen und somit die Vererbungsstruktur zu optimieren. Die Anwendung der jeweiligen Indikatormethode sowie Restrukturierung erfolgte stichprobenartig, da die Anzahl der Indikatoren sehr umfangreich war. Es wurden für das XBMC-Projekt 512 und für das HtmlUnit-Projekt 4066 Indikatoren erkannt. Diese Evaluation zeigte die Funktionsfähigkeit und Notwendigkeit der entwickelten Methode und das Praxisvorkommen der definierten Indikatoren.

Die Anwendung der entwickelten Methode und Werkzeugunterstützung ermöglicht das Analysieren, Bewerten sowie Optimieren von Vererbungsstrukturen in objektorientierten Softwaresystemen. Dies stellte das Ziel dieser Arbeit dar und konnte dahingehend erfüllt werden.

7.1 Kritische Würdigung

Anhand der in dem theoretischen Teil verwendeten Literatur konnten einige der Indikatoren definiert und beschrieben werden. Vor allem durch die Arbeit von [Gam+94; Fow98; Mar09] war dies möglich. Diese Autoren haben bereits viel zum Thema Erkennung von Softwarestrukturproblemen und deren Lösungen beigetragen. Es sind jedoch keine genauen Problematiken zu Vererbungsstrukturen definiert bzw. diese nur teilweise beschrieben. Es fehlt an einem Gesamtwerk an Definitionen der Vererbungsstrukturprobleme, mit der systematischen Analyse, Bewertung und Optimierung. Dort setzt diese Arbeit an und definiert für Vererbungsstrukturprobleme einige Indikatoren und Methoden zur systematischen Analyse und Bewertung, sodass diese optimiert werden können.

Mithilfe der verwendeten Literatur und der evaluierten Projekte wurde ersichtlich, dass diese Problematiken nicht nur theoretischer Natur sind. Die getätigte Teilevaluation in Kapitel 6 hat gezeigt, dass die beschriebenen Indikatoren in Bezug auf die Praxis relevant sind. Es wurde eine große Anzahl an Indikatoren in den zu evaluierten Projekten erkannt. Aus den erkannten Indikatoren wurden stichprobenartig Indikatoren ausgewählt und näher im Kapitel 6 beschrieben und evaluiert. Für jeden der ausgewählten Indikatoren wurde die entsprechende Methode zur Restrukturierungsmethodenfindung angewandt. Die getätigte Teilevaluation half die Indikatoren und deren Praxisnutzen zu bewerten.

Der BAT-Indikator ermöglicht das Erkennen von degenerierten Vererbungsstrukturen. Ein DC-Indikator weist auf eine fehlende Abstraktionsschicht bzw. fehlende Nutzung von bereits existierenden Abstraktionsschichten hin. Durch Mangel an Ergebnissen bzw. Funden konnte der DÜ-Indikator nicht eingehend evaluiert bzw. eingeschätzt werden. Der KIF-Indikator ermöglicht das Erkennen von fehlenden Wiederverwendungsmechanismen, Abstraktionen und Kapselungen wie Vererbung oder Komposition. Es ist möglich mithilfe des MV-Indikators Design Schwachstellen bzw. Mehrdeutigkeiten und somit verbundene Logik- bzw. Umsetzungsfehler zu erkennen. Das Erkennen von fehlenden Abstraktionsschichten, fehlender Wiederverwendung von Schnittstellen und Methoden, die von Details abhängen, ist mithilfe des PAKS-Indikators möglich. Der PVH-Indikator weist auf stark degenerierte abhängige Vererbungsstrukturen hin. Die Erkennung der Abhängigkeit der Superklasse von deren Subklassen ist mithilfe des SKS-Indikators möglich. Des Weiteren besteht die Möglichkeit der Erkennung einer fehlerhaften Umsetzung des `Factory Method` Patterns innerhalb der Superklasse, um deren Subklassen zu erstellen.

Die Erkennung des Indikator SVK konnte aus Gründen des Zeit- und Implementierungsaufwands nicht umgesetzt werden, weshalb eine Evaluation dieses Indikators nicht möglich war. Durch die Erkennung und Restrukturierung der erkannten Indikatoren ist es möglich Vererbungsstrukturen zu optimieren und die Wartbarkeit und Erweiterbarkeit zu erhöhen.

Die definierten Indikatoren beschreiben Vererbungsstrukturproblematiken in objektorientierten Softwaresystemen. Das erstellte Werkzeug IChecker kann daher in allen objektorientierten Sprachen, die mithilfe von Sotograph analysiert werden können, die beschriebenen Indikatoren erkennen. Die auf Quellcode-Ebene zu erkennenden Indikatoren, können durch Klocwork nur für Java und C++ erkannt werden. Diese Einschränkung kann durch eigene Analyse des Quellcodes oder durch Erweiterung von Klocwork aufgelöst werden. Die erkannten Indikatoren werden in XML serialisiert. XML ist ein bekanntes und weitverbreitetes Austauschformat [Qui15] es kann von anderen Programmen bspw. Como via XML-Parser eingelesen werden. Des Weiteren konnte durch die Verwendung von XML eine XSLT Datei erzeugt werden, welches die Konvertierung von HTML aus XML ermöglicht. Die Indikatoren können innerhalb eines Browsers mithilfe des HTML dargestellt werden. Das ermöglichte den Verzicht auf eine GUI. Como hat zudem die Möglichkeit die Daten direkt via HTML darzustellen und benötigt keine extra GUI zur Datendarstellung. Für einige der zu erkennenden Indikatoren werden PNG-Dateien erzeugt, die die Vererbungsstrukturproblematiken darstellen. Diese Darstellung ermöglicht das Nachvollziehen der erkannten Problematiken und der derzeitigen Verwendung des betrachteten Quellcodes. Bspw. wird für den BAT-Indikator angezeigt, welche Klassen im Vererbungsbaum geprüft werden. Anhand dessen kann der Fall weiter eingeschätzt werden. Dies erleichtert die Analyse des erkannten Indikators zur Findung einer geeigneten Restrukturierungsmethode.

Das Werkzeug IChecker verwendet eine Einstellungsdatei und mehrere Ressourcendateien. In diesen sind die Konstanten, Pfadangaben sowie Datenbankverbindungsinformationen zentral gespeichert. Das ermöglicht das Ändern eines Wertes innerhalb einer Datei, sodass die Dateien, die diese Konstanten nutzen, nicht weiter verändert werden müssen. Mithilfe der Einstellungsdatei ist es außerdem möglich die Pfadangaben sowie Datenbankverbindungsinformationen nach der Kompilierung zu ändern. D. h. wird das Werkzeug IChecker auf einem neuen System verwendet, müssen zur korrekten Ausführung des Werkzeugs nur die Daten in der Einstellungsdatei angepasst werden. Es wird somit eine dynamische Verwendung des Systems ermöglicht.

7.2 Ausblick

Im Ausblick sollte der SVK-Indikator erkannt werden, um eine eingehende Evaluation dieses Indikators und der entwickelten SVK-Methode zu ermöglichen. Der Klocwork-Checker wurde bereits für diesen Indikator verfasst. Die erkannten Klocwork-Defekte müssen nach der Erkennung durch Klocwork von IChecker eingehend analysiert werden. Für diese Analyse sind vor allem Abfragen an die Sotograph-Datenbank notwendig.

Die Erkennung der Indikatoren, die mithilfe von Klocwork erkannt werden, sollte im Ausblick auf die Sprache C# erweitert werden. Dies sollte innerhalb einer Aktualisierung von Klocworks möglich sein. Die nächste Klocwork-Version unterstützt bereits benutzerdefinierte C#-Checker, jedoch erst bis zur C# Version 4.0. Ist die Erstellung von benutzerdefinierten C#-Checkern für eine aktuelle Version möglich, müssen geeignete Checker erstellt werden.

Die Erkennung des BAT-Indikators ermöglicht zurzeit nur die Erkennung von Typüberprüfungsoperatoren. Die Erkennung von Typcode muss in unmittelbarem Ausblick stehen, sodass alle möglichen BAT-Indikatoren erkannt und restrukturiert werden können. Die Erkennung sollte schrittweise entwickelt werden. Als erstes sollten `switch-case`-Anweisungen erkannt werden die dann auf Typcode geprüft werden können. Die falsch-positiv Rate des DC-Indikators sollte fortlaufend durch Erweiterung der Ressourcendatei, welche die `typedefs` zur Filterung beinhaltet, reduziert werden.

Die Filterung für konvertierte Parameter sollte des Weiteren angepasst werden. Durch das Einlesen der Methodensignatur aus der Quelldatei können die Parameter bestimmt werden, damit Konvertierungen von Rückgabewerten oder Attributen direkt ausgefiltert werden können.

Die Pfade sowie Datenbankverbindungsinformationen können über eine Einstellungsdatei angepasst werden. Im Ausblick sollte dies über die Programmparameterübergabe möglich sein. Es erhöht die Benutzerfreundlichkeit des Werkzeugs und die Einstellungsdatei muss nicht bei kurzen Änderungen angepasst werden.

Die Integration des IChecker Werkzeugs in das Como-Werkzeug steht ebenso in Ausblick. Die Nutzung der Vererbungsstrukturanalyse bzw. die Darstellung der erkannten Indikatoren sollt innerhalb Como möglich sein. Mithilfe der serialisierten Daten und der erzeugten XSLT-Datei wäre es möglich die Darstellung durch HTML oder XML vorzunehmen.

Abkürzungsverzeichnis

8

AIST National Institute of Advanced Industrial Science and Technology

AST abstract syntax tree

B. M. Berner & Mattner

BAT `Bedingte Anweisungen mit Typüberprüfungen`

CQA Code Quality Analysis

DÜ `Delegation Überbeanspruchung`

DC `Downcasting`

DIP Dependency Inversion Principle

DRY Don't Repeat Yourself

GUI Graphical User Interface

HTML Hypertext Markup Language

ISP Interface Segregation Principle

JSON JavaScript Object Notation

KAST Klocwork abstract syntax tree

KIF `Klassen mit Identischer Funktionalität`

LSP Liskov Substitution Principle

MV `Mehrfachvererbung`

OCP Open Close Principle

OOD Object-oriented design

OOP Object-oriented programming

PAKS `Partiell Analoge Klassenschnittstellen`

PVH `Parallele Vererbungshierarchien`

SKS `Superklasse kennt Subklasse`

SOLID Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion

SQL Structured Query Language

SRP Single Responsibility Principle

SVK `Subklassen variieren nur in Konstanten`

UML Unified Modeling Language

XML Extensible Markup Language

XSLT Extensible Stylesheet Language Transformations

Glossar
9

Exception Englisch für Ausnahme oder Ausnahmefall. Exceptions beschreiben einen Ausnahme- oder Fehlerfall innerhalb eines Programmes.

Anti-Pattern Ein Anti-Pattern stellt eine verbreitete Lösung eines Problems dar, die oft zu negativen Konsequenzen führt [Uni15].

Code Smell Ist ein Hinweis auf eine Problematik, vor allem in objektorientierten Softwaresystemen, die einer Restrukturierung bedarf. Der Begriff und bekannte Hinweise oder smells (engl. Gerüche) wurden von Fowler definiert [vgl. Fow98, S. 63].

Lucene Apache Lucene ist ein Open Source Projekt und stellt eine hoch performante Textsuchmaschine geschrieben in Java dar. Diese Technologie ist für jede Anwendung geeignet, die eine Volltextsuche benötigt [Fou15].

MySQL Die MySQL-Software wird von Oracle entwickelt und liefert einen robusten SQL-Datenbankserver. Dieser ist für missionskritische, schwer belastbare Systeme aber auch für die embedded Nutzung gedacht [MyS15].

OOD Objektorientierter Design oder Entwurf ist der Prozess in dem die Anforderungen in eine Spezifikation umgewandelt werden [vgl. Phi10, S. 8].

OOP Objektorientierte Programmierung ist ein Prozess in dem das zuvor erstellte objektorientierte Design in ein arbeitendes Programm konvertiert wird [vgl. Phi10, S. 8].

Open Source Open Source Software ist Software die frei verfügbar, verwendbar, veränderbar und von jedermann verteilt werden kann. Sie wird durch viele Personen erzeugt und unter bestimmten Lizenzen, die der Open Source Definition entsprechen, verbreitet [org15].

Pattern Pattern werden grundsätzlich in dieser Arbeit mit Entwurfsmustern, die von [Gam+94] definiert wurden, assoziiert.

PostgreSQL PostgreSQL stellt ein objektrelationales Datenbanksystem, welches Open Source ist, dar [Pos15].

XPath Stellt eine Sprache zum Adressieren von Teilen in XML-Dokumenten dar. Sie wurde für die Verwendung in XSLT entworfen [CD15].

Abbildungsverzeichnis

3.1 Methode 15

3.2 SKS-Methode 16

3.3 BAT-Methode 18

3.4 KIF-Indikator 20

3.5 KIF-Methode 21

3.6 PAKS-Indikator 22

3.7 PAKS-Methode 23

3.8 PVH-Indikator 24

3.9 PVH-Methode 25

3.10 DÜ-Indikator 26

3.11 DÜ-Methode 26

3.12 SVK-Indikator 27

3.13 SVK-Methode 28

3.14 DC-Methode 30

3.15 MV-Indikator 31

3.16 MV-Methode 32

5.1 Core-Namensraum 38

5.2 Database-Namensraum 39

5.3 IO-Namesensraum 40

5.4 Ausgabebeispiel 41

5.5 Graphics-Namensraum 42

5.6 Ausgabebeispiel Graphviz 42

5.7 Detection-Namensraum 43

5.8 BAT-Indikator Ausgabe 45

5.9 DC-Indikator Ausgabe 47

5.10 DÜ-Indikator Ausgabe 48

5.11 CodeClones-Namesensraum 48

5.12 KIF-Indikator Ausgabe 49

5.13 MV-Indikator Ausgabe 50

5.14 PAKS-Indikator Ausgabe 50

5.15 PVH-Indikator Ausgabe 51
5.16 SKS-Indikator Ausgabe 52
5.17 ICCore-Namensraum 53

6.1 HtmlUnit Vererbungsstrukturen 56
6.2 HtmlUnit BAT-Indikator 56
6.3 XBMC KIF-Indikator 61
6.4 XBMC MV-Indikator 62
6.5 HtmlUnit MV-Indikator 63
6.6 XBMC PAKS-Indikator 64
6.7 HtmlUnit PAKS-Indikator 65
6.8 XBMC PVH-Indikator 67
6.9 HtmlUnit PVH-Indikator 68
6.10 XBMC SKS-Indikator 69
6.11 HtmlUnit SKS-Indikator 69

Tabellenverzeichnis

2.1 Projektgrößen 14

3.1 Gewichtung der Indikatoren 33

Beispielverzeichnis

2.1 Castings 6

3.1 BAT-Indikator 17

3.2 AnimalNoisePrinter 22

3.3 DC-Indikator 28

5.1 Klassennamen-Regex 45

5.2 DC-Regex 46

6.1 XBMC BAT-Indikator 54

6.2 HtmlUnit BAT-Indikator 55

6.3 XBMC DC-Indikator 58

6.4 HtmlUnit DC-Indikator 58

6.5 Weiterer HtmlUnit DC-Indikator 59

6.6 XBMC DÜ-Indikator 59

Literatur

[14] *C++. Type conversions.* 2014. URL: http://cplusplus.com/doc/tutorial/typecasting/ (besucht am 14. 04. 2015).

[15a] *C++ ref. dynamic_cast conversion.* 2015. URL: http://en.cppreference.com/w/cpp/language/dynamic_cast (besucht am 16. 04. 2015).

[15b] *C++ ref. typeid Operator.* 2015. URL: http://en.cppreference.com/w/cpp/language/typeid (besucht am 17. 04. 2015).

[15c] *IBM Knowledge Center. Chapter 15. C++ Inheritance.* 2015. URL: http://www-01.ibm.com/support/knowledgecenter/SSQ2R2_9.0.0/com.ibm.tpf.toolkit.compilers.doc/ref/langref_os390/cbclr21020.htm (besucht am 17. 06. 2015).

[15d] *IBM Knowledge Center. The typeid operator (C++ only).* 2015. URL: http://www-01.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.cbclx01/the_typeid_operator.htm (besucht am 17. 04. 2015).

[15e] *Microsoft Developer Network. Inheritance and Interfaces.* 2015. URL: https://msdn.microsoft.com/en-us/library/ms973861.aspx (besucht am 17. 06. 2015).

[15f] *Microsoft Developer Network. dynamic_cast Operator.* 2015. URL: https://msdn.microsoft.com/en-us/library/cby9kycs.aspx (besucht am 16. 04. 2015).

[15g] *Microsoft Developer Network. typeid Operator.* 2015. URL: https://msdn.microsoft.com/en-us/library/fyf39xec.aspx (besucht am 16. 04. 2015).

[15h] *Microsoft Developer Network. is (C# Reference).* 2015. URL: https://msdn.microsoft.com/de-de/library/scekt9xw.aspx (besucht am 27. 04. 2015).

[15i] *OMG Unified Modeling Language TM (OMG UML).* formal/15-03-01. Version 2.5. Object Managment Group. März 2015. URL: http://www.omg.org/spec/UML/2.5/ (besucht am 24. 08. 2015).

[Aho+14] A.V. Aho u. a. *Compilers. Principles, Techniques, and Tools (Second Edition).* Pearson Education Limited, 2014. ISBN: 978-1-292-02434-9.

[AT15a] National Institute of Advanced Industrial Science und Technology. *About AIST.* 2015. URL: https://www.aist.go.jp/aist_e/about_aist/index.html (besucht am 22. 06. 2015).

[AT15b] National Institute of Advanced Industrial Science und Technology. *AIST CCFinderX.* 2015. URL: http://www.ccfinder.net/ccfinderxos.html (besucht am 22. 06. 2015).

[BM15] B. M. *About Berner und Mattner.* 2015. URL: http://www.berner-mattner.com/ (besucht am 07. 04. 2015).

[Bro+98] William J. Brown u. a. *AntiPatterns. Refactoring Software, Architectures, and Projects in Crisis.* John Wiley & Sons, Inc., 1998. ISBN: 978-0-471-19713-3.

[CD15] James Clark und Steve DeRose. *XML Path Language (XPath).* 2015. URL: http://www.w3.org/TR/xpath/ (besucht am 30. 09. 2015).

[de] Academic dictionaries und encyclopedias. *Keyword: Polymorphismus.* URL: http://universal_lexikon.deacademic.com/28187/Polymorphismus (besucht am 08. 04. 2015).

[Dow12] Allen B. Downey. *Think Python.* O'Reilly Media Inc., 2012. ISBN: 978-1-449-33072-9.

[Fou15] The Apache Software Foundation. *Apache Lucene Core.* 2015. URL: https://lucene.apache.org/core/ (besucht am 30. 09. 2015).

[Fow98] Martin Fowler. *Refactoring. Improving the Design of Existing Code.* Addison Wesley, 1998. ISBN: 978-0201485677. URL: http://esigatesting.inta.gov.ar/webdav/files/pdf/sample-signed.pdf (besucht am 28. 04. 2015).

[Gam+94] Erich Gamma u. a. *Design patterns. Elements of Reusable Object-Oriented Software.* Pearson Eductation Inc., 1994. ISBN: 978-0201633610.

[Gar06] Javier Garzás. *Object-Oriented Design Knowledge. Principles, Heuristics and Best Practices.* IGI Global, 2006. URL: https://books.google.de/books?id=8Vq9AQAAQBAJ&pg=PA320&lpg=PA320&dq=superclass+knows+subclass (besucht am 22. 09. 2015).

[Gra15a] Graphviz. *The DOT Language*. 2015. URL: http://www.graphviz.org/content/dot-language (besucht am 28. 09. 2015).

[Gra15b] Graphviz. *Welcome to Graphviz*. 2015. URL: http://www.graphviz.org/ (besucht am 28. 09. 2015).

[hel15a] hello2morrow. *About Us*. 2015. URL: https://www.hello2morrow.com/company/company (besucht am 22. 06. 2015).

[hel15b] hello2morrow. *Sotograph*. 2015. URL: https://www.hello2morrow.com/products/sotograph (besucht am 22. 06. 2015).

[HT99] Andrew Hunt und David Thomas. *The Pragmatic Programmer. From Journeyman to Master*. Addison Wesley, 1999. ISBN: 0-201-61622-X. URL: https://robot.bolink.org/ebooks/The%20Pragmatic%20Programmer%20-%20From%20Journeyman%20To%20Master%20By%20Andrew%20Hunt%20and%20David%20Thomas%20-%20Addison%20Wesley%20-%201999.pdf (besucht am 22. 06. 2015).

[Inc15] Gargoyle Software Inc. *HtmlUnit*. 2015. URL: http://htmlunit.sourceforge.net/ (besucht am 13. 09. 2015).

[KE09] Prof. Dr. Alfons Kemper und Dr. André Eickler. *Datenbanksysteme. Eine Einführung*. Oldenburg Verlag München, 2009. ISBN: 978-3-486-59018-0.

[Klo12] Klocwork. *Writing custom C/C++ checkers*. 2012. URL: http://docs.klocwork.com/Insight-10.0/Writing_custom_C/C++_checkers (besucht am 05. 08. 2015).

[Klo15a] Klocwork. *About Klocwork*. 2015. URL: http://www.klocwork.com/company (besucht am 24. 08. 2015).

[Klo15b] Klocwork. *System requirements*. 2015. URL: https://developer.klocwork.com/documentation/klocwork/en/current/system-requirements#SupportedC#languagespecifications (besucht am 05. 08. 2015).

[Klo15c] Klocwork. *Writing custom C# checkers*. 2015. URL: https://developer.klocwork.com/documentation/klocwork/en/current/writing-custom-c-checkers (besucht am 05. 08. 2015).

[Kod15] Kodi. *About*. 2015. URL: http://kodi.tv/about/ (besucht am 13. 09. 2015).

[KS08] H. Kegel und F. Steimann. »Systematically refactoring inheritance to delegation in java«. In: *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*. Mai 2008, S. 431–440. DOI: 10.1145/1368088.1368147.

[LR06a] Bernhard Lahres und Gregor Raýman. *Praxisbuch Objektorientierung. Professionelle Entwurfsverfahren*. Rheinwerk Computing, 2006. ISBN: 978-3-89842-624-4. URL: http://openbook.rheinwerk-verlag.de/oo/oo_04_strukturvonooprogrammen_01_001.htm#Rxxob04strukturvonooprogrammen01001040015541f02e130 (besucht am 06. 09. 2015).

[LR06b] Martin Lippert und Stephen Roock. *Refactoring in Large Software Projects. Performing Complex Restructurings Successfully*. John Wiley & Sons, Inc., 2006. ISBN: 978-0-470-85892-9. URL: https://books.google.de/books?id=bCEYuB83ROcC&printsec=frontcover&hl=de#v=onepage&q&f=false (besucht am 20. 05. 2015).

[LW93] Barbara Liskov und Jeannette M Wing. *Family values: A behavioral notion of subtyping*. Techn. Ber. DTIC Document, Juli 1993.

[LW99] Barbara Liskov und Jeannette M Wing. *Behavioral Subtyping Using Invariants and Constraints*. Techn. Ber. New York, NY, USA, Juli 1999. URL: http://reports-archive.adm.cs.cmu.edu/anon/1999/CMU-CS-99-156.pdf (besucht am 05. 09. 2015).

[Mar03] Robert C. Martin. »SRP: The Single Responsibility Principle«. In: *Agile Software Development: Principles, Patterns, and Practices*. 2003. Kap. 8. ISBN: 978-0135974445. URL: https://docs.google.com/file/d/0ByOwmqah_nuGNHEtcU5OekdDMkk/edit (besucht am 17. 04. 2015).

[Mar09] Robert C. Martin. *Clean Code. A Handbook of Agile Software Craftmanship*. Pearson Education Inc., 2009. ISBN: 978-0-13-235088-4.

[Mar12] Robert C. Martin. *The Principles of OOD*. 2012. URL: http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod (besucht am 17. 04. 2015).

[Mar96a] Robert C. Martin. *The Dependency Inversion Principle*. Object Mentor Inc, Mai 1996. URL: http://www.objectmentor.com/resources/articles/dip.pdf (besucht am 17. 04. 2015).

[Mar96b] Robert C. Martin. *The Interface Segregation Principle*. Object Mentor Inc, Aug. 1996. URL: http://www.objectmentor.com/resources/articles/isp.pdf (besucht am 17. 04. 2015).

[Mar96c] Robert C. Martin. *The Liskov Substitution Principle*. Object Mentor Inc, März 1996. URL: http://www.objectmentor.com/resources/articles/lsp.pdf (besucht am 17. 04. 2015).

[Mar96d] Robert C. Martin. »The Open-Closed Principle«. In: *More C++ gems* (Jan. 1996). URL: http://www.objectmentor.com/resources/articles/ocp.pdf (besucht am 17. 04. 2015).

[MLC05] Raúl Marticorena, Carlos Lópe und Yania Crespo. *Parallel Inheritance Hierarchy. Detection from a Static View of the System*. Techn. Ber. Glasgow, UK: 6th International Workshop on Object Oriented Reenginering (WOOR), 2005.

[MyS15] MySQL. *Chapter 1 General Information*. 2015. URL: http://dev.mysql.com/doc/refman/5.7/en/introduction.html (besucht am 30. 09. 2015).

[Net15] Microsoft Developer Network. *What's New for Visual C#*. 2015. URL: https://msdn.microsoft.com/en-us/library/hh156499.aspx (besucht am 05. 08. 2015).

[Ora08] Oracle. *Class Instanceof*. 2008. URL: http://docs.oracle.com/cd/E13155_01/wlp/docs103/javadoc/com/bea/p13n/expression/operator/Instanceof.html (besucht am 01. 04. 2015).

[Ora15a] Oracle. *Defining an Interface*. 2015. URL: https://docs.oracle.com/javase/tutorial/java/IandI/interfaceDef.html (besucht am 12. 06. 2015).

[Ora15b] Oracle. *Inheritance*. 2015. URL: https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html (besucht am 30. 09. 2015).

[Ora15c] Oracle. *Interfaces*. 2015. URL: https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html (besucht am 12. 06. 2015).

[org15] Opensource [dot] org. *Welcome to The Open Source Initiative*. 2015. URL: http://opensource.org/ (besucht am 30. 09. 2015).

[OXF] OXFORD-Dictionary. *Keyword: polymorphism*. URL: http://www.oxforddictionaries.com/definition/english/polymorphism (besucht am 08. 04. 2015).

[Phi10] Dusty Phillips. *Python 3 Object Oriented Programming*. Packt Publishing Ltd., 2010. URL: ftp://91.193.237.1/pub/docs/linux-support/programming/Python/Python%203%20Object%20Oriented%20Programming%20[Dusty%20Phillips]%20(2010)/Python.3.Object.Oriented.Programming.Dusty.Phillips.2010.pdf (besucht am 08. 04. 2015).

[Pos15] PostgreSQL. *Welcome to the PostgreSQL Wiki!* 2015. URL: https://wiki.postgresql.org/wiki/Main_Page (besucht am 30. 09. 2015).

[Qui15] Liam R. E. Quin. *XML ESSENTIALS*. 2015. URL: http://www.w3.org/standards/xml/core (besucht am 13. 09. 2015).

[Sin14] Chaitanya Singh. *Polymorphism in Java – Method Overloading and Overriding*. 2014. URL: http://beginnersbook.com/2013/03/polymorphism-in-java/ (besucht am 08. 04. 2015).

[Spo07] Siegfried Spolwig. *Was ist Polymorphie?* 2007. URL: http://oszhdl.be.schule.de/gymnasium/faecher/informatik/oop/polymorphie.htm (besucht am 08. 04. 2015).

[Str13] Bjarne Stroustrup. *The C++ Programming Language (hardcover) (4th Edition)*. Pearson Education Inc., 2013. ISBN: 978-0-321-95832-7.

[Ull11] Christian Ullenboom. *Java ist auch eine Insel. 5.9.3 Typen mit dem instanceof-Operator testen*. 2011. URL: http://openbook.rheinwerk-verlag.de/javainsel/javainsel_05_009.html#dodtp14c07ec4-50ad-4e31-a26d-43c37526d8d0 (besucht am 01. 04. 2015).

[Uni15] Iowa State University. *Anti-Patterns*. 2015. URL: http://web.cs.iastate.edu/~hridesh/teaching/362/07/01/notes/antipatterns.pdf (besucht am 30. 09. 2015).

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Ort, Datum **Name**