HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

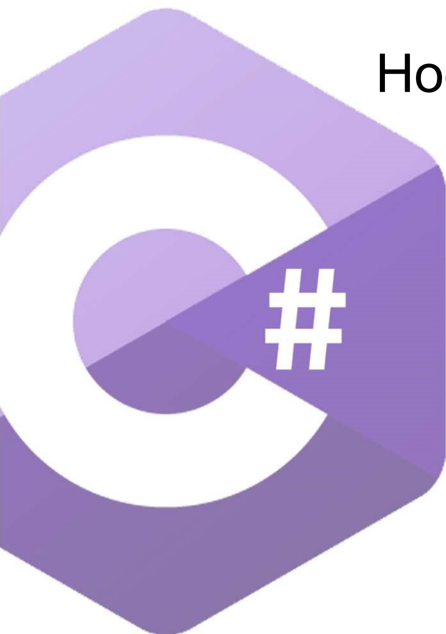
ins
INSTITUTE FOR
NETWORKED SOLUTIONS

Statische Code Analyse

Studienarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Frühjahrssemester 2018



Autor(en): Egger Joel, Stocker Marcel
Betreuer: Prof. Bläser Luc

Versionierung

Stocker	01.03.18	Erstellung Dokument
Egger	21.04.18	Dokumentation Kapitel 1.1
Egger	04.05.18	Überarbeitung aller Kapitel
Stocker	05.05.18	Dokumentation Kap. 1.1
Stocker	07.05.18	Überarbeitung aller Kapitel
Egger	12.05.18	Beginn der Auswertung
Stocker	12.05.18	Korrekturen
Stocker	14.05.18	Korrekturen
Egger	20.05.18	Korrekturen
Stocker	22.05.18	Überarbeitung
Stocker / Egger	27.05.18	Überarbeitung und Schlussfolgerung
Stocker / Egger	30.05.18	Abgabe

Inhaltsverzeichnis

1 Einleitung und Übersicht 8
1.1 Statische Code-Analysen 9
1.1.1 Lexikalische Analyse 9
1.1.2 Code-Stil Analyse 10
1.1.3 Metrik Berechnungen 16
1.2 Softwarearchitektur 21
1.2.1 Projekte 21
1.2.2 Wichtigste Klassen & Interfaces 22
1.3 Datenbanken 24
2 Evaluation 25
2.1 Log4Net 25
2.1.1 Messdaten 25
2.1.2 Häufigkeit der Refactorings 25
2.1.3 Nötige Anpassungen 26
2.2 Prism 28
2.2.1 Messdaten 28
2.2.2 Häufigkeit der Refactorings 28
2.2.3 Nötige Anpassungen 28
2.3 Wox 29
2.3.1 Messdaten 29
2.3.2 Häufigkeit der Refactorings 29
2.3.3 Nötige Anpassungen 30
3 Schlussfolgerungen 31
3.1 Zusammenfassung 31
3.2 Bewertung der Resultate 31
3.3 Nicht erreichte Ziele 31
3.3.1 Variablen und Property korrigieren 31
3.3.2 Korrelation zwischen Klassenname und Methodenname 31
3.3.3 Wörterbuch Analyse mit einer einzigen Datenbank 31
3.3.4 Verlinkte Variablenbezeichner ersetzen 31
3.4 Weitere Schritte 31
4 Anhang 32

Aufgabenstellung

Aufgabenstellung Studienarbeit für Joël Egger und Marcel Stocker:

Code Design Analyse in C#

1. Auftraggeber und Betreuer

Diese Studienarbeit findet für das INS Institut für vernetzte Systeme statt.

Betreuer HSR:

- Prof. Dr. Luc Bläser, Institut für vernetzte Systeme, lblaeser@hsr.ch

2. Ausgangslage

Die Code Qualität von Software variiert stark. Ein Software-Entwickler erkennt in der Regel sehr schnell, meist schon durch oberflächliche Ansicht, ob ein Programmcode eher gut oder schlecht ist. Dafür ausschlagend sind verschiedene Aspekte, wie Design-Regeln oder Stil-Muster. Solche offensichtliche Findings werden meist in Code Reviews zuerst gefunden, sind für den Reviewer aber eher uninteressant und lenken von der eigentlich tiefen Code-Inspektion oft ab.

Es wäre nun nützlich, wenn man einen Code-Review automatisch mittels einer statischen Analyse durchführen könnte. Als ersten Schritt fokussieren wir uns auf die Aspekte, welche ein menschlicher Reviewer durch eher oberflächliche Durchsicht finden würde. Einerseits gibt es schon zahlreiche statische Analyse-Tools zur Messung von Code-Metriken, Bestimmung von Technical Dept und Aufspüren von Bug-Patterns und Anti-Patterns. Andererseits sind deren Analyse meist noch zu begrenzt oder zu lokal, um eine ähnlich aussagekräftige Beurteilung wie ein oberflächlicher menschlicher Code-Review zu machen.

3. Ziele und Aufgabenstellung

Das Ziel dieser Studienarbeit ist es, eine statische Code-Analyse zu entwickeln, welche eine grundsächliche Qualitätsaussage über Code-Design und Stil macht. Es ist als erster Schritt hinsichtlich eines automatischen Code-Review-Tools zu verstehen.

Die Analyse bezieht sich auf C# für das Visual Studio IDE. Hierzu kann die .NET Compiler Platform (Roslyn) eingesetzt werden.

Die Arbeit hat folgende spezifische Ziele:

- Sammlung von grundsätzlichen Design- und Stil-Merkmalen in qualitativen Source Code, die automatisch erkennbar wären, wie z.B.

- Design: klare Begriffe in Benennung, keine Abkürzungen oder übergenerische Namen
- Design: Kohäsion einer Klasse (semantischer Bezug der Member-Namen zur Klasse)
- Design: Obere Metrik-Schranken für die Komplexität einer Klasse oder Methode.
- Stil: Konsistente Formatierung
- Stil: Keine Schreibfehler in Namen
- Stil: Antimuster, wie x == true oder y == false
- Und so weiter

- Konzept über die Erkennung dieser Merkmale, z.B. Einsatz eines Wörterbuchs, Thesaurus, Ontologie-Datenbank.
- Auswahl der gesammelten Merkmale für die Realisierung in einer statischen Analyse in Absprache mit dem Betreuer.
- Entwurf, Implementation und Testen eines Prototyps zur statischen Analyse der bestimmten Merkmale für C# basierend auf der .NET Compiler Platform.
- Experimentelle Evaluation des Prototyps z.B. anhand von Open-Source Projekte mit Fazit der gewonnenen Erkenntnisse.
- Recherche über den Stand der Technik auf diesem Gebiet.

4. Zur Durchführung

Mit dem HSR-Betreuer finden wöchentliche Besprechungen statt. Zusätzliche Besprechungen sind nach Bedarf durch die Studierenden zu veranlassen.

Alle Besprechungen sind von den Studenten mit einer Traktandenliste vorzubereiten und die Ergebnisse in einem Protokoll zu dokumentieren, welches dem Betreuer per E-Mail zugestellt wird.

Für die Durchführung der Arbeit ist ein Projektplan zu erstellen. Dabei ist auf einen kontinuierlichen und sichtbaren Arbeitsfortschritt zu achten. An Meilensteinen gemäss Projektplan sind einzelne Arbeitsresultate in vorläufigen Versionen abzugeben. Über die abgegebenen Arbeitsresultate erhalten die Studierenden ein vorläufiges Feedback. Eine definitive Beurteilung erfolgt auf Grund der am Abgabetermin abgelieferten Dokumentation.

5. Dokumentation

Über diese Arbeit ist eine Dokumentation gemäss den Richtlinien der Abteilung Informatik zu verfassen. Die zu erstellenden Dokumente sind im Projektplan festzuhalten. Alle Dokumente sind nachzuführen, d.h. sie sollten den Stand der Arbeit bei der Abgabe in konsistenter Form dokumentieren. Die Dokumentation ist vollständig digital dem Studiengang und dem Betreuer abzugeben. Auf Wunsch ist für den Auftraggeber und den Betreuer eine gedruckte Version zu erstellen.

6. Termine

Siehe auch Terminplan auf dem Skripteserver Informatik -> Fachbereich -> Studienarbeit_Informatik

19.02.18	Beginn der Studienarbeit, Ausgabe der Aufgabenstellung durch den Betreuer. Vorlagen sowie eine ausführliche Anleitung betreffend Dokumentation stehen auf dem Skripteserver zur Verfügung.
28.05.18	Die Studierenden geben das Abstract zur Kontrolle an ihren Betreuer/Examinator frei. Die Studierenden erhalten vorgängig vom Studiengangsekretariat die Aufforderung mit den Zugangsdaten zur Online-Erfassung des Abstracts im DAB-Tool.
01.06.18	Der Betreuer/Examinator gibt das Dokument mit dem korrekten und vollständigen Abstract zur Weiterverarbeitung an das Studiengangsekretariat frei.
01.06.18	Hochladen aller verlangten Dokumente auf archiv-i.hsr. Abgabe des Berichts an den Betreuer bis 17.00 Uhr.

7. Beurteilung

Eine erfolgreiche Studienarbeit erhält 8 ECTS-Punkten (1 ECTS Punkt entspricht einer Arbeitsleistung von ca. 25 bis 30 Stunden), siehe auch die Modulbeschreibung der Studienarbeit Informatik der HSR.

Gesichtspunkt	Gewicht
1. Organisation, Durchführung (Projektplanung u. Nachführung Arbeit gemäss Projektplan, Selbständigkeit, Einsatz, Zusammenarbeit mit Auftraggeber, Betreuer)	1/5
2. Bericht (Inhalt des Projektschlussberichts, Gliederung, Darstellung, Sprache der gesamten Dokumentation)	1/5
3. Inhalt *)	(3/5)
3.1 Problemanalyse (Vorstudie, Literaturstudium, Anforderungsspezifikation, Anforderungsanalyse, Domainanalyse)	1/5
3.2 Lösungsentwurf (Lösungsvarianten und deren Beurteilung, Variantenentscheid, Konzept, Entwurf)	1/5
3.3 Realisierung und Test	1/5

*) Die Gliederung des Gesichtspunktes “3. Inhalt” in einzelne Unterpunkte kann den Gegebenheiten der Arbeit angepasst werden. Das Gesamtgewicht des Gesichtspunktes bleibt hingegen bei 50%.

Im Übrigen gelten die Bestimmungen der Abt. Informatik zur Durchführung von Studienarbeiten.

Rapperswil, den 9. Februar 2018

Der verantwortliche Dozent

Prof. Dr. Luc Bläser
Hochschule für Technik Rapperswil

Abstract

Wir haben eine neue statische Design- und Stil-Analyse für C# entwickelt. Diese prüft den Code während dem Programmieren darauf, ob sich Rechtschreibfehler eingeschlichen haben, ob die Benennung von Klassen, Interfaces, Variablen, etc. den allgemeinen Richtlinien entsprechen und ob die bekanntesten Code Metriken eingehalten wurden. Die gefundenen Fehler-/Korrekturelemente werden in der IDE farblich markiert.

Die zahlreichen Analysen wurden in folgende drei Kategorien unterteilt:

Lexikalische Analyse

Zum einen wird jeder Bezeichner in ihre einzelnen Wörter unterteilt und jedes einzelne Wort wird mit der Rechtschreib-Datenbank überprüft. Zum anderen wird ermittelt, um welchen Bezeichnertyp es sich handelt und ob dieser die richtigen Wortarten enthält. Bei Klassennamen braucht es zum Beispiel ein Nomen am Ende

Code-Stil Analyse

In der Code-Stil Analyse wird der Code dahingehend überprüft, ob dieser noch zu vereinfachen wäre. Hierzu kommen mehrere Möglichkeiten in Frage, welche analysiert werden könnten.

Metrik Berechnungen

In dieser Kategorie werden bekannte Metriken berechnet. Darunter fallen Metriken wie: *Lines of Code* (LOC), oder *Lack of Cohesion in Methods* (LCOM*), und andere.

Das Projekt wurde in einem VSIX Projekt realisiert und wird somit als Visual Studio Extension zur Verfügung gestellt. Das Tool wurde über die Open Source Projekte Log4Net, Prism und Wox laufen gelassen. Dabei kam heraus, dass Log4Net am meisten auf die Code-Analyse angesprungen ist. Die gefundenen Code Smells betrafen vor allem Schreibfehler (darunter auch unnötige Abkürzungen von Wörtern), Vereinfachungen mit Typ-Inferenz (var Keyword) und zu lange Klassen oder Methoden.

Die implementierten Analysen liefern hilfreiche Meldungen zu Design- und Stil-Problemen. Dank dessen kann man sich einen guten Ein-/Überblick sowohl über die Komplexität als auch über die Qualität des Codes machen.

Management Summary

Ausgangslage

Die statische Code Analyse soll Programmierern dabei helfen besseren Code zu schreiben.

Vorgehen, Technologien

Die zu implementierenden Analysen wurden aus den Anforderungen ermittelt. Dabei handelt es sich um häufig auftauchende Fehler beim Programmieren, als auch um lexikalische Korrekturen. Die *lexikalische Analyse* verwendet Datenbanken zum Überprüfen der Rechtschreibung, als auch der Wortart. So kann sichergestellt werden, dass allgemeine Programmierrichtlinien eingehalten werden. Die *Code-Stil Analyse* überprüft den Code und versucht diesen zu vereinfachen. Die *Metrik Berechnungen* überprüfen den Code auf allgemeine Software Metriken wie Lines of Code (LOC), oder Lack of Cohesion in Methods (LCOM*).

Ergebnisse

Das Projekt wurde mit Roslyn, dem Microsoft Code..., für Visual Studio und C# entwickelt. Zahlreiche Programmierunterstützende Funktionen (Korrekturmassnahmen) wurden implementiert und wurden in folgende drei Kategorien unterteilt:

- Wörterbuch-basierte Codeanalyse
- Code-Stil Analyse
- Metrik Berechnungen

Die Analysen finden jeweils dann statt, wenn der Programmierer für ca. eine halbe Sekunde nichts auf der Tastatur eingibt. Dabei wird nur das gerade geöffnete Dokument untersucht.

1 Einleitung und Übersicht

Code Smells ist eine Form des Codes, die zwar eine korrekte Ausführung bewerkstelligen, jedoch schlechte Eigenschaften aufweisen, wie statischer Code, untestbarkeit, nicht erweiterbar, u.v.m. Diese Code Smells können durch sogenanntes Refactoring (Überarbeiten des Codes) in ihrer Form verbessert werden. Eine gute Literatur dazu bietet sich in Martin Fowler's *Refactoring* [1991] und im darauf aufbauenden Buch von Joshua Kerievsky *Refactoring to Patterns* [2004].

Programmierer stehen oft unter Stress, da sie Ihre Lösung termingerecht abgeben müssen. Es kann also durch Stress, oder auch durch Unwissenheit zur Erzeugung von Code Smells kommen. Ein einfaches sind einfache stilistische Fehler wie boolesche if-Bedingungen wie hier: `if(varX == true)`, bis hin zu Klassenbezeichnungen, die nicht mit einem Nomen enden, Methoden, die kein Verb beinhalten, oder Tippfehler in einzelnen Wörtern. Dieses Projekt soll den Code während dem Programmieren analysieren und Korrekturen anbieten.

Zur Einfachheit wurde das Projekt mit dem Namen SASKIA abgekürzt, als Annäherung zu "Semester Arbeit, Statische Code Analyse". Es wurde als Extension für Visual Studio entwickelt, für die Programmiersprache C#. Die Extension wurde dann auf einige GitHub Projekte angewendet. Folgende Auswertungen kamen zu Stande:

Dieses Dokument gliedert sich in folgende Themen:

- Das Kapitel 1.1 "**Statische Code-Analysen**" befasst sich mit den umgesetzten Programmier-Unterstützungen. Dieses Kapitel wurde in drei Unterkategorien zerlegt :
  - *Lexikalische Analyse*: Dieses Kapitel befasst sich mit dem Analysieren von Rechtschreibungsfehlern und Benennung von Klassen, Variablen, etc.
  - *Code-Stil Analyse*: Befasst sich mit dem Analysieren der Code-Zusammenstellung, sodass überflüssige/komplizierte Code Teile vereinfacht werden
  - *Metrik Berechnungen*: Befasst sich mit allgemein bekannten Code Metriken (z.B. Lines of Code, Conditional Complexity, etc.)
- Im Kapitel 1.2 **"Softwarearchitektur**" wird erklärt, wie das Projekt software-technisch aufgebaut wurde.
- Im Kapitel 3 "**Schlussfolgerungen** " befasst sich mit dem Fazit der realisierten Code-Analysen.

1.1 Statische Code-Analysen

In der statischen Code Analyse fokussieren wir und auf die Erkennung von verschiedenen Arten von Code Smells. Das folgende Kapitel beschreibt alle Analysen und Metrik-Prüfungen, die durch SASKIA vorgenommen werden können, unterteilt in lexikalische Refactorings, Code-Stil-Refactorings und Metrik-Prüfungen.
Zu beachten gibt es, dass die Refactorings jeweils nur kleine Code-Anpassungen vornehmen und es Sinn macht, diese zu kombinieren. So kann beispielsweise der Code `bool b = !((4 < x) && (x < 12));` zuerst über das DeMorganSimplifierRefactoring zu `bool b = !(4 < x) || !(x < 12);` und danach über NotOperatorInversionRefactoring zu `bool b = 4 >= x || x >= 12;` vereinfacht werden.

1.1.1 Lexikalische Analyse

Rechtschreibfehler, als auch die richtige Benennung von Klassen, Interfaces, Variablen, etc. werden hier analysiert. Zum einen wird jede Bezeichnung (z.B. ein Klassenname 'HotelBenutzer') wird in ihre einzelnen Wörter unterteilt. Jedes einzelne Wort wird mit der Rechtschreib-Datenbank überprüft. Zum anderen wird ermittelt, ob es sich um einen Klassennamen, einen Variablennamen, etc. handelt. Bei einem Klassennamen wird mit der Wortart-Datenbank überprüft, ob die Klassenbezeichnung mit einem Nomen endet. Ähnlich wird mit Interfaces, etc. vorgegangen.

TypoRefactoring

Motivation

Es kann aus verschiedenen Gründen vorkommen, dass der Programmierer ein einzelnes Wort ,z.B. bei der Benennung einer Methode, mit einem Tippfehler versieht. Hat der Programmierer diese Methode mit einem Tippfehler versehen, kann es schwierig sein, jene Methode wieder zu finden.

Lösung

- Klassen-, Interface-, Struct-, Enum-, Methoden-, Variablen- und Property-Bezeichnungen werden auf Tippfehler überprüft. Zusammengesetzte Wörter werden zerlegt und auch auf Tippfehler überprüft.
- Fehlerhafte Bezeichner werden in der IDE farblich markiert.
- SASKIA zeigt entsprechend Korrekturvorschläge an

Beispiele

Ausgangslage	Verbesserung
`class Appartment {}`	`class Apartment {}`
`class AppartmentRoom {}`	`class ApartmentRoom {}`
`class LuxuryAppartmentRoom {}`	`class LuxuryApartmentRoom {}`
`private void Methodd() {}`	`private void Method() {}`

`class Apartment {`
`private int _roooms;`
`private int Rooms {`
`get { return _roooms; }`
`}`
`}`
	`class Apartment {`
	`private int _rooms;`
	`private int Rooms {`
	`get { return _rooms; }`
	`}`
	`}`

Etc.	

WordTypeRefactoring

Motivation

In jeder Programmiersprache gibt es Namens-Konventionen. Klassen enden meistens mit einem Nomen. Methoden beinhalten Verben, etc. Ein unerfahrener Programmierer kennt diese eventuell noch nicht und gibt den entsprechenden Klassen, Methoden, etc. falsche Namen.

Lösung

- Klassen-, Interface-, Struct- und Enum-Namen werden auf korrekte Namens-Konventionen überprüft.
- Als Wörterbuch dient ein privates Projekt, namens «English-Dictionary-SQLite»[1]
- Fehlerhafte Wörter werden ersetzt

Beispiele

Ausgangslage	Verbesserung
`class Mustern {}` (Verb)	`class Muster {}` (Nomen)
`class GemaeldeMustern {}`	`class GemaeldeMuster {}`
`interface IMustern {}` (Verb)	`cnterface IMuster {}` (Nomen)
Etc.	

In den Beispielen wurden Wörter in der Deutschen Sprache abgebildet zum besseren Verständnis. Die Extension wird aber nur die Englische Sprache unterstützen. Dies birgt zugleich ein Problem. Die Englische Sprache besitzt unzählige Nomen, die zugleich als Verb benutzt werden können und umgekehrt. Zum Beispiel: „a book“ -> „to book“. Die Ermittlung, ob der Name, z.B. einer Klasse, Sinn ergibt mit so einem „gemischten Wort“ wurde nach einiger Recherchezeit keine weitere Beachtung geschenkt.

1.1.2 Code-Stil Analyse

In der Code-Stil Analyse wird der Code dahingehend überprüft, ob dieser noch zu vereinfachen wäre. Hierzu kommen mehrere Möglichkeiten in Frage, welche Analysiert werden könnten. Genauere Details im entsprechend benannten Kapitel.

BooleanConstantComparisonRefactoring

Motivation

Häufig schreiben unerfahrene Programmierer Code, in welchem Vergleiche mit booleschen Konstanten stattfinden. Dadurch wird der Code unleserlicher.

Lösung

SASKIA soll Vergleiche mit Boolean-Konstanten erkennen und Refactorings anbieten, um diese zu kürzen.

Die folgenden Refactorings sollten durch SASKIA vorgenommen werden können:

- Vergleiche mit **true** werden weggelassen.
- Vergleiche mit **false** werden durch den Not-Operator realisiert.
- Vergleiche mit **!=** erfolgen analog.
- Ausdrücke mit binären Operatoren, welche mit **false** verglichen werden, müssen geklammert werden, wobei Vergleiche mit **true** nicht geklammert werden sollten.
- Abgrenzung: Dieses Refactoring vereinfacht komplexe Ausdrücke nicht zu Boolean-Konstanten. Dies wird durch **BooleanConstantSimplifierRefactoring** realisiert.

Beispiele

Ausgangslage	Verbesserung
`return boolVariable == true;`	`return boolVariable;`
`var x = false == BoolFunction();`	`var x = !BoolReturningFunction();`
`CallMethod(4 < 6 == false);`	`CallMethod(!(4 < 6));`
`var x = boolVar != true;`	`var x = !boolVar;`

[1] https://github.com/AyeshJayasekara/English-Dictionary-SQLite, Letzter Zugriff: 30.04.2018

BooleanConstantSimplifierRefactoring

Motivation
Häufig schreiben Programmierer konstante Ausdrücke, die durch ein Literal ersetzt werden könnten. Dieses Refactoring vereinfacht deswegen konstante Boolean-Ausdrücke und ersetzt sie mit einem Boolean-Literal.

Lösung
- Vereinfachung von **NOT**-Operatoren vor konstanten booleschen Ausdrücken.
- Vereinfachung von logischen Short-Circuit-Operatoren && und ||.
- Geklammerte boolesche Ausdrücke, deren Klammer unnötig ist, werden ausgeklammert.

Beispiele

Ausgangslage	Verbesserung
`A(!false && true);`	`A(true);`
`return !(!false);`	`return false;`
`if (true && false || true) { ....`	`if (true) { ....`

IfReturnBooleanRefactoring

Motivation
Viele Programmieranfänger schreiben bei Boolean-Methoden häufig ein If-Statement und geben im Then-Block **true** und im else Block **false** zurück oder umgekehrt. Dies soll durch ein Refactoring abgekürzt werden können.

Lösung
Das Refactoring-Tool ersetzt das if-Statement durch ein einfaches return-Statement. Dadurch wird der Code leserlicher. Bei Verneinung der if-Condition müssen binäre Operatoren wie im Beispiel unten geklammert werden.
Dabei müssen auch die folgenden Fälle berücksichtigt werden:
- Der **Then**- und **Else**-Block können beliebig in Blöcken verschachtelt sein.
- Das Literal kann beliebig in runden Klammern verschachtelt sein.

Beispiele

Ausgangslage	Verbesserung

`if (x == 4) {`
`    return false;`
`} else {`
`    return true;`
`}`
	`return !(x == 4);`

`if (A())`
`    return ((true));`
`else {`
`    return false;`
`}`
	`return A();`

DeMorganSimplifierRefactoring

Motivation
Häufig werden von Programmierern AND und OR-Verknüpfungen geschrieben, welche dann wiederum negiert werden (z.B. if (!(x < 12 && x > 5))). Bei komplexeren Bedingungen kann dies jedoch ziemlich unübersichtlich werden, sodass man den Ausdruck über das DeMorgans-Law vereinfachen kann.

Lösung
SASKIA erkennt alle Verneinungen von OR und AND-Verknüpfungen und schlägt vor den Ausdruck durch das DeMorgan-Law zu vereinfachen. Durch das Refactoring **NotOperatorInversion** lässt sich dann der Ausdruck weiter vereinfachen.

Beispiele

Ausgangslage	Verbesserung
`bool expr = !(x == 4 || x == 12 || x == 3);`	`bool expr = !(x == 4 || x == 12) && !(x == 3);`
`if (!(x == 4 || x == 12)) {}`	`if (!(x == 4) && !(x == 12)) {}` Lässt sich durch das NotOperatorInversionRefactoring zu `if (x != 4 && x != 12) { }` weiter vereinfachen.

NotOperatorInversionRefactoring

Motivation

Es kommt vor, dass Programmierer einen binären Ausdruck verneinen, anstatt direkt den Umkehroperator zu verwenden. So trifft man beispielsweise auf Ausdrücke wie !(x < 4), welche besser direkt als x >= 4 geschrieben werden würden. Es wäre also sinnvoll, dazu ein Refactoring anzubieten.

Lösung

SASKIA schlägt bei Verneinungen von binären Operatoren die folgenden Verbesserungen vor. Die folgenden Operatoren werden unterstützt:

- "Kleiner als" (<) wird durch die Verneinung zu "grösser gleich" (>=)
- "Grösser als" (>) wird durch die Verneinung zu "kleiner gleich" (<=)
- "Kleiner gleich" (<=) wird durch die Verneinung zu "grösser als" (>)
- "Grösser gleich" (>=) wird durch die Verneinung zu "kleiner als" (<)
- Der Vergleichsoperator (==) wird durch die Verneinung zu "nicht gleich" (!=) und umgekehrt.

Beispiele

Ausgangslage	Verbesserung

`if (!(x == 4) && !(x >= 12)) {`
`    ...`
`}`
	`if (x != 4 && x < 12) {`
	`    ...`
	`}`

`Console.WriteLine(!(x < 12));`	`Console.WriteLine(x >= 12);`

IfAndElseBlockEqualsRefactoring

Motivation

Es kommt vor, dass Programmierer im Hitze des Gefechts identische if- und else-Blöcke schreiben. Dabei lässt sich durch dieses Refactoring das if-else-Statement entfernen. Selbstverständlich muss das Refactoring berücksichtigen, dass Seiteneffekt-behaftete if-Conditions nicht verloren gehen.

Lösung

Das Refactoring vergleicht der Syntaxbaum vom Then-Block mit dem Syntaxbaum vom Else-Block ohne Berücksichtigung von Whitespaces, Kommentaren und ob der if-Block oder else-Block in geschweiften Klammern steht. Falls die if-Bedingung nicht seiteneffektfrei ist, muss sie im Code vorhanden bleiben.

Die folgenden if-Conditions werden präventiv als Seiteneffekt-behaftet angesehen:

- Methodenaufrufe / Eventaufrufe
- Propertyzugriffe
- Zugriffe auf Felder, die nicht von einem Propertyzugriff unterschieden werden können, da sie auf ein anderes File zugreifen müssten.

Beispiele

<table>
<tr><th>Ausgangslage</th><th>Verbesserung</th></tr>
<tr><td>x und y sind Felder
<pre>if (this.x == this.y) {
    Console.WriteLine("hello");
} else {
    Console.WriteLine("hello");
}</pre></td><td><pre>Console.WriteLine("hello");</pre></td></tr>
<tr><td>X und Y sind Properties
<pre>if (base.X == base.Y) {
    Console.WriteLine("hello");
} else {
    // comment
    Console.WriteLine("hello");
}</pre></td><td><pre>bool condition = base.X == base.Y
Console.WriteLine("hello");</pre></td></tr>
</table>

IllegalFieldAccessRefactoring

Motivation

In den meisten Softwareengineer-Umfeldern ist es Praxis, alle Felder **private** zu machen. Klassen sollten nur über Properties (Getter & Setter) und andere Methoden kommunizieren. SASKIA korrigiert ebenfalls **protected**, **protected internal** und **internal**-Felder zu **private** da der Zugriff von Klassen im selben Assembly oder Unterklassen auf Feldern ebenfalls die Klassenkopplung unnötig erhöht. SASKIA empfiehlt dem Programmierer klassenübergreifende Zugriffe nur über Properties und Methoden zu machen. Einzig als const deklarierte Felder werden von der Überprüfung ausgeschlossen.

Lösung

SASKIA meldet alle Felder, die nicht **private** sind und bietet an es zu refactoren.

Beispiele

<table>
<tr><th>Ausgangslage</th><th>Verbesserung</th></tr>
<tr><td><pre>class Bird {
    protected internal List<Wing> wings;
    public int age;
    internal static List<Bird> birdPool;
}</pre></td><td><pre>class Bird {
    private List<Wing> wings;
    private int age;
    private static List<Bird> birdPool;
}</pre></td></tr>
</table>

IntegerConstantSimplifierRefactoring

Motivation

Programmierer schreiben gelegentlich Integer-Ausdrücke im Source-Code, welche jedoch nur aus Literalen bestehen und auch durch ein einzelnes Integer-Literal ausgedrückt werden könnten. Dadurch wird der Source-Code leserlicher.

Lösung

SASKIA soll ein Refactoring bereitstellen, bei dem konstante Integer-Ausdrücke durch ein einfaches Literal ersetzt werden können. Dadurch wird der Wert über das Visitor-Pattern evaluiert. Folgende Operationen werden unterstützt:

- Die unären Vorzeichen + und -
- Der "bitwise complement"-Operator ~
- Die Rechenoperatoren +, -, *, /, %
- Der Left-Shift << und Right-Shift >>-Operator
- Die "bitwise and, or und xor"-Operatoren &, |, ^

Die Operatoren ++ und -- werden von SASKIA nicht berücksichtigt. Ebenfalls kann SASKIA bei Divisionen und Modulo nur eine Vereinfachung vornehmen, sofern nicht durch 0 geteilt

wird. Falls durch das Literal 0 geteilt wird, meldet Visual Studio bereits eine Fehlermeldung. Die Erkennung von Overflows wird bei diesem Refactoring nicht abgedeckt.

Beispiele

Ausgangslage	Verbesserung
`return 4 + 12 * 7;`	`return 88;`
`var x = (4 + 12) - 18 & 5;`	`var x = 4;`

LongConstantSimplifierRefactoring

Motivation

Äquivalent zu Integern sollte das **IntegerConstantRefactoring** auch für Longs funktionieren. Deshalb wird hier auf das vorherige Unterkapitel verwiesen. Dennoch sollen ein paar Beispiele das Refactoring erläutern. Zu beachten gibt es, dass Long die Operatoren << und >> nicht unterstützt.

Beispiele

Ausgangslage	Verbesserung
`Console.WriteLine(4L + 20L % 3L);`	`Console.WriteLine(6L);`
`var number = 25L * 2L / 5L;`	`var number = 10L;`

PotentialStaticMethodRefactoring

Motivation

Methoden, welche keine Abhängigkeiten auf das Objekt haben, können in C# als static deklariert werden. Dadurch wird sichergestellt, dass der Aufruf der Methode keine Instanz benötigt. Zudem lassen sich durch statische-Methoden die Performance leicht erhöhen. Um fremden Code nicht zu breaken sollte dieses Refactoring jedoch nur auf **private**-Methoden angewendet werden.

Lösung

Alle Methoden werden von SASKIA geprüft. Erfüllt die Methode alle folgenden Punkte, kann die Methode static gemacht werden:

- Keine Referenzen auf **this** oder **base**
- Keine nicht-statischen Methodenaufrufe in der eigenen Klasse vorhanden
- Keine nicht-statischen Property- oder Felderzugriffe in der eigenen Klasse vorhanden.

Durch SASKIA sollte die Methode direkt statisch gemacht werden können. Es werden nur **private**-Methoden berücksichtigt.

Beispiele

Ausgangslage

```
class MathUtils {
    public int[] MinMax(int lhs, int rhs) {
        if (lhs > rhs) {
            Swap(ref lhs, ref rhs);
        }

        return new[] { lhs, rhs };
    }

    private void Swap(ref int lhs, ref int rhs) {
        var temp = lhs;
        lhs = second;
        rhs = temp;
    }
}
```

Verbesserung

```
class MathUtils {
    public int[] MinMax(int lhs, int rhs) {
        if (lhs > rhs) {
            Swap(ref lhs, ref rhs);
        }

        return new[] { lhs, rhs };
    }

    private static void Swap(ref int lhs, ref int
rhs) {
        var temp = lhs;
        lhs = second;
        rhs = temp;
    }
}
```

TypeIdentifierConventionRefactoring

Motivation

Typbezeichnungen werden bei C# gross und im CamelCase-Format geschrieben. Underlines werden für Typnamen in C# nicht verwendet. SASKIA soll unkonventionell benannte Typbezeichnungen korrigieren können.

Lösung

Die folgenden Korrekturen können durch SASKIA bei allen Typbezeichnungen vorgenommen werden:

- Underlines sollten in Camel-Case umgeschrieben werden können.
- Klassennamen, Delegatenamen, Structnamen, Enumnamen sollten grossgeschrieben werden und ohne ein spezifisches Präfix.
- Interfacenamen sollten ebenfalls grossgeschrieben werden und mit einem grossen "I" beginnen.

Ausgangslage	Verbesserung

`public class natural_aquarium {`
`    ...`
`}`
	`public class NaturalAquarium {`
	`    ...`
	`}`

`public interface _convertible {`
`    ....`
`}`
	`public interface IConvertible {`
	`    ....`
	`}`

MethodPropertyIdentifierConvention

Motivation

Methoden und Properties werden bei C# gross und im CamelCase-Format geschrieben. Underlines werden für Methodennamen in C# nicht verwendet. Das Refactoring-Tool SASKIA sollte dies melden.

Lösung

Die folgenden Korrekturen können durch SASKIA vorgenommen werden:

- kleingeschriebene Methoden und Propertynamen werden durch grossgeschriebene Namen ersetzt.
- Underlines im Methodennamen werden durch die CamelCase-Schreibweise ersetzt.
- Führende Underlines werden entfernt.

Beispiele

Ausgangslage	Verbesserung
`void swim_to_surface() {` `}`	`void SwimToSurface() {` `}`

`public string _first_name`
`{`
`    get; private set;`
`}`
	`public string FirstName`
	`{`
	`    get; private set;`
	`}`

UseOfVarRefactoring

Motivation

In C# ist es möglich den Typ einer lokalen Variable vom Compiler bestimmen zu lassen, um nicht explizit angeben zu müssen. Dies geschieht über das Keyword **var**. Dies ist oft vorteilhaft, da sich lange Typen wie generische Typen abkürzen lassen. Aber auch bei Typen wie string oder DateTime wäre es redundant den Typ bei einer lokalen Variable angeben zu müssen. Um das Refactoring einfacher zu halten, wird jedoch var nur vorgeschlagen, wenn der zugewiesene Datentyp exakt identisch mit dem Typ der Variablendeklaration ist.

Lösung

SASKIA schlägt die Verwendung des Keywordsvar in folgenden Fällen vor:

- nur bei lokalen Variablen, die nicht als **const** markiert sind.
- nur bei Klassentypen (nicht bei primitiven Typen wie int x = 3.4f)

Beispiele

Ausgangslage	Verbesserung

`int number = 4;`
`string message = "hallo";`
`Customer customer = new Customer();`
`ICustomer customer = new Customer();`
	`int number = 4;`
	`var message = "hallo";`
	`var customer = new Customer();`
	`ICustomer customer = new Customer();`

WhitespaceFixRefactoring

Motivation
Häufig werden Programme geschrieben bei denen die korrekte Einrückung des Codes vernachlässigt wird. Dies ist jedoch keine gute Idee, da eine inkonsistente Einrückung des Codes die Lesbarkeit beeinträchtigt.

Lösung
Durch SASKIA lässt sich auf über den **namespace**-Block direkt das ganze namespace richtig einrücken.

1.1.3 Metrik Berechnungen

ConditionalComplexityRefactoring

Motivation
McCabe kam der Gedanke, dass Softwarekomponenten ab einer gewissen Komplexität für den Menschen nicht mehr begreifbar sind und erfand eine sehr verbreitete Formel für die Messung der Komplexität innerhalb einer Methode, wobei e die Anzahl Kanten und n die Anzahl Knoten im Kontrollflussgraph sind. Kurz gesagt ist M höher, wenn die Methode mehr Verzweigungen wie **if**-Statements, Schleifen, **try**-catch-Blöcke oder subtilere Sachen wie **&&** oder **||** enthält.

M = e – n + 2
Mehr zur McCabe-Metrik: https://www.st.cs.uni-saarland.de/edu/se2/metriken.pdf

Eine einfachere Formel, die zum gleichen Ergebnis kommt lautet wie folgt:
M = "Anzahl bedingte Sprünge" + 1

Zudem wird eine Komplexität M >= 10 als kritisch angesehen, wonach sich auch SASKIA richtet.

Lösung
Die Metrik unterstützt alle uns bekannten Verzweigungen (bedingten Sprünge), die C# kennt:

- Die binären Operatoren **&&, ||**, welche im Grunde genommen auch if-Statements sind.
- Die Case-Anweisung in switch-Statements
- Jeder catch-Block in einer Exception und deren Filter-Klauseln (**when**)
- Conditional-Access-Expressions wie customer?.FirstName
- Conditional-Expressions wie x == null ? 4 : 5
- If- und ElseIf-Blöcke
- Die drei Schleifentypen **for, foreach und while**
- Abgrenzung: Unbedingte Sprünge wie **else, return, goto, continue, break**, … sind keine Erhöhung der Komplexität

Beispiele

Ausgangslage	Metrikberechnung

<table>
<tr><td><pre>long Fi(int i) {
    if (i < 0) return -1;
    if (i < 2) return 1;

    return Fi(i - 1) + Fi(i - 2);
}</pre></td><td>M = 2 "if" + 1 = 3</td></tr>
<tr><td><pre>public static void Sort(int[] data)
{
    int i, j;
    int N = data.Length;

    for (j=N-1; j>0; j--) {
        for (i=0; i<j; i++) {
           if (data [i] > data [i + 1])
                exchange (data, i, i + 1);
        }
    }
}</pre></td><td>M = 2 "for" + 1 "if" + 1 = 4</td></tr>
</table>

DepthOfInheritanceRefactoring

Motivation

Wenn man im Sourcecode zu viele Klassenhierarchien hat, dann wird es schnell unverständlich.
Deshalb soll SASKIA warnen, wenn die Klassenhierarchie über zu viele Stufen geht.
Die DIT-Metrik einer Klasse lässt sich durch Anzahl Vererbungsstufen zwischen der Klasse und der Klasse **object** bestimmen.

Siehe: https://www.st.cs.uni-saarland.de/edu/se2/metriken.pdf

Lösung

Bei jeder Klasse wird zur Parent-Klasse navigiert, bis die Klasse **object** explizit als Parent-Klasse angegeben wird oder keine Parent-Klasse spezifiziert ist. Dabei wird bei jedem Schritt ein Zähler erhöht.

Für den ersten Schritt unterstützt die Metrik nur Klassen, die im selben File liegen. Später kann man die Metrik auch File-übergreifend umsetzen.

Beispiele

<table>
<tr><th>Ausgangslage</th><th>Metrikberechnung</th></tr>
<tr><td><pre>class Animal {
}

class Mammal : Animal {
}

class Elephant : Mammal {
}

class IndianElephant : Elephant {
}</pre></td><td>Die Klasse Animal hat DIT = 1.
Die Klasse Mammal hat DIT = 2.
Die Klasse Elephant hat DIT = 3.
Die Klasse IndianElephant hat DIT = 4.</td></tr>
</table>

<table>
<tr><td>

```
interface ISerializable {
    byte[] GetBytes();
}

class Entity : object, ISerializable {
    public abstract byte[] GetBytes();
}

class Customer : Entity {
    public override byte[] GetBytes()
{
        return new[] { 1, 2, 3 };
    }
}
```

</td><td>Die Klasse Entity hat DIT = 1.
Die Klasse Customer hat DIT = 2.</td></tr>
<tr><td></td><td></td></tr>
</table>

LackOfCohesionRefactoring

Motivation

Wichtige Prinzipien in der Softwareentwicklung sind die **hohe Kohäsion** innerhalb eines Moduls, sowie das Single-Responsibility-Prinzip. Häufig wird gerade auf Klassenebene gegen dieses Prinzip verstossen.
Mehr zur Metrik: http://ecet.ecs.uni-ruse.bg/cst06/Docs/cp/SII/II.10.pdf

Lösungs

SASKIA wendet die Metrik **LCOM** an, um eine Aussage über die Kohäsion innerhalb einer Klasse zu machen. Dabei prüft LCOM, ob die Attribute von der Mehrheit der Methoden in einer Klasse benötigt werden. Dabei muss SASKIA auf die Symboltabelle zurückgreifen, damit lokale Variablen, welche Felder überdecken das Ergebnis der Metrik nicht verfälschen.

Die Formel zur Berechnung der LCOM-Metrik lautet wie folgt:
LCOM = (m - avg(m(A))) / (m – 1)

Dabei sind **m** die Anzahl Methoden innerhalb der Klasse und **avg(m(A))** ist der Durchschnitt der Methoden, die auf ein Attribut zugreifen.
Ein LCOM-Wert in der Nähe von 1 bedeutet schlechte Kohäsion, während ein Wert in der Nähe von 0 eine gute Kohäsion bedeutet.

Um Divisionen mit Null zu verhindern, kann die Formel nur für Klassen mit mindestens 2 Methoden angewendet werden.

Beispiele

<table>
<tr><th>Ausgangslage</th><th>Metrikberechnung</th></tr>
<tr><td>

```
class Fish {
    private string name;
    private int age;

    public override string ToString() {
        return $"{name} {age}";
    }

    public void IncreaseAge() {
        ++age;
    }
}
```

</td><td>Anzahl Methoden m = 2
Methodenzugriffe auf das Attribut name:
m(A1) = 1
Methodenzugriffe auf das Attribut age:
m(A2) = 2
Durchschnittliche Zugriffe auf ein Attribut:
avg(m(A)) = 1.5

LCOM = (2 - 1.5) / (2 - 1) = 0.5 (Mittelhohe Kohäsion)</td></tr>
<tr><td>

```
class SqlConnection {
    private int ressourceId;
```

</td><td>Anzahl Methoden m = 3</td></tr>
</table>

<pre>   public void Open() {
        ressourceId = NativeOpen();
    }
 	
    public SqlCommand
CreateCommand(string sql) {
        return new SqlCommand(sql,
ressourceId);
    }
 	
    public void Close() {
        NativeClose(ressourceId);
    }
}</pre>
	Methodenzugriffe auf das Attribut ressourceId: m(A1) = Durchschnittliche Zugriffe auf ein Attribut = 3
	
	LCOM = (3 - 3) / (3 - 1) = 0 (Hohe Kohäsion)

LinesOfCodeRefactoring

Motivation

Lange Klassen und Methoden verleiten dazu gegen ziemlich alle Software-Engineering-Prinzipien zu verstossen. Dazu wenige Beispiele:

- Lange Klassen und Methoden haben meist eine **tiefe Kohäsion**.
- Sie bestehen meist aus **Spaghetticode**.
- Wird der Code nicht in logische Teile unterteilt, entsteht häufig ein **Comment Smell.**
- Das **Single-Responsibility-Prinzip** wird bei einer langen Klasse meist nicht eingehalten.

Lösung

SASKIA meldet, wenn eine Methode oder eine Klasse eine gewisse Anzahl Zeilen überschritten hat und gibt dem Programmierer eine entsprechende Meldung aus. Dazu zählt SASKIA lediglich die Anzahl Zeilenumbrüche in einer Klasse oder einer Methode.

Beispiele

<pre>class SqlConnection {
    private int ressourceId;
 	
    public void Open() {
        ressourceId = NativeOpen();
    }
 	
    public SqlCommand
CreateCommand(string sql) {
        return new SqlCommand(sql,
ressourceId);
    }
 	
    public void Close() {
        NativeClose(ressourceId);
    }
}</pre>
	Die Klasse SqlConnection hat 15 LOC. Dies ist eher kurz für eine Klasse, weshalb SASKIA dies nicht als Warnung anzeigen wird.

LongParameterListRefactoring

Motivation

Methoden, welche lange Parameterlisten aufweisen erhöhen die Kopplung zwischen den Klassen und die Komplexität der Methodenaufrufe. Zudem wird der Code verständlicher, wenn die Methoden weniger Parameter haben. Auch sind zu viele Parameter eine häufige Quelle für versteckten duplizierten Code.

Lösung

Für alle Methoden werden die Parameter gezählt. Wird eine Parameterzahl (z.B. 4) überschritten, dann meldet SASKIA, dass die Parameterliste zu lang sein könnte.

- Methoden, die zu lange Parameterlisten haben werden von SASKIA markiert.

- Parameter, mit dem **params**-Modifier zählen als 1 Parameter (muss speziell abgehandelt werden)
- Parameter, welche mit **out** oder **ref** markiert wurden dürfen auch nur einmal gezählt werden (muss speziell abgehandelt werden).

Beispiele

Ausgangslage	Metrik Berechnung
`private void Swap(ref int left, ref int right) { .... }`	Die Methode hat nur zwei Parameter und wird deshalb nicht als kritisch angesehen.
`public int Update(int id, string name, string surname, string streetAddress, string streetAddress2, string postcode, string town, string city, string nationality, string age, string gender, string job) { }`	Diese Methode hat 12 Parameter. SASKIA warnt hier vor einer "**Long Parameter List**".

1.2 Softwarearchitektur

1.2.1 Projekte

Im Folgenden werden die Projekte in der SASKIA-Solution kurz beschrieben.

SASKIA

Das Saskia-Projekt ist das ausführbare VSIX-Projekt, welches vom VisualStudio als Plugin gestartet wird. Es enthält die zwei Klassen **CodeSmellCodeFixProvider** und **CodeSmellDiagnosticAnalyzer**, welche die von Roslyn bekannten Klassen **CodeFixProvider** und **DiagnosticAnalyzer** um Funktionalitäten erweitern, die von allen Refactorings benötigt werden. Die Unterklassen sind ebenfalls in diesem Projekt abgelegt, implementieren die Refactorings jedoch nicht selbstständig. Sie dienen lediglich als Adapter für die Refactorings, welche im **Refactoring**-Projekt abgelegt sind.

Refactoring

Das Refactoring-Projekt enthält alle Refactoring-Klassen (Klassen, welche **IRefactoring** implementieren). Das IRefactoring-Interface schreibt Hook-Methoden vor, welche im SASKIA-Projekt vom **CodeSmellCodeFixProvider/CodeSmellDiagnosticAnalyzer** aufgerufen werden. Die Implementation des Refactorings wird über den Konstruktoraufruf der Unterklassen gesetzt.
Zudem sind in diesem Projekt auch Helper-Klassen für die Refactorings abgelegt, wie die Bestimmung der Wortart eines Wortes.

RefactoringTesting

Dieses Projekt enthält die Unit Tests für das Projekt **Refactoring**. Dazu wird für jede getestete Klasse eine Testklasse erstellt.

1.2.2 Wichtigste Klassen & Interfaces

Dieses Kapitel beschreibt die wichtigsten, für das Verständnis relevanten, Klassen der SASKIA-Solution.

SASKIA.CodeSmellDiagnosticAnalyzer

Diese Klasse enthält den gemeinsamen Code aller DiagnosticAnalyzer-Klassen. Dazu gehören die folgenden Verantwortlichkeiten:

- Initialize-Methode, welche die Refactorings auf bestimmte SyntaxNodes registriert.
- Erzeugen von einheitlichen Diagnoseobjekten
- Aufrufen der IRefactoring-Hook-Methoden in der richtigen Reihenfolge

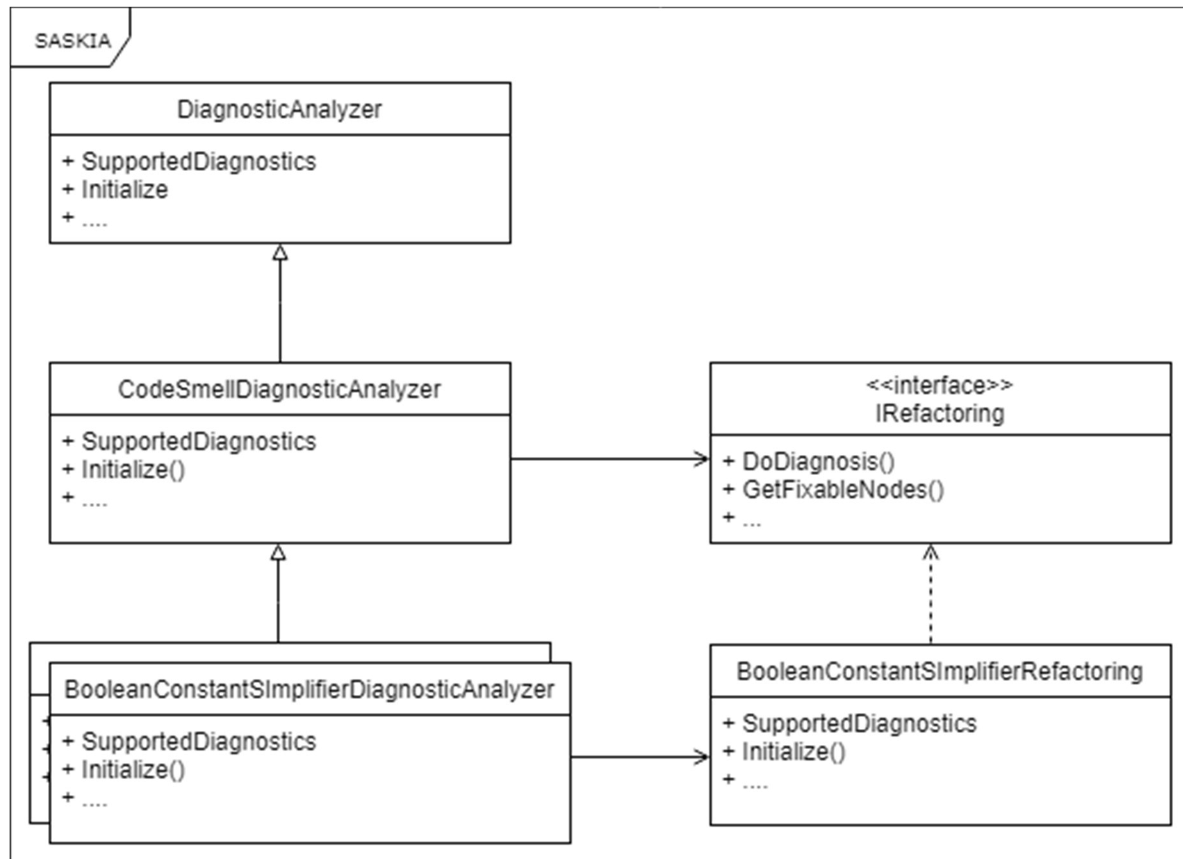


Abbildung 1: DiagnosticAnalyzer Vererbungshierarchie

Wie in Abbildung 1 zu erkennen ist, setzt die Klasse **BooleanConstantSimplifier-DiagnosticAnalyzer** über den Konstruktoraufruf seiner Basisklasse das zu verwendende Refactoring (**BooleanConstantSimplifierRefactoring**). CodeSmellDiagnosticAnalyzer ruft auf das IRefactoring-Interface die nötigen Hook-Methoden auf, welche von **BooleanConstantRefactoring** implementiert werden.

SASKIA.CodeSmellCodeFixProvider

Diese Klasse beherbergt den gemeinsamen Code alles CodeFix-Providers und ruft ebenfalls die Hookmethoden auf das IRefactoring-Interface auf.
Zu den Verantwortlichkeiten gehören

- Registrieren des Code-Actions
- Der allgemeine Code-Flow eines Refactorings wie SyntaxTree durch anderen ersetzen. Die Unterschiede der Refactorings geschehen über die Hook-Methoden.

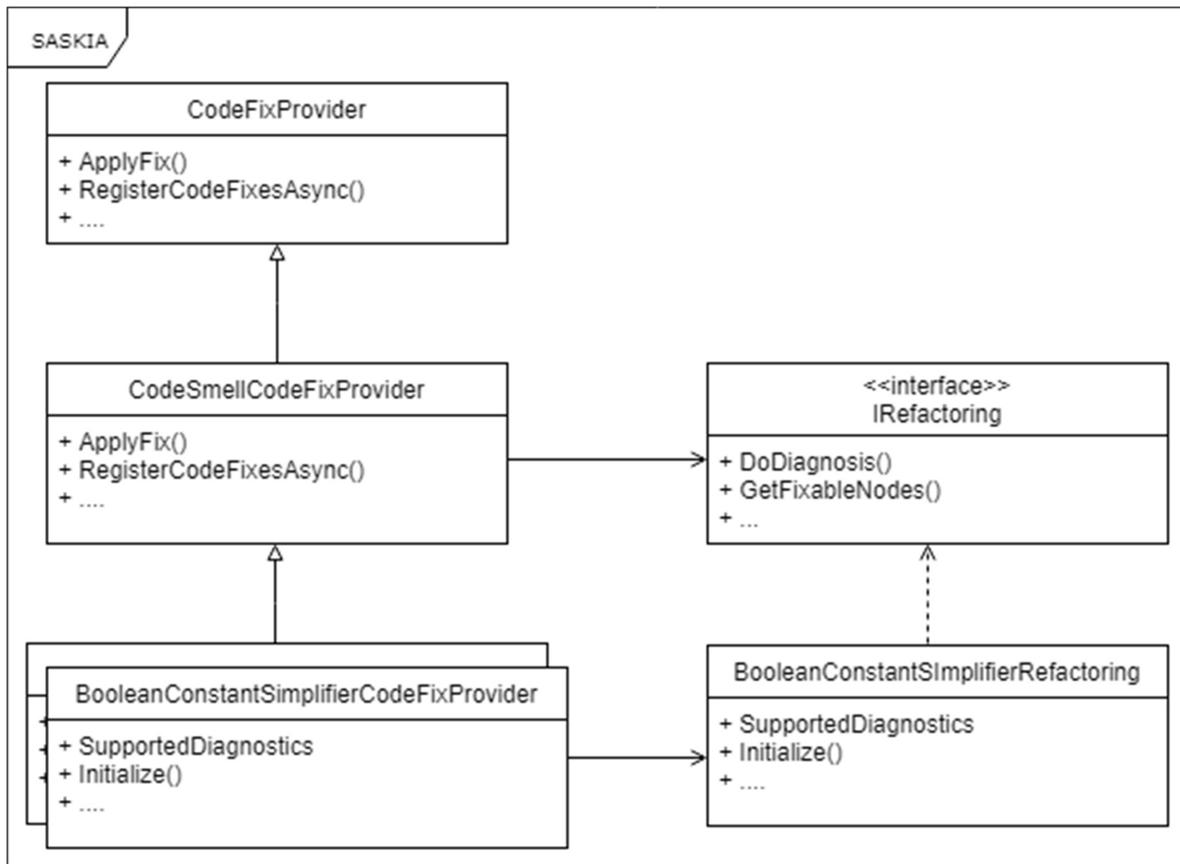


Abbildung 2: CodeFixProvider Vererbungshierarchie

In Abbildung 2: Analog zu den DiagnosticAnalyzern setzen ebenfalls die Unterklassen von **CodeSmell-CodeFixProvider** die Implementation des IRefactorings über den Konstruktoraufruf bei der Basisklasse. Auch hier kennt der **CodeSmellCodeFixProvider** nur das Interface IRefactoring und nicht die Implementation. Die CodeFixes werden von der Klasse **CodeSmell-CodeFixProvider** vorgenommen, jedoch werden auf dem **IRefactoring**-Interface die Hook-Methoden aufgerufen.

Refactoring.IRefactoring

Das **IRefactoring**-Interface schreibt vor, welche Methoden durch die Refactorings implementiert werden müssen. Diese Methoden werden dann von den Klassen **CodeSmellDiagnosticAnalyzer** und **CodeSmellCodeFixProvider** aufgerufen. Da es in den DiagnosticAnalyzer und CodeFixProvider meist sehr ähnlichen Code hat, wurde entschieden das Refactoring in einer Klasse (den Unterklassen von **IRefactoring**) zu implementieren.

Werden neue Refactorings erstellt müssen die Methoden von IRefactoring bekannt sein. Deshalb werden sie in der folgenden Tabelle kurz beschrieben.

Methode / Property	Zweck
`string DiagnosticId { get; }`	Eindeutige ID für das Refactoring (z.B. SASKIA001)
`string Title { get; }`	Titel, welcher z.B. in der „Error View“ zu sehen ist.
`string Description { get; }`	Beschreibung, welche sichtbar ist, wenn man das Refactoring ausführen möchte
`IEnumerable<SyntaxKind> GetSyntaxKindsToRecognize();`	Gibt eine Auflistung aller SyntaxNodes (wie ClassDeclaration) zurück, welche das Refactoring auslösen sollen

`DiagnosticInfo DoDiagnosis(SyntaxNode node);`	Untersucht den SyntaxNode und gibt als Resultat zurück, ob der Knoten korrekt ist.
`IEnumerable<SyntaxNode> GetFixableNodes(SyntaxNode node);`	Der Methode wird ein Syntax-Knoten übergeben und als Rückgabe gibt die Methode eine Auflistung der ersetzenden SyntaxKnoten zurück. Wird **null** zurückgegeben so bedeutet dies, dass kein Syntax-Knoten ersetzt wird.
`SyntaxNode GetReplaceableNode(SyntaxToken token);`	Durch diese Methode kann ein Refactoring aus dem ausgewählten SyntaxToken (Stelle im Code bei der das Refactoring ausgelöst wurde) den zu ersetzenden Knoten bestimmen.

1.3 Datenbanken

Tippfehler Analyse

Um Tippfehler zu analysieren, wird eine Datenbank namens Hunspell[2] verwendet. Hunspell ist der Spell-Checker von Libre-Office. Für die Einbindung von Hunspell wird die Extension NHunspell[3] verwendet. NHunspell ist ein privates Open Source Projekt, welches auf Hunspell aufbaut. Auf die Implementation wird in der Installationsanleitung näher eingegangen (siehe Anhang). NHunspell ist Lizenziert unter *GNU Lesser General Public License version 3* (siehe Fussnote[4]).

Wortart Analyse

Da Hunspell (siehe Kapitel *TypoRefactoring*) nur für Spell-Check benutzt werden kann, muss hier eine andere/weitere Wörterbuch Datenbank benutzt werden. In der Recherche wurden wir auf die Datenbank *English-Dictionary-SQlite*[5] aufmerksam. Diese Datenbank ist auch ein privates Open Source Projekt und steht unter der *MIT Licence*[6]. Im Vergleich zu Hunspell ist der Zugriff auf die SQLite Datenbank langsam. Aus diesem Grund wird in der Extension die Datenbank beim ersten Zugriff ins RAM kopiert. Dies hat eine Signifikante Leistungssteigerung zur Folge. Auf die Implementation wird in der Installationsanleitung näher eingegangen (siehe Anhang).

[2] http://hunspell.github.io/, letzter Zugriff: 30.04.2018
[3] https://www.codeproject.com/Articles/33658/NHunspell-Hunspell-for-the-NET-platform, letzter Zugriff: 29.04.2018
[4] https://opensource.org/licenses/lgpl-3.0.html, letzter Zugriff: 14.05.2018
[5] https://github.com/AyeshJayasekara/English-Dictionary-SQLite, letzter Zugriff: 27.05.2018
[6] https://opensource.org/licenses/MIT, letzter Zufgriff: 27.05.2018

2 Evaluation

Das folgende Kapitel beschreibt die Evaluation des SASKIA-Projekts durch das Anwenden des Refactorings auf bestehende C#-Projekte, die aus dem GitHub gezogen wurden. Die von SASKIA zur Evaluation benötigte Dauer berechnet sich aus der Zeit bei der SASKIA das letzte Refactoring protokolliert hat minus der Zeit bei der SASKIA die erste Meldung protokolliert hat. Demzufolge ist das Aufstarten der Experimentalinstanz nicht in der Zeit eingerechnet. Die Messzeiten wurden auf einem PC erhoben mit **8GB Arbeitsspeicher** und einem **Intel ® Core ™ i7-2630QM CPU mit 2 GHz**. Man kann davon ausgehen, dass die Analyze der Solutions in Produktion schneller ist, da das Logging in die Files ebenfalls zeitaufwändig ist.

2.1 Log4Net

Im Folgenden werden die Resultate der Auswertung des Projekts Log4Net beschrieben. Log4 Net ist ein verbreitetes Logging-Framework für C#-Applikationen und kann über GitHub unter folgendem Link heruntergeladen werden:
https://github.com/apache/logging-log4net

2.1.1 Messdaten

Die folgenden Messdaten wurden durch die Evaluation von SASKIA erhoben.

Dauer der Evaluation (1. Durchgang)	17:54 Minuten
Dauer der Evaluation (2. Durchgang)	09:59 Minuten
Anzahl der C#-Files	357

Um das Log4Net-Projekt zu analysieren benötigte SASKIA **17:54 Minuten**. Log4Net enthält **357** C#-Files. Nach den Korrekturen des Refactorings dauerte es noch **14 Minuten**.

2.1.2 Häufigkeit der Refactorings

Die folgende Tabelle enthält die Anzahl von SASKIA gemeldeten Warnungen total und berechnet relativ zur Anzahl C#-Files vor und nach den unten protokollierten Massnahmen.

Refactoring	Vorkommnisse 1. Durchgang	Vorkommnisse 2. Durchgang
BooleanConstantComparison	4 (0,011)	2 (0,006)
ConditionalComplexity	18 (0,050)	18 (0,050)
DeMorganSimplifier	14 (0,039)	13 (0,036)
IllegalFieldAccess	76 (0,213)	54 (0,151)
IntegerConstantSimplifier	51 (0,143)	7 (0,020)
LackOfCohesion	43 (0,120)	28 (0,078)
LinesOfCode	1012 (2,835)	324 (0,908)
LongConstantSimplifier	49 (0,137)	7 (0,020)
LongParameterList	38 (0,106)	10 (0,028)
MethodPropertyIdentifierConvention	25 (0,070)	1 (0,003)
NotOperatorInversion	1 (0,003)	1 (0,003)
PotentialStaticMethod	25 (0,070)	10 (0,028)
TypeIdentifierConvention	9 (0,025)	2 (0,006)
UseOfVar	489 (1,370)	201 (0,563)
WhitespaceFix	43 (0,120)	264 (0,739)
TypoRefactoring	1223 (3,426)	461 (1,291)
WordTypeRefactoring	330 (0,924)	302 (0,845)

2.1.3 Nötige Anpassungen

Refactoring	Anpassungen	behoben
BooleanConstantComparison	NEIN Sämtliche Vergleiche mit Boolean würden sich vereinfachen lassen.	JA
ConditionalComplexity	NEIN Die Methoden, welche von SASKIA gemeldet wurden waren öfters 40-70 Zeilen lang und hatten enorm viele Verzweigungen (sehr unleserlicher Code).	JA
DeMorganSimplifier	JA Enthält noch ein Bug, sodass SASKIA wegen eines Null-Pointer abstürzt.	JA
IllegalFieldAccess	JA Log4Net enthält zwar einige protected-Fields, welche SASKIA weiterhin melden soll. Häufig meldet SASKIA jedoch Konstanten, welche mit «public const» erstellt wurden, was nicht der Fall sein soll.	JA
IntegerConstantSimplifier	JA SASKIA möchte Integerkonstanten mit einem unären Minus davor (wie -1 zu -1) vereinfachen, was natürlich nicht funktioniert und korrigiert werden.	JA
LackOfCohesion	JA SASKIA meldet zuviele LackOfCohesion-Metrik-Verstösse. Dies vielfach auch bei kleinen Klassen und Klassen mit statischen Methoden. Deshalb wäre es sinnvoll statische Klassenmembers nicht in die Berechnung einfliessen zu lassen und Klassen mit weniger als 5 Objektmethoden (ohne Konstruktur) gar nicht zu melden oder den Treshold zu erhöhen. Ebenfalls ist über das Logfile erkennbar, dass dieses Refactoring gelegentlich wegen eines Dictionary-Zugriffs Exceptions wirft.	JA
LinesOfCode	JA SASKIA meldet zuviele LinesOfCode-Smells, da die Document-Kommentare fälschlicherweise mitgezählt werden. Dies muss angepasst werden.	JA
LongConstantSimplifier	NEIN	JA
LongParameterList	JA Methoden mit 4 Parametern gibt es in der Praxis noch recht viele. Deshalb wird neu Long Parameter List erst ab 5 Parametern gemeldet.	JA
MethodPropertyIdentifierConvention	JA Bei Event-Callback-Methoden sind teilweise bei C# Underlines üblich. Leider ist es praktisch nicht möglich solche Methoden zu	JA

	erkennen. Jedoch sollten Methoden, die als extern deklariert sind nicht auf die Namensrichtlinie geprüft werden. Bei Properties reagiert SASKIA korrekt.	
NotOperatorInversion	NEIN SASKIA reagiert bei diesem Refactoring richtig.	JA
PotentialStaticMethod	NEIN Sämtlich durchgeführte Stichproben waren Methoden, die besser statisch deklariert werden könnten.	JA
TypeIdentifierConvention	JA SASKIA markiert den Klassennamen «Level2EventLogEntryType» als falsch. Zahlen sollten somit erlaubt sein (z.B. auch bei UTF8 sinnvoll)	JA
UseOfVar	NEIN Es fällt auf, dass die meisten erfahrenen Programmierer wie Dozenten var verwenden. Möchte jemand diese Meldung nicht, kann man sie selbstständig deaktivieren.	JA
WhitespaceFix	NEIN Es ist wichtig, dass der Code, die richtigen Einrückungen hat.	JA
WordType	NEIN Wenn der entsprechende Worttyp nicht gefunden wird, dann ist es gut, wenn SASKIA dies meldet.	JA
TypoRefactoring	JA Wörter mit 1 Buchstaben sollten nicht geprüft werden.	JA
WordTypeRefactoring	NEIN Wenn der entsprechende Worttyp nicht gefunden wird, dann ist es gut, wenn SASKIA dies meldet.	JA

2.2 Prism

Im Folgenden werden die Resultate der Auswertung des Projekts Prism beschrieben. Prism ist ein Framework, welches von Microsoft als Unterstützung für MVVM mit WPF angeboten wird. Prism ist ebenfalls OpenSource und kann unter folgendem Link heruntergeladen werden: https://github.com/PrismLibrary/Prism.

2.2.1 Messdaten

Die folgenden Messdaten wurden durch die Evaluation von SASKIA erhoben.

Dauer der Evaluation (1. Durchgang)	7:54 Minuten
Dauer der Evaluation (2. Durchgang)	5:03 Minuten
Anzahl der C#-Files	825

2.2.2 Häufigkeit der Refactorings

Die folgende Tabelle enthält die Anzahl von SASKIA gemeldeten Warnungen total und berechnet relativ zur Anzahl C#-Files.

Refactoring	Vorkommnisse 1.Durchgang	Vorkommnisse 2.Durchgang
BooleanConstantComparison	1 (0,001)	1 (0,001)
ConditionalComplexity	0 (0,000)	0 (0,000)
DeMorganSimplifier	0 (0,000)	0 (0,000)
IllegalFieldAccess	67 (0,081)	32 (0,039)
IntegerConstantSimplifier	0 (0,000)	0 (0,000)
LackOfCohesion	48 (0,058)	24 (0,029)
LinesOfCode	40 (0,048)	86 (0,104)
LongConstantSimplifier	0 (0,000)	0 (0,000)
LongParameterList	1 (0,001)	1 (0,001)
MethodPropertyIdentifierConvention	18 (0,022)	3 (0,004)
NotOperatorInversion	0 (0,000)	0 (0,000)
PotentialStaticMethod	2 (0,002)	1 (0,001)
TypeIdentifierConvention	0 (0,000)	0 (0,000)
UseOfVar	109 (0,132)	54 (0,650)
WhitespaceFix	269 (0,326)	144 (0,175)
TypoRefactoring	261 (0,316)	105 (0,127)
WordTypeRefactoring	0 (0,000)	84 (0,102)

2.2.3 Nötige Anpassungen

Refactoring	Anpassungen	behoben
BooleanConstantComparison	NEIN SASKIA hat eine Korrektur gefunden, die Sinn machen würde	JA
IllegalFieldAccess	NEIN Prism hat sehr viele public, internal und protected-Felder. Häufig sind diese zwar in Nested-Klassen aber es ist trotzdem nicht gut.	JA
LackOfCohesion	JA Prism scheint die Prinzipien des Software Engineerings besser zu erfüllen als Log4Net. Es gibt aber immer noch zuviele Warnungen betreffend Lack Of Cohesion. Der Threshold muss nochmals erhöht werden.	JA

LinesOfCode	NEIN Kurze Klassen sind wichtig. Dies gilt auch für Microsoft. SASKIA wird deshalb nicht angepasst.	JA
LongParameterList	NEIN Es wurde nur 1 Methode gefunden, welche eindeutig zuviele Parameter hat.	JA
MethodPropertyIdentifierConvention	JA Wie bei Log4Net wird auch hier bei jeder Event-Callback-Methode wegen des Underlines eine Warnung gemeldet. Deshalb wird SASKIA so angepasst, dass genau 1 Underline im Methodennamen erlaubt sind, sofern die Methode die Parametertypen object und EventArgs besitzt.	JA
PotentialStaticMethod	NEIN Keine Anpassung notwendig	JA
UseOfVar	NEIN Das Keyword var wäre in allen untersuchten Fällen die bessere Wahl gewesen.	JA
Whitespace	NEIN Code sollte einheitlich und richtig eingerückt sein.	JA

2.3 Wox

Im Folgenden werden die Resultate der Auswertung des Projekts Wox beschrieben. Wox ist ein OpenSource-Projekt. Es ist ein Hilfstool zur Filesuche in Windows und kann unter folgendem Link bezogen werden: https://github.com/Wox-launcher/Wox.

2.3.1 Messdaten

Die folgenden Messdaten wurden durch die Evaluation von SASKIA erhoben.

Dauer der Evaluation (1. Durchgang)	3:29 Minuten
Dauer der Evaluation (2. Durchgang)	2:20 Minuten
Anzahl der C#-Files	220

2.3.2 Häufigkeit der Refactorings

Refactoring	Vorkommnisse 1. Durchgang	Vorkommnisse 2. Durchgang
BooleanConstantComparison	1 (0,005)	0 (0,000)
ConditionalComplexity	5 (0,023)	4 (0,018)
DeMorganSimplifier	0 (0,00)	0 (0,000)
IllegalFieldAccess	35 (0,159)	34 (0,155)
IntegerConstantSimplifier	0 (0,00)	0 (0,000)
LackOfCohesion	10 (0,045)	10 (0,045)
LinesOfCode	12 (0,055)	30 (0,136)
LongConstantSimplifier	0 (0,00)	0 (0,000)
LongParameterList	2 (0,009)	0 (0,000)
MethodPropertyIdentifierConvention	14 (0,064)	1 (0,005)
NotOperatorInversion	1 (0,005)	0 (0,000)
PotentialStaticMethod	11 (0,005)	6 (0,027)
TypeIdentifierConvention	0 (0,00)	0 (0,000)
UseOfVar	71 (0,323)	27 (0,123)
WhitespaceFix	111 (0,505)	54 (0,245)
TypoRefactoring	0 (0,00)	202 (0,918)
WordTypeRefactoring	0 (0,00)	46 (0,209)

2.3.3 Nötige Anpassungen

Refactoring	Anpassungen	behoben
BooleanConstantComparison	NEIN SASKIA hat eine Korrektur gefunden, die Sinn machen würde	JA
ConditionalComplexity	NEIN Die Fälle, die SASKIA aufgedeckt hat, sind alles schwerwiegende Fälle von Conditional Complexity.	JA
IllegalFieldAccess	NEIN Die aufgedeckten Fälle waren alles public-Felder, die verhindert werden hätten müssen. Dies gilt auch für «public static readonly»-Felder.	JA
LackOfCohesion	JA Wenn eine Klasse gar keine Felder hat, dann soll LackOfCohesion nicht anspringen.	
LinesOfCode	NEIN Die gefundenen Klassen sind alle zu lang.	JA
LongParameterList	NEIN Zwei Methoden haben zuviele Parameter. Dies ist legitim.	JA
MethodPropertyIdentifierConvention	NEIN Es hat Methoden in diesem Projekt, welche klein geschrieben sind etc. Diese sollten von SASKIA bewusst angezeigt werden.	JA
NotOperatorInversion	NEIN Das gefundene Refactoring ist gerechtfertigt.	JA
PotentialStaticMethod	NEIN Alle angezeigten Methoden können ohne weiteres zu static gemacht werden.	JA
UseOfVar	JA Auf Anmerkung von Herr Bläser muss dieses Refactoring wegen Fehlern bei einigen Polymorphismus-Fällen noch angepasst werden. Das Refactoring wird angepasst, sodass nur noch var vorgeschlagen wird, wenn bei einer Zuweisung der statische und dynamische Typ übereinstimmen.	JA
Whitespace	NEIN Es ist wichtig für die Lesbarkeit den Code richtig einzurücken.	JA

3 Schlussfolgerungen

3.1 Zusammenfassung

Im Verlaufe der Arbeit konnten viele Ziele erreicht werden. SASKIA ist bereits im Stande viele einfachere Refactorings durchzuführen. Dazu gehören beispielsweise kleinere Codesmellstrukturen, wie Vergleiche mit boolschen Literalen oder identische if- und else-Blöcke, sowie die Erkennung von Tippfehlern und Wortartverwechslungen. Zudem erkennt SASKIA einige der in der Praxis am gängigsten Metriken wie Long Parameter List oder Conditional Complexity.

3.2 Bewertung der Resultate

Mit den getesteten Projekten von GitHub, darunter bekannte Software wie Log4Net oder Prism, konnte SASKIA auf Anwendbarkeit getestet werden. Mit diesen Tests konnten wir anpassen, wie streng SASKIA auf einzelne Code Smells anspringen bzw. achten sollte. Die daraus entstandenen Resultate sind plausibel und bieten dem Programmierer eine gute Übersicht über die Qualität des Codes.

3.3 Nicht erreichte Ziele

Einige nicht erreichte Ziele wurden bereits in Kurzform erklärt und werden hier im Detail behandelt, andere werden hier ergänzt.

3.3.1 Variablen und Property korrigieren

Im Vergleich zur Analyse von Klassennamen, welche mit einem Nomen enden müssen, ist die Analyse von Variablen-/Propertynamen nicht ganz so trivial. Der Grund dafür ist, dass man eigentlich alle Wortarten (Nomen, Verb, etc.) als Variable-/Propertynamen angegeben werden kann.

3.3.2 Korrelation zwischen Klassenname und Methodenname

Um zu überprüfen, ob eine Methode zulässig ist in Abhängigkeit zum Klassennamen, reicht eine simple Wörterbuch-Datenbank nicht aus. Dazu muss eine Graph-Datenbank benutzt werden, welche Beziehungen zwischen Wörtern beherbergt. Solche Datenbanken sind zurzeit nicht offline verfügbar. Dafür gibt es einige Online-Datenbanken, welche von Firmen (z.B. Microsoft) zur Verfügung gestellt werden. Da SASKIA in kurzen Abständen die entsprechenden Analysen durchführt, ist davon abzuraten, jedes Mal eine Verbindung zum Internet herzustellen um die Abfragen zu machen. Zudem sind diese Dienste (im Falle von Microsoft Azure Services) kostenpflichtig.

3.3.3 Wörterbuch Analyse mit einer einzigen Datenbank

Auch hierzu gibt es keine offline Datenbank/Library, die zugleich Wortarten als auch Rechtschreibung überprüfen kann. Online bieten sich wieder Microsoft Azure Services an, welche kostenpflichtig sind und ungeeignet bei den zahlreichen Abfragen von SASKIA.

3.3.4 Verlinkte Variablenbezeichner ersetzen

Variablenbezeichner, die auf die Analysen "TypoRefactoring" oder "WordTypeRefactoring" angesprungen sind, konnten problemlos korrigiert werden. Wenn aber jener Bezeichner innerhalb einer Property verwendet wurde (ein Link auf eine Variable), wurden diese Links nicht mit korrigiert. Dasselbe in umgekehrt: wurde der Link korrigiert, wurde der ursprüngliche Bezeichner nicht korrigiert. Zugunsten anderen Analysen wurde diese Analyse vernachlässigt.

3.4 Weitere Schritte

Besonderes Potential bieten die lexikalischen Analysen. Zum Beispiel können wir als weiteres das im Kapitel 3.3.4 erwähnte Ziel implementieren. Zum anderen könnte man dem Programmierer die Möglichkeit geben aus einer Liste, mit mehreren Korrekturvorschlägen, auszuwählen.

4 Anhang

A - Persönliche Berichte

Joël Egger

Mit der Studienarbeit SASKIA bin ich im Grossen und Ganzen recht zufrieden. Ich konnte aus der Arbeit viel in den Bereichen Refactoring und Compilerbau lernen. Zudem war es eine gute Erfahrung für die Abwicklung eines eher wissenschaftlichen Projekts. Die Zusammenarbeit mit Marcel Stocker und dem Dozenten Dr. Luc Bläser verlief immer sehr zufriedenstellend. Des Weiteren bin ich froh, dass wir ein sehr interessantes Projekt abwickeln konnten, welches uns auch Spass gemacht hat.

Ein bisschen schwieriger war es teilweise, die technischen Probleme in den Griff zu bekommen. Häufig gab es Fehler, welche ein Bug von Roslyn darstellen, welche jedoch fast nicht dokumentiert waren. Deshalb mussten wir oft mit Mitarbeitern der HSR, welche im Roslyn-Bereich Erfahrung mit sich brachten, Rücksprache halten oder in Foren sehr lange suchen, um ein Problem zu beheben.

Die wichtigsten Erkenntnisse, die ich aus dem Projekt ziehen kann sind, dass es sich lohnt viele Unit-Tests zu schreiben. Leider haben wir eher relativ spät begonnen zu dokumentieren. Manchmal wäre es besser gewesen, das System direkt zu dokumentieren, damit man auf diesem Thema noch à jour ist.

Marcel Stocker

In Grund genommen hatte ich mit diesem Projekt viel Freude am Programmieren. Es war eine spannende Aufgabe und man konnte sich vorstellen, wie nützlich das Ergebnis dieses Projekts für andere Programmierer sein kann. Ich hatte auch das Glück mit Joël Egger zusammen zu arbeiten. Nach dem Anlauf des Projekts konnten wir gut die Aufgaben untereinander aufteilen und waren immer auf derselben Wellenlänge. Auch die Zusammenarbeit mit Prof. Bläser war sehr angenehm und ich fühlte mich immer gut unterstützt und konnte gut vom Wissen von Prof. Bläser lernen.

Die Arbeit mit Roslyn war meistens gut, manchmal weniger gut. Die paar Mal als es nicht so gut lief, bin ich nicht sicher, ob es an Roslyn, oder Visual Studio lag. Insgesamt zählen dazu Probleme wie: Roslyn setzt nicht mehr an und analysiert gar nichts mehr (bei allen Workstations), VM Ware Fusion Lizenz auf meinem Macbook ist abgelaufen und ein guter Ersatz musste erarbeitet werden, synchronisationsprobleme mit GitHub. Zweiteres war ein rechter Murks und konnte selbst mit Bootcamp nicht gelöst werden. Ich musste somit auf andere Workstations ausweichen. Diese technischen Probleme waren der grösste Zeitfresser und waren sehr unangenehm und nervenzerreissend. Denn im Grund genommen will ich einfach nur Programmieren.

Im Grossen und Ganzen bin ich jedoch sehr zufrieden. Ich denke jedoch, dass wenn diese technischen Probleme nicht gewesen wären, könnten wir ein umfangreicheres Ergebnis abliefern.

B - Installationsanleitung

Installation VSIX Projekt

Im Folgenden wird beschrieben, wie das VSIX-Projekt korrekt aufgesetzt wird, damit Roslyn korrekt funktioniert.

1. Für SASKIA wird die Projektvorlage **VSIX Project** im Visual Studio verwendet. Für diese Projektvorlage muss Visual Studio SDK installiert werden: https://msdn.microsoft.com/en-us/library/bb166441.aspx
2. Da SASKIA auf Roslyn (Microsoft Compiler Platform) basiert, werden die folgenden NuGet-Pakete gebraucht.
   a. Microsoft.CodeAnalysis.Analyzer (v.2.6.0)
   b. Microsoft.CodeAnalysis.Common (v.2.6.1)
   c. Microsoft.CodeAnalysis.Csharp (v.2.6.1)
   d. Microsoft.CodeAnalysis.Csharp.Features (v.2.6.1)
   e. Microsoft.CodeAnalysis.Csharp.Workspaces (v.2.6.1)
   f. Microsoft.CodeAnalysis.Features (v.2.6.1)
   g. Microsoft.CodeAnalysis.Workspaces.Common (v.2.6.1)
3. Damit VisualStudio weiss, dass es bei diesem VSIX Project nach DiagnosticAnalyzer und CodeFixes-Klassen suchen muss, müssen im **source.extension.vsixmanifest** unter **Assets** noch die folgenden Einträge vorgenommen werden.
   a. Klick auf **New Asset** und **Microsoft.VisualStudio.Analyzer, A project in current solution** und **SASKIA** auswählen.
   b. Klick auf **New Asset** und **Microsoft.VisualStudio.MefComponent, A project in current solution** und **SASKIA** auswählen.
4. **Hinweis:** Beim Ausführen des Projekts wird eine experimentelle Instanz des Visual Studios gestartet. Damit das VisualStudio die neu hinzugefügten Klassen lädt, muss jeweils der folgende Ordner gelöscht werden: %localappdata%\Microsoft\VisualStudio\15.0_[*HASH*]Exp

Die Ausführung des Programms Reset the Visual Studio Experimental [Version] genügt dafür nicht.

Installation NHunspell

1. Download NHunspell als NuGet Paket in Visual Studio.
2. Download des Wörterbuchs (en_us.aff und en_us.dic) von angehägter Website[7]
3. Kopiere die Wörterbuch Dateien in die Projekt-Mappe (nicht die Solution, sondern in jenem Projekt wo sie gebraucht wird)
4. Markiere die Dateien als Resource und “copy always”

Installation English-Dictionary-SQLite

1. Download der English-Dictionary-SQLite Dateien von folgender Source[8]
2. Kopiere die Wörterbuch Dateien in die Projekt-Mappe
3. Markiere die Dateien als Resource und “copy always”
4. Setze im Code, wo das Wörterbuch verwendet wir den Ort der Resource-Dateien, mit der Methode “`NativePath(string path)`”.

[7] https://github.com/ropensci/hunspell/tree/master/inst/dict, letzter Zugriff: 06.05.18
[8] https://github.com/AyeshJayasekara/English-Dictionary-SQLite, Letzter Zugriff: 30.04.2018

C – Eigenständigkeitserklärung

Eigenständigkeitserklärung

Ich erkläre hiermit,

- dass ich die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt habe, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde,
- dass ich sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben habe.
- das ich keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise genutzt habe.

Ort, Datum: St. Gallen, 07.05.18

Name, Unterschrift: Joël Egger

Eigenständigkeitserklärung

Ich erkläre hiermit,

- dass ich die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt habe, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde,
- dass ich sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben habe.
- das ich keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise genutzt habe.

Ort, Datum: St. Gallen, 07.05.18

Name, Unterschrift: Marcel Stocker