



Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Master-Thesis

Visual Studio-Erweiterung zur statischen Code-Analyse

andrena objects ag

Manuel Naujoks

Betreut durch

Prof. Dr. Thomas Fuchß

Karlsruhe, den 30. August 2012

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und das darin beschriebene Programm Usus.NET selbstständig programmiert und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Karlsruhe, den 30. August 2012
Manuel Naujoks

Zusammenfassung

Das Ziel dieser Arbeit war die Entwicklung und Dokumentation einer Visual Studio-Erweiterung, die dem Entwickler direktes Feedback anhand von **Softwaremetriken** und daraus abgeleiteten Qualitätskenngrößen gibt. Die Arbeit wurde im Rahmen einer Master-Thesis bei dem Softwareentwicklungshaus andrena objects ag durchgeführt. Dabei diente das Plugin Usus für Eclipse als Vorlage. Die erstellte Erweiterung kann eine **statische Code-Analyse** von .NET-Projekten in Visual Studio 2010 durchführen. Dabei werden bestimmte **Codemetriken** berechnet, die bei andrena zum Einsatz kommen. Für die Berechnung werden die **Assemblies**, die der Compiler bei jedem Kompilervorgang erzeugt, mit der **Common Compiler Infrastructure** analysiert. Ein wesentliches Feature der Erweiterung ist die nahtlose Integration in Visual Studio, die es ermöglicht, die **Metriken** und Qualitätskenngrößen direkt in der Entwicklungsumgebung anzuzeigen.

Die entwickelte Erweiterung ist in der Lage, Softwareentwickler bei der Entwicklung von **Clean Code** zu unterstützen. **Clean Code** wurde von Robert C. Martin in seinem Buch „Clean Code“ vorgestellt und verfolgt das Ziel Quellcode besser verstehen und warten zu können. Das automatisierte Feedback erlaubt dem Entwickler, sich über Problemfälle in seinem Code zu informieren und diese schnell zu lokalisieren. Um eine Einschätzung bezüglich der Merkmale von **Clean Code** vorzunehmen, nutzt die Erweiterung die **statischen Verteilungen** der **Metriken**. Die Analyse dieser **Verteilungen** hat zu der Definition einer neuen **Clean Code-Metrik** geführt. Zusätzlich kann das Tool andrena's **Softwarequalitätsindex** berechnen, der eine weitere Interpretation der inneren Qualität der Software ermöglicht. Die Erweiterung protokolliert die Veränderung dieses **Softwarequalitätsindex** über die Zeit und stellt sie in einem Schaubild dar.

Um die entwickelte Erweiterung und dessen Möglichkeiten der Entwicklerunterstützung zu evaluieren, wurde eine Refactoring-Beispielaufgabe aus dem andrena-Kurs „Agile Software Engineering“ analysiert. Diese Untersuchung bestätigt, dass die Erweiterung die Veränderungen des Quellcodes erkennt und dass die Refactorings zu Veränderungen der **Metriken**, Qualitätskenngrößen und besonders der neuen **Clean Code-Metrik** führen. Um die Leistungsfähigkeit des Werkzeugs zu messen, wurden die **statischen Code-Analysen** einiger Industrie- und Open-Source-Projekte durchgeführt, deren Laufzeit erfasst und die Ergebnisse dokumentiert.

Die entwickelte Visual Studio-Erweiterung trägt den Namen Usus.NET und ist als Open-Source-Projekt unter <https://github.com/usus/Usus.NET> veröffentlicht. Die innere Softwarequalität des Usus.NET-Projekts wird von Usus.NET mit einem andrena-**Softwarequalitätsindex** von 82,2% als sehr positiv bewertet. Die neue **Clean Code-Metrik** der Methoden bestätigt dies. Damit stellt Usus.NET eine Grundlage dar, auf dessen Basis weitere Entwicklungen zur Verbesserung der Codequalität vorgenommen werden können.

Abstract

The aim of this thesis was to develop and document a Visual Studio extension that is capable of providing direct development feedback based on [software metrics](#) and quality indicators. The extension was developed during a postgraduate internship at the software company andrena objects ag. The Eclipse plugin Usus was used as a guide in terms of project orientation. The developed extension can perform a [static code analysis](#) of .NET projects inside Visual Studio 2010. Several [code metrics](#) that are used in daily business at andrena, are calculated from the compiler generated [assemblies](#) using the [Common Compiler Infrastructure](#). One major feature is the seamless integration into Visual Studio, that enables the display of the [metrics](#) and indicators directly within the integrated development environment.

The developed extension helps developers to write [clean code](#). [Clean code](#) was introduced by Robert C. Martins book “[Clean Code](#)“ and aims at improving readability and maintainability of source code. Automated feedback allows the developer to localize problematic areas within the code. For assessment of the code concerning the characteristics of [clean code](#), the extension makes use of the [statistical distributions](#) of the [metric results](#). An analysis of these [distributions](#) has led to the definition of a new [clean code metric](#). Additionally, the tool can calculate andrena’s [software quality index](#), which enables further interpretation of inner software quality. The extension records the changes of this [software quality index](#) and visualizes them in a diagram.

The developed extension and its opportunities to support the developer are evaluated. A refactoring exercise of the andrena course “[Agile Software Engineering](#)“ is analyzed to verify that changes in the code are picked up by the extension and lead to changes in the different [metrics](#), indicators and especially in the new [clean code metric](#). In order to measure the performance of the tool, the [static code analyses](#) of a number of real-world projects are timed and documented.

The developed Visual Studio extension is referred to as Usus.NET and is published as an open source project at <https://github.com/usus/Usus.NET>. The inner quality of the Usus.NET project is rated with an andrena [software quality index](#) of 82,2%, which is an excellent rating. The new [clean code metric](#) of the methods confirms this. Usus.NET is a foundation and allows for further work on improvement of code quality.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Aufgabenstellung	2
1.2	andrena objects ag	2
1.3	Aufbau dieses Dokuments	3
2	Grundlagen	5
2.1	Allgemeine Grundlagen	5
2.1.1	Clean Code	5
2.1.2	Eclipse	6
2.1.3	Visual Studio	6
2.2	Technische Grundlagen	7
2.2.1	Objektorientierung	7
2.2.2	Graphentheorie	8
2.2.3	Statische Code-Analyse	9
3	Anforderungen	11
3.1	Interessenvertreter	11
3.2	Ziele	12
3.3	Architektur	13
3.4	Use Cases	14
3.5	Produkt-Backlog	15
4	Vorgehensweise	17
4.1	Praktischer Teil	17
4.2	Theoretischer Teil	17
4.3	Tatsächlicher Ablauf	18
5	Usus	21
5.1	Eclipse-Plugin	21
5.1.1	Usus Cockpit	21
5.1.2	Usus Info	22
5.1.3	Usus Hotspots	22
5.1.4	Usus Histogram	23
5.1.5	Usus Class Graph & Usus Package Graph	23
5.2	Metriken	23
5.2.1	Pro Methode	24
5.2.2	Pro Klasse	26
5.2.3	Projektübergreifend	28
5.3	Zusammenfassung	31

6	Andere Tools	33
6.1	Visual Studio Metrics	33
6.1.1	Visual Studio Code Metrics Power Tool	34
6.1.2	Code Metrics Viewer	34
6.2	NDepend	35
6.2.1	Code Query Language	35
6.2.2	Abstraktheit und Instabilität	35
6.3	CCD-Addin	37
6.3.1	Prinzipien	38
6.3.2	Praktiken	38
6.4	Zusammenfassung	38
7	Evaluierung der Technologien	41
7.1	FxCop	42
7.1.1	Umgebung	42
7.1.2	API	43
7.2	Common Compiler Infrastructure	44
7.2.1	CCI Metadata	45
7.2.2	CCI Code and AST Components	46
7.3	NRefactory	47
7.3.1	AST aus Quellcode	47
7.3.2	Mono.Cecil	48
7.4	Codename „Roslyn“	48
7.4.1	Schichten	49
7.4.2	Syntax Tree	49
7.4.3	Semantik	50
7.4.4	Workspaces	51
7.5	Zusammenfassung	52
8	Usus.NET	55
8.1	Metrikberechnung	55
8.1.1	Zyklomatische Komplexität	57
8.1.2	Methodenlänge	58
8.1.3	Kumulierte Komponentenabhängigkeit	59
8.1.4	Klassengröße	61
8.1.5	Nicht-statische öffentliche Felder	61
8.1.6	Namespaces mit zyklischen Abhängigkeiten	61
8.2	Objektmodell	65
8.2.1	Methodenberichte	65
8.2.2	Klassenberichte	66
8.2.3	Namespace-Berichte	67
8.3	Metrikgewichtung	68
8.3.1	Hotspots	68
8.3.2	Statistiken	69
8.3.3	Ignorierbares	70
8.4	Metrikverteilung	71
8.5	Testen	72

9	Usus.NET als Visual Studio-Erweiterung	75
9.1	Oberflächen	75
9.1.1	Hubs	75
9.1.2	Cockpit	77
9.1.3	Info	77
9.1.4	Hotspots	78
9.1.5	Histogramm	79
9.1.6	Class Graph & Package Graph	80
9.1.7	Clean Code-Grade	80
9.1.8	SQI	81
9.2	Visual Studio Integration	81
9.2.1	Events	82
9.2.2	Fenster	82
9.2.3	Menüpunkt	84
9.2.4	Zeilensprünge	85
9.2.5	Editor Adornment	85
10	Clean Code-Unterstützung	89
10.1	Kleine Metriken	89
10.2	Approximation	90
10.3	Abweichung	91
10.4	Refactoring-Vorschläge	92
11	andrena-Softwarequalitätsindex	93
11.1	Parameter	93
11.2	Berechnung in Usus.NET	95
11.2.1	Relevante Code-Zeilen	96
11.2.2	Compiler-Warnungen	97
11.2.3	Testabdeckung	98
11.3	Veränderung	98
12	Usus.NET-Evaluation	101
12.1	Laufzeitmessungen	101
12.2	Fallbeispiel andrena-Kurs	102
12.3	Ergebnis	103
13	Zusammenfassung	107
14	Fazit und Ausblick	109
Anhang		xxi
A.1	Manuskript über den Prozess- und Softwarequalitätsindex	xxi
A.2	Artikel über Isis	xxi
A.3	Testpläne	xxi
A.4	Messergebnisse der Laufzeitmessungen	xxi
A.5	Visual Studio-Erweiterung Usus.NET	xxi

Abbildungsverzeichnis

3.1	Grobe Architektur des zu entwickelnden Systems	13
3.2	Anwendungsfälle des zu entwickelnden Systems	14
3.3	Produkt-Backlog des zu entwickelnden Systems	15
4.1	Ursprünglicher Projektplan	19
4.2	Tatsächlicher Projektablauf	20
5.1	Usus Cockpit zeigt Übersicht über alle Projekte	21
5.2	Usus Info zeigt Übersicht über eine Methode oder Klasse	22
5.3	Usus Hotspots zeigt die Stellen im Code, die eine besonders negative Ausprägung der Metriken haben	22
5.4	Usus Histogramm zeigt die statistische Verteilung der Werte der ausgewählten Metrik	23
5.5	Usus Graph Schaubilder zeigen die Abhängigkeiten der Klassen oder Pakete voneinander an	24
5.6	Ablaufgraph G des Quellcodes 5.1	25
5.7	Abhängigkeitsgraph einer Klasse mit aufsummierten Abhängigkeiten	27
6.1	Metrik-Fenster in Visual Studio 2010 Premium	33
6.2	Architekturdiagramm in Visual Studio 2010 Ultimate auf Basis von Paketen, aufklappbar bis auf Methodenebene	34
6.3	Schematische Darstellung des „Abstraktheit und Instabilität“-Diagramm in NDepend	37
7.1	FxCop Runner	42
7.2	FxCop Runner in der Kommandozeile	43
7.3	InternalsVisibleTo-Attribute der Assembly Micorsoft.Cci.dll	45
7.4	Tabellarisches Ergebnis der Evaluierung der verschiedenen Technologien	52
8.1	Architektur von Usus.NET	55
8.2	Klassen, die an der Metrikberechnung von Usus.NET beteiligt sind	56
8.3	Schematische Funktionsweise des Algorithmus Vertex Reduction	63
8.4	Objektmodell und Klassen des vollständigen Metrikberichts	65
9.1	Oberflächen und ViewModels in Verbindung mit dem ViewHub-Objekt	76
9.2	Usus.NET Cockpit-Fenster	77
9.3	Usus.NET Info-Fenster	78
9.4	Usus.NET Hotspots-Fenster	78
9.5	Usus.NET Namespace-Zyklus-Dialog	79
9.6	Usus.NET Histogramm-Fenster	80
9.7	Usus.NET Clean Code-Grade-Fenster	80

Abbildungsverzeichnis

9.8	Usus.NET SQI-Fenster	81
9.9	Oberklassen der Usus.NET-Fenster	83
9.10	Usus.NET-Menüpunkt in der Menüleiste von Visual Studio	84
10.1	Exponentialverteilung für großes und kleines λ	90
10.2	Geometrische Verteilung für großes und kleines λ	91
11.1	Zeitliche Entwicklung des SQI im Usus.NET SQI-Fenster	98
12.1	Laufzeiten der statischen Code-Analyse verschiedener Industrie- und Open-Source-Projekte	105
12.2	Vorher-Metriken des andrena Refactoring-Beispiels	106
12.3	Nachher-Metriken des andrena Refactoring-Beispiels	106

Tabellenverzeichnis

8.1	Abbildung von Knoten zu starker Zusammenhangskomponente zu anderen Knoten	64
11.1	Relative Größen und Zweidrittelkonstanten aller Isis-Metriken	94
11.2	Zweidrittelkonstanten f_{mk} der Isis-Monolithkorrekturfaktoren	95
11.3	SQI-Gewichte	95

1 Einleitung

Als vor ungefähr 50 Jahren der erste Computer gebaut wurde [Wei61], konnte sich wahrscheinlich niemand vorstellen, wie schnell sich diese Rechenmaschinen verbreiten und weiterentwickeln würden. Die technischen Fortschritte, unter anderem in der Prozesortechnologie, haben dafür gesorgt, dass Ende 2008 weltweit mehr als eine Milliarde PCs tatsächlich verwendet wurden [wor12]. Dank Moore's Law¹ besitzen sogar heutige Smartphones eine erstaunliche Rechenleistung, die wahrscheinlich auch weiterhin steigen wird. Doch mit der rasanten Entwicklung immer fähigerer Hardware, entstehen auch immer neue Möglichkeiten für Software. Aus diesem Grund mussten sich auch die Programmiersprachen weiterentwickeln. Der O'REILLY-Verlag hat dazu ein Poster erstellt, das zeigt, wie Programmiersprachen andere Programmiersprachen beeinflusst haben [O'R].

Immer umfangreichere Programmiersprachen und Frameworks stellen die Entwickler vor neue Herausforderungen. Diese neuen Möglichkeiten und die damit in Verbindung stehenden Erwartungen der Nutzer, gestalten die Softwareentwicklung zunehmend schwieriger. Die Anforderungen an die moderne Softwareentwicklung sind sogar so hoch, dass viele Projekte scheitern. Die Standish Group veröffentlicht mit dem CHAOS Report² einen jährlichen Bericht, der die Erfolge und die Fehlschläge von IT-Projekten statistisch aufbereitet. Stefan Hagen interpretiert den CHAOS Report in seinem Blog-Post [Hag10] und vergleicht diesen mit anderen Statistiken, um die Ursachen der Fehlschläge zu finden. Um die Probleme zu lösen, wurden verschiedene Modelle von Entwicklungsprozessen formalisiert und angewendet, die beispielsweise auch von Walt Scacchi in seiner Veröffentlichung beschrieben werden [Sca01].

Eine Lösung ist das agile Projektmanagement. Mike Cohn zitiert in seinem Blog-Post [Coh12] den CHAOS Report von 2011 und beschreibt agile Projekte im Durchschnitt als dreimal so erfolgreich wie nicht-agile Projekte. Für die Entwickler bedeutet das konkret, dass sie schneller auf Änderungen der Anforderungen reagieren müssen. Dabei ist die Gefahr groß, dass die Entwickler Schaden an der äußeren und der inneren Qualität der Software anrichten. Mit der äußeren Qualität ist der Funktionsumfang gemeint. Die innere Qualität bezieht sich auf die Struktur und die Form des Quelltext. Während die Verschlechterung der äußeren Qualität in Form von Bugs relativ schnell offensichtlich wird, ist die Verschlechterung der inneren Qualität subtiler. Das kann dazu führen, dass zukünftige Änderungen komplizierter und langsamer vorgenommen werden können. Um dem entgegen zu wirken, haben berühmte Entwickler Bücher geschrieben, die Hilfestellungen enthalten. Zwei der bekanntesten Werke sind Robert C. Martin's Bücher „Clean Code“ [Mar09a] und „Clean Coder“ [Mar11]. Diese Hilfestellungen adressieren jeden Entwickler und versuchen ihn zu motivieren, verantwortungsbewusst mit Quellcode zu arbeiten. Um diese Hilfestel-

¹Mehr Informationen: „Moore's Law Inspires Intel Innovation“ <http://www.intel.com/content/www/us/en/silicon-innovations/moores-law-technology.html>

²Mehr Informationen: „CHAOS Knowledge Center“ <http://blog.standishgroup.com/pmresearch>

lungen auf den eigentlichen Gegenstand der Softwareentwicklung, nämlich den Quellcode selber, zu übertragen, kann eine **statische Code-Analyse** genutzt werden. Die **statische Code-Analyse** berechnet **Metriken** des System, die einen Teil der inneren Qualität reflektieren. Moderne Entwicklungsumgebungen bieten momentan nur wenig Möglichkeiten den Quelltext zu analysieren und zu interpretieren. Einige separate Werkzeuge bieten etwas mehr Unterstützung und versuchen, dem Entwicklern so viele Informationen zu liefern, dass er eine bessere Einsicht in das Projekt bekommt.

1.1 Aufgabenstellung

Genau an dieser Stelle setzt diese Master-These an. Es soll ein Werkzeug für .NET-Code entwickelt werden, das dem Entwickler mehr relevante Informationen über eine Codebasis liefert, als anderen Werkzeuge dies tun. Diese Informationen sollen den Entwickler unterstützen, den Quellcode verantwortungsbewusster zu verändern, sodass der Code eher dem von Robert C. Martin beschriebenen **Clean Code** entspricht. Das zu entwickelnde Tool soll außerdem direkt in Visual Studio integriert sein, sodass der Entwickler das Feedback automatisiert erhält und kein weiteres Werkzeug starten muss. Bei der Entwicklung der Visual Studio-Erweiterung soll ein Programm aus der Java-Welt als Orientierung dienen. Die Eclipse-Erweiterung **Usus** analysiert den Code und berechnet und gewichtet Kennzahlen, die dem Entwickler helfen können, eine bessere Einsicht in seinen Code zu bekommen. In dieser Master-These soll ein Tool entwickelt werden, das einen vergleichbarem Funktionsumfang hat und zusätzliche Interpretationsmöglichkeiten bietet. Dabei soll die **Verteilung** der **Metriken** untersucht und mit **Clean Code** in Verbindung gebracht werden. Zusätzlich soll der **andrena-Softwarequalitätsindex** berechnet werden, um eine Codebasis noch schneller einschätzen zu können.

1.2 andrena objects ag

Diese Master-These wird bei dem Unternehmen andrena objects ag in Karlsruhe durchgeführt. andrena ist ein Softwareentwicklungs- und beratungshaus, das keine eigenen Produkte vertreibt, sondern sein Know How in der agilen Softwareentwicklung in Form von Projektunterstützung, Lösungen (Auftragsarbeiten) und Training an seine Kunden weitergibt. Dabei spielen die in Unterabschnitt 2.1.1 beschriebenen Konzepte, Extreme Programming und **Clean Code**, eine besondere Rolle. Um die moderne Softwareentwicklung immer weiter zu verbessern, ist andrena daran interessiert, sowohl die Fähigkeiten der Entwickler als auch die Fähigkeiten der eingesetzten Werkzeuge weiterzuentwickeln. Aus diesem Grund soll diese Master-These die technologische Weiterentwicklung der agilen Entwicklung unterstützen. Jedes Projekt, an dem andrena beteiligt war, konnte, laut eigenen Angaben, bisher erfolgreich beendet werden [ao12]. Diesen außergewöhnlichen Erfolg begründet andrena mit dem Konzept des *agilen Software Engineering* und genau dieses Konzept soll, durch das in dieser Master-These zu entwickelnde Programm, weiter unterstützt werden.

1.3 Aufbau dieses Dokuments

Die vorliegende Master-These beschreibt die Entwicklung einer Visual Studio-Erweiterung zur **statischen Code-Analyse**. Mit einer agilen Anforderungsanalyse werden zuerst die Anforderungen an das System und eine Planung der Entwicklungsaufgaben dokumentiert. Dadurch entsteht eine konkretere Vorstellung der Software, die im Folgenden entwickelt werden soll. Anschließend wird die zeitliche Planung und nachträglich die tatsächliche Durchführung beschrieben. Anhand bereits bestehender Werkzeuge und theoretischen Ideen wird analysiert, wie diese Software die definierten Anforderungen realisieren kann. Dadurch werden die grundlegenden Funktionen des Systems spezifiziert. Nach dieser Analysephase wird die Entwicklung des Programms beschrieben, indem zuerst auf die zentralen Komponenten eingegangen wird, die die eigentlichen Berechnungen anstellen. Danach wird die Integration dieser zentralen Komponenten in Visual Studio dokumentiert. Bis zu dieser Stelle wurde dann also ein System beschrieben, welches eine **statische Code-Analyse** direkt in Visual Studio durchführen kann. Auf dieser Grundlage aufbauend, können dann weitere Funktionen konzipiert und implementiert werden. Es werden dann Funktionen spezifiziert, die das **Clean Code-Level** und den **Softwarequalitätsindex** einer Codebasis berechnen. In diesem Zusammenhang wird dann auch die dafür nötige Implementierung beschrieben. Abschließend wird die entwickelte Software evaluiert, indem zum Einen die Laufzeit der **Analyse** verschiedener Projekte gemessen wird. Zum anderen wird ein Refactoring-Beispiel von *andrena* analysiert, bevor die Ergebnisse beider Evaluierungen beschrieben werden. Das Dokument endet mit einer Zusammenfassung und einem Ausblick, indem mögliche Weiterentwicklungen des Systems vorgeschlagen werden.

In diesem Dokument sind Querverweise farblich dargestellt. Adressen zu Websites und Verweise auf Quellen des Literaturverzeichnis sind dabei **blau** hervorgehoben. Verweise auf Kapitel, Abschnitte, Abbildungen, Tabellen, Listings, Formeln, Definitionen und Bezeichnungen im Glossar sind **dunkelblau** gekennzeichnet. In dem Glossar werden häufig auftretende Bezeichnungen, die in dieser Master-These von hoher Bedeutung sind, zusätzlich an einer zentralen Stelle beschrieben. Durch die farbige Hervorhebung wird auf diese Bedeutung explizit hingewiesen. Die digitale Version dieser Master-These erlaubt es, über einen Klick auf diese farbigen Stellen direkt zu der Position der referenzierten Stelle zu springen. Wörter in *kursiver* Form haben eine implizite Bedeutung und beziehen sich auf einen besonderen Kontext. Um auf externe Informationen hinzuweisen, wurden Fussnoten und das Literaturverzeichnis verwendet. Im Literaturverzeichnis werden essentielle und häufig mehrfach referenzierte Quellen aufgeführt, die für den Zusammenhang wichtig sind. Die Fussnoten dagegen beschreiben weniger wichtige Quellen, die nur an der Stelle interessant sind, an der sie verwendet werden. Zusätzlich werden Definitionsblöcke verwendet, um Begriffe mit einer besonderen Bedeutung kurz zu beschreiben. Wenn die besonderen Begriffe eine umfangreichere Beschreibung erfordern, wird diese Beschreibung in einem eigenen Abschnitt vorgenommen.

2 Grundlagen

In diesem Kapitel sollen die Grundlagen vermittelt werden, die für ein Verständnis einer mit Quellcode arbeitenden Erweiterung für Visual Studio erforderlich sind. Dazu wird in Abschnitt 2.1 zunächst auf allgemeine Grundlagen wie Quellcode, sowie Entwicklungsumgebungen eingegangen. Anschließend werden in Abschnitt 2.2 die technischen Grundlagen vorgestellt.

2.1 Allgemeine Grundlagen

In diesem Abschnitt wird zunächst auf die *Clean Code*-Idee von Robert C. Martin sowie einige seiner Prinzipien eingegangen. Anschließend werden die beiden Entwicklungsumgebungen Eclipse und Visual Studio vorgestellt. Beide Programme sind in dieser Master-Thesis von Bedeutung, da die zu entwickelnde Erweiterung in ähnlicher Form bereits für Eclipse existiert und für Visual Studio ebenfalls erstellt werden soll.

2.1.1 Clean Code

Ein Softwaresystem wird in Form von Quellcode erstellt. Dieser Code wird anschließend kompiliert um ein ausführbares Programm zu erhalten. Robert C. Martin beschreibt Code in seinem Buch „Clean Code“ als Sprache, in der die Entwickler die Anforderungen an die Software maschinenlesbar zum Ausdruck bringen [Mar09a]. Weiter beschreibt er in seinem Buch auch Prinzipien und Best Practices, die das Erstellen von verständlicherem und wartbarerem Quellcode unterstützen. Die selben Prinzipien wurden auch von Ralf Westphal und Stefan Lieser aufgegriffen und im Rahmen eines Wertesystems mit sieben Graden vorgestellt [WL11]. Martin's Buch „Clean Code“ bleibt auch für Westphal und Lieser die grundlegende Lektüre. Die Prinzipien und Best Practices der *Clean Code*-Bewegung umfassen viele Aspekte, die bereits durch Kent Beck in seinem Buch über die Methodik *Extreme Programming* (XP) [Bec99] eingeführt wurden. Drei wichtige Bestandteile von XP, die ebenfalls in den Graden des *Clean Code*-Wertesystems von Westphal und Lieser auftreten, werden jetzt wie folgt definiert.

Definition 1 (Automatisiertes Testen) *Automatisiertes Testen ist eine der wichtigsten Praktiken in der Clean Code-Bewegung. Dabei werden Testfälle in Verbindung mit dem Code implementiert. Diese Testfälle sind per Knopfdruck ausführbar und informieren den Entwickler darüber, ob das System noch alle spezifizierten Funktionen erfüllt.*

Laut Robert C. Martin ist es die Aufgabe eines Entwicklers, keinen Schaden in einem Softwaresystem anzurichten [Mar11]. Damit meint er Schaden an der Funktionalität der Anwendung (äußere Qualität) und Schaden an der Struktur und Form des Quelltext (innere Qualität). Beides kann mit automatisierten Testfällen verhindert werden.

Definition 2 (Ständiges Refactoring) Ständiges Refactoring *ist der effektivste Schutz gegen Schäden an der Struktur und der damit verbundenen inneren Qualität eines Softwaresystems. Durch evolutionäres Wachstum und viele Anpassungen kann es vorgekommen, dass die Struktur einer Codebasis neue Anpassungen nicht optimal unterstützt. Dieses Ändern der Struktur, damit Anpassungen wieder besser unterstützt werden, wird als Refactoring bezeichnet.*

Durch Tests kann sichergestellt werden, dass Refactorings den Funktionsumfang und damit die äußere Qualität nicht beeinträchtigen. Martin Fowler beschreibt solche Refactorings und deren Vorteile in seinem Buch [Fow99].

Definition 3 (Schnelle Codereviews) Schnelle Codereviews *sind eine weitere Voraussetzung um Code entwickeln zu können, der wirklich lesbar, verständlich und wartbar ist. Der entscheidende Punkt ist das Feedback. Je schneller der Entwickler Feedback bekommt, um so früher können Probleme im Quellcode erkannt und mit relativ wenig Aufwand korrigiert werden.*

2.1.2 Eclipse

Eclipse¹ ist eine *integrierte Entwicklungsumgebung* (engl. Integrated Development Environment, IDE) der Eclipse Foundation. Die Anwendung ist ein kostenfreies Open-Source-Programm und wird hauptsächlich für die Entwicklung von Software mit der Programmiersprache Java² verwendet. Eclipse unterstützt externe Erweiterungen in Form von Plugins.

2.1.3 Visual Studio

Visual Studio³ ist eine integrierte Entwicklungsumgebung von Microsoft. Im Gegensatz zu Eclipse ist Visual Studio ab der Professional-Version ein kommerzielles Produkt. Die Entwicklung für das .NET Framework⁴ von Microsoft kann mit Visual Studio durchgeführt werden, wobei mehrere Programmiersprachen unterstützt werden. Zu den bekanntesten und am weitesten verbreiteten Sprachen des .NET Frameworks zählen C[‡] und Visual Basic.NET.

Visual Studio erlaubt die Installation von externen Erweiterungen als Addin oder Erweiterung ab der Professional-Version. Die kostenfreie Express Edition besitzt einen sehr eingeschränkten Funktionsumfang. Der Unterschied zwischen einem Visual Studio-Addin und einer Visual Studio-Erweiterung ist der, dass Addins bereits in früheren Versionen unterstützt wurden und ein Entwickler die Schnittstellen in Visual Studio direkt anprogrammieren kann. Die Erweiterungen werden ab Visual Studio 2010 unterstützt und benötigen zur Entwicklung das Visual Studio 2010 SDK⁵, erlauben aber eine modernere Nutzung der Automatisierung-API. Aaron Marten beschreibt in seinem Artikel [Mar10]

¹Download und mehr Informationen: „Eclipse Foundation“ <http://www.eclipse.org/>

²Download und mehr Informationen: „Java“ <http://www.oracle.com/us/technologies/java/index.html>

³Mehr Informationen: „Microsoft Visual Studio 2010 Produktfamilie“ <http://www.microsoft.com/germany/visualstudio/products/default.aspx>

⁴Download und mehr Informationen: „Microsoft .NET Framework“ <http://www.microsoft.com/net>

⁵Download: „Visual Studio 2010 SP1 SDK“ <http://www.microsoft.com/download/en/details.aspx?id=21835>

die Möglichkeiten, wie mit der neuen Erweiterungstechnologie (VSIX) Addin-ähnliche Erweiterungen erstellt werden können.

2.2 Technische Grundlagen

In diesem Abschnitt werden die technischen Grundlagen erläutert. Dazu wird zunächst der Begriff der Objektorientierung eingeführt, da sämtliche Softwaresysteme, die in dieser Master-Thesis betrachtet werden, objektorientiert sind. Anschließend werden Graphen und Bäume beschrieben, da verschiedene [Softwariemetriken](#) anhand dieser Datenstruktur berechnet werden können. Abschließend wird das Konzept der statischen Code-Analyse vorgestellt, da die zu entwickelnde Erweiterung eben diese durchführen können soll.

2.2.1 Objektorientierung

Die Objektorientierung ermöglicht laut Grady Booch, die hohe Komplexität von Softwaresystemen zu beherrschen [\[Boo94\]](#). Das ist möglich, da diese Methode die Dinge der realen Welt als Objekte sieht und dadurch die Problemdomäne verständlich und anschaulich macht. Ein objektorientiertes Softwaresystem besteht aus fünf wesentlichen Komponenten, die jetzt wie folgt definiert werden. Die lateinischen Übersetzungen sind aus dem Buch von Bernd Oestereich übernommen [\[Oes01\]](#).

Definition 4 (Klasse) Klasse kommt aus dem lateinischen von *classis* und bedeutet „Aufgebot“. Eine Klasse der Typ aller seiner Objekte. Klassen können abstrakt sein. Eine abstrakte Klasse, oder auch Schnittstelle genannt, kann nicht als Objekt erzeugt werden. Eine nicht-abstrakte Klasse, also eine konkrete Klasse, muss von dieser Klasse eine Vererbung der Struktur und des Verhaltens durchführen. Von dieser ererbenden konkreten Klasse kann dann ein Objekt erzeugt werden, das sowohl vom Typ der konkreten Klasse als auch vom Typ der abstrakten Klasse ist.

Grady Booch bezeichnet eine Klasse auch als Menge von Objekten der gleichen Struktur und des gleichen Verhaltens.

Definition 5 (Objekt) Objekt kommt ebenfalls aus dem lateinischen von *obicere* und bedeutet „entgegenhalten“. Ein Objekt ist eine Instanz einer Klasse und stellt die Funktionen und Daten einer logischen Einheit in einem Softwaresystem dar.

Grady Booch beschreibt ein Objekt als ein Etwas, das einen Zustand, ein Verhalten und eine Identität hat [\[Boo94\]](#). Laut Bertrand Meyer kann auf ein Objekt per Referenz zugegriffen werden [\[Mey88\]](#).

Definition 6 (Attribut) Attribut kommt auch aus dem lateinischen von *attributum* und bedeutet „das Beigefügte“, was einer Eigenschaft oder einem Kennzeichen einer Sache entspricht. Die Daten, die ein Objekt ausmachen, werden anhand von Attributen gespeichert. Auf die Attribute kann von den Operationen des Objekts aus zugegriffen werden kann. Attribute werden auch als Felder bezeichnet und lassen sich in Klassenfelder und Instanzfelder unterscheiden. Klassenfelder können von allen Objekten der Klasse gemeinsam genutzt werden, während Instanzfelder unterschiedliche Werte für konkrete Objekte haben können.

Definition 7 (Operation) Operation kommt aus dem lateinischen von operatio und bedeutet „Handlung“. Eine konkrete Aktion, die anhand einer definierten Vorschrift durchgeführt wird, bezeichnet Oestereich in diesem Sinne als Operation. Das Verhalten von Objekten wird anhand ihrer Operationen festgelegt. Andere Bezeichnungen für Operation ist Funktion oder Methode. Methoden lassen sich in Klassenmethoden und Instanzmethoden aufteilen. Klassenmethoden können nicht auf die Instanzfelder sondern nur auf die Klassenfelder zugreifen. Instanzmethoden können auf Instanzfelder und auf Klassenfelder zugreifen.

Definition 8 (Paket) Paket ist ein hierarchisches Gruppierungskonstrukt, das es erlaubt Klassen und andere Pakete zu gruppieren. Pakete werden auch als Namespaces bezeichnet, da sie eine benannte Zusammenfassung von Klassen und anderen Namespaces sind.

Ein Paket wird von Bernd Oestereich als eine Ansammlung von Modellelementen bezeichnet [Oes01]. Dabei sind Modellelemente Klassen oder andere Pakete und dienen der besseren Strukturierung des Systems. Grady Booch beschreibt zwei weitere Mittel von objektorientierten Softwaresystemen [Boo94], die der Abstraktion dienen und die es erlauben, Verantwortlichkeiten einfacher zu organisieren. Beide Konzepte werden jetzt ebenfalls definiert.

Definition 9 (Assoziation) Assoziation kommt aus dem lateinischen von associare und bedeutet „verbinden“. Eine Assoziation verbindet zwei Klassen, sodass Objekte der einen Klasse auf Objekte der verbundenen Klasse zugreifen können. Damit ist die Klasse, von der die Assoziation ausgeht, von der anderen Klasse abhängig. Anders ausgedrückt hat diese Klasse eine Abhängigkeit von der anderen.

Bernd Oestereich beschreibt eine Assoziation als eine *Hat-eine-Beziehung*, die angibt, dass eine Klasse mit einer anderen Klasse zusammenarbeitet. Grady Booch bezeichnet diese Zusammenarbeit auch als semantische Verbindung.

Definition 10 (Vererbung) Vererbung erlaubt es Klassen, das Verhalten und die Struktur anderer Klassen zu erben oder zu spezialisieren. Ein Objekt der erbenden Klasse hat dabei erstmal die gleiche Struktur und das gleiche Verhalten, wie ein Objekt der vererbenden Klasse oder Oberklasse genannt. Diese Struktur und das Verhalten können von der erbenden Klasse verändert werden. Die erbende Klasse hat eine Abhängigkeit von der Oberklasse.

Vererbung wird von Grady Booch auch als eine *Ist-ein-Beziehung* beschrieben. Mit diesen sieben Konzepten kann ein objektorientiertes Softwaresystem vollständig beschrieben werden.

2.2.2 Graphentheorie

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest und Clifford Stein beschreiben einen Graph in ihrem Buch „Introduction to Algorithms“ [CLRS09] als Tupel $G = (V, E)$. Der erste Wert, V , ist eine endliche Menge an Knoten und der zweite, E , ist eine endliche Menge an Kanten zwischen diesen Knoten. Eine Kante ist ein Paar aus $V \times V$ und beinhaltet die beiden Knoten, die sie verbindet. Cormen und co unterscheiden in gerichtete und

ungerichtete Graphen. Bei Letzterem verbindet eine Kante die Knoten in beiden Richtungen, wobei eine gerichtete Kante die Verbindungsrichtung anhand der Knoten-Reihenfolge im Kanten-Tupel vorgibt. Ein gerichteter Graph wird auch als Digraph bezeichnet. Weiter definieren Cormen und seine Koautoren einen Baum als zusammenhängenden azyklischen Graph, indem jeder Knoten von einem Wurzelknoten aus erreichbar ist.

2.2.3 Statische Code-Analyse

Artur Wagner beschreibt in seiner Seminar-Ausarbeitung [Wag] die *statische Code-Analyse* als Prozess, der von einem Compiler-ähnlichen Programm gestartet wird. Dieses Programm führt den Code nicht aus und erzeugt auch keinen ausführbaren Code, sondern nimmt eine *Analyse* des Codes vor. Wagner erwähnt noch, dass eine *statische Code-Analyse* keinen Korrektheitsnachweis liefert sondern lediglich eine Art Systemzusammenfassung erzeugt. Eine solche *Analyse* findet dabei automatisiert statt. Christof Ebert, Reiner Dumke, Manfred Bundschuh und Andreas Schmietendorf stellen ebenfalls den Vergleich mit einem Compiler an. In ihrem Buch [EDBS05] erwähnen sie zusätzlich noch ein Qualitätsmodell, das mit dem Ergebnis der *Analyse* in Verbindung gebracht wird und so eine einfache Interpretierbarkeit des analysierten Systems ermöglicht. Zusätzlich beschreiben sie den Vorgang als Sammeln von *Metriken* und deren Bewertung anhand von Kriterien.

Nadine Vehring beschreibt eine *Metrik* in ihrer Seminar-Ausarbeitung [Veh07] als Maß dafür, wie sehr eine „Entität“ ein „Attribut“ erfüllt. Als „Entität“ bezeichnet Vehring einen Teil des analysierten Systems, also beispielsweise eine Methode, Klasse oder eine Menge von Klassen. Weiterhin beschreibt sie ein „Attribut“ als Eigenschaft, die im betrachteten System, also bei den „Entitäten“ gemessen werden soll. In der vorliegenden Master-Thesis wird eine *Metrik* wie folgt definiert.

Definition 11 (Metrik) *Metrik ist der Begriff für eine bestimmte und tatsächlich gemessene Eigenschaft, die Methoden, Klassen, Paketen, oder Mengen von diesen, im zu analysierenden System besitzen. Damit sind die Metriken das direkte Ergebnis einer statischen Code-Analyse. Der Begriff wird häufig auch synonym für die Eigenschaft verwendet, die gemessen werden soll.*

Definition 12 (Statistik der Metrik) *Die Statistik der Metrik ist die Bezeichnung für die zusammengefassten Werte einer Metrik in einem System. Diese Werte können vor und nach der Aggregation gewichtet und bewertet werden.*

Das einzelne Methoden oder Klassen in einer Statistik nicht berücksichtigt werden, kann bei der Berechnung der Statistik ebenfalls festgelegt werden. Marc Philipp und Nicole Rauch unterscheiden in ihrem Artikel [PR10] ebenfalls zwischen *Metrik* und Statistik. Die *Metriken* eines Systems können auch manuell bestimmt werden. In dieser Master-Thesis werden *Metriken* aber immer als automatisiert bestimmbar und damit als Ergebnis einer automatisierten statischen Code-Analyse betrachtet. Das Konzept der *statischen Code-Analyse* kann auch genutzt werden, um subjektive explizite Probleme im Code zu lokalisieren. Dazu gehören beispielsweise die Verwendung von obsoleten Klassen und Methoden, sowie fehlende Klassenkommentare und unkonventionelle Benennungen. In dieser Master-Thesis wird das Konzept der *statischen Code-Analyse* ausschließlich für die Bestimmung der *Metriken* verwendet, die in Abschnitt 5.2 beschrieben werden.

3 Anforderungen

In diesem Kapitel werden die Anforderungen an die Visual Studio-Erweiterung zur statischen Code-Analyse beschrieben, die im Rahmen dieser Master-Thesis entwickelt werden soll. Dabei werden die Anforderungen in einem agilen Kontext wie Sprint 0¹ bestimmt und in Form von Zielen, Interessenvertreter, Use Cases und User Stories definiert. Dieses Kapitel gibt damit eine Antwort auf die Frage, was während dieser Master-Thesis gemacht werden soll.

3.1 Interessenvertreter

Die *Interessenvertreter* (engl. Stakeholder) im Kontext dieser Master-Thesis sind Gruppen von Personen, die einen Einfluss auf Anforderungen des Systems haben können. Die folgenden fünf Gruppen wurden dafür identifiziert.

Interessenvertreter 1 (andrena objects ag) *Als Arbeitgeber stellt andrena das Umfeld dar, indem das Projekt durchgeführt und das System implementiert wird. Sie legt Wert auf agile Softwareentwicklung und möchte Entwicklungsprozesse und Praktiken gewinnbringend einsetzen. Die andrena objects ag hat bereits ein Plugin zur statischen Code-Analyse für Java und Eclipse entwickelt und möchte ihren Entwicklern ein ähnliches Programm auch für die .NET-Umgebung zur Verfügung stellen können.*

Interessenvertreter 2 (Leser dieser Thesis) *Die Leser der vorliegenden Thesis haben ebenfalls ein Interesse an dem zu entwickelnden Produkt. Dieses Interesse ist dabei hauptsächlich theoretischer Natur. Sie haben einen Informatik-artigen Hintergrund und möchten einen Einblick in die Algorithmen und die Implementierungsdetails sämtlicher implementierten Aspekte des Systems erhalten.*

Interessenvertreter 3 (.NET-Entwickler mit wenig Erfahrung) *Entwickler mit wenig Erfahrung arbeiten an Projekten mit relativ wenig Quellcode und wollen auf einfache Weise einen Überblick über Problemfälle des Systems behalten. Mithilfe des Systems wollen sie diese rechtzeitig beseitigen und sicherstellen, dass dadurch keine neuen Probleme entstehen.*

Interessenvertreter 4 (.NET-Entwickler mit viel Erfahrung) *Entwickler mit viel Erfahrung arbeiten an Projekten mit viel und kompliziertem Quellcode. Sie möchten das System einsetzen um Tendenzen im System zu erkennen und Spezialfälle zu analysieren. Auch sie wollen einen besseren Überblick erhalten und das System aus möglichst vielen verschiedenen Perspektiven sehen.*

¹Agile Anforderungsanalyse: „10 Things to do in Sprint 0“ <http://www.pmscrum.com/blog/2011/06/10-things-do-sprint-0>

Interessenvertreter 5 (.NET, Clean Code-Entwickler) *Clean Coder legen sehr viel Wert auf Quellcode, der möglichst einfach zu verstehen ist und damit schnell an neue Anforderungen angepasst werden kann. Sie erwarten von dem System Feedback und Unterstützung bei der Entwicklung von sauberem Code. Außerdem möchten sie auf Stellen im Code hingewiesen werden, die nicht ihren Vorstellungen entsprechen.*

Die Beschreibungen der Interessengruppen sind grob und allgemein gehalten, während versucht wurde, die direkten Bedürfnisse der jeweiligen Gruppe zu benennen. Natürlich ist die Erklärung damit nicht vollständig oder exklusiv. So kann beispielsweise auch ein erfahrener .NET-Entwickler ein Interesse an **Clean Code** haben.

3.2 Ziele

Nachdem die beteiligten Interessengruppen definiert wurden, werden in diesem Abschnitt die Ziele des zu entwickelnden Systems beschrieben. Peter Hruschka und Chris Rupp erklären in ihrem Buch [HR02] ein Ziel als „ein erstrebenswerter Zustand“ in der Zukunft, den es zu erreichen gilt¹. Damit ist ein Ziel eine abstrakte Form einer Anforderung an das System, die sich auch in einem agilen Projekt nicht so schnell ändert wie spezielle Anforderungen. In der vorliegenden Arbeit werden nun die folgenden vier Ziele definiert.

Ziel 1 (Einsicht in die Codebasis) *Große Entwicklungsprojekte haben eine große Codebasis, die es den Entwicklern erschwert, Aussagen über den Quellcode des Systems zu treffen. Mithilfe statischer Code-Analyse kann eine Einsicht in diesen Quellcode gewährleistet und so zum Vorteil aller Interessenvertreter genutzt werden. Eigenschaften des Quellcodes können als **Metriken** berechnet und bewertet werden. Dieses Ziel ist erreicht, sobald ein Entwickler, der die Codebasis nicht oder nur unzureichend kennt, Eigenschaften des Codes erkennen und darauf reagieren kann.*

Ziel 2 (Erkennen von Problemfällen) *In einer vorhandenen Codebasis besteht die Gefahr, den Überblick über Problemfälle zu verlieren. Ein Problem kann beispielsweise eine Klasse sein, die zu viele Abhängigkeiten zu anderen Klassen hat, oder eine Methode, die zu viel Funktionalität besitzt. Das Erkennen und Anzeigen dieser Stellen ist ein Vorteil, von dem alle Interessenvertreter profitieren können und der es ermöglicht, Schwachstellen systematisch zu entfernen. Dieses Ziel ist erreicht, sobald ein Entwickler, der die Codebasis nicht oder nur unzureichend kennt, Probleme im Code erkennen und darauf reagieren kann.*

Ziel 3 (Förderung von Clean Code) *Das Entwickeln von Quellcode kann auf eine sehr rücksichtslose Art und Weise geschehen, sodass spätere Anpassungen nur schwer vorzunehmen sind. Durch motivierendes Feedback während der Programmierung, kann die Entwicklung von **Clean Code** gefördert werden. Das führt zu einer Codebasis, die flexibler auf Anpassungen reagieren kann. Dieses Ziel ist erreicht, sobald ein Entwickler mehr **Clean Code-Prinzipien** und -Praktiken beachtet und einhält, als er dies ohne Unterstützung des Systems getan hätte.*

¹Seite 26

Ziel 4 (Interpretation der Softwarequalität) Um eine bestehende Codebasis zu bewerten, ist eine detaillierte Kenntnis über alle Klassen, deren Interaktionen und vielem mehr, erforderlich. Außerdem muss eine Bewertung durch ein Codereview manuell jedes mal neu erfolgen, ist subjektiv und fehleranfällig. Eine teilweise automatisierte Interpretation anhand von Regeln und Gewichtungen, die auf Erfahrungen basieren, kann die menschliche Bewertung optimieren und einem Entwickler jederzeit mit verhältnismäßig geringem Aufwand eine Qualitätsinterpretation ermöglichen. Dieses Ziel ist erreicht, sobald ein Entwickler, der die Codebasis nicht oder nur unzureichend kennt, die Qualität des Quellcodes durch Interpretation einschätzen und mit anderen Systemen vergleichen kann.

3.3 Architektur

In diesem Abschnitt wird die grundlegende Architektur des zu entwickelnden Systems erläutert. Im Rahmen dieser Master-Thesis soll ein System entwickelt werden, dass die in Abschnitt 3.2 genannten Ziele erreicht. Zusätzlich soll dieses System als Erweiterung in Visual Studio laufen und den Quelltext (kompiliert und/oder unkompiliert) in einer .NET-Sprache, wie beispielsweise C#, analysieren. Abbildung 3.1 zeigt die beiden Komponenten *Visualisierung* und *Statische Code-Analyse*, um die es sich im weiteren Verlauf handeln wird. Die Komponente, die die statische Code-Analyse durchführt, verwendet drei

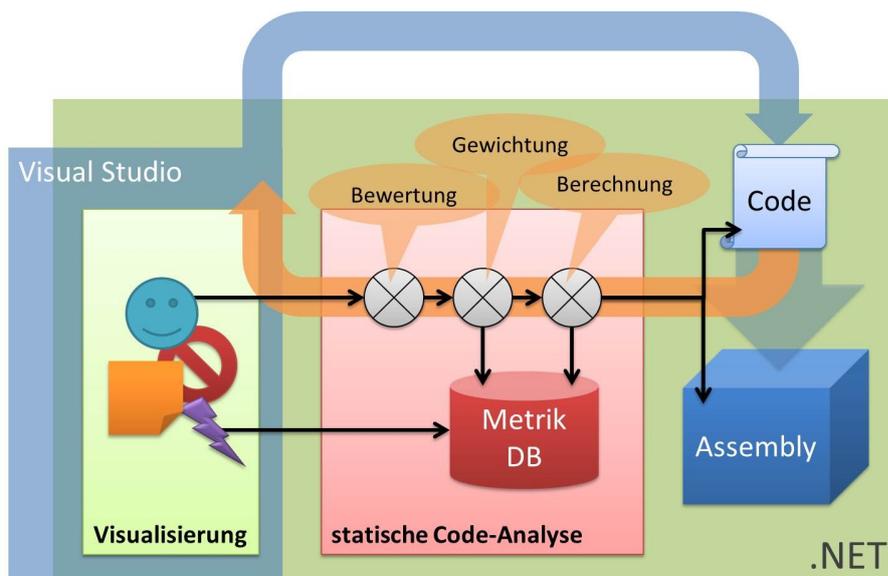


Abbildung 3.1: Grobe Architektur des zu entwickelnden Systems

Aktionen, die einem kompletten Analyseprozess entsprechen. Die schwarzen Pfeile symbolisieren dabei Zugriffe auf Daten. Die Aktion *Berechnung* betrachtet das Softwaresystem und kann dafür auf den Quellcode und das erzeugte Kompilat zugreifen. Das Ergebnis dieser Betrachtung wird in Form von *Metriken* in einer Datenbank gespeichert, die die Daten für die Aktion *Gewichtung* bereitstellt. Diese gewichtet die Daten und erzeugt Statistiken, die ebenfalls in der Datenbank hinterlegt werden. Die Aktion *Bewertung* versucht anhand der gewichteten Daten eine Aussage über die Qualität des Softwaresystems zu treffen. Nachdem diese drei Aktionen abgelaufen sind, ist der Analyseprozess beendet.

Die Komponente *Visualisierung*, die in Visual Studio ausgeführt wird, kann diesen Analyseprozess starten und auf dessen Daten zugreifen. Sie zeigt die berechneten **Metriken** in geeigneten Darstellungsformen an und gibt, basierend auf der Bewertung, ein entsprechendes Entwicklerfeedback. Dieses Feedback bezieht sich direkt auf den Quellcode, der von der **statischen Code-Analyse** verarbeitet wurde. Nach einer Veränderung des Quelltext, kann der Prozess erneut gestartet werden, um auch diese Veränderung zu analysieren. So entsteht theoretisch ein kontinuierlicher Verbesserungskreislauf. Dieser Zyklus wird durch die breiten transparenten Pfeile angezeigt und entspricht dem Informationsfluss in dieser Umgebung.

3.4 Use Cases

Aus den in Abschnitt 3.2 festgelegten Zielen dieses Projekts lassen sich zunächst vier Anwendungsfälle bestimmen, die in Abbildung 3.2 dargestellt sind. Die Akteure entsprechen

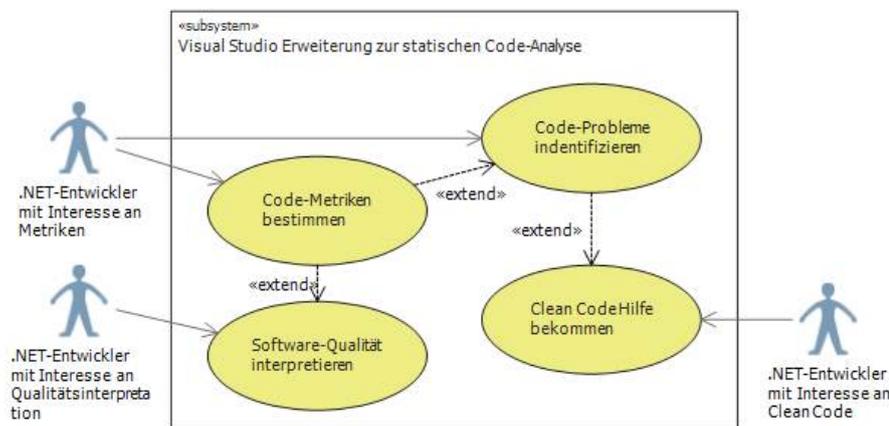


Abbildung 3.2: Anwendungsfälle des zu entwickelnden Systems

ungefähr den in Abschnitt 3.1 definierten Interessenvertretern. Alle Akteure sind .NET-Entwickler und unterscheiden sich durch ihr Interesse. *.NET-Entwickler mit Interesse an Metriken* entspricht ungefähr dem .NET-Entwickler mit wenig Erfahrung (Interessenvertreter 3). Seine Interaktion mit dem System beschränkt sich auf die Bestimmung von **Codemetriken** sowie der Identifizierung von Code-Problemen als Verletzung definierter **Metrik-Grenzen**. Der .NET-Entwickler mit viel Erfahrung (Interessenvertreter 4) findet sich im Akteur *.NET-Entwickler mit Interesse an Qualitätsinterpretation* wieder. Er benutzt das System um eine Interpretation der Qualität zu erhalten, die anhand der **Metriken** vorgenommen wird. Der Akteur *.NET-Entwickler mit Interesse an Clean Code* ist an den Clean Code-Entwickler (Interessenvertreter 5) angelehnt. Er lässt sich durch das System bei der Erzeugung von **Clean Code**, wie er in Unterabschnitt 2.1.1 beschrieben wurde, unterstützen. Diese Hilfestellung geschieht anhand der Code-Probleme, die durch **Code-metriken** erkannt werden. Es ist ebenfalls zu sehen, dass der Anwendungsfall *Codemetriken bestimmen* alle anderen Anwendungsfälle direkt oder indirekt ergänzt. Damit ist er ein entscheidender Bestandteil des Systems, wenn nicht sogar der wichtigste. *andrena objects ag* (Interessenvertreter 1) und *Leser des Thesis* (Interessenvertreter 2) haben keine direkte Interaktion mit dem System und werden bei der Betrachtung der Use Cases ignoriert.

3.5 Produkt-Backlog

Die Anwendungsfälle lassen sich in *User Stories* („Anwendererzählungen“) unterteilen. Abbildung 3.3 zeigt diese Aufteilung in der horizontalen und die Verteilung auf geplanten Iterationen in der vertikalen Dimension. Dieses zwei-dimensionale Backlog stellt die konkreten Anforderungen aus Benutzersicht an das komplette System dar. Die User Sto-

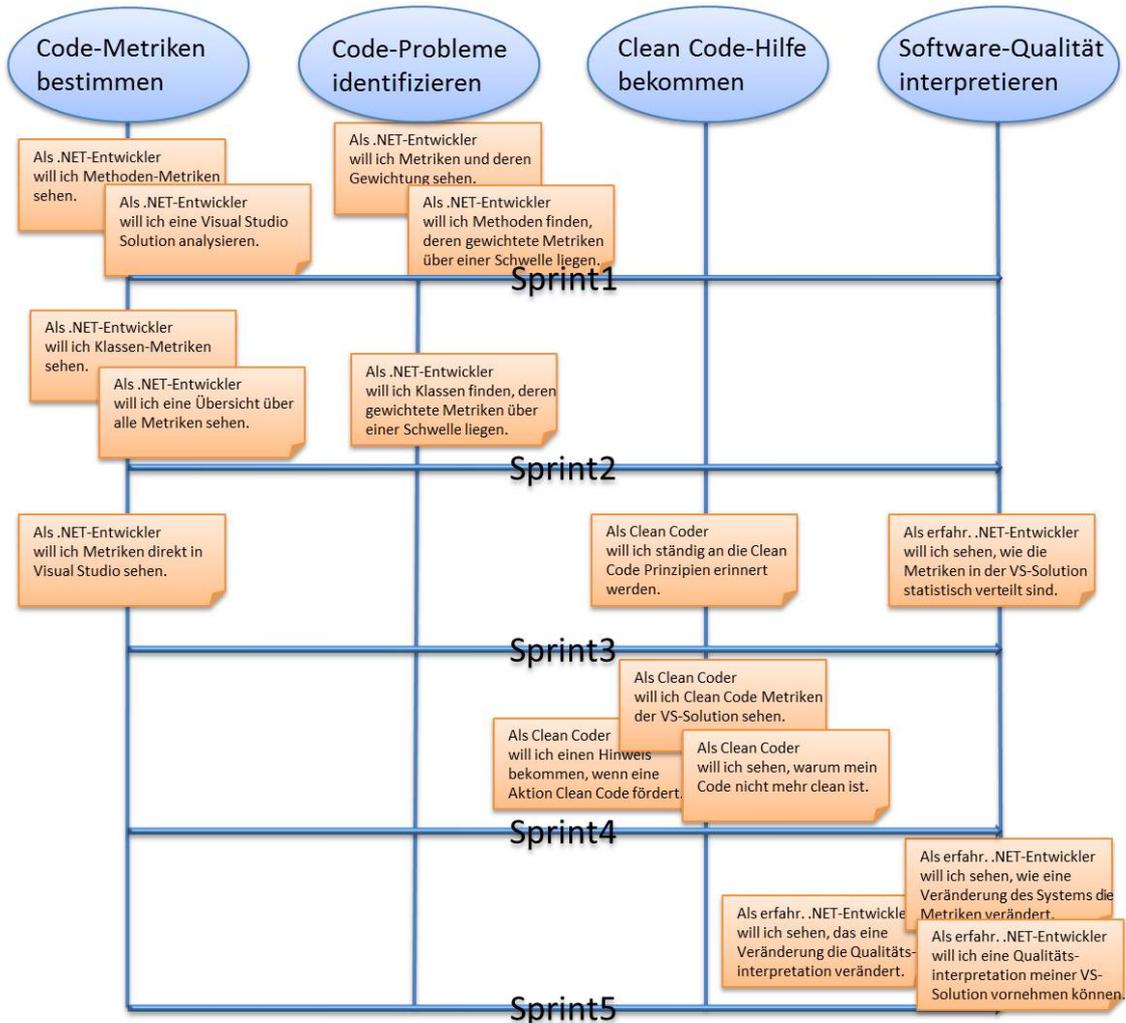


Abbildung 3.3: Produkt-Backlog des zu entwickelnden Systems

ries sind allgemein gehalten, da auf ein großes Design vor der Implementierung aus Agilitätsgründen verzichtet wurde. Pro Iteration werden die User Stories dann in konkrete Entwicklungs-Aufgaben unterteilt. Die Begründung jeder Story wurde nicht explizit aufgeschrieben, da die Beziehung zu den Use Cases und damit den Zielen aus Abschnitt 3.2 erhalten und offensichtlich geblieben ist. Eine Priorisierung ist ebenfalls implizit durch das Aufteilen auf die Iterationen entstanden, sodass eine explizite Business Value-Analyse nicht notwendig ist. Da die Entwicklungsgeschwindigkeit noch nicht bekannt ist, ist die Planung der Iterationen vorläufig. Eine Änderung kann also jederzeit erfolgen, wenn neue

Use Cases und User Stories dazukommen, andere wegfallen oder nicht alle in einer Iteration geschafft werden. Zum Zeitpunkt dieser Planung lassen sich die bereits gefunden User Stories in schätzungswiese fünf Iterationen implementieren.

Was die Abnahmekriterien jeder Anwendererzählung betrifft, so wurde auch hier kein Mehrwert in der expliziten Definition gesehen. Pro Story kann ohne viel Aufwand ein exemplarischer manueller Akzeptanztest durchgeführt werden, der zeigt, ob das Ziel erreicht wurde. Zusätzlich können automatisierte Akzeptanztests zu Beginn der jeweiligen Iterationen festgelegt werden.

4 Vorgehensweise

Nachdem die Anforderungen an das System, das in dieser Master-These entwickelt werden soll, in Kapitel 3 definiert wurden, wird in diesem Kapitel die Vorgehensweise beschrieben. Dieses Kapitel gibt damit eine Antwort auf die Frage, wann die Anforderungen realisiert werden sollen. Dabei wird zuerst eine grobe Zeitplanung vorgestellt, die zu Beginn der These erstellt wurde. Anschließend wird dieser ursprüngliche Plan mit der tatsächlichen zeitlichen Durchführung verglichen, dessen Plan gegen Ende der These erstellt wurde. Abbildung 4.1 zeigt die zeitliche Aufteilung der Aufgaben zu Beginn der Master-These und nimmt eine Aufteilung in einen praktischen und einen theoretischen Teil vor. Beide Teile überlagern sich, sodass eine Dokumentation der Implementierung zeitnah erfolgen kann und keine große Schreibphase gegen Ende des sechsmonatigen Zeitrahmens notwendig ist.

4.1 Praktischer Teil

Der praktische Teil besteht aus einer Evaluierungsphase und der eigentlichen Implementierungsphase. Während der Evaluierung werden verschiedene Technologien betrachtet, mit denen eine [statische Code-Analyse](#) durchgeführt werden kann. Das Ergebnis der Evaluierung wird die Grundlage der nächsten Phase darstellen. In der zweiten Phase werden die User Stories aus Abschnitt 3.5 in fünf zweiwöchigen (10 Tage) Iterationen implementiert.

Iteration 1 16.05.12 bis 29.05.12

Iteration 2 01.06.12 bis 14.06.12

Iteration 3 19.06.12 bis 02.07.12

Iteration 4 05.07.12 bis 18.07.12

Iteration 5 01.08.12 bis 14.08.12

Nach jeder Iteration liegt eine funktionstüchtige Version der Visual Studio-Erweiterung zur statischen Code-Analyse vor. Außerdem folgen allen Iterationen Phasen der schriftlichen Zusammenfassung.

4.2 Theoretischer Teil

Der theoretische Teil entspricht hauptsächlich der Dokumentation aller Ergebnisse und deren Beschreibung in Form der schriftlichen Ausarbeitung dieser Master-These. So werden in einer Planungsphase zunächst Grundlagen (Kapitel 2) beschrieben, sowie eine agile Anforderungsanalyse (Kapitel 3) durchgeführt. Außerdem wird das Usus-Programm, dessen Funktionsumfang einen maßgeblichen Einfluss auf das entwickelte System hat, näher

betrachtet und dessen **Metriken** beschrieben (Kapitel 5). Anschließend wird das Ergebnis der Evaluierungsphase beschrieben (Kapitel 7), bevor die tatsächlichen Iterationen in Form von schriftlichen Zusammenfassungen abgeschlossen werden (Kapitel 8, 9, 10 und 11). Nach der Zusammenfassung der vierten Iteration wird eine Fallstudie auf Basis eines andrena-Kurs zum Thema Refactoring durchgeführt. Bei diesem Kurs wird das in dem praktischen Teil entwickelte System eingesetzt und ermittelt, wie gut das Werkzeug einen Entwickler unterstützt (Kapitel 12). Unmittelbar vor der fünften und vorläufig letzten Iteration wird eine Analyse von **statistischen Verteilungen** der **Metrik**Metriken sowie deren Bedeutung erfolgen (Kapitel 10).

4.3 Tatsächlicher Ablauf

Nach der ersten Iteration bestand das in dieser Master-Thesis entwickelte Programm nur aus einem Kommandozeilenwerkzeug, welches **Assemblies** einlesen und deren Methoden analysieren konnte. In der anschließenden Zusammenfassung sind große Teile von Kapitel 8 entstanden. Die zweite Iteration hat eben dieses Kommandozeilenwerkzeug um die Fähigkeit auch Klassen analysieren zu können erweitert. Kapitel 8 konnte danach abgeschlossen werden. Erst nach der dritten Iteration war das Programm zum ersten Mal als Erweiterung in Visual Studio lauffähig und führte die **Analysen** bei jedem Build-Vorgang durch. Die daraus resultierende Zusammenfassung war aufwendiger als erwartet und dauerte vier Tage. Dabei ist Kapitel 9 entstanden. Die darauf folgende vierte Iteration musste daher um eine halbe Woche verschoben werden. Der neue Zeitraum dieser Iteration ist 09.07.12 bis 20.07.12.

Nach dieser vierten Iteration hatte das Programm als Visual Studio-Erweiterung die Fähigkeit eine **Clean Code-Metrik** sowie den **Softwarequalitätsindex** zu bestimmen. Ein besseres Verständnis der Use Cases *Clean Code-Hilfe bekommen* (Ziel 3) und *Softwarequalität interpretieren* (Ziel 4) hat dazu geführt, dass die für die fünfte Iteration geplanten User Stories aus Abschnitt 3.5 bereits in der vierten Iteration implementiert werden konnten. Die anschließende Zusammenfassung dieser Iteration dauerte dadurch länger. Für den **Clean Code**-Teil, der in Kapitel 10 beschrieben wird, und den **SQI**-Teil, der in Kapitel 11 vorgestellt wird, wurden jeweils drei Tage benötigt. Da für diese Zusammenfassung aber nur zwei Tage eingeplant waren, wurde das Refactoring-Fallbeispiel geringer priorisiert und als Teil der fünften Iteration vorgenommen. Diese fünfte und letzte Iteration brauchte daraufhin nicht wieder verschoben zu werden.

Während der fünften Iteration wurden einige Fehler des Programms behoben, sowie die Performance gemessen und verbessert. Außerdem wurde die **Analyse** des andrena-Fallbeispiels durchgeführt. Die Zusammenfassungen der Laufzeitmessungen und des Fallbeispiels konnten in jeweils zwei Tagen abgeschlossen und in Kapitel 12 als Evaluierung der Visual Studio-Erweiterung beschrieben werden. Da die fünftägige Analyse der **Metrik-Verteilungen** als Teil der **Clean Code-Metrik** bereits in der vierten Iteration implementiert werden konnte, konnten diese Tage gespart und die zeitliche Verspätung wieder aufgeholt werden. Abbildung 4.2 stellt den tatsächlichen Projektablauf grafisch dar.

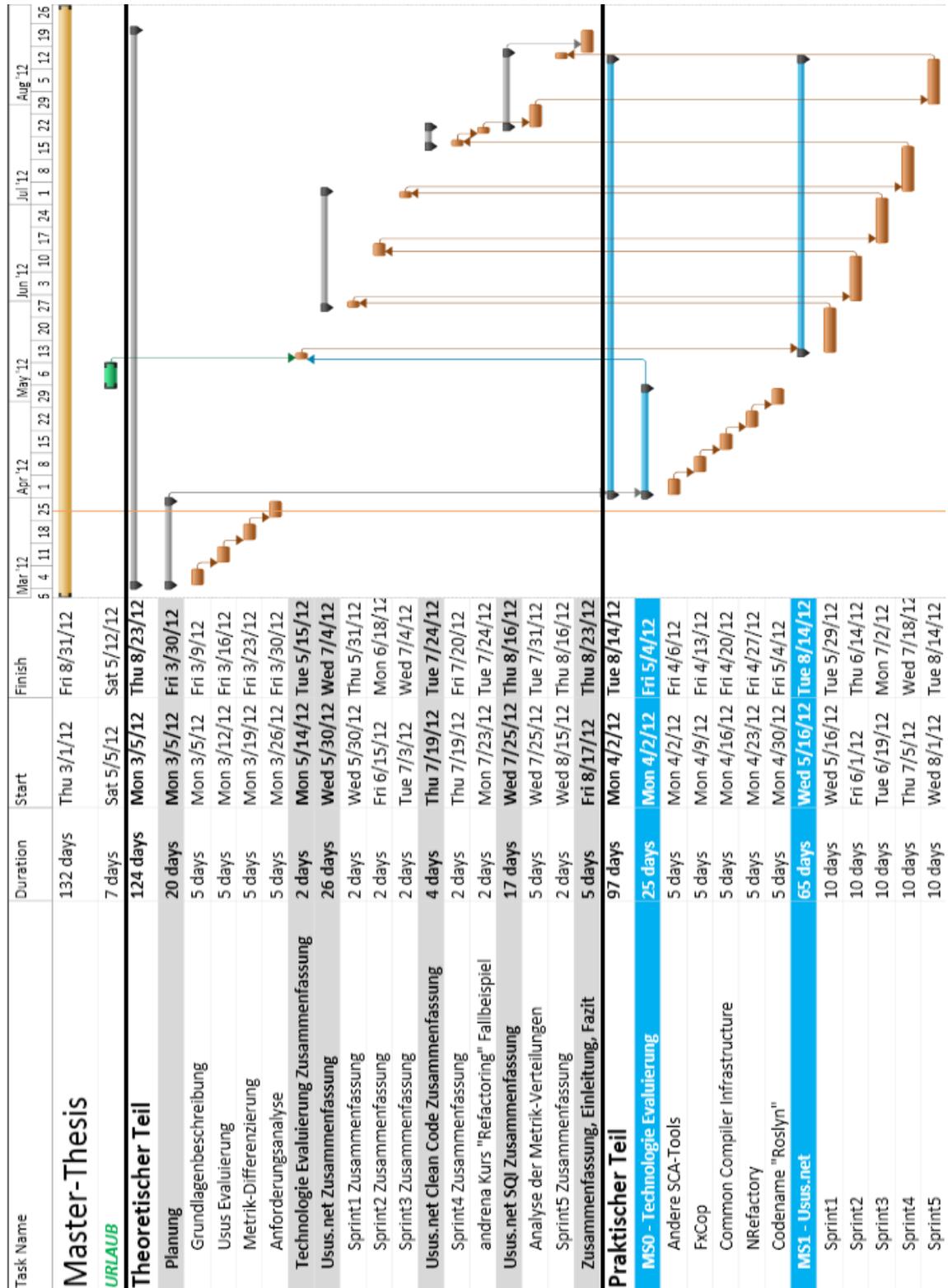


Abbildung 4.1: Ursprünglicher Projektplan

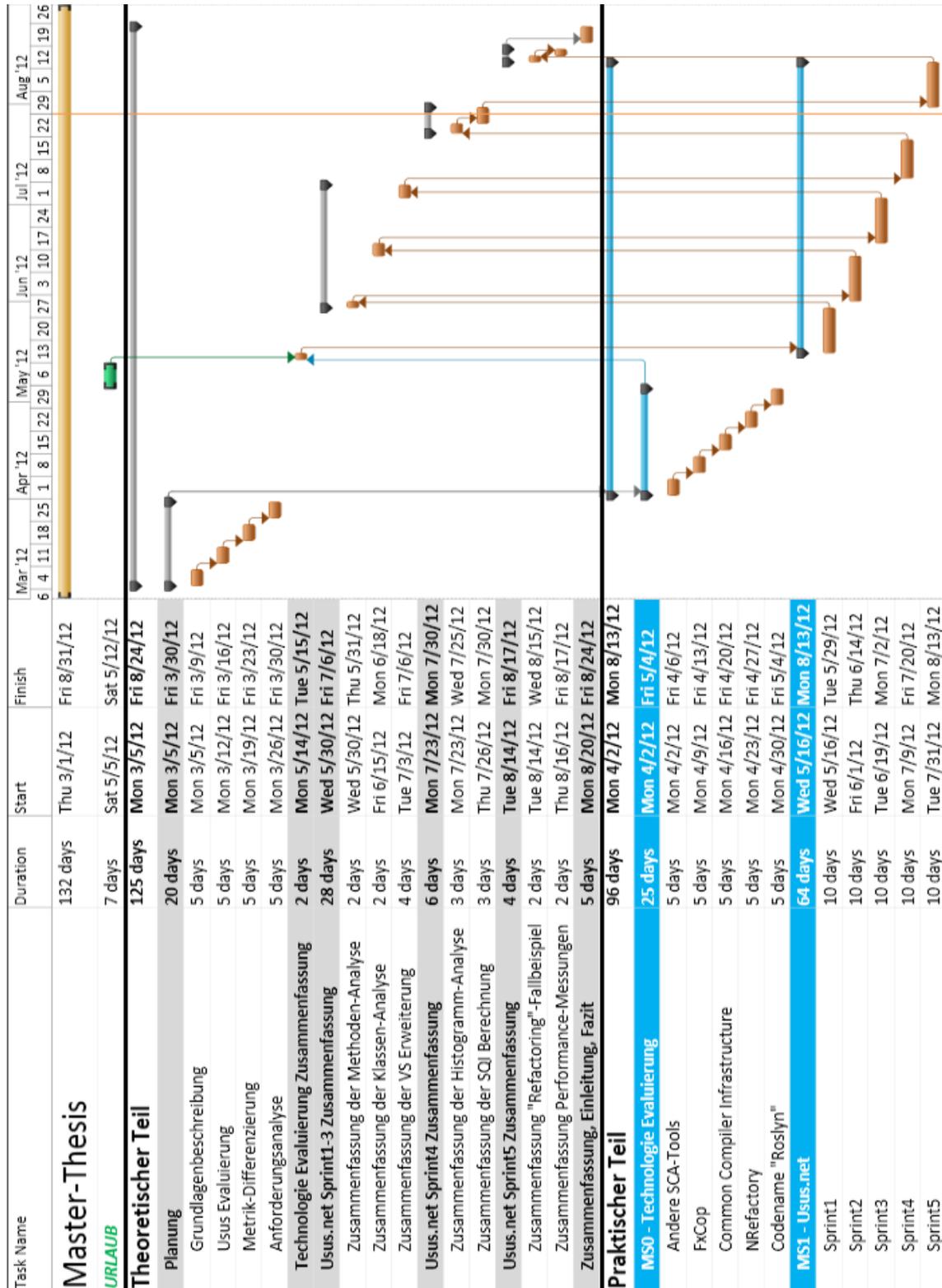


Abbildung 4.2: Tatsächlicher Projektablauf

5 Usus

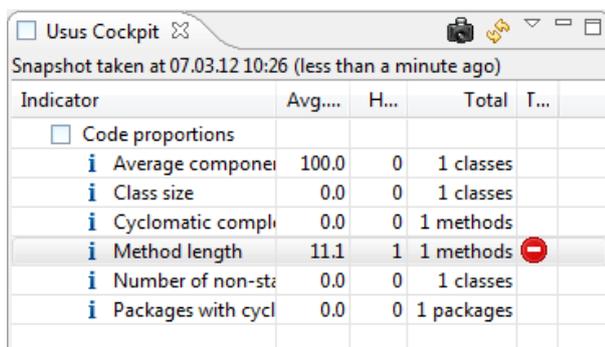
Das Wort *usus* kommt aus dem Lateinischen und bedeutet „das, was üblich ist“. In diesem Kapitel wird das Usus-Plugin [FRS⁺10] für die Java-Entwicklungsumgebung Eclipse vorgestellt, sowie auf die **Metriken**, die es berechnet, eingegangen. Das Usus-Programm ist im Rahmen einer Initiative von andrena (Abschnitt 1.2) entstanden. Usus entspricht im Groben dem Werkzeug für Eclipse und Java, das im Rahmen dieser Master-Thesis auch für Visual Studio und .NET entwickelt werden soll. Aus diesem Grund wird dessen Funktionsumfang an dieser Stelle betrachtet, um ihn besser nachimplementieren zu können. Eine direkte Portierung ist anhand der Technologieunterschiede von Visual Studio und Eclipse, sowie .NET und Java höchstwahrscheinlich nicht möglich, sodass diese Option in der vorliegenden Master-Thesis nicht weiter verfolgt wird.

5.1 Eclipse-Plugin

Über den Menüeintrag **Help / Install New Software** lässt sich das Usus-Plugin für Eclipse installieren. Dort muss zunächst eine neue Software Site mit der URL <http://projectusus.googlecode.com/svn/updates/> hinzugefügt und anschließend „Project Usus“ ausgewählt werden. Nach der Installation steht die „Project Usus perspective“ zur Verfügung, die die folgenden Fenster enthält.

5.1.1 Usus Cockpit

In diesem Fenster werden die **Usus-Metriken** angezeigt, die für alle Projekte, die Usus betrachtet, berechnet werden. Zusätzlich wird der Trend pro **Metrik** dargestellt, also ob sich die Ausprägung der **Metrik** verbessert oder verschlechtert hat. Die Verbesserung wird dabei zwischen zwei erstellten Snapshots gemessen, die entweder manuell oder durch einen neuen Speichervorgang ausgelöst werden können. Die in Abbildung 5.1 dargestellten Statistiken



The screenshot shows the 'Usus Cockpit' window with a table of metrics. The table has columns for 'Indicator', 'Avg....', 'H...', 'Total', and 'T...'. The 'Method length' row is highlighted in grey and has a red minus sign in the 'T...' column, indicating a decrease. The 'Code proportions' checkbox is unchecked.

Indicator	Avg....	H...	Total	T...
<input type="checkbox"/> Code proportions				
i Average component	100.0	0	1 classes	
i Class size	0.0	0	1 classes	
i Cyclomatic complex	0.0	0	1 methods	
i Method length	11.1	1	1 methods	-
i Number of non-sta	0.0	0	1 classes	
i Packages with cycl	0.0	0	1 packages	

Abbildung 5.1: Usus Cockpit zeigt Übersicht über alle Projekte

der **Metriken** errechnen sich aus der Aggregation der Paket-, Klassen- oder Methoden-Eigenschaften.

5.1.2 Usus Info

Dieses Fenster lässt sich im Kontext einer Methode oder einer Klasse öffnen und zeigt **Metriken**, die anhand der Eigenschaften des Kontextes ermittelt werden können. Das in

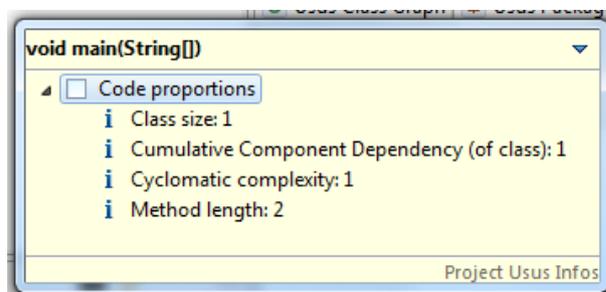


Abbildung 5.2: Usus Info zeigt Übersicht über eine Methode oder Klasse

Abbildung 5.2 gezeigte Info-Fenster lässt sich mit **Ctrl-U** öffnen und mit **Esc** wieder schließen.

5.1.3 Usus Hotspots

Dieses Fenster zeigt sogenannte Hotspots, also Stellen im Quellcode, dessen **Metriken** definierte Grenze oder einen Schwellwert überschreiten. Hotspots lassen sich für jede **Metrik** einsehen, die im Usus Cockpit angezeigt wird. Zusätzlich wird der Trend pro Hotspot angezeigt, also ob sich die **Metrik** des Hotspots verbessert oder verschlechtert. Das in Ab-

The screenshot shows a window titled 'Method length: Hotspots are methods with more than 9 statements. Rating fu'. Below the title, there is a table with four columns: 'Value', 'Name', 'Path', and 'Trend'. The table contains four rows of data. The first row has a value of 24 and a path of '\BadPractice_\src\misc'. The second row has a value of 13 and a path of '\BadPractice_\src\coding'. The third and fourth rows have values of 12 and 10, and paths of '\BadPractice_\src\coding\tests'. The 'Trend' column for all rows shows a value of 0. There are also some icons and a search bar at the top of the window.

Value	Name	Path	Trend
24	ArtikelpreisKalkulator...	\BadPractice_\src\misc	0
13	DogHandlerFuerBew...	\BadPractice_\src\coding	0
12	DogHandlerTest.train...	\BadPractice_\src\coding\tests	0
10	DogHandlerTest.train...	\BadPractice_\src\coding\tests	0

Abbildung 5.3: Usus Hotspots zeigt die Stellen im Code, die eine besonders negative Ausprägung der **Metriken** haben

Abbildung 5.3 gezeigte Hotspot-Fenster zeigt Hotspots immer nur für eine **Metrik** an. Der Wechsel zu einer anderen Hotspot-Metrik erfolgt über einen Doppelklick auf die Metrikanzeige im Usus Cockpit. Über einen Doppelklick auf einen Hotspot lässt sich entweder zu der dazugehörigen Methode im Quelltext oder dem entsprechenden Paket oder der Klasse in einem der beiden Usus Graph-Ansichten navigieren.

5.1.4 Usus Histogram

Dieses Fenster zeigt die absolute Häufigkeitsverteilung der Ausprägungen einer **Metrik** über alle Projekte an, die Usus betrachtet. Die verwendete **Metrik** wird dabei auf der x-Achse angezeigt, während die Anzahl der Ausprägungen auf der y-Achse dargestellt wird. Die **Verteilung** lässt sich für eine der **Metriken** einsehen, die im Usus Cockpit angezeigt werden. Das in Abbildung 5.4 gezeigte Histogramm-Fenster zeigt die **Verteilung** der **Metrik**

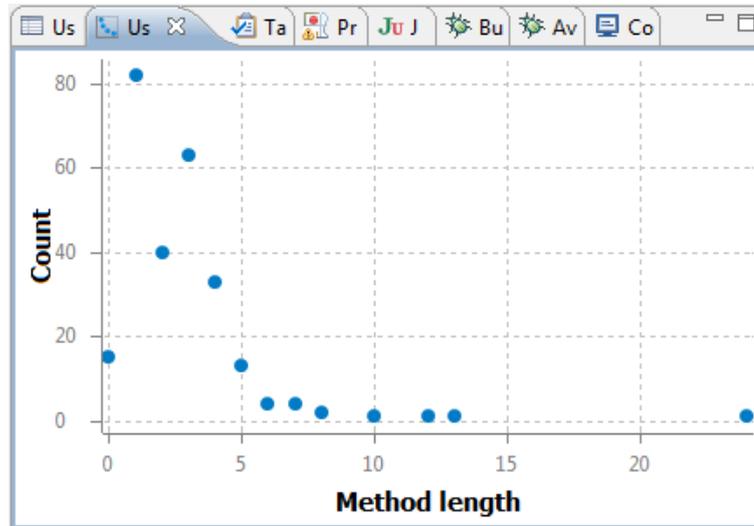


Abbildung 5.4: Usus Histogramm zeigt die **statistische Verteilung** der Werte der ausgewählten **Metrik**

an, die zuvor über einen Einfachklick im Usus Cockpit markiert wurde. Es werden immer nur die Daten für eine **Metrik** angezeigt. Die Ansicht kann vergrößert, verkleinert, skaliert und als Grafik gespeichert werden.

5.1.5 Usus Class Graph & Usus Package Graph

Diese Fensterkombination zeigt die Abhängigkeiten der betrachteten Projekte entweder auf Klassenebene oder auf Paketebene an. Auf Paketebene lassen sich optional nur die Pakete anzeigen, die sich in einem Zyklus von Abhängigkeiten zu anderen Paketen befinden. Auf Klassenebene lassen sich optional nur die Klassen anzeigen, die über eine Abhängigkeit über Paketgrenzen hinweg verfügen. Dabei werden auch nur eben diese paketübergreifenden Abhängigkeiten angezeigt. Das in Abbildung 5.5 dargestellte Fenster besteht eigentlich aus zwei Fenstern. Eine Fenster zeigt den Usus Class Graph, während das andere den Usus Package Graph darstellt. Zwischen den beiden Ansichten kann beliebig gewechselt werden. Die Knoten in den Graphen lassen sich mit der Maus frei positionieren. Zusätzlich lassen sich die Darstellungen der Graphen auch automatisch anordnen.

5.2 Metriken

In den Fenstern Usus Cockpit und Usus Info zeigt das Eclipse-Plugin die Werte verschiedener **Metriken** an. Das Usus Info-Fenster zeigt im Kontext einer Methode neben den **Methodenmetriken** auch die **Klassenmetriken** an. In diesem Abschnitt wird daher zuerst auf die

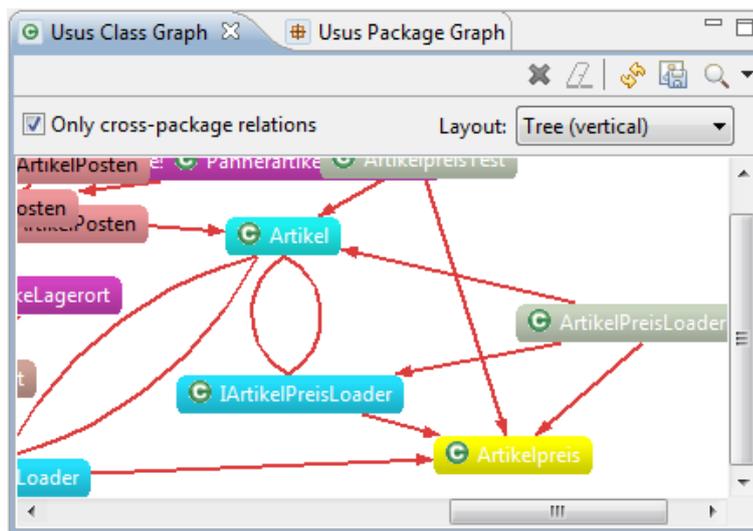


Abbildung 5.5: Usus Graph Schaubilder zeigen die Abhängigkeiten der Klassen oder Pakete voneinander an

Metriken eingegangen, die das Usus-Plugin für Methoden berechnet, bevor die **Klassenmetriken** näher betrachtet werden. Abschließend werden die **Metriken** beschrieben, die im Usus Cockpit angezeigt werden. Diese Beschreibungen sind von Bedeutung, da sämtliche **Metriken** im Rahmen der Usus-Erweiterung für Visual Studio nachimplementiert werden sollen.

5.2.1 Pro Methode

Wenn das Usus Info-Fenster im Kontext einer Methode geöffnet wird, wird die **zyklomatische Komplexität** sowie die **Methodenlänge** angezeigt.

Zyklomatische Komplexität

Die **Metrik Zyklomatische Komplexität** (engl. Cyclomatic Complexity, CC) wurde von Thomas J. McCabe vorgestellt [McC76], um Methoden anhand von linear unabhängigen Ablaufpfaden in Bezug auf Komplexität zu bewerten. McCabe bezieht sich in seinem Artikel auf die Graphentheorie und errechnet die Komplexität eines Ablaufgraphen wie folgt.

$$v(G) = e - n + 2p \quad (5.1)$$

- G = Ablaufgraph
- $v(G)$ = zyklomatische Komplexität von G
- e = Anzahl Kanten im Ablaufgraph G
- n = Anzahl Knoten im Ablaufgraph G
- p = Anzahl Zusammenhangskomponenten in G

Ernest Wallmüller beschreibt die **zyklomatische Komplexität** in seinem Buch [Wal01] auch als Anzahl aller entscheidungstreffenden Stellen in der Methode. Im Falle einer Verkettung

von binären Entscheidungen zu logischen Ausdrücken, zählt jede Entscheidung als eine solche Stelle. Diese einfachere Rechnung ergibt sich als

$$v(G) = 1 + \left(\sum_{b \in Bs(G)} 1 \right) \quad (5.2)$$

wobei $Bs(G)$ die Menge aller binären Entscheidungen im Ablaufgraph G darstellt. Voraussetzung für diese Rechnung ist, dass die Methode nur einen Eingang und nur einen Ausgang besitzt. Der Quellcode in Listing 5.1 soll die Grundlage für eine demonstrative Berechnung der zyklomatischen Komplexität darstellen.

Listing 5.1: Einfache if-Verschachtelung

```

1 public void doSomething() {
2     if (condition1) {
3         if (condition2 || condition3)
4             do1();
5     }
6 }

```

Nach Formel 5.2 ergeben sich drei Entscheidungsstellen, welche durch die drei binären Bedingungen dargestellt werden. Die zyklomatische Komplexität entspricht damit $v(G) = 1 + 3 = 4$. Die Berechnung anhand Formel 5.1 basiert auf der Struktur des Ablaufgraphen,

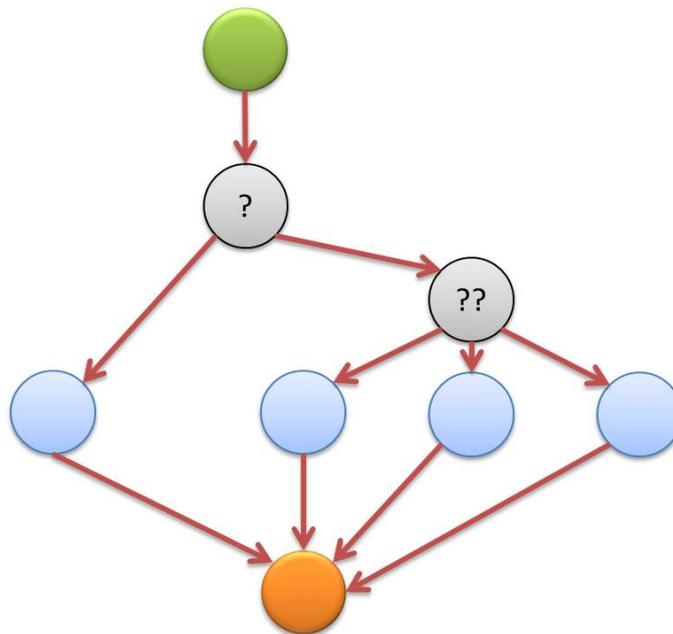


Abbildung 5.6: Ablaufgraph G des Quellcodes 5.1

der in Abbildung 5.6 dargestellt ist. Hier entspricht die zyklomatische Komplexität $v(G) = 10 - 8 + 2 * 1 = 4$. Die Ergebnisse beider Rechnungen sind identisch. Die Eigenschaften einer Methode, die für die Berechnung der zyklomatischen Komplexität erforderlich sind, sind also entweder die Anzahl der binären Entscheidungen oder der vollständige Ablaufgraph.

Methodenlänge

Die Länge einer Methode kann auf unterschiedliche Weise ermittelt werden. Eine Unterscheidung der Möglichkeiten wird von Mark Lorenz und Jeff Kidds in [LK94] vorgenommen. In der vorliegenden Arbeit werden zwei Möglichkeiten wie folgt definiert.

Definition 13 (Anzahl Code-Zeilen) *Die Anzahl der Code-Zeilen (engl. Lines of code) entspricht der tatsächlichen Anzahl an Zeilenumbrüchen ohne leere Zeilen und Kommentarzeilen. Diese Längenangabe ist stark vom Entwicklerstil abhängig und kann sich daher unterschiedlich ausprägen, je nachdem wie beispielsweise eine Parameterliste umgebrochen wird.*

Definition 14 (Anzahl der Anweisungen) *Die Anzahl der Anweisungen (engl. Number of statements) ist eine stabilere Längenangabe. Eine Anweisung ist jeder durch ein Semikolon abgeschlossene Ausdruck, sowie Bedingungs- und Wiederholungsanweisungen.*

Die im Usus Info-Fenster angezeigte **Methodenlänge** entspricht der Anzahl der Anweisungen der Methode. Eine Berechnung kann daher über die Aufsummierung der Semikola und der `if`-, `switch`-, `for`-, `while`- und `try-catch`-Anweisungen erfolgen.

5.2.2 Pro Klasse

Wenn das Usus Info-Fenster im Kontext einer Klasse geöffnet wird, wird die **Klassengröße** sowie die **kumulierte Komponentenabhängigkeit** der Klasse dargestellt.

Klassengröße

Ähnlich der **Methodenlänge** lässt sich auch die Größe einer Klasse auf verschiedene Weise berechnen. Lorenz und Jeff unterscheiden mehrere Möglichkeiten [LK94]. In der vorliegenden Master-Thesis werden im folgenden zwei Varianten der **Klassengröße** definiert.

Definition 15 (Anzahl der Methoden) *Indem die Anzahl der Methoden betrachtet wird, können Klassen erkannt werden, die zu viele oder zu wenige Funktionen erfüllen. Weitere Unterscheidungsmöglichkeiten sind Methoden in Klassen- und Instanzmethoden aufzuteilen oder Methoden anhand der Sichtbarkeit zu klassifizieren. Ein Konstruktor würde sich wie eine statische Methode, also eine Klassenmethode, verhalten.*

Definition 16 (Anzahl der Felder) *Indem die Anzahl der Felder betrachtet wird, können Klassen erkannt werden, die zu viele Informationen verwalten. Auch hier ist eine weitere Unterteilung in Klassen- und Instanzfelder möglich. Die Sichtbarkeit der Felder erlaubt eine weitere Einschränkung.*

Das Usus Info-Fenster zeigt als **Klassengröße** die Anzahl der Instanzmethoden, der Klassenmethoden sowie der Konstruktoren an. Dabei wird die Sichtbarkeit der Methode oder des Konstruktors nicht berücksichtigt. Das Usus-Plugin fokussiert damit auf die Funktion der Klassen und nicht auf die Information und das Wissen einer Klasse, welches in den Feldern liegt. Die **Klassengröße** kann also berechnet werden, wenn alle Methoden und Konstruktoren einer Klasse ermittelt werden können.

Kumulierte Komponentenabhängigkeit

Die Metrik *Kumulierte Komponentenabhängigkeit* (engl. Cumulative Component Dependency, CCD) ist laut Peter Grogono [Gro02] eine Metrik, die für Systeme und Untersysteme ermittelt wird. Dabei werden für jede Klasse (Komponente) in diesem System die Anzahl der Klassen ermittelt, von denen die betrachtete Klasse direkt und indirekt abhängt. Eine Klasse ist immer auch von sich selbst abhängig. Marc Philipp und Nicole Rauch bezeichnen diese Abhängigkeiten in ihrem Artikel [PR10] als reflexiv und transitiv. Anschließend werden die Abhängigkeiten aller betrachteten Klassen aufsummiert und ergeben den CCD-Wert des Systems. In dem Usus Info-Fenster wird dieser Abhängigkeitswert einer betrachteten Klasse als „CCD (of class)“ bezeichnet. Um die Anzahl der Klassen zu bestimmen, von denen eine betrachtete Klasse abhängig ist, ist mindestens der vollständige Abhängigkeitsgraph der Klasse erforderlich. Mit einem Durchmusterungsalgorithmus kann die Erreichbarkeitsmenge (Reach-Menge) der Klasse (Startknoten) in diesem Abhängigkeitsgraph mit linearer Laufzeit $O(V + E)$ ermittelt werden. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest und Clifford Stein beschreiben in ihrem Buch „Introduction to Algorithms“ [CLRS09] die beiden Algorithmen *Breadth First Search* (BFS) und *Depth First Search* (DFS) für diesen Zweck. Abbildung 5.7 zeigt sieben

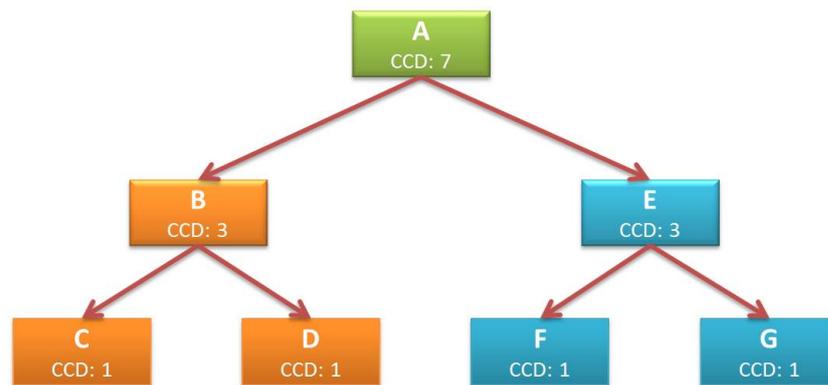


Abbildung 5.7: Abhängigkeitsgraph einer Klasse mit aufsummierten Abhängigkeiten

Klassen, die in einer hierarchischen Abhängigkeitsstruktur stehen. Offensichtlich ist Klasse A von allen anderen Klassen direkt oder indirekt abhängig und hat damit den CCD-Wert sieben. Dies entspricht der Kardinalität der Reach-Menge von A, $\{A, B, C, D, E, F, G\}$, die ermittelt wird, indem BFS oder DFS mit Klasse A als Startknoten im Abhängigkeitsgraph gestartet wird. Dabei werden die durch den Algorithmus markierten Knoten als Ergebnis des Algorithmus behandelt. Bei dem Abhängigkeitsgraph muss es sich nicht um einen Baum handeln. Formel 5.3 zeigt die Berechnungsvorschrift unter Verwendung des DFS-Algorithmus.

$$ccd(c) = |DFS(dG, c)| \quad (5.3)$$

dG = Abhängigkeitsgraph des Systems

c = Klasse im System $\in dG$

$ccd(c)$ = CCD-Wert der Klasse c

Die Berechnung des **CCD**-Werts einer Klasse kann also durchgeführt werden, sobald ein Abhängigkeitsgraph erzeugt werden kann. Um einen solchen Graphen zu erzeugen, müssen die direkten Abhängigkeiten aller Klassen ermittelt werden können. Jede Klasse wird dann einem Knoten zugeordnet und jede Abhängigkeit einer gerichteten Kante. Eine grafische Darstellung (siehe Unterabschnitt 5.1.5) ist dann ebenfalls möglich. Die direkten Abhängigkeiten einer Klasse können bestimmt werden, wenn die Typen aller Felder, Variablen, Methodenparameter, Oberklasse, Interfaces und sämtlicher Methodenaufrufe entfernter Klassen identifiziert werden können. Abhängigkeiten zu Klassen außerhalb des betrachteten Systems, wie beispielsweise `String` oder `Object`, können ignoriert werden.

5.2.3 Projektübergreifend

Neben den **Metriken**, die Usus für Methoden und Klassen berechnet, werden im Usus Cockpit Statistiken zu der Codebasis angezeigt. Dafür werden die ermittelten **Metriken** so bewertet, wie es Marc Philipp und Nicole Rauch in ihrem Artikel [PR10] beschreiben. In diesem Unterabschnitt werden die verschiedenen Statistiken vorgestellt und auf ihre Bewertung eingegangen. Weiterhin werden die Schwellwerte der Statistiken definiert, anhand derer eine Klasse, Methode oder Paket als Hotspot (siehe Unter-Unterabschnitt 5.1.3) eingestuft wird.

Durchschnittliche Komponentenabhängigkeit

Die **Metrik** *Durchschnittliche Komponentenabhängigkeit* (engl. Average Component Dependency, ACD) ist laut Peter Grogono [Gro02] eine **Metrik**, die für Systeme und Untersysteme ermittelt wird. Dabei wird der Mittelwert der in Unterabschnitt 5.2.2 beschriebenen **kumulierten Komponentenabhängigkeiten** der Klassen des Systems wie in Formel 5.4 berechnet, wobei n die Anzahl der Klassen ist.

$$acd = \frac{ccd}{n} \quad (5.4)$$

Da Usus die **CCD**-Werte nicht für Systeme oder Untersysteme ermittelt, sondern die **CCD**-Werte der Klassen bestimmt ohne sie aufzusummieren (siehe Unterabschnitt 5.2.2), kann der **ACD**-Wert anhand einer Menge von Klassen Cs berechnet werden, wie in Formel 5.5 dargestellt. Die Rechnung ist äquivalent zu Formel 5.4.

$$acd(Cs) = \frac{\sum_{c \in Cs} ccd(C)}{|Cs|} \quad (5.5)$$

Peter Grogono beschreibt die Bedeutung des **ACD**-Werts als durchschnittliche Anzahl an Komponenten, die durch eine Änderung einer Komponente betroffen sind und eventuell ebenfalls geändert werden müssen. Der **ACD**-Wert wird im Usus Cockpit als Statistik in Prozent angezeigt. Dazu wird nochmal der Mittelwert über die betrachteten Klassen gebildet, was in Formel 5.6 zu sehen ist und der Bewertungsfunktion von Philipp und Rauch entspricht.

$$acd'(Cs) = \frac{acd(Cs)}{|Cs|} \quad (5.6)$$

Eine Klasse wird von Usus dann als Hotspot betrachtet, wenn ihr **CCD**-Wert über einer Schwelle liegt, die von der Projektgröße abhängig ist. Die Projektgröße wird dabei als

Anzahl der Klassen festgelegt. Anhand der Tooltip-Erklärung im Usus Cockpit liegt diese Schwelle für kleine Projekte bei 15% der Klassenanzahl, während bei großen Projekten 5% der Klassenanzahl verwendet wird. Dafür haben Philipp und Rauch Formel 5.7 mithilfe von Erfahrungswerten definiert, um die Berechnung des CCD-Schwellwert-Faktors L_{ccd} anhand der Menge aller Klassen Cs im System durchführen zu können.

$$L_{ccd}(Cs) = \frac{1,5}{2(\log_5 |Cs|)} \quad (5.7)$$

Um den tatsächlichen Schwellwert für eine Menge von Klassen Cs zu bestimmen, muss der Faktor $L_{ccd}(Cs)$ noch mit der Anzahl der Klassen $|Cs|$ multipliziert werden.

Durchschnittliche Klassengröße

Eine Klasse wird von Usus als Hotspot gesehen, sobald die in Unterabschnitt 5.2.2 beschriebene **Klassengröße** den Schwellwert 12 übersteigt. Das haben Philipp und Rauch festgelegt. Die im Usus Cockpit angezeigte **durchschnittliche Klassengröße** betrachtet nur die Klassen, die bereits als Hotspot markiert wurden. Das liegt an der Bewertungsfunktion $rating_{cs}$ von Philipp und Rauch, die in Formel 5.8 angegeben ist.

$$rating_{cs}(cs) = \begin{cases} \left(\frac{cs}{12}\right) - 1, & \text{wenn } cs > 12 \\ 0, & \text{sonst} \end{cases} \quad (5.8)$$

Die Bewertungsfunktion der **Klassengröße** cs ist direkt von dem Schwellwert 12 abhängig und bewertet alle Klassen, dessen Größe 12 oder weniger beträgt mit 0. Um die **durchschnittliche Klassengröße** acs einer Menge von Klassen Cs zu berechnen, bildet Usus den Mittelwert aller bewerteten Klassengrößen. Dazu wird die Formel 5.9 verwendet.

$$acs(Cs) = \frac{\sum_{c \in Cs} rating_{cs}(c)}{|Cs|} \quad (5.9)$$

Dabei gehen die mit 0 bewerteten **Klassengrößen** ebenfalls in die Durchschnittsberechnung mit ein.

Durchschnittliche zyklomatische Komplexität

Die Berechnung der **durchschnittlichen zyklomatischen Komplexität** findet auf ähnliche Weise statt. Philipp und Rauch haben dafür den Schwellwert 4 gewählt. Damit werden Methoden ignoriert, die vier oder weniger unabhängige Ablaufpfade besitzen oder anders ausgedrückt, weniger als vier verschiedene Entscheidungen treffen. Die Bewertungsfunktion $rating_{cc}$ eines wie in Unterabschnitt 5.2.1 berechneten zyklomatischen Komplexitätswert sieht damit folgendermaßen aus.

$$rating_{cc}(cc) = \begin{cases} \left(\frac{cc}{4}\right) - 1, & \text{wenn } cc > 4 \\ 0, & \text{sonst} \end{cases} \quad (5.10)$$

Anschließend kann der Mittelwert der bewerteten Komplexitäten gebildet werden, indem durch die Anzahl der Methoden dividiert wird.

Durchschnittliche Methodenlänge

Auch die Berechnung der **durchschnittlichen Methodenlänge** findet ähnlich statt. Philipp und Rauch haben den Schwellwert für diese **Metrik** auf 9 festgelegt [PR10]. Methoden mit 9 oder weniger Anweisungen werden damit ignoriert. Die Bewertungsfunktion $rating_{ml}$ sieht dann folgendermaßen aus.

$$rating_{ml}(ml) = \begin{cases} \left(\frac{ml}{9}\right) - 1, & \text{wenn } ml > 9 \\ 0, & \text{sonst} \end{cases} \quad (5.11)$$

Aus den bewerteten Längen kann wieder der Mittelwert berechnet werden, indem durch die Anzahl der Methoden dividiert wird.

Anzahl nicht-statischer öffentlicher Felder

Wenn eine Klasse mindestens ein öffentliches Feld hat, das nicht statisch oder eine Konstante ist, dann betrachtet Usus diese Klasse als einen Hotspot. Dabei wird jede Klasse mit 1 bewertet, die mindestens eines dieser Felder besitzt. Der Schwellwert ist ebenfalls 1. Die Anzahl der betroffenen Klassen wird wie jede andere **Metrik** im Usus Cockpit über die Anzahl aller Klassen gemittelt und somit als Prozent dargestellt.

Pakete mit zyklischen Abhängigkeiten

Für Klassen wurde in Unterabschnitt 5.2.2 ein Abhängigkeitsgraph unabhängig vom Paket ermittelt, in dem sich die betrachtete Klasse befindet. Um **Pakete mit zyklischen Abhängigkeiten** zu identifizieren, müssen alle Klassenknoten eines Pakets zu einem Paketknoten in einem neuen Abhängigkeitsgraph auf Paketebene zusammengefasst werden. Die Kanten zwischen den Klassen werden zu Kanten zwischen den Paketen. Anschließend können alle trivialen Kreise im Abhängigkeitsgraph der Pakete entfernt und Zyklen gesucht werden. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest und Clifford Stein beschreiben in ihrem Buch „Introduction to Algorithms“ [CLRS09] den Algorithmus *Strongly Connected Components* (SCC, starke Zusammenhangskomponenten) mit linearer Laufzeit $O(V + E)$ für diesen Zweck. Mit diesem Algorithmus können alle starken Zusammenhangskomponenten (SCCs) eines Graphen ermittelt werden. Da per Definition von Cormen und co alle Knoten in einer SCC sich gegenseitig erreichen können, handelt es sich um Kreise. Folglich enthalten alle starken Zusammenhankomponenten, die mehr als eine Paket beinhalten, Pakete, die auf einem Kreis im Abhängigkeitsgraph liegen.

$$cyclicPackages(pdG) = \sum_{scc \in SCCs(pdG)} \begin{cases} |scc|, & \text{wenn } |scc| > 1 \\ 0, & \text{sonst} \end{cases} \quad (5.12)$$

- pdG = Paket-Abhängigkeitsgraph
- $SCCs(pdG)$ = Menge aller starken Zusammenhangskomponenten in pdG
- scc = Starke Zusammenhangskomponente als Menge von Paketen
- $cyclicPackages(pdG)$ = Anzahl aller Pakete in allen SCCs in pdG

Formel 5.12 zeigt dabei die Aufsummierung aller Pakete in nicht-trivialen starken Zusammenhangskomponenten eines Abhängigkeitsgraph auf Paketebene. Abschließend kann das Ergebnis, also der *cyclicPackages*-Wert, durch die Anzahl der betrachteten Pakete dividiert werden, um einen durchschnittlichen **Paketzyklus-Wert** zu bestimmen, der im Usus Cockpit angezeigt wird. Alle Pakete, die 2 oder mehr zyklische Abhängigkeiten besitzen werden als Hotspots behandelt.

5.3 Zusammenfassung

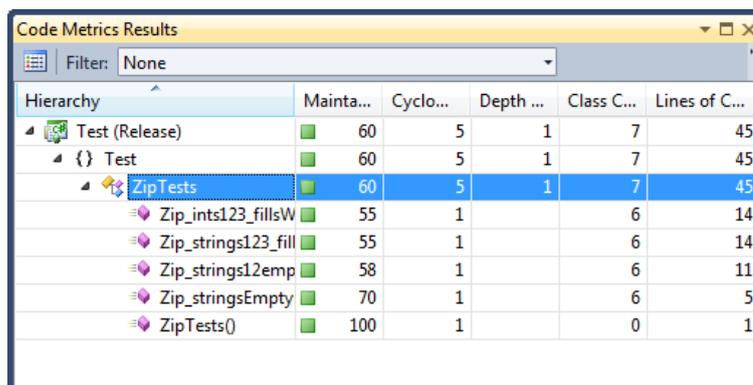
In diesem Kapitel wurde Usus für Java betrachtet. Dabei wurden zuerst die einzelnen Fenster und deren Funktionen beschrieben. Anschließend wurden die **Metriken**, die das Programm berechnet, gewichtet und bewertet, ausführlich dokumentiert. Jede **Metrik**, sei es auf Methoden-, Klassen oder Paketebene, wurde dabei analysiert und deren Berechnung beschrieben. In Bezug auf die in Abschnitt 3.2 definierten Ziele ermöglicht der Funktionsumfang von Usus für Java eine Einsicht in die Codebasis (Ziel 1) und macht problematische Stellen im Code offenbar (Ziel 2). Leider wird die Entwicklung von **Clean Code** (Ziel 3) dabei nicht direkt unterstützt. Auch ermöglicht Usus keine einfache Interpretation der Codequalität (Ziel 4). Es wird zwar eine Gewichtung der berechneten **Metriken** vorgenommen und im Cockpit-Fenster angezeigt, allerdings sind diese Werte nicht sehr transparent und ihre Bedeutung ist nicht aussagekräftig genug. Dennoch sind diese Werte wertvoll und sollten daher auch entsprechende Beachtung finden. Die in dieser Master-Thesis zu entwickelnde Visual Studio-Erweiterung wird diese Werte ebenfalls berechnen, da sie eine Einsicht in den Code ermöglichen und dazu dienen, Probleme im Quellcode zu lokalisieren. Die Beschreibungen in diesem Kapitel dienen dabei als Spezifikation. Sämtliche Funktionen von Usus für Java sollten auch in der Version für das .NET Framework enthalten sein. Es liegt daher nahe, dort auch ähnliche Fenster zu haben. Ein Entwickler, der sich mit einem Usus auskennt, sollte sich dadurch auch schnell an das andere gewöhnen können.

6 Andere Tools

Nachdem in dem vorherigen Kapitel das Eclipse-Plugin Usus vorgestellt wurde, werden in diesem Kapitel einige andere Werkzeuge beschrieben, die ebenfalls eine **statische Code-Analyse** durchführen. Während Usus die direkte Inspiration für das in dieser Master-Thesis zu entwickelnde Tool darstellt, werden in diesem Kapitel einige Visual Studio-Erweiterungen betrachtet. Diese Erweiterungen versuchen bereits, die in Kapitel 3 definierten Anforderungen zu erfüllen und sind dabei auf Projekte spezialisiert, die für das .NET Framework entwickelt werden.

6.1 Visual Studio Metrics

In diesem Abschnitt werden die Möglichkeiten beschrieben, die Visual Studio von Haus aus bietet, um Software zu analysieren. Ab Visual Studio 2010 Premium können zu jeder Solution (Menge von Projekten) verschiedene **Codemetriken** direkt berechnet werden. Die Anzeige der Ergebnisse ist in Abbildung 6.1 dargestellt. Bei den berechneten **Metriken**



Hierarchy	Mainta...	Cyclo...	Depth ...	Class C...	Lines of C...
Test (Release)	60	5	1	7	45
Test	60	5	1	7	45
ZipTests	60	5	1	7	45
Zip_ints123_fillsW	55	1		6	14
Zip_strings123_fill	55	1		6	14
Zip_strings12emp	58	1		6	11
Zip_stringsEmpty	70	1		6	5
ZipTests()	100	1		0	1

Abbildung 6.1: Metrik-Fenster in Visual Studio 2010 Premium

handelt es sich um „Lines of Code“ (eigentlich „Number of Statements“), „Class Coupling“ (direkte Klassenabhängigkeiten), „Cyclomatic Complexity“ und „Maintainability Index“ (Microsofts Indikator für Wartbarkeit). Jede dieser **Metriken** wird auf Methodenebene erhoben und zusammenfassend in der hierarchischen Struktur nach oben (Klasse, Paket, Assembly) propagiert. Zusätzlich wird die **Metrik** „Depth of Inheritance“ (Anzahl der direkten und indirekten Oberklassen ohne Interfaces) auf Klassenebene bestimmt und ebenfalls nach oben aggregiert. Der Wartbarkeitsindex sowie dessen Berechnung wird von Zain Naboulsi in seinem Artikel [Nab11] umfangreich erläutert. Neben der einfachen Darstellung erlaubt Visual Studio das Filtern der Methoden und Klassen anhand der **Metriken** und zeigt neben der Ausprägung des Wartbarkeitsindex eine Ampel an, die nach Microsofts Empfinden die Methode/Klasse als gut wartbar, weniger gut wartbar und nicht so gut wartbar klassifiziert.

In Visual Studio 2010 Ultimate können zudem mehrere grafische Funktionen genutzt werden. So lassen sich zum Beispiel Methoden automatisch als Sequenzdiagramme darstellen, was einen Hinweis auf deren Komplexität und Abhängigkeiten geben kann. Zusätzlich kann der Abhängigkeitsgraph einer Solution mithilfe des Architecture Explorer visualisiert und so zyklische Abhängigkeiten von Klassen und Paketen schnell gefunden werden. Abbildung 6.2 zeigt das Ergebnis als Graph an. Das Ergebnis kann auch als Matrix dargestellt werden kann. Leider sind diese Funktion zur **Metrik**-Berechnung und zur Visualisierung in

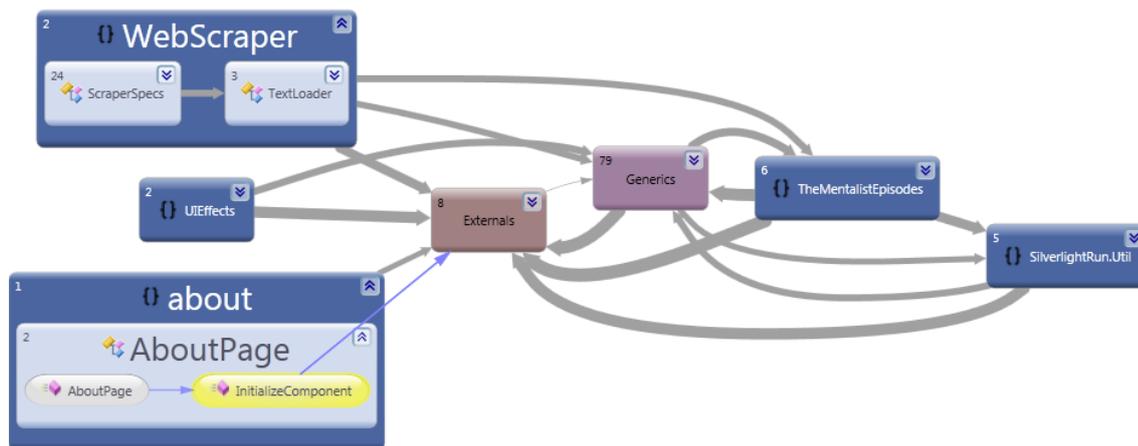


Abbildung 6.2: Architekturdiagramm in Visual Studio 2010 Ultimate auf Basis von Paketen, aufklappbar bis auf Methodenebene

der am weitesten verbreiten Professional-Version von Visual Studio nicht verfügbar. Zwei kostenlose Programme, die die gleichen **Codemetriken** berechnen, werden in den folgenden Unterabschnitten vorgestellt.

6.1.1 Visual Studio Code Metrics Power Tool

Das Kommandozeilenwerkzeug Visual Studio Code Metrics Power Tool¹ wird kostenfrei von Microsoft zum Download angeboten. Es erlaubt die Berechnung der gleichen **Metriken** wie die in Visual Studio 2010 Premium integrierte Funktionalität und erzeugt eine XML-Datei. Diese Datei enthält die **Metriken** für Projekte, Namespaces, Typen und Methoden in hierarchischer Form.

6.1.2 Code Metrics Viewer

Die Erweiterung Code Metrics Viewer² für Visual Studio 2010 wird von Matthias Friedrich zum kostenlosen Download angeboten. Das Tool nutzt das Kommandozeilenwerkzeug Visual Studio Code Metrics Power Tool um **Metriken** direkt in der Entwicklungsumgebung grafisch anzeigen zu können. In einem Visual Studio-Fenster, das dem aus Abbildung 6.1 nachempfunden ist, wird der Inhalt der durch das Power Tool erzeugten XML-Datei visualisiert. Es lassen sich ebenfalls verschiedene Filter einstellen. Der Hauptunterschied zu

¹Download: „Visual Studio Code Metrics PowerTool 10.0“ <http://www.microsoft.com/download/en/details.aspx?id=9422>

²Download: „Code Metrics Viewer extension“ <http://visualstudiogallery.msdn.microsoft.com/9f35524b-a784-4dbc-bd7b-6babb7a5a3b3>

der integrierten Funktionalität besteht darin, dass die Ampel zu jeder **Metrik** angezeigt wird.

6.2 NDepend

Für die grafische Darstellung des Abhängigkeitsgraphen kann das kommerzielle Werkzeug NDepend¹ (299€ für eine Einzelplatzlizenz) genutzt werden. Dieses Tool ist neben der umfangreichen Darstellung von Zusammenhängen und Abhängigkeiten in der Lage, viele verschiedene **Metriken** auf Anwendungs-, Projekt-, Namespace-, Klassen- und Methodenebene zu berechnen. Die Abhängigkeiten können, wie im Architecture Explorer in Visual Studio Ultimate, in Form einer Matrix oder in Form eines Diagramms veranschaulicht werden. Dafür analysiert NDepend genau wie das Code Metrics Power Tool die pro Projekt von dem Compiler erzeugten **Assembly**-Dateien (exe oder dll). Das Programm ist neben .NET auch für Java und C++ erhältlich und kann als eigenständige Anwendung oder als Visual Studio-Addin verwendet werden.

6.2.1 Code Query Language

Abhängig von der **Metrik**- und Abhängigkeitsberechnung kann NDepend Warnungen und Hinweise anzeigen, die den Programmierer rechtzeitig auf schwierige Stellen im Code hinweisen und es ihm erlauben, dort gezielt einzugreifen. Es lassen sich auch eigene Warnungen und Berichte mithilfe der deklarativen Sprache CQL (Code Query Language) erstellen und ausführen.

6.2.2 Abstraktheit und Instabilität

Eine sehr interessante **Metrik** im NDepend **Metrik**-Portfolio ist *Abstraktheit und Instabilität*. Diese Kombination zweier **Metriken** geht auf einen Artikel von Robert C. Martin [Mar94] zurück. In diesem Artikel beschreibt Martin verschiedene **Metriken** um Abhängigkeiten zwischen Kategorien zu messen. Eine Kategorie definiert er als Gruppe zusammenhängender Klassen, die gemeinsam benutzt werden. Diese Definition kann somit auf Pakete (Namespaces) oder auf **Assemblies** angewendet werden. Als erste **Metrik** stellt Martin *Afferent Couplings* (hinführende Kupplungen $Ca(k)$) einer Kategorie k als Anzahl der Klassen vor, die nicht in der Kategorie sind und eine Abhängigkeit von Klassen in der Kategorie haben. Daraus ergibt sich Formel 6.1.

$$Ca(k) = \sum_{c_o \in Cs \setminus Cs(k)} \begin{cases} 1, & \text{wenn } \exists (c_o, c_i) \in dG \mid c_i \in Cs(k) \\ 0, & \text{sonst} \end{cases} \quad (6.1)$$

- Cs = Menge aller Klassen im System
- $Cs(k)$ = Menge aller Klassen in der Kategorie k
- dG = Abhängigkeitsgraph des Systems

¹Mehr Informationen: „NDepend tutorial with explanations and screenshots“ <http://www.ndepend.com/GettingStarted.aspx#Tuto>

Martin bezieht sich auf die Abhängigkeiten zwischen Kategorien. In Formel 6.1 und den folgenden Formeln wird der bereits aus Unterabschnitt 5.2.2 bekannte Abhängigkeitsgraph dG auf Klassenebene verwendet um die Abhängigkeiten abzubilden. Die Grenze einer Kategorie k wird in Bezug auf die Abhängigkeiten der Klassen durch die Funktion $Cs(k)$ bestimmt. Während Formel 6.1 alle Abhängigkeiten in Richtung der betrachteten Kategorie beschreibt, definiert Formel 6.2 alle Abhängigkeiten aus der Richtung der betrachteten Kategorie. Die **Metrik Efferent Couplings** (wegführende Kupplungen $Ce(k)$) bezeichnet die Anzahl der Klassen innerhalb einer Kategorie k , die Abhängigkeiten zu Klassen außerhalb der Kategorie besitzen.

$$Ce(k) = \sum_{c_i \in Cs(k)} \begin{cases} 1, & \text{wenn } \exists (c_i, c_o) \in dG \mid c_o \in Cs \setminus Cs(k) \\ 0, & \text{sonst} \end{cases} \quad (6.2)$$

Die Formeln 6.1 und 6.2 besitzen die gleiche Struktur und unterscheiden sich nur durch die Richtung der gesuchten Kante im Abhängigkeitsgraph dG und der Menge der Klassen, die gezählt werden soll.

Mit der Anzahl der hinführenden und wegführenden Abhängigkeiten kann jetzt die **Metrik Instability** (Instabilität $I(k)$) einer Kategorie k wie in Formel 6.3 bestimmt werden. Martin beschreibt diese **Metrik** als Maß dafür, wie die betrachtete Kategorie beeinträchtigt ist, wenn sich das restliche System ändert.

$$I(k) = \frac{Ce(k)}{Ca(k) + Ce(k)} \quad (6.3)$$

Eine Instabilität von 1.0 bedeutet maximale Instabilität und wird erreicht, wenn nur wegführende Abhängigkeiten vorhanden sind, also die Klassen innerhalb der Kategorie von Klassen außerhalb abhängig sind. Im Falle einer Änderung im restlichen System ist diese Kategorie sehr wahrscheinlich ebenfalls betroffen. Eine Instabilität von 0.0 bedeutet maximale Stabilität, da keine wegführenden Abhängigkeiten vorhanden sind und Abhängigkeiten nur von Klassen außerhalb der Kategorie zu Klassen in der Kategorie bestehen. Eine Änderung des restlichen Systems hätte also keinen Einfluss auf die betrachtete Kategorie. Instabilität kann nur bestimmt werden, wenn mindestens ein Abhängigkeit (wegführend oder hinführend) vorhanden ist.

Robert C. Martin erläutert in seinem Artikel zudem den Zusammenhang von Instabilität und Abstraktheit. Dafür schreibt er, dass Abhängigkeiten von instabilen Kategorien zu vermeiden sind. Da Kategorien mit abstrakten Klassen und Schnittstellen hinführende Abhängigkeit einer externen Klasse motivieren, empfiehlt er keine abstrakten Typen innerhalb einer instabilen Kategorie zu positionieren. Um den Grad der Abstraktheit $A(k)$ einer Kategorie k zu bestimmen, hat Martin die **Metrik Abstractness** als Verhältnis zwischen abstrakten Klassen und Schnittstellen zu konkreten Klassen beschrieben. Dieses Verhältnis kann mit Formel 6.4 berechnet werden.

$$A(k) = \frac{|aCs(k)|}{|Cs(kp)|} \quad (6.4)$$

$aCs(k)$ = Menge der abstrakten Klassen und Schnittstellen in der Kategorie k

Genau wie bei der Instabilität liegt auch die Ausprägung der Abstraktheit zwischen 0.0 und 1.0. Eine Abstraktheit von 0.0 entspricht einer konkreten Kategorie ohne abstrakte Typen, während eine Abstraktheit von 1.0 für eine Kategorie mit ausschließlich abstrakten Typen steht.

Die Abstraktheit und die Instabilität können kombiniert werden. NDepend greift die Idee von Martin auf und trägt jede *Assembly* der betrachteten Projektumgebung in einem Diagramm ähnlich Abbildung 6.3 entsprechend ein. Robert C. Martin beschreibt eine

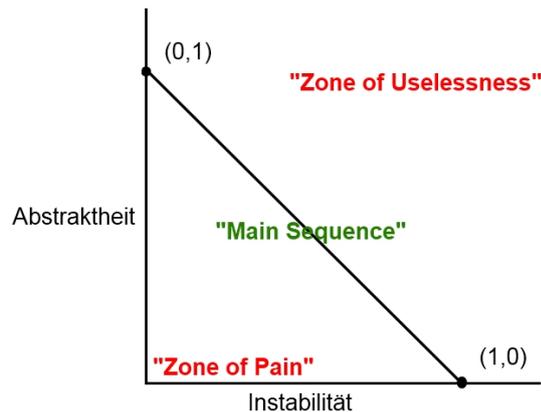


Abbildung 6.3: Schematische Darstellung des „Abstraktheit und Instabilität“-Diagramm in NDepend

Kategorie, die in der Hauptsequenz liegt, als ausgeglichen, da sie zum Teil erweiterbar und zum Teil stabil ist. Laut Martin sind die beiden Endpunkte der Hauptsequenz die optimalen Punkte. Nach seiner Erfahrung können nur ungefähr die Hälfte der Kategorien in einem System diese optimalen Positionen einnehmen. Die andere Hälfte sollte möglichst nah an der Hauptsequenz liegen, was durch eine weitere *Metrik* in Form einer Abstandsberechnung festgehalten werden könnte.

6.3 CCD-Addin

In Kapitel 3 wurde im Rahmen der Anforderungsanalyse der Anwendungsfall *Clean Code-Hilfe bekommen* identifiziert. Das Visual Studio-Addin CcdAddIn¹ von Alexander Zeitler versucht eine solche Hilfestellung in passiver Form zu geben. In Unterabschnitt 2.1.1 wurde die Idee von *Clean Code* bereits erläutert und auf das sieben-Grade-System von Ralf Westphal und Stefan Lieser [WL11] eingegangen. Das Programm integriert sich in die IDE und zeigt zu jedem Grad die entsprechenden Prinzipien und Praktiken an, sodass der Entwickler immer an seinen aktuellen Grad und an die damit verbundenen Regeln erinnert wird. Jede Praktik und jedes Prinzip ist anklickbar, sodass das Addin weitere Informationen anzeigen kann. Leider zeigt das Addin die Informationen nur an und bietet sonst keine weitere Hilfestellung.

¹Download: „CcdAddIn is a visual Studio Add-In that displays the CCD values according to your current CCD Grade“ <https://github.com/AlexZeitler/CcdAddIn>

6.3.1 Prinzipien

In diesem Unterabschnitt werden die einzelnen Grade vorgestellt. Da der *schwarze* Grad lediglich einer Interessenbekundung entspricht und der *weiße* Grad das komplette Spektrum abdeckt, werden im Folgenden nur die verbleibenden fünf Grade erwähnt. Die folgenden Prinzipien wurden direkt von Westphal und Lieser [WL11] übernommen.

Rot „Don´t Repeat Yourself“ (DRY), „Keep it simple, stupid“ (KISS), Vorsicht vor Optimierungen!, „Favour Composition over Inheritance“ (FCoI)

Orange „Single Level of Abstraction“ (SLA), „Single Responsibility Principle“ (SRP), „Separation of Concerns“ (SoC), Source Code Konventionen

Gelb „Interface Segregation Principle“ (ISP), „Dependency Inversion Principle“ (DIP), „Liskov Substitution Principle“ (LSP), „Principle of Least Astonishment“ (PLA), „Information Hiding Principle“ (IHP)

Grün „Open Closed Principle“ (OCP), „Tell, don´t ask“ (TDA), „Law of Demeter“ (LD)

Blau Entwurf und Implementation überlappen nicht, Implementation spiegelt Entwurf, „You Ain´t Gonna Need It“ (YAGNI)

6.3.2 Praktiken

Neben den Prinzipien existieren zu jedem Grad auf Praktiken, die ebenfalls direkt von Westphal und Lieser [WL11] übernommen wurden.

Rot die Pfadfinderregel beachten, „Root Cause Analysis“, ein Versionskontrollsystem einsetzen, einfache Refaktorisierungsmuster anwenden, Täglich reflektieren

Orange Issue Tracking, Automatisierte Integrationstests, Reviews, Lesen, Lesen, Lesen

Gelb Automatisierte Unit Tests, Mockups (Testattrappen), Code Coverage-Analyse, Teilnahme an Fachveranstaltungen, Komplexe Refaktorisierungen

Grün Continuous Integration, [Statische Code-Analyse \(Metriken\)](#), „Inversion of Control“ (IOC) Container, Erfahrung weitergeben, Messen von Fehlern

Blau Continuous Delivery, Iterative Entwicklung, Komponentenorientierung, Test first

6.4 Zusammenfassung

In diesem Kapitel wurden einige Möglichkeiten beschrieben, wie [Metriken](#) für .NET-Quellcode berechnet werden können. Doch weder die integrierten Funktionen in Visual Studio, noch die darauf aufbauenden Erweiterungen können die [Metriken](#) berechnen, gewichten und bewerten, die Usus für Java bestimmen kann. NDepend kann dies auch nicht aber NDepend erlaubt es eigene Regeln zu definieren. Mit diesen eigenen Regeln könnten wahrscheinlich einige aber sicher nicht alle Cockpit-[Metriken](#) und Hotspot-Funktionen von Usus nachgebaut werden. Dafür bietet NDepend jede Menge andere interessante [Metriken](#) und Funktionen, die für die in dieser Master-Thesis zu entwickelnde Erweiterung nicht relevant sind. Zuletzt wurde auch eine Erweiterung betrachtet, die einfach nur die [Clean](#)

Code-Grade von Ralf Westphal und Stefan Lieser anzeigt. Auch wenn dies Clean Code nur indirekt fördert, kann die zu entwickelnde Erweiterung diese Idee aufgreifen. In Bezug auf die in Abschnitt 3.2 definierten Ziele, kann keines der betrachteten Werkzeuge eine Interpretation der Softwarequalität (Ziel 4) oder aktive Clean Code-Unterstützung (Ziel 3) bieten.

7 Evaluierung der Technologien

Nachdem in Kapitel 3 die Anforderungen an die zu entwickelnde Visual Studio-Erweiterung beschrieben wurden, werden in diesem Kapitel verschiedene Technologien in Betracht gezogen, um ein Werkzeug zu entwickeln, das einen ähnlichen Funktionsumfang wie das in Kapitel 5 beschriebene Usus-Plugin besitzt. In diesem Kapitel liegt der Schwerpunkt auf der **statischen Code-Analyse**, mit dessen Hilfe die **Metriken** aus Unterabschnitt 5.2 berechnet werden sollen. Zu diesem Zweck werden Möglichkeiten untersucht um Quellcode zu analysieren und um die Rohdaten zu ermitteln, die für die Berechnung der vorgestellten **Metriken** benötigt werden. Diese **Analyse** wird auch von Artur Wagner in seiner Ausarbeitung [Wag] mit der Analyse, die ein Compiler durchführt, verglichen. Allerdings ist es möglich, eine **statische Code-Analyse** auch nach dem Kompilieren durchzuführen, indem das Kompilat betrachtet wird. Dies ist im Fall von Java und .NET möglich, da es sich bei dem erzeugten Format um eine Zwischensprache handelt, die erst unmittelbar vor dem Ausführen in Maschinensprache übersetzt wird. Die Technologien, die in diesem Kapitel behandelt werden, nutzen entweder den Quellcode oder das Compiler-Ergebnis um eine **statische Code-Analyse** durchzuführen. Um die Technologien anhand ihrer Eignung zu evaluieren, wird die Bewertung nach verschiedenen Kriterien vorgenommen. Die Kriterien sind nach Wichtigkeit absteigend sortiert.

1. Können alle Informationen gesammelt werden, die benötigt werden um die von Usus berechneten **Metriken** zu berechnen?
2. Ist die Technologie allgemein verfügbar und kann als Bestandteil einer Anwendung veröffentlicht werden?
3. Ist die Technologie kostenfrei verwendbar?
4. Ist die Technologie einfach zu verwenden und kann leicht eingesetzt werden?
5. Unterstützt die Technologie alle Versionen der .NET-Laufzeitumgebungen unter Windows?
6. Ist die Technologie unabhängig von anderen Laufzeitumgebungen und anderen externen Komponenten?
7. Kann die Technologie ein unvollständiges System verarbeiten, das Klassen und Pakete verwendet, die nicht im betrachteten System definiert sind?
8. Ist die Technologie in der Lage ein System zu analysieren, das nicht nur mit C# entwickelt wurde sondern auch mit VB.NET?
9. Kann die Technologie mit dem kompilierten Quellcode, also der Assembly, arbeiten?
10. Kann die Technologie mit dem unkompilierten Quellcode arbeiten?

Die Kriterien sind in Form von Fragen so formuliert, dass eine bejahende Antwort als erstrebenswert gilt. Anhand dieser Kriterien können verschiedene Technologien, die in den folgenden Abschnitten beschrieben werden, evaluiert werden. Das Ergebnis wird abschließend im letzten Abschnitt dieses Kapitels in Form einer Zusammenfassung vorgestellt. Die Auswahl der betrachteten Technologien wurde abhängig vom subjektiven Bekanntheitsgrad eingeschränkt und ist nicht vollständig.

7.1 FxCop

Jason Kresowaty beschreibt FxCop¹ in seiner Ausarbeitung [Kre08] als Werkzeug zur statischen Code-Analyse für Assemblies, die mit C#, VB.NET und allen anderen .NET-Sprachen entwickelt wurden. Dabei werden Regeln ausgeführt, die nach Problemen im Sinne von Verletzungen von Konventionen und Richtlinien suchen. Diese Analyse ist möglich, da der Compiler aus dem C#- oder VB.NET-Quellcode Befehle in der *Common Intermediate Language* (CIL) erzeugt. Im ECMA-335-Standard beschreibt Microsoft diese Zwischensprache als Bestandteil der Common Language Infrastructure ausführlich [Mic12a].

7.1.1 Umgebung

Die FxCop-Technologie besteht aus zwei Teilen. Der erste Teil ist die FxCop-Anwendung und der zweite Teil sind die Regeln, die genutzt werden um die statische Analyse zu nutzen. Microsoft veröffentlicht mit dem Programm eine Menge von Regeln, um Assemblies anhand den von Krzysztof Cwalina und Brad Abrams veröffentlichten Konventionen und Richtlinien [CA08] zu untersuchen. Eine Regel, die eine Berechnung von Metriken zur späteren Verwendung durchführt, ist nicht vorhanden. Es lassen sich allerdings eigene Regeln definieren, die FxCop genauso ausführen kann, und mit der das Programm beliebig erweitert werden kann. Eine solche eigene Regel könnte alle notwendigen Informationen sammeln, um ein Programm mit einem Usus-ähnlichen Funktionsumfang zu entwickeln. Kresowaty beschreibt in seiner Ausarbeitung [Kre08] wie so eine Regel erstellt werden kann. Regeln lassen sich auch automatisiert ausführen, wenn eine Analyse mit der grafischen Oberfläche, die in Abbildung 7.1 dargestellt ist, nicht sinnvoll ist. Abbildung 7.2 zeigt die Kommandozeilenversion des FxCop Runners.

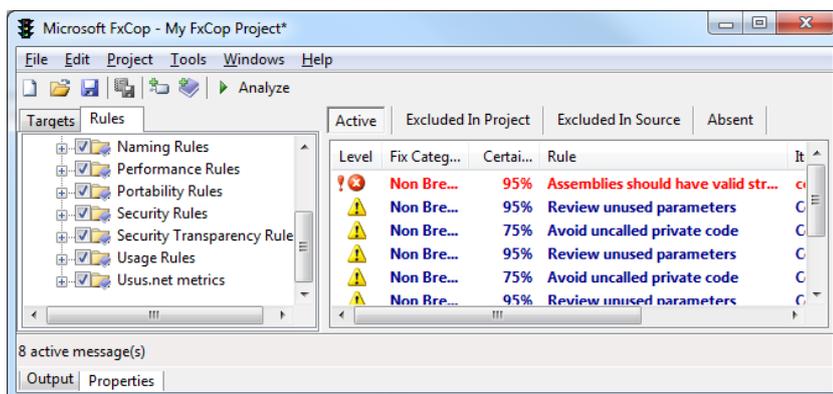


Abbildung 7.1: FxCop Runner

¹Download: „FxCop 10.0“ <http://www.microsoft.com/en-us/download/details.aspx?id=6544>

```
Administrator: C:\Windows\system32\cmd.exe
NetRule.dll" /rule:"D:\manuel\Git\GitHub\MTSS12\source\TechEval_FxCop\UsusNetRule\bin\Debug\UsusNetRule.dll" /console
Microsoft (R) FxCop Command-Line Tool, Version 10.0 (10.0.30319.1) X86
Copyright (C) Microsoft Corporation. All Rights Reserved.

Loaded ususnetrule.dll...
Loaded UsusNetRule.dll...
Initializing Introspection engine...
Analyzing...
Analysis Complete.

NOTE: One or more referenced assemblies could not be found. Use the '/directory'
or '/reference' switch to specify additional assembly reference search paths.

Project : warning : CA0060 : The indirectly-referenced assembly 'Microsoft.VisualStudio.CodeAnalysis, Version=10.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a' could not be found. This assembly is not required for analysis, however, analysis results could be incomplete. This assembly was referenced by: D:\manuel\Git\GitHub\MTSS12\source\TechEval_FxCop\UsusNetRule\bin\Debug\FxCopSdk.dll.
Done:00:00:00.9540000

C:\Users\mannajoks>
```

Abbildung 7.2: FxCop Runner in der Kommandozeile

Eine FxCop-Regel entspricht einer Klassenbibliothek, die eine Regelklasse und eine Konfigurationsdatei enthält. Während FxCop den Code analysiert, erzeugt es Instanzen der Regelklasse und nutzt diese um in der Codebasis Problemfälle zu finden. Problemfälle können dann von der Regelklasse in Problemlisten eingetragen werden, die FxCop entweder grafisch anzeigt oder in eine Report-Datei exportiert. Zusätzlich können Problemfälle als kritisch gekennzeichnet und mit verschiedenen Texten und Beschreibungen versehen werden um eine möglichst detaillierte Beschreibung der Schwierigkeit zu ermöglichen. FxCop ist also auf das Finden von Problemen und nicht auf objektive Datensammlung spezialisiert. Um eine *Assembly* möglichst schnell analysieren zu können, erzeugt FxCop mehrere Instanzen einer Regel und führt diese in mehreren Threads parallel aus. Jason Kresowaty betont in seiner Ausarbeitung [Kre08], dass sich die Regeln nicht Thread-sicher verhalten. Damit mit FxCop weiterverwendbare Daten gesammelt werden können, müssen diese auf eine geeignete Weise exportiert werden. Entweder können diese Daten in Form von Zeichenketten als Probleme verpackt oder direkt an ein Ziel gespeichert werden. Diese beiden Lösungen sind umständlich und nicht intuitiv und werden von der Nebenläufigkeit von FxCop noch erschwert.

7.1.2 API

Regeln für FxCop können mit C[#] in Form einer Klassenbibliothek entwickelt werden, wie es Jason Kresowaty in seiner Ausarbeitung [Kre08] vorstellt. Die dadurch erzeugte *Dynamic Link Library* (dll-Datei) kann dann von FxCop verwendet werden. Dazu muss die DLL über eine spezielle Metadatei als eingebettete Ressource verfügen, die die Regel dem Runner bekanntmacht. Die Regel selbst wird in einer Klasse implementiert, die von einer abstrakten Oberklasse für Regeln abgeleitet ist und somit Analysemethoden überladen kann. Mit diesen Methoden kann das Objektmodell der zu analysierenden *Assembly* untersucht werden. Alle Basisklassen und Klassen, die das Objektmodell repräsentieren befinden sich im Namespace `Microsoft.FxCop.Sdk`. Das *Application Programming Interface* (API) um FxCop-Regeln zu programmieren enthält Teile, die dem in das .NET Framework integrierten `System.Reflection`-API ähnlich sind. Beide APIs enthalten Klassen um das Objektmodell von .NET-Sprachen zu repräsentieren.

Um Operationen von einer Datenstruktur zu trennen, beschreiben Erich Gamma, Richard Helm, Ralph E. Johnson und John Vlissides das Besuchermuster (Visitor Pattern) in [GHJV95]. Mit dem Besuchermuster ermöglicht die FxCop-API die gleiche Datenstruktur (Objektmodell) mit vielen Regeln (Besuchern) zu analysieren. In der Regelklasse, die von `BaseIntrospectionRule` abgeleitet ist, können oberflächliche Besuchermethoden überladen werden. Diese oberflächlichen Methoden bestimmen, an welcher Stelle im Objektmodell eine weitere Überprüfung vorgenommen werden soll. Zum Beispiel wird die `Check(Member)`-Methode in Listing 7.1 für jedes Property oder Methode in jeder Klasse aufgerufen. Die `Check(TypeNode)`-Methode wird zusätzlich für jede Klasse oder Interface ausgeführt.

Listing 7.1: Beispiele der FxCop-API

```

1 ProblemCollection Check(Member member) {...}
2 ProblemCollection Check(TypeNode type) {...}
3 void VisitMemberBinding(MemberBinding memberBinding) {...}
4 void VisitBranch(Branch branch) {...}
5 void VisitAssignmentStatement(AssignmentStatement assignment) {...}

```

Nachdem sich die Regel mit einer dieser oberflächlichen `Check`-Methoden in den Kontrollfluss der *Analyse* integriert, kann eine tiefer gehende Analyse erfolgen. Dazu wird beispielsweise ein `BinaryReadOnlyVisitor`-Besucher erstellt, der die aktuelle Stelle im Objektmodell detaillierter besuchen kann. In der Klasse, die von `BinaryReadOnlyVisitor` erbt, können `Visit`-Methoden überladen werden. Listing 7.1 zeigt einige dieser Methoden. Beispielsweise kann in der `VisitMemberBinding`-Methode jeder Methodenaufruf und in der `VisitAssignmentStatement`-Methode jede Zuweisung analysiert werden. Auf diese Weise und in Verbindung mit sämtlichen Parameter- und Rückgabetypen kann aus dem Objektmodell der für die zu berechnenden *Metriken* erforderliche Abhängigkeitsgraph erstellt werden. Die Anzahl der Anweisungen pro Methode kann entweder über die vom Compiler erzeugte Zwischensprache (engl. Common Intermediate Language, CIL) oder über die `Statement`-Objekte von FxCop erfolgen. Während die CIL-Anweisungen die C[#]-Anweisungen sehr viel granularer abbilden, entsprechen die FxCop-Anweisungen zusammengefassten CIL-Anweisungen. Eine Entsprechung der C[#]-Anweisungen konnte ohne detaillierte Betrachtung der konkreten `Statement`-Typen nicht ermittelt werden. Die Anzahl der Zeilen kann dafür auf eine sehr einfache Weise bestimmt werden, da jedes `Statement` über einen `SourceContext` und dort über Start- und Endzeilennummern verfügt. Anhand der Liste aller Anweisungen kann die erste Zeile als kleinste Startzeile und die letzte als größte Endzeile gesucht und mittels der Differenz die *Methodenlänge* berechnet werden. Die *zyklomatische Komplexität* kann bestimmt werden, da jede Verzweigung im Code von der `VisitBranch`-Methode besucht wird.

7.2 Common Compiler Infrastructure

Die *Common Compiler Infrastructure* (CCI) ist eine Sammlung von Bibliotheken, die Compiler-ähnliche Funktionen bereitstellen und von Microsoft Research entwickelt wurde [BFV12]. Auf Codeplex lässt sich die Komponente *CCI Metadata*¹ finden. Guy Smith beschreibt sie als Obermenge von `System.Reflection`, `System.Reflection.Emit` und

¹Download und mehr Informationen: „CCI: Metadata API“ <http://ccimetadata.codeplex.com/>

System.CodeDom, also den Mechanismen des .NET Framework, die zur Laufzeit Reflexion und Code-Erzeugung möglich machen [Smi09a]. Smith beschreibt CCI Metadata als Werkzeug, das mit den vom Compiler erzeugten CIL-Anweisungen arbeitet. Um einfacher mit Methoden zu arbeiten, stellt Smith mit CCI Code and AST Components¹ eine weitere Komponente vor [Smi10]. CCI Code and AST Components erleichtert das Arbeiten mit Methoden, da es von den CIL-Anweisungen abstrahiert und Quellcode-ähnliche Bäume nutzt. Beide Komponenten sind unter der Microsoft Public License veröffentlicht. Die CCI wird in einer ähnlichen Variante unter anderem von dem Code Metrics Power Tool aus Unterabschnitt 6.1.1 und FxCop aus Abschnitt 7.1 genutzt. Die Assembly Microsoft.Cci.dll, die mit diesen Tools ausgeliefert wird, besitzt viele Funktionen, die nur intern oder von erlaubten Assemblies aufrufbar sind. Funktionen, die beispielsweise konkrete Methodenmetriken berechnen, sind nur von Programmen aufrufbar, die in der CCI-Assembly aufgelistet sind, wie in Abbildung 7.3 dargestellt.

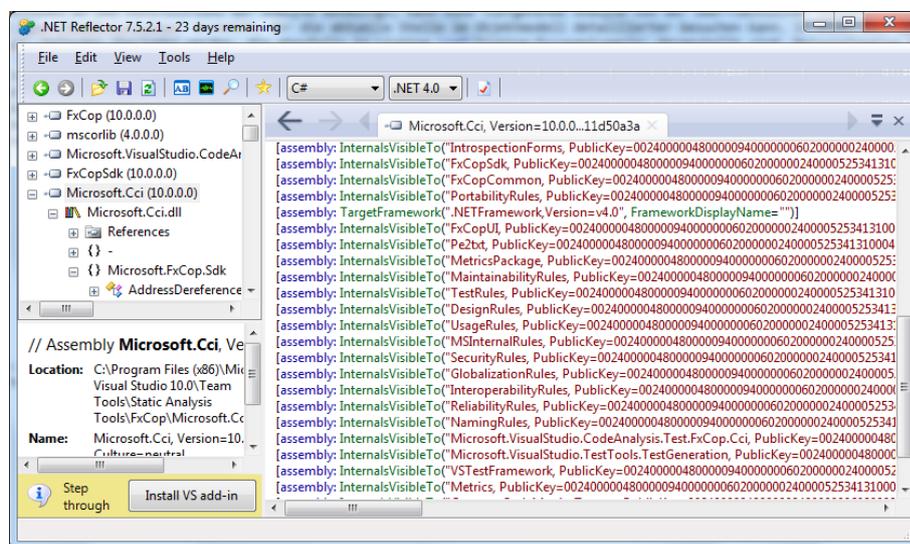


Abbildung 7.3: InternalsVisibleTo-Attribute der Assembly Microsoft.Cci.dll

7.2.1 CCI Metadata

Im Gegensatz zu der FxCop-API kann eine .NET-Assembly mit CCI direkt untersucht werden, ohne dass die Analyse in Form einer einschränkenden Regel parallelisiert erfolgt. CCI überlässt dem Verwender der Bibliothek ob und wie die Untersuchung stattfinden soll. Mit einem `PeReader.DefaultHost` können Assemblies eingelesen werden, die von CCI als PE-Dateien bezeichnet werden. PE-Dateien werden in der vorliegenden Master-Thesis wie folgt definiert.

Definition 17 (Portable Executable) Eine Portable Executable-Datei (PE-Datei), also eine (wörtlich übersetzt) „transportierbare“ und ausführbare Datei, ist das Ergebnis einer Kompilierung eines Visual Studio .NET-Projekts. Die Datei ist entweder eine Bibliothek (dll-Datei) oder eine direkt ausführbare Datei (exe-Datei).

¹Download und mehr Informationen: „CCI: Code Model and AST API“ <http://cciaast.codeplex.com/>

Zusätzlich kann die vom Compiler erzeugte `pdb-Datei` mit einem `PdbReader` importiert werden. Eine `pdb-Datei` wird hier wie folgt definiert.

Definition 18 (Program Database) *Die Program Database-Datei (pdb-Datei) enthält die Datei- und Zeilenzuordnung der kompilierten Funktionen in einer PE-Datei. Laut Microsoft [Mic11] enthält sie generell Debug- und Projektinformationen.*

Mit der `IAssembly.GetAllTypes()`-Methode können alle Typen einer `Assembly` ermittelt werden. Alle Methoden und Properties eines Typen, können mit der Eigenschaft `INamedTypeDefinition.Methods` bestimmt werden. Der Inhalt von Methoden und Properties kann in Form von CIL-Anweisungen genauer betrachtet werden. Diese Anweisungen lassen sich mit der Eigenschaft `Operations` ermitteln, die auf dem `IMethodBody`-Interface definiert ist. So lassen sich beispielsweise alle lokalen Variablen und Methodenaufrufe lokalisieren, um Abhängigkeiten der Methode zu anderen Typen zu finden. Die CIL-Anweisung eines Methodenaufrufs enthält immer auch den Typ, der das Aufrufziel darstellt. Zusammen mit Attributen, Parametern und Rückgabewerten kann die Abhängigkeitsmenge eines Typen ermittelt werden. Die **zyklomatische Komplexität** einer Methode kann ebenfalls über die CIL-Anweisungen bestimmt werden, was durch das Code Metrics Power Tool aus Unterabschnitt 6.1.1 sowie Steve Gilham [Gil11] demonstriert wird. Um die **Methodenlänge** einer Funktion aus den CIL-Anweisungen zu ermitteln, kann auch hier die Funktionalität durch das Code Metrics Power Tool inspiriert werden. Wenn zu der `.NET-Assembly` zusätzlich eine `pdb-Datei` existiert, kann außerdem die Start- und Endzeile jeder Anweisung auf die Methode bezogen werden, ähnlich der Lösung für die `FxCop`-Regel aus Unterabschnitt 7.1.2.

7.2.2 CCI Code and AST Components

Soweit wurden nur Funktionen von `CCI Metadata` betrachtet. `CCI Code and AST Components` erlaubt es das Modell des Quellcodes einer Methode in Form eines **abstrakten Syntaxbaums** zu analysieren. Philip Newcomb beschreibt einen `AST` in seiner Präsentation über den `AST Metamodel Standard` [New05]. In der vorliegenden Master-Thesis wird ein `AST` wie folgt definiert.

Definition 19 (Abstrakter Syntaxbaum) *Ein abstrakter Syntaxbaum (engl. Abstract Syntax Tree, AST) ist eine formale Repräsentation der syntaktischen Struktur einer Software auch innerhalb von Methoden. Besonders in Methoden enthält der AST einen Knoten für jede Anweisung. Wenn diese Anweisung aus mehreren anderen Anweisungen besteht, besitzt der Knoten für diese Anweisungen entsprechende Unterknoten.*

Um einen solchen Baum zu betrachten, können Besucher verwendet werden. Beispielsweise erlaubt eine Objekt vom Typ `CodeTraverser` jede strukturelle Stelle der `Assembly` mit überladbaren Methoden, von denen einige in Listing 7.2 aufgeführt sind, zu besuchen.

Listing 7.2: `CodeTraverser`-Methoden von `CCI Code and AST Components`

```

1 TraverseChildren(IConditionalStatement conditionalStatement) {...} //if
2 TraverseChildren(IConditional conditional) {...} //bool expression
3 TraverseChildren(IForStatement forStatement) {...}
4 TraverseChildren(ISwitchStatement switchStatement) {...}

```

Für jede Stelle im Code leitet `CodeTraverser` den Aufruf an die registrierten `ICodeVisitor`-Objekte weiter und ermöglicht so die Analyse des `AST`. Das Interface `ICodeVisitor` besitzt für jede `Traverse`-Methode eine entsprechende `Visit`-Methode. Da `CodeTraverser` das Code-Modell von links nach rechts und „Depth First“ besucht, können zwei Besucher registriert werden. Einer wird aufgerufen bevor ein Syntax-Element im Baum besucht wird und der andere wird aufgerufen nachdem der Besuch abgeschlossen ist.

`CCI Code and AST Components` ermöglicht das syntaktische Besuchen von Sprachelementen, indem von CIL abstrahiert wird. Es existieren Besuchermethoden für alle Sprachelemente, die aus C# 4.0 bekannt sind, wie beispielsweise Lambda-Ausdrücke, Generics, Attribute und vielen mehr. Die herunterladbare Visual Studio-Solution der `CCI Code and AST Components` enthält das Beispiel „PeToTextViaCodeModel“, das sämtliche Strukturelemente einer `dll`- oder `exe`-Datei besucht und wieder C#-Quelltext erzeugt. Das Ergebnis entspricht Code, der dem Original sehr ähnlich ist und an professionelle Decompiler erinnert. Die `CCI` erlaubt es außerdem eine geladene `Assembly` zu verändern und anschließend wieder zu exportieren.

7.3 NRefactory

Auch in der Mono-Welt existieren Werkzeuge für Compiler-ähnliche Funktionen. Das Team hinter Mono beschreibt Mono als Implementierung des .NET Frameworks für Windows, Linux, Mac OS und verschiedene mobile Plattformen [Tea12]. In diesem Umfeld ist `NRefactory`¹ Teil der freien .NET-Entwicklungsumgebung `SharpDevelop`² und arbeitet im Gegensatz zur `CCI` und `FxCop` nicht nur auf `Assembly`-Ebene, sondern hauptsächlich mit Quellcode.

7.3.1 AST aus Quellcode

Mit `NRefactory` ist es möglich einen `AST` aus C#-Quelltext zu erzeugen. Unterstützung für VB.NET ist nach Angaben in der Funktionsbeschreibung der Bibliothek [ICS12] noch nicht implementiert. Anschließend können `AST`-Besucher als direkte Implementierung der `IAstVisitor`-Schnittstelle oder als Ableitung von `DepthFirstAstVisitor` erstellt werden und mit dem abstrakten `Syntaxbaum` arbeiten. Listing 7.3 zeigt einige wenige Methoden, die im `IAstVisitor`-Interface definiert sind.

Listing 7.3: `IAstVisitor`-Methoden von `NRefactory`

```

1 VisitIfElseStatement(IfElseStatement ifElseStatement) {...}
2 VisitBinaryOperatorExpression(BinaryOperatorExpression expression) {...}
3 VisitPrimitiveExpression(PrimitiveExpression expression) {...}
4 VisitNewLine(NewLineNode newLineNode) {...}
5 VisitInvocationExpression(InvocationExpression expression) {...}

```

Mit einem solchen Besucher können strukturelle Code-Elemente wie `if`-Anweisungen sehr gut besucht werden, was die Berechnung der zyklomatischen Komplexität ermöglicht, ohne kompilieren zu müssen. Wie in Unterabschnitt 5.2.1 beschrieben, brauchen dafür nur alle Entscheidungsstellen besucht und aufsummiert werden. Die Methodenlänge kann ebenfalls

¹Download: „Repository for `NRefactory` 5“ <https://github.com/icsharpcode/NRefactory/>

²Download und mehr Informationen: „`SharpDevelop`“ <http://www.icsharpcode.net/OpenSource/SD/>

einfach bestimmt werden. Es ist möglich entweder alle Statements oder alle Zeilenumbrüche zu besuchen und aufzusummieren. Der [AST](#) kann auch verändert und anschließend neuer Code erzeugt werden. Allerdings kann der Abhängigkeitsgraph nicht durch die alleinige Erzeugung des [AST](#) erstellt werden.

7.3.2 Mono.Cecil

NRefactory ordnet Methodenaufrufe in Form von `InvocationExpression`-Objekten erst dann der definierenden Stelle zu, wenn alle Typen und Methoden im [AST](#) mit einem `CSharpAstResolver`-Objekt komplett aufgelöst werden. Dazu verwendet NRefactory die Mono.Cecil-Bibliothek, die Jean-Baptiste Evain als API zum Erzeugen, Analysieren und Verändern von [.NET-Assemblies](#) beschreibt [Eva11]. Die [Assemblies](#), in denen sich die Typen befinden, auf die der Code zugreift, werden von Mono.Cecil geladen. Listing 7.4 zeigt wie ein `CSharpAstResolver`-Objekt zur Auflösung des abstrakten Syntaxbaums erzeugt werden kann.

Listing 7.4: Erzeugung eines `CSharpAstResolver`-Objekts

```
1 CSharpAstResolver CreateResolver(  
2     CompilationUnit parsed, params Type[] types)  
3 {  
4     IProjectContent project = new CSharpProjectContent();  
5     CSharpParsedFile file = parsed.ToTypeSystem();  
6     project = project.UpdateProjectContent(null, file);  
7     project = project.AddAssemblyReferences(LoadAssemblies(types));  
8     return new CSharpAstResolver(  
9         project.CreateCompilation(), parsed, file);  
10 }
```

Der Code orientiert sich an dem Code-Beispiel von Daniel Grunwald in dem NRefactory-Repository [ICS12]. Das übergebene `CompilationUnit`-Objekt ist der Wurzelknoten des [AST](#), der aus dem Quellcode-Fragment erstellt wurde. Dieser [AST](#) wird für die Typzuordnung vorbereitet und gemeinsam mit allen anderen [Assemblies](#), die für die Auflösung von externen Typen verwendet werden sollen, zusammengefasst. Das zusammenfassende Objekt ist von Typ `CSharpProjectContent`. Um diese [Assemblies](#) zu finden, wird jede durch einen in ihr enthaltenen, nicht weiter relevanten Typ identifiziert. Für die Erzeugung des `CSharpAstResolver`-Objekts, wird der unvorbereitete [AST](#), der vorbereitete [AST](#) und das Kompilat des `CSharpProjectContent`-Objekts übergeben. Die eigentliche Auflösung wird dann über einen Aufruf der `ApplyNavigator`-Methode auf einem `CSharpAstResolver`-Objekt synchron durchgeführt. Diese Funktion erwartet ein Objekt vom Typ `IResolveVisitorNavigator` und teilt diesem alle aufgelösten Details zu allen Knoten im [AST](#) mit. Ähnlich wie die [CCI](#) erlaubt auch Mono.Cecil das manuelle Einlesen von [Assemblies](#) und [pdb-Dateien](#) um diese anhand der [CIL](#)-Anweisungen zu analysieren. [Assemblies](#) und [pdb-Dateien](#) können außerdem auch verändert und gespeichert werden.

7.4 Codename „Roslyn“

Karen Ng, Matt Warren, Peter Golde und Anders Hejlsberg beschreiben in ihrem Dokument [NWDH11] ein Projekt um den `C#`- und den `VB.NET`-Compiler als Dienst zur

Verfügung zu stellen. Sie beschreiben das Problem, dass Compiler viel isoliertes und ungeteiltes Wissen über die zu kompilierende Sprache besitzen. Dieses Wissen ist auch für andere Anwendungen, wie beispielsweise Entwicklungsumgebungen, Werkzeuge die Refactorings unterstützen und Code-Analyse-Tools von Bedeutung. Momentan müssen diese Programme das Wissen der Compiler neu erfinden und Teile des Compilers neu implementieren. Im Rahmen des Projekts *Roslyn*¹ sollen die Compiler ihr Wissen teilen können. Dazu werden sie als Dienste zur Verfügung gestellt und bieten APIs an. Diese APIs können dann von sämtlichen Code-orientierten Werkzeugen genutzt werden. Kirill Osenkov stellt in seinem Blog-Post [Ose11] die CTP (Community Technology Preview) von Roslyn vor und erwähnt ein weiteres Ziel. Die bisherigen Compiler sollen in einer Sprache, die auf der .NET-Laufzeitumgebung (Common Language Runtime, CLR) ausgeführt wird, neu geschrieben werden. Auf diese Weise soll das Team hinter Visual Studio schneller und bessere Features entwickeln können.

7.4.1 Schichten

In ihrem Dokument [NWDGH11] beschreiben Hejlsberg und co die traditionelle Compiler-Pipeline der .NET-Umgebung. Die Roslyn-Compiler werden diese Pipeline ebenfalls haben. Ein Parser erzeugt einen Syntaxbaum anhand der Grammatik der Sprache. Darauf folgt die Deklarierungsphase. In dieser Phase werden die Deklarierungen im Code zusammen mit importierten Metadaten in Form von benannten Einheiten, sogenannten Symbols, erstellt. Danach folgt die Binder-Phase, in der Typen im Syntaxbaum den Symbols zugeordnet werden. In der letzten Phase wird alle vom Compiler gesammelte Information in Form von CIL-Anweisungen in eine *Assembly*-Datei exportiert. Project Roslyn bietet für jede dieser Phasen eine entsprechende API im Rahmen einer Compiler API Schicht, die auf der Compiler-Pipeline aufbaut. Auf diese Compiler-API-Schicht setzt Roslyn noch eine Language Service-Schicht, die mit dem Objektmodell der Compiler-Phasen arbeitet und beispielsweise Refactorings unterstützt.

Hejlsberg und co unterscheiden zwischen verschiedenen Architekturschichten, die getrennt von den oben beschriebenen Schichten über der Compiler-Pipeline beschrieben werden. Die Compiler-APIs erlauben den Zugriff auf das syntaktische und das semantische Objektmodell der Compiler-Pipeline und stellen sie dadurch zur Verfügung. Die nächste Schicht ist die Scripting-API. Sie bietet eine Umgebung für das Ausführen von Code-Schnipseln in Form von Ausdrücken und Anweisungen. Mit dieser Umgebung kann eine interaktive Programmierschleife (Read Eval Print Loop, REPL) realisiert werden. Die Workspace-API stellt die nächste Schicht dar und ermöglicht eine *Analyse* auf der Basis von kompletten Visual Studio-Solutions. Die letzte Schicht ist die Services-API. Sie ist die einzige Schicht, die eine Abhängigkeit von Visual Studio hat. Sämtliche IDE-Funktionen, wie beispielsweise Visual Studio IntelliSense, Refactorings und Formatierungsfunktionen, befinden sich in dieser Schicht.

7.4.2 Syntax Tree

In diesem Unterabschnitt werden die Möglichkeiten beschrieben, wie der Syntaxbaum der Compiler-API verwendet werden kann. Interessant ist zu sehen, dass Roslyn im Gegensatz

¹Download und mehr Informationen: „Microsoft Roslyn June 2012 CTP“ <http://msdn.com/roslyn>

zu NRefactory die Datenstruktur nicht als **abstrakten Syntaxbaum** bezeichnet, sondern lediglich als Syntaxbaum. Philip Newcomb beschreibt in seiner Präsentation über den AST Metamodel Standard [New05] den Unterschied als Nähe zum tatsächlichen Quellcode. Der Syntax Tree von Roslyn orientiert sich sehr am Quellcode. Hejlsberg und co legen in ihrem Dokument [NWGH11] besonderen Wert auf die Vollständigkeit des Baums. Jedes Leerzeichen und sogar syntaktische Fehler sind Bestandteil des Syntaxbaums.

Listing 7.5 zeigt die Erzeugung eines Syntaxbaums aus einem Quellcodeausschnitt, der in der Variable `sourceText` enthalten ist. Ähnlich wie NRefactory bietet auch Roslyn die Möglichkeit, den Baum mit LINQ-Abfragen zu analysieren, da die Eigenschaft `DescendentNodes` ein `IEnumerable` zurückliefert. LINQ¹ ist ein Sprachkonstrukt, das es erlaubt, mit Keywords wie `from`, `where` und `select` SQL-ähnliche Abfragen direkt in C# oder VB.NET zu schreiben.

Listing 7.5: Erzeugung eines Syntax Tree mit der Roslyn Compiler-API

```
1 SyntaxTree tree = SyntaxTree.ParseCompilationUnit(sourceText);
2 methods = tree.Root.DescendentNodes().OfType<MethodDeclarationSyntax>();
3 ...
4 ifs = method.DescendentNodes().OfType<IfStatementSyntax>();
5 invocations = method.DescendentNodes().OfType<InvocationExpressionSyntax>();
```

Listing 7.5 zeigt auch, wie alle Methodendeklarationen in `sourceText` gefunden werden. In jeder dieser Methoden kann anschließend nach `if`-Anweisungen und ähnlichen Entscheidungsträgern gesucht werden, um die **zyklomatische Komplexität**, wie sie in Unterabschnitt 5.2.1 beschrieben wurde, auszurechnen. Um die Typen zu finden, von denen eine Methode abhängig ist, können neben sämtlichen Variablen alle Methodenaufrufe gefunden werden. Daraus kann anschließend der Abhängigkeitsgraph bestimmt werden. Ähnlich zu NRefactory kann auch für den Roslyn-Baum ein Besucher als Ableitung von `SyntaxWalker` erstellt werden. Über entsprechende Überladungen kann dann auf alle Knoten im Syntaxbaum reagiert werden. Leerzeichen, Zeilenumbrüche und andere Elemente, die keine nicht syntaktische Relevanz haben, sind im Baum in Form von `Trivia`-Objekten vorhanden und lassen sich ebenfalls finden. Eine Manipulation des Code-Baums ist auch möglich, indem ein neuer Baum erzeugt wird. Bestehende Bäume sind laut Hejlsberg und co unveränderlich.

7.4.3 Semantik

In diesem Unterabschnitt wird beschrieben, wie die Compiler-API die Semantik von Code verwaltet. Nach der Erzeugung der syntaktischen Repräsentation von Code, dem Syntax Tree, werden beispielsweise die Methodenaufrufe nicht automatisch der Stelle zugeordnet, die sie deklarieren. Ähnlich wie bei NRefactory wird dazu eine Auflösung benötigt, die den Baum mit den Symbols verbindet. Listing 7.6 zeigt wie ein Syntaxbaum kompiliert werden kann. Durch die Kompilierung werden alle Verweise, die sich wie im Beispiel auf die `mcorlib` (Bibliothek der Basistypen wie `object`, `string`, `DateTime`, usw) beziehen, aufgelöst.

¹Mehr Informationen: „LINQ (Language-Integrated Query)“ <http://msdn.microsoft.com/de-de/library/bb397926.aspx>

Listing 7.6: Erzeugung eines semantischen Modells aus Syntax-Bäumen

```

1 Compilation compilation = Compilation.Create(name)
2                               .AddReferences(mscorlib)
3                               .AddSyntaxTrees(tree);
4 SemanticModel semanticModel = compilation.GetSemanticModel(tree);

```

Das Kompilieren findet im Speicher statt und das Ergebnis wird nicht in Form einer *Assembly* in eine Datei geschrieben. Aus dem Kompilat kann das semantische Modell erzeugt werden. Dieses Modell kann anschließend genutzt werden, um zu sämtlichen syntaktischen Elementen im Syntax Tree semantische Eigenschaften zu erhalten.

Listing 7.7: Ermitteln der semantischen Information eines Methodenaufrufs

```

1 SemanticInfo invocationSymbols = semanticModel.GetSemanticInfo(invocation);
2 string typeName = invocationSymbols.Symbol.ContainingType.Name;

```

Dafür kann die `GetSemanticInfo`-Methode mit dem entsprechenden Element aufgerufen werden, wie in Listing 7.7 dargestellt. Das semantische Modell enthält alle Symbols des Kompilats und nicht nur die, die für die Auflösung der Typen benötigt werden die im Syntaxbaum deklariert wurden. Wenn also über alle Typen im semantischen Modell iteriert wird, werden nicht nur die Typen aus dem Baum, sondern auch alle Typen aller referenzierten *Assemblies* betrachtet.

7.4.4 Workspaces

In diesem Unterabschnitt werden die Möglichkeiten der Compiler-API in Bezug auf die *Analyse* von ganzen Visual Studio-Solutions beschrieben. Eine *Solution* ist eine sln-Datei, die es Visual Studio ermöglicht mehrere Projekte mit mehreren Quellcodedateien zu verwalten und zu kompilieren. In einer Solution sind in der Regel alle Abhängigkeiten der beinhalteten Projekte entweder durch andere beinhaltete Projekte oder durch referenzierte *Assemblies* auflösbar. Die Roslyn Services-API erlaubt es im Kontext einer laufenden Visual Studio-Instanz einen sogenannten Workspace zu verwenden, der die Visual Studio-Umgebung repräsentiert und den Syntaxbaum ändert, sobald sich der Quellcode ändert. Ein Workspace kann auch ohne Abhängigkeit von Visual Studio erzeugt werden, indem die Solution-Datei direkt geladen wird, wie in Listing 7.8 zu sehen ist.

Listing 7.8: Erzeugung eines Workspace aus einer Visual Studio-Solution-Datei

```

1 IWorkspace workspace = Workspace.LoadSolution(solutionFile);
2 ISolution solution = ws.CurrentSolution;
3 ...
4 SyntaxTree tree = document.GetSyntaxTree();
5 SemanticModel semantic = document.GetSemanticModel();

```

In einem *ISolution*-Objekt kann über alle enthaltenen Projekte iteriert werden. In jedem Projekt können alle Dokumente als *IDocument*-Objekte betrachtet werden. Zu jedem Dokument kann der zugehörige Syntaxbaum und das semantische Modell bestimmt werden. Da die Workspace-Repräsentation das Erzeugen und Kompilieren der Dateien übernimmt, kann die *statische Code-Analyse* direkt auf Solution-Ebene durchgeführt werden. Der dazu nötige Syntaxbaum und das semantische Modell wurden bereits in den Unterabschnitten 7.4.2 und 7.4.3 beschrieben.

7.5 Zusammenfassung

In den oberen Abschnitten wurden die vier Technologien FxCop, CCI, NRefactory und Projekt Roslyn kurz vorgestellt. Zu jeder Technologie wurden Eigenheiten und Besonderheiten beschrieben, die sich mehr oder weniger auf die Eignung der Technologie in Bezug auf ein Usus-ähnliches Programm für .NET auswirken. Abbildung 7.4 visualisiert die sich daraus ergebenden bewerteten Verbindungen zwischen den Technologien und den Eingangs festgelegten Fragen. Es ist zu sehen, dass die für die Berechnung der **Metriken** erforderlichen Informationen von allen untersuchten Lösungen ermittelt werden können. Bei zwei Lösungen allerdings mit Vorbedingung.

	FxCop	CCI	NRefactory	Project Roslyn
Metrik-Informationen bestimmbar?	✓	✓	⚠	⚠
Verfügbar?	✓	✓	✓	✗
Kostenfrei?	✓	✓	✓	✓
Einfach einsetzbar?	✗	✓	✓	✓
Für alle .NET Versionen?	✓	✓	✓	✓
Unabhängige Komponente?	✗	✓	✓	✗
Tauglich für unvollständige Software?	✓	✓	⚠	⚠
Für C# und VB.net?	✓	✓	✗	✓
Für Assembly?	✓	✓	✓	✗
Für Code?	✗	✗	✓	✓

Ja ✓

Eingeschränkt ⚠

Nein ✗

Abbildung 7.4: Tabellarisches Ergebnis der Evaluierung der verschiedenen Technologien

Zuerst wurde in Abschnitt 7.1 FxCop betrachtet. Diese Technologie wird im Rahmen der Evaluierung als nicht tauglich bewertet. Der Grund dieser Entscheidung ist auf die Umgebung zurückzuführen. Die eigentliche **statische Code-Analyse** könnte von einer eigenen speziellen Regel durchgeführt werden, die im Kontext der FxCop Runner-Anwendung ausgeführt wird. Dadurch wird die Kommunikation zwischen Regel und aufrufendem Programm erschwert was zu einer größeren Distanz zwischen Informationssammlung und Informationsaufbereitung führt.

Anschließend wurde in Abschnitt 7.2 die **Common Compiler Infrastructure** betrachtet. Im Rahmen der Evaluierung wird diese Technologie als sehr geeignet erachtet. Als einzigen Nachteil wurde festgestellt, dass CCI mit **Assemblies** und CIL-Anweisungen arbeitet und daher keinen direkten Bezug zu dem originalen Quellcode hat. Da das zu analysierende Programm in Form einer **Assembly** vorliegen muss, wurden alle Abhängigkeiten bereits zur Kompilierzeit aufgelöst und Ziele von Methodenaufrufen bestimmt. CCI kann diese auslesen und braucht nicht selbst zu kompilieren.

Danach wurde in Abschnitt 7.3 NRefactory untersucht. Diese Technologie wird in der Evaluierung als geeignet angesehen. Die Besonderheit von NRefactory ist, dass diese Bibliothek Unterstützung für die **Analyse** von Quelltext bietet (zumindest für C#, VB.NET

kommt wahrscheinlich bald dazu). Diese Form der [Analyse](#) hat einige Einschränkungen. Beispielsweise kann das tatsächliche Aufrufziel eines Methodenaufrufs durch die alleinige Betrachtung von Quelltext nicht ermittelt werden. Dafür muss NRefactory kompilieren, was dann nur in einer vollständigen Umgebung möglich ist. Um mit [Assemblies](#) zu arbeiten wird Mono.Cecil genutzt, das hier in Verbindung mit NRefactory gesehen wird. Während der Evaluierung wurden die Möglichkeiten der [Assembly](#)-Analyse mithilfe von CCI als umfangreicher festgestellt.

Abschließend wurde in Abschnitt 7.4 das Projekt mit dem Codename Roslyn vorgestellt. In dieser Evaluierung wurde diese Technologie als nicht geeignet bewertet. Diese Entscheidung wurde aufgrund der Tatsache getroffen, dass Roslyn noch nicht offiziell verfügbar ist, sondern derzeit nur als CTP bezogen werden kann (was sich nach dieser Master-These wahrscheinlich bald ändert). Weiterhin ist die Installation der Roslyn-Umgebung erforderlich. Langfristig ist diese Technologie die bessere Lösung, da sie direkt von Microsoft kommt und im Gegensatz zu NRefactory wesentlich umfangreichere Möglichkeiten, wie beispielsweise die Workspace-API, bietet. Roslyn arbeitet mit Quellcode und muss zur Auflösung von Methodenaufrufen ebenfalls kompilieren was wieder nur in einer vollständigen Umgebung funktioniert. Dank der Workspace-API wird diese aber automatisch verwaltet.

Zusammenfassend ist zu sagen, dass die [Common Compiler Infrastructure](#) derzeit die beste Möglichkeit darstellt, eine [statische Code-Analyse](#) durchzuführen. Dabei werden die [Assemblies](#) analysiert, die die verschiedenen .NET-Compiler erzeugen. Sollte die [Analyse](#) von reinem Quellcode in einem Sonderfall erforderlich sein, kann eine Kombination mit NRefactory genutzt werden. Sobald Projekt Roslyn allerdings einen offiziellen Zustand erreicht, sollte eine mögliche Technologierersetzung in Betracht gezogen werden. Bis dahin unterstützt die Tatsache, dass FxCop und andere Programme ebenfalls die CCI verwenden, die Entscheidung zugunsten der [Common Compiler Infrastructure](#).

8 Usus.NET

Nachdem das Usus-Plugin für Java einfach *Usus* heißt, wird die Visual Studio-Erweiterung, die in der vorliegenden Master-Thesis entwickelt wird, im weiteren Verlauf als *Usus.NET* bezeichnet. Die Architektur von Usus.NET, das in Abschnitt 3.3 noch grob als System bezeichnet wurde, besteht aus drei Teilen, die in Abbildung 8.1 abgebildet sind. **Usus.net**

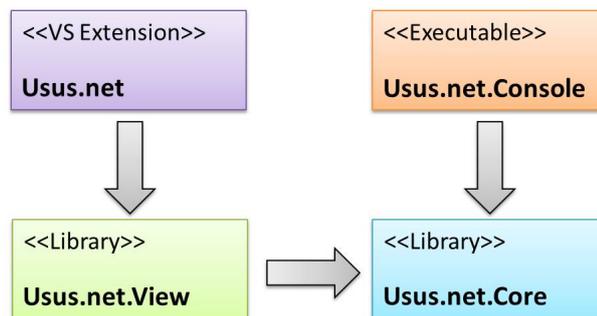


Abbildung 8.1: Architektur von Usus.NET

als Visual Studio-Erweiterung verwendet Oberflächenelemente, die in einer Bibliothek zusammengefasst und unabhängig von Visual Studio sind. Die Oberflächen sind Bestandteil der Bibliothek **Usus.net.View**. Diese verwendet die **Usus.net.Core**-Bibliothek, die den allgemeinen Funktionsumfang von Usus enthält und die **statische Code-Analyse** von **Assemblies**, die Bewertung der **Metriken** und die Hotspot-Analyse durchführen kann. Diese Bibliothek kann direkt in andere Anwendungen integriert werden, wie beispielsweise in **Usus.net.Console**, um Usus.NET auch über die Kommandozeile bedienen zu können. In diesem Kapitel wird ausschließlich auf die zentralen Funktionen von Usus.NET eingegangen, die in **Usus.net.Core** implementiert sind. In diesem Zusammenhang wird die konkrete Berechnung der **Metriken** mit der **Common Compiler Infrastructure** beschrieben, sowie auf das zugrunde liegende Objektmodell eingegangen. Das Objektmodell dient dazu die Ergebnisse der **statischen Code-Analyse** weiter zu verarbeiten und als Bericht zur Verfügung zu stellen. Abschließend wird das Verifikations-Framework vorgestellt, mit dem **Usus.net.Core** selbst getestet wurde.

8.1 Metrikberechnung

In diesem Abschnitt wird beschrieben, wie die **Metriken** aus Abschnitt 5.2 von Usus.NET berechnet werden. In Abschnitt 7.2 wurde erklärt wie **Assemblies** mit CCI analysiert werden können. Deswegen beginnt auch die Berechnung der **Metriken** mit einem Pfad zu einer PE-Datei, wie sie in Unterabschnitt 7.2.1 definiert wurde. Abbildung 8.2 zeigt die Klassen, die an der Berechnung beteiligt sind. Diese Klassen befinden sich alle im Namespace **andrena.Usus.net.Core.Metrics**. Ein **AssemblyVisitor**-Objekt ist in der Lage, eine **PE-Datei** einzulesen und definierte Typen und Methoden zu lokalisieren. Dies ist mit

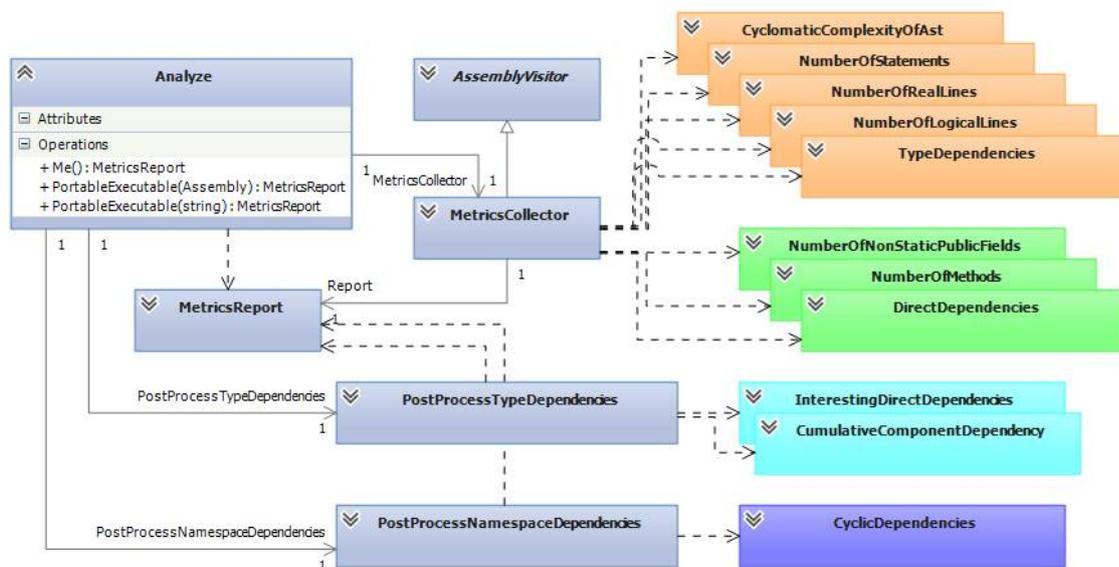


Abbildung 8.2: Klassen, die an der Metrikberechnung von Usus.NET beteiligt sind

CCI Metadata möglich. Die Klasse `MetricsCollector` erbt die `Analyse`-Funktionalität von `AssemblyVisitor` und berechnet für die gefundenen Typen und Methoden die **Metriken**, die ohne den Kontext berechenbar sind. Für Methoden sind dies *zyklomatische Komplexität*, *Anzahl der Anweisungen*, *Anzahl der tatsächlichen Codezeilen*, *Anzahl der logischen Codezeilen* und *Abhängigkeiten von Typen*. Als **Klassenmetriken** sind *Anzahl der nicht-statischen öffentlichen Felder*, *Anzahl der Methoden* und *direkte Abhängigkeiten von Typen* kontextfrei. Alle weiteren **Metriken** sind nur dann berechenbar, wenn die kontextfreien **Metriken** vollständig bestimmt wurden. So lassen sich beispielsweise die *interessanten direkten Typ-Abhängigkeiten* einer Klasse erst ermitteln, wenn alle deklarierten Typen im betrachteten System bekannt sind. Das gleiche gilt für die *zyklischen Abhängigkeiten von Namespaces*, die nur in einem vollständigen Graph der Namespaces gefunden werden können. Diese kontext-sensitiven **Metriken** werden von Usus.NET im Rahmen einer nachträglichen Bearbeitung der **Metriken** berechnet. Dazu wird zum einen die Klasse `PostProcessTypeDependencies` verwendet, die aus allen interessanten Abhängigkeiten den Abhängigkeitsgraph erzeugt um anschließend die **kumulierte Komponentenabhängigkeit** zu bestimmen. Zum anderen wird die Klasse `PostProcessNamespaceDependencies` genutzt um aus dem Abhängigkeitsgraph der Typen einen Abhängigkeitsgraph der Namespaces zu erstellen. Dieser kann dann genutzt werden, um die **zyklischen Abhängigkeiten der Namespaces** zu suchen.

Für jede zu bestimmende **Metrik** existiert eine Klasse, die die Berechnung durchführen kann. Alle Ergebnisse werden in einem `MetricsReport`-Objekt gesammelt und in Form eines Berichts im `MetricsCollector`-Objekt hinterlegt. Der Aufrufer kann diesen Bericht jederzeit einsehen. Abschnitt 8.2 beschreibt diesen Bericht als Bestandteil des Objektmodells ausführlicher, während Listing 8.1 einen vereinfachten Codeausschnitt aus der Bibliothek `Usus.net.Console` zeigt, der die **statische Code-Analyse** durchführt und sämtliche **Metriken** aller Methoden einfach auf der Console ausgibt.

Listing 8.1: Aufruf der Code-Analyse von `Usus.net.Core` in `Usus.net.Console`

```

1 var metrics = Analyze.PortableExecutable(assemblyToAnalyze);
2 //alternativ: Analyze.Me();
3 foreach (var method in metrics.Methods) {
4     Console.WriteLine("Signature: " + method.Signature);
5     Console.WriteLine("CC: " + method.CyclomaticComplexity);
6     ...
7 }

```

Nachdem die Infrastruktur der Metrikberechnung hiermit beschrieben wurde, werden in den folgenden Unterabschnitten die konkreten Berechnungen der einzelnen [Metriken](#) vorgestellt.

8.1.1 Zyklomatische Komplexität

Die [zyklomatische Komplexität](#) einer Methode gibt an, wie viele unterschiedliche Ablaufpfade durch diese Methode existieren. In Unterabschnitt 5.2.1 wurde diese [Metrik](#) ausführlicher erklärt. In Usus.NET kann die Berechnung auf zwei verschiedene Arten durchgeführt werden. Zum einen kann ausschließlich [CCI Metadata](#) verwendet und alle CIL-Anweisungen die eine Entscheidung treffen, aufsummiert werden. Eine einfache Abwandlung des Algorithmus, der unter anderem von Steve Gilham vorgestellt [[Gil11](#)] und laut ihm angeblich auch in frühen Versionen von NDepend (Abschnitt 6.2) verwendet wurde, ist in der Klasse `CyclomaticComplexityOfIl` implementiert. Die andere Variante, die in Usus.NET standardmäßig verwendet wird, nutzt [CCI Code and AST Components](#). Die Klasse `CyclomaticComplexityOfAst` erzeugt ein spezielles Besucher-Objekt vom Typ `CyclomaticComplexityCalculator`, das den aus dem Methodenrumpf erzeugten abstrakten Syntaxbaum besucht. Die Erzeugung des [AST](#) aus der Methodendefinition übernimmt CCI. Listing 8.2 zeigt den Code, der die [zyklomatische Komplexität](#) in `Usus.net.Core` bestimmt.

Listing 8.2: Statement-Analyse zur Bestimmung der zyklomatischen Komplexität

```

1 var methodBody = method.Decompile(pdb, host);
2 var cyclomaticComplexityCalculator = new CyclomaticComplexityCalculator();
3 cyclomaticComplexityCalculator.Traverse(methodBody.Statements());
4 var result = cyclomaticComplexityCalculator.Result;

```

Das `CyclomaticComplexityCalculator`-Objekt erbt die Funktionalität, sämtliche Anweisungen (engl. Statements) in einem Methodenrumpf zu besuchen, von `CodeTraverser`, einer Klasse in der CCI. Jede Bedingung, jede `if`-, `while`-, `for`-, `case`- und `catch`-Anweisung erhöht einen anfangs mit 1 initiierten Zähler. Obwohl `CodeTraverser` auch eine Behandlungsroutine für `foreach`-Anweisungen anbietet, ist die CCI momentan nicht in der Lage diese als solche erfolgreich zu erkennen. Dies hat zur Folge, dass `foreach`-Anweisungen, die vom Compiler in `while-if-finally`-Anweisungen übersetzt werden, auch also solche erkannt werden. Durch das `while` und das `if` entstehen also zwei Stellen, die eine Entscheidung treffen, während im Quellcode mit `foreach` nur eine erkennbar ist. Die in Usus.NET berechnete [zyklomatische Komplexität](#) einer Methode entspricht also der Komplexität, die der Compiler generiert und nicht der, die durch den Quellcode suggeriert wird. Für die Berechnung wird immer nur die aktuell Methode betrachtet. Bedingungen in anderen Methoden, die von der aktuellen Methode aufgerufen werden, werden nicht berücksichtigt.

CCI Code and AST Components erlaubt die Statement-Analyse unabhängig davon, ob eine `pdb`-Datei vorhanden ist oder nicht. Die zyklomatische Komplexität kann also in beiden Fällen korrekt berechnet werden.

8.1.2 Methodenlänge

Die Länge einer Methode sagt etwas darüber aus, wie viele Dinge diese Methode tut. Eine kurze Methode ist einfacher zu verstehen und zu verändern als eine lange. In Unterabschnitt 5.2.1 wurde diese Metrik und ihre Varianten genauer beschrieben. Usus.NET bestimmt drei verschiedene Methodenlängen.

Als sehr nachvollziehbare Längenangabe wird wohl die Anzahl der Codezeilen gesehen, die Logik enthalten. Dieser Wert wird von der Klasse `NumberOfLogicalLines` bestimmt, indem nur CCI Metadata verwendet wird. Alle CIL-Anweisungen, außer leeren Anweisungen (`Nop`), Block-Verlassen-Anweisungen (`Leave.S`) und Methodenende-Anweisungen (`Ret`) werden dabei betrachtet. Listing 8.3 zeigt den Code, der die Codezeilen der betroffenen CIL-Anweisungen (Operationen) einer Methode findet, doppelte Werte entfernt und deren Anzahl bestimmt.

Listing 8.3: Operation-Analyse zur Bestimmung der Anzahl der logischen Codezeilen

```
1 var locations = method.LocatedOperations(pdb);
2 var result = locations.GetAllStartLinesOfInterestingOpCodes()
3     .Distinct().Count();
```

Die Zuordnung der Codezeilen des Quelltexts vor der Kompilierung zu den CIL-Anweisungen danach, kann nur durchgeführt werden, wenn zu der zu analysierenden `PE-Datei` eine `pdb-Datei` existiert. Wie in Unterabschnitt 7.2.1 definiert, beinhaltet die `pdb-Datei` die erforderlichen Zeilenangaben. Die CCI unterstützt die Analyse dieser Dateien ebenfalls mithilfe der Klasse `PdbReader`. Wenn keine `pdb-Datei` zur Verfügung steht, oder es sich bei der Methode um eine Iterator-Methode mit `yield return` handelt, kann die CCI die Zeileninformationen nicht bestimmen. In diesem Fall ist der Ergebniswert der `NumberOfLogicalLines`-Metrik -1.

Standardmässig verwendet Usus.NET die Anzahl der logischen Codezeilen als `Methodenlänge`. Sollte deren Bestimmung aufgrund einer fehlenden `pdb-Datei` oder spezieller Compiler-Funktionen, wie beispielsweise Iterator-Methoden, nicht möglich sein, wird die Anzahl der logischen Anweisungen (Statements) verwendet. Die `NumberOfStatements`-Klasse verwendet CCI Code and AST Components um den Methodenrumpf zu dekompileieren und einen `AST` zu erzeugen. Dieser kann anschließend analysiert und sämtliche Anweisungen besucht werden. Dieses Vorgehen ist vergleichbar mit der Bestimmung der zyklomatischen Komplexität. Listing 8.4 zeigt wie dafür ein `StatementCollector`-Objekt verwendet wird.

Listing 8.4: Statement-Analyse zur Bestimmung der Anzahl der Anweisungen

```
1 var methodBody = method.Decompile(pdb, host);
2 var statementCollector = new StatementCollector(pdb);
3 statementCollector.Traverse(methodBody.Statements());
4 var result = return statementCollector.Result.Count();
```

Wenn die `pdb-Datei` nicht vorhanden ist, enthält die `pdb-Variable` keine Referenz auf ein `PdbReader`-Objekt, sondern lediglich `null`. Das Dekompilieren der Methode funktioniert trotzdem. Neben der Anzahl der Anweisungen kann Usus.NET auch die Anzahl der CIL-Anweisungen, den sogenannten Operationen, bestimmen. Dafür wird kein `AST` erzeugt, sodass nur die `CCI Metadata`-Infrastruktur benötigt wird. Allerdings besteht eine Methode aus einer Vielzahl von Operationen, die keinen direkten Zusammenhang mit der Länge des Quellcodes vor dem Kompilieren erkennen lassen. Die Anzahl der CIL-Anweisungen kann daher nicht als nachvollziehbare `Methodenlänge` gesehen werden und wird daher nicht weiter berücksichtigt.

Als dritte nachvollziehbare `Methodenlänge` kann Usus.NET die Anzahl der tatsächlichen Zeilen einer Methode ausrechnen. Genau wie bei der Anzahl der logischen Codezeilen ist hierfür `CCI Metadata` sowie eine `pdb-Datei` erforderlich. In der Klasse `NumberOfRealLines` werden die Zeilen aller CIL-Anweisungen eines Methodenrumpfs ermittelt. Durch die Differenz der ersten und letzten Zeile kann die Anzahl der Zeilenumbrüche bestimmt werden. Listing 8.5 zeigt diese Berechnung.

Listing 8.5: Operation-Analyse zur Bestimmung der Anzahl der tatsächlichen Zeilen

```
1 var locations = method.LocatedOperations(pdb);
2 var firstLine = locations.GetAllValidLines(1 => 1.EndLine).Min();
3 var lastLine = locations.GetAllValidLines(1 => 1.EndLine).Max();
4 var result = Math.Max(0, lastLine - firstLine - 1);
```

Wenn die `pdb-Datei` nicht existiert, ist der Wert der `NumberOfRealLines-Metrik` wieder `-1`. Da die Anzahl der tatsächlichen Zeilen einer Methode wenig Auskunft über die logische Größe der Methode gibt, bevorzugt Usus.NET die Anzahl der logischen Zeilen als `Methodenlänge`. Und nur wenn dieser Wert nicht ermittelt werden kann, fällt Usus.NET auf die Anzahl der logischen Anweisungen zurück.

8.1.3 Kumulierte Komponentenabhängigkeit

Die Anzahl aller Klassen, von denen eine Klasse im objektorientierten Sinne (siehe Unterabschnitt 2.2.1) direkt und indirekt abhängig ist, ist auch als `kumulierte Komponentenabhängigkeit` (CCD) bekannt. Diese `Metrik` wurde bereits ausführlich in Unterabschnitt 5.2.2 vorgestellt. Damit die direkten Abhängigkeiten einer Klasse bestimmt werden können, müssen zunächst die direkten Klassenabhängigkeiten jeder Methode ermittelt werden. Jede Verwendung eines Typen innerhalb der Methodensignatur sowie dem Methodenrumpf muss erkannt werden. Usus.NET verwendet dazu die Klasse `TypeDependencies`.

Listing 8.6: Typen-Analyse zur Bestimmung der Klassenabhängigkeiten einer Methode

```
1 var dependenciesOfMethod = Enumerable.Empty<string>()
2     .Union(TypeDependenciesOfSignature.Of(method))
3     .Union(TypeDependenciesOfVariables.Of(method))
4     .Union(TypeDependenciesOfCallOperations.Of(method))
5     .Union(TypeDependenciesOfNewOperations.Of(method))
6     .Union(TypeDependenciesOfCatches.Of(method))
7     .Union(TypeDependenciesOfTypeMentions.Of(method));
```

Mithilfe der `CCI Metadata`-Infrastruktur werden die CIL-Anweisungen der Methode nach Referenzen auf Typen untersucht. Die dabei verwendeten Strategien sind in Listing 8.6 zu

sehen. Die Ergebnisse dieser Strategien werden in einer Liste konsolidiert, wobei doppelte Vorkommen ignoriert werden. Die Variable `method` enthält eine Referenz auf ein Objekt vom Typ `IMethodDefinition`, welches in `CCI Metadata` definiert ist. Nachdem zu jeder Methode alle Abhängigkeiten bestimmt wurden, können die Abhängigkeiten einer Klasse einfach ermittelt werden, wie in Listing 8.7 dargestellt.

Listing 8.7: Typen-Analyse zur Bestimmung der Klassenabhängigkeiten einer Klasse

```

1 var dependenciesOfType = Enumerable.Empty<string>()
2     .Union(type.FullName().Return())
3     .Union(GetMethodTypes(methods))
4     .Union(GetFieldTypes(type.Fields))
5     .Union(GetAncestorTypes(type))
6     .Union(GetGenericConstraints(type));

```

Dazu werden die Abhängigkeitslisten der Methoden zusammengefasst und mit den Typen der Felder, den Typen der Oberklassen und den Typen eventueller generischer Einschränkungen kombiniert. Eine Klasse ist auch von sich selbst abhängig. Diese Konsolidierung der Listen erfolgt in der Klasse `DirectDependencies`. Doppelte Vorkommen eines Typen werden wieder ignoriert. Die Variable `type` enthält eine Referenz auf ein Objekt vom Typ `INamedTypeDefinition`, welches in `CCI Metadata` definiert ist. `methods` enthält eine Sequenz von `MethodMetricsReport`-Objekten, die den Methoden der Klasse in `type` zugeordnet sind.

Um Typen wie `object`, `string` und `int`, die in der Base Class Library (BCL) des .NET Frameworks definiert sind, zu ignorieren, brauchen nur die interessanten Typen betrachtet werden. Dies sind Klassen, die in den analysierten `Assemblies` selbst deklariert und definiert wurden. Dieser Filter ist in der Klasse `InterestingDirectDependencies` definiert. Da zu diesem Zeitpunkt alle deklarierten Typen des Systems bereits bekannt sein müssen, kann der Filter erst im Rahmen der nachträglichen Bearbeitung durch die `PostProcessTypeDependencies`-Klasse angewendet werden. Jeder in den `Assemblies` deklarierte Typ ist anschließend nur noch von anderen in den `Assemblies` deklarierten Typen abhängig. Aus diesem Netz der Abhängigkeiten erzeugt Usus.NET mithilfe der QuickGraph¹-Bibliothek einen Abhängigkeitsgraph auf Typeebene. Um jetzt alle direkten und indirekten Abhängigkeiten einer Klasse `type` zu bestimmen, kann der Algorithmus Depth First Search verwendet werden. In Unterabschnitt 5.2.2 wurde der Algorithmus bereits ausgewählt, um die Erreichbarkeitsmenge des `type`-Knoten zu finden. Wie in Listing 8.8 dargestellt, ist die **kumulierte Komponentenabhängigkeit** einer Klasse `type` die Anzahl der nicht vom Compiler erzeugten Typen, in der Erreichbarkeitsmenge von `type`.

Listing 8.8: Typen-Analyse zur Bestimmung der **kumulierten Komponentenabhängigkeit**

```

1 var result = typeGraph.Reach(type).Vertices
2     .Count(t => !t.CompilerGenerated);

```

Die Implementierung der `Reach`-Funktion verwendet die `DepthFirstSearchAlgorithm<T, Edge<T>>`-Klasse von QuickGraph und setzt den Startknoten auf `type`. Zusätzlich wird die Suche abgebrochen, nachdem die Erreichbarkeitsmenge des Startknotens gefunden wird.

¹QuickGraph: „Generic Graph Data Structures and Algorithms for .NET“ <http://quickgraph.codeplex.com/>

8.1.4 Klassengröße

Die **Klassengröße** gehört zu den **Metriken** die kontextfrei bestimmt werden können. In Unterabschnitt 5.2.2 wurde die **Klassengröße** in Usus als Anzahl der Methoden festgelegt. Die API der **CCI Metadata** erlaubt eine bedingte Aufsummierung aller Methoden in einer Klassendefinition, wie in Listing 8.9 gezeigt.

Listing 8.9: Typen-Analyse zur Bestimmung der Anzahl der Methoden

```
1 var result = type.Methods.Count(m => !m.IsDefaultCtor());
```

Da jeder nicht-statische Typ immer einen Default-Konstruktor enthält, wird dieser in der **Klassengröße** nicht berücksichtigt. Alle anderen Konstruktoren werden gezählt. Neben Konstruktoren und Destruktoren erzeugen der C#- und VB.NET-Compiler aus Properties, Indexern, Operatoren und Events ebenfalls Methoden, die von **CCI Metadata** auch als solche erkannt werden. Gerade in den **get-** und **set-**Blöcken eines Properties wird oft Logik implementiert, sodass auch hier mehr als nur Datenzugriff passiert und Methoden-ähnliche Konstrukte entstehen. Eine Unterscheidung, ob ein Property-Block Logik enthält oder nicht (wie im Falle von Auto Implemented Properties), kann Usus.NET nicht vornehmen. Usus.NET sieht deswegen sowohl im **get-** als auch im **set-**Block jeweils eine Methode. Wenn Properties generell ignoriert werden würden, könnte die **Klassengröße** sehr leicht manipuliert werden, indem Methoden als Properties implementiert werden. Da Indexer sich wie Properties verhalten wird auch hier wieder eine Methode pro Zugriffsrichtung gezählt. Operatoren werden als statische Methoden definiert und beeinflussen daher ebenfalls die **Klassengröße**. Das letzte Sprachkonstrukt das ebenfalls in Methoden resultiert, ist das Event. Ein als **event** definiertes Feld beinhaltet ähnlich wie Properties zwei Blöcke. Über den **add-**Block kann ein **EventHandler** registriert und den **remove-**Block kann dieser wieder entfernt werden. Auch hier wird wieder pro Block eine Methode erkannt, da in diesen Event-Blöcken ebenfalls Logik implementiert werden kann.

8.1.5 Nicht-statische öffentliche Felder

Die Anzahl der **öffentlichen Felder**, die nicht statisch sind, kann ebenfalls im Rahmen der kontextfreien Bestimmung der **Klassenmetriken** ermittelt werden. Diese **Metrik** wurde bereits in Unterabschnitt 5.2.3 vorgestellt. Listing 8.10 zeigt diese bedingte Aufsummierung der Felder einer Klasse.

Listing 8.10: Typen-Analyse zur Bestimmung der nicht-statischen öffentlichen Felder

```
1 var result = type.Fields.Count(f => !f.IsStatic
2     && f.Visibility == TypeMemberVisibility.Public);
```

Auch hier ermöglicht die API der **CCI Metadata** eine Iteration über alle Felder einer Typ-Definition.

8.1.6 Namespaces mit zyklischen Abhängigkeiten

Die Anzahl der **zyklischen Abhängigkeiten** ist die einzige **Metrik**, die Usus.NET für Namespaces bestimmen kann. In Unterabschnitt 5.2.3 wurde hergeleitet, dass dafür der Algorithmus Strongly Connected Components verwendet werden kann. Für die Berechnung wird also der vollständige Abhängigkeitsgraph auf Namespace-Ebene benötigt. Um die Berechnung der **kumulierten Komponentenabhängigkeit** in Unterabschnitt 8.1.3 durchführen

zu können, wurde der vollständige Abhängigkeitsgraph auf Typebene bereits erstellt. Aus diesem Graph kann der Namespace-Graph reduziert werden. Damit ist offensichtlich, dass auch diese **Metrik** im Rahmen der nachträglichen Bearbeitung kontextabhängig bestimmt wird. Die Klasse `PostProcessNamespaceDependencies` führt die nötigen Berechnungen durch, indem zuerst der Abhängigkeitsgraph auf Namespace-Ebene erstellt wird. Danach werden die Kreise ermittelt und den entsprechenden Namespaces zugewiesen.

Namespace-Graph

Die Erstellung des Graphen aller Namespaces geschieht wie in Listing 8.11 dargestellt.

Listing 8.11: Erzeugung des Abhängigkeitsgraphen auf Namespace-Ebene

```

1 namespaceGraph = metrics.GraphOfTypes.Cast(t => t.AsNamespaceWithTypes());
2 foreach (var namespaceGroup in namespaceGraph.Vertices
3         .GroupBy(n => n.Itself.Name)) {
4     namespaceGraph.Reduce(
5         namespaceGroup.AsNamespaceWithTypes(),
6         namespaceGroup);
7 }

```

Der Abhängigkeitsgraph auf Klassenebene ist ein Graph vom Typ `MutableGraph`, der Knoten vom Typ `TypeMetricsReport` enthält. Dieser Graph wird zunächst in einen neuen `MutableGraph`-Graphen mit Knoten vom Typ `NamespaceMetricsWithTypeMetrics` überführt. Dies ist notwendig, da Knoten im nächsten Schritt zusammengefasst werden und alle Knoten in einem Graph den gleichen Knotentyp haben müssen. Jeder Knoten vom Typ `TypeMetricsReport` wird zu einem `NamespaceMetricsWithTypeMetrics`-Knoten umgestaltet. Das Ergebnis dieser `Cast`-Operation ist also ein identischer Graph mit einem unterschiedlichen Knotentyp. Anschließend können alle Knoten, die den gleichen Namespace (`n.Itself.Name`) besitzen zusammengefasst werden. Alle diese gruppierten Knoten (Typen) des gleichen Namespace können dann mit der `Reduce`-Operation des Graphen zu einem einzigen Knoten zusammengefasst werden, der alle Typen des Namespace repräsentiert. Der erste Parameter von `Reduce` ist der neue Knoten, der die Knoten in der Menge, die als zweiter Parameter übergeben wird, ersetzen soll. Selbstverständlich müssen alle Kanten zwischen Knoten außerhalb und Knoten innerhalb der zu reduzierenden Menge zu Kanten zwischen dem neuen Knoten und den Knoten außerhalb der reduzierten Menge umgestaltet werden.

In der `QuickGraph`-Bibliothek konnte kein Algorithmus gefunden werden, der diesen Reduktionsvorgang komplett übernimmt. `QuickGraph` bietet lediglich die `MergeVertex`-Operation an, die es erlaubt, einen einzigen Knoten aufzulösen und die Knoten seiner eingehenden Kanten mit den Knoten seiner ausgehenden Kanten zu verbinden. Auf dieser Basis kann der Algorithmus in Listing 8.12 aufsetzen, der in der vorliegenden Master-Thesis als *Vertex Reduction* bezeichnet wird. Der erste Parameter dieses Algorithmus ist der neue Knoten `reducedVertex`, der eine Menge an anderen Knoten zusammenfassen soll. Der zweite Parameter des Algorithmus sind die Knoten, die zusammengefasst werden sollen, welche als Menge von Knoten in `vertices` vorliegen. Der neue Knoten wird in den Graph eingefügt und mit allen Knoten, die er ersetzen soll, *stark* verbunden. Anschließend können alle Knoten, die ersetzt werden sollen, mithilfe der `MergeVertex`-Operation aufgelöst werden.

Listing 8.12: Vertex Reduction - Reduktion von Knotenmengen in einem Digraph

```

1 graph.AddVertex(reducedVertex);
2 foreach (var vertex in vertices) {
3     graph.AddEdge(new Edge<T>(reducedVertex, vertex));
4     graph.AddEdge(new Edge<T>(vertex, reducedVertex));
5     graph.MergeVertex(vertex, (s, t) => new Edge<T>(s, t));
6 }

```

Durch die beidseitigen Verbindungen der aufzulösenden Knoten mit dem neuen Knoten werden alle Kanten der alten Knoten auf den neuen Knoten umbogen. Abbildung 8.3 zeigt die schematische Funktionsweise des Algorithmus Vertex Reduction an einem Beispiel. Die beiden mittleren Knoten sollen zu einem zusammengefasst werden, ohne

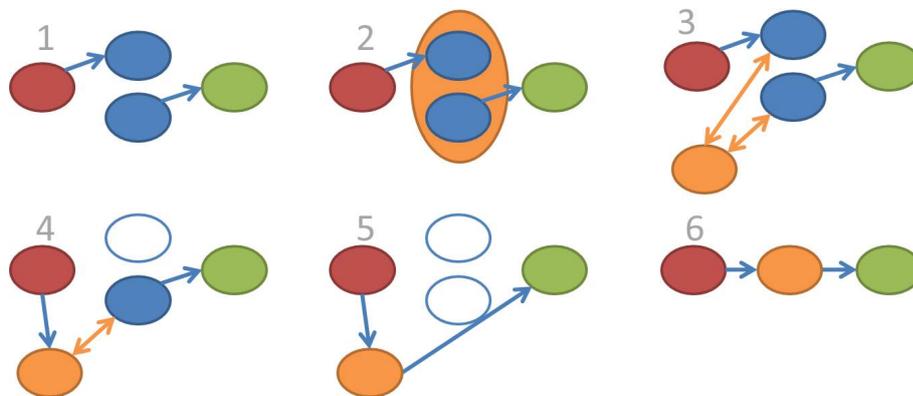


Abbildung 8.3: Schematische Funktionsweise des Algorithmus Vertex Reduction

dass die Kanten verloren gehen. Diese Graphenreduktion, wie sie für die Erstellung des Abhängigkeitsgraph auf Namespace-Ebene erforderlich ist, kann mit der wiederholten Anwendung des Algorithmus Vertex Reduction erreicht werden. Bei jeder Anwendung wird dem Graph ein neuer `NamespaceMetricsWithTypeMetrics`-Knoten hinzugefügt, der alle `NamespaceMetricsWithTypeMetrics`-Knoten des gleichen Namespace ersetzt. Sobald der Algorithmus für alle Gruppen von Typen, die sich im gleichen Namespace befinden angewendet wurde, ist der Abhängigkeitsgraph auf Namespace-Ebene vollständig.

Namespace-Kreise

Die QuickGraph-Bibliothek bietet eine Implementierung des Algorithmus zur Erkennung der starken Zusammenhangskomponenten. Die Methode `StronglyConnectedComponents` kann dazu auf dem QuickGraph-Graphen vom Typ `BidirectionalGraph<V, Edge<V>>` aufrufen werden. Das Ergebnis ist eine Zuordnung von Knoten `V` zu der eindeutigen Identifikationsnummer der Strongly Connected Component in Form eines `IDictionary<V, int>`-Objekts. Um die Kreise den enthalten Knoten zuordnen zu können, ist eine Zuordnung von Knoten zu einer Menge von Knoten (dem Kreis) erforderlich. Tabelle 8.1 zeigt die Transformation der ursprünglichen Zuordnung, um die gewünschte Abbildung von Knoten auf Knotenmenge zu erhalten. `IE<V>` steht für eine Sequenz von Objekten vom Typ `V` und ist eine Abkürzung des Typen `IEnumerable<V>`. Die Transformation hat einen einmaligen Aufwand von $O(n)$, da die initiale Abbildung einmal vollständig invertiert wer-

(1)	$V \rightarrow int$	Ergebnis der SCC-Methode
(2)	$int \rightarrow IE < V >$	Umkehrung der Abbildung (1)
(3)	$V \rightarrow int \rightarrow int \rightarrow IE < V >$	Funktionale Komposition von (1) und (2)
(4)	$V \rightarrow IE < V >$	Reduktion der funktionalen Komposition (3)

Tabelle 8.1: Abbildung von Knoten zu starker Zusammenhangskomponente zu anderen Knoten

den muss (2). Diese neue Abbildung wird in einem `StronglyConnectedComponents<V>`-Objekt gekapselt. Um die Anzahl der [zyklischen Abhängigkeiten](#) eines konkreten Namespace zu ermitteln, muss die `StronglyConnectedComponents`-Methode auf dem Graph auf Namespace-Abhängigkeiten aufgerufen werden. Auf das Ergebnis kann die in Tabelle 8.1 gezeigte Transformation angewendet werden. Durch diese Transformation wird ein Objekt vom Typ `StronglyConnectedComponents<NamespaceMetricsWithTypeMetrics>` erzeugt. Dieses Objekt kann jederzeit nach allen oder speziellen Kreisen gefragt werden. Die [zyklischen Abhängigkeiten eines Namespace](#) werden in der nachträglichen Bearbeitung der [Metriken](#) von der Klasse `CyclicDependencies` bestimmt. Listing 8.13 zeigt wie das Objekt, das die starken Zusammenhangskomponenten repräsentiert (also das Ergebnis der Komposition aus Tabelle 8.1), nach dem Kreis gefragt werden kann, auf dem sich der Knoten des Namespace `namespaceWithTypes` befindet.

Listing 8.13: Namespace-Analyse zur Bestimmung der [zyklischen Abhängigkeiten](#)

```
1 var result = from n in cycles.OfVertex(namespaceWithTypes).Vertices
2           select n.Itself;
```

Nachdem alle Namespaces Informationen über die Kreise, auf denen sie sich eventuell befinden, erhalten haben, ist die nachträgliche Bearbeitung der [statischen Code-Analyse](#) beendet. Das vollständige `MetricsReport`-Objekt steht dem Aufrufer der Analysemethoden der `Analyze`-Klasse zur Verfügung.

Vertikale Kreise

Namespaces sind hierarchisch. Jeder Typ befindet sich direkt in einem Namespace, wobei dieser wieder Bestandteil eines übergeordneten Namespace sein kann. Der Typ ist also mehreren Namespaces zugeordnet, dem direkten und den indirekten. Usus.NET verwendet den Namespace, der den Typ direkt enthält, als Namespace der Klasse oder des Interfaces. Wenn die Zyklen ermittelt werden, werden also auch Referenzen innerhalb den Hierarchieebenen beachtet. Eine Abhängigkeit eines Typen in Namespace A von einem Typen in Namespace A.B erzeugt für Usus.NET also einen Namespace-Kreis, wenn ein Typ in Namespace A.B einen Typen aus A referenziert. Dieser Kreis ist vertikal, da er zwischen den Ebenen der Namespace-Hierarchie existiert. Prinzipiell sind in .NET alle öffentlichen Typen der übergeordneten Namespaces von den untergeordneten Namespaces ohne `using`-Anweisungen sichtbar. Wenn Usus.NET vertikale Zyklen ignorieren würde, könnten fragwürdige Klassenabhängigkeiten unerkannt bleiben.

8.2 Objektmodell

Im vorherigen Abschnitt wurde die Berechnung der **Metriken** ausführlich beschrieben. Dieser Abschnitt stellt das Objektmodell des Metrikberichts vor. Alle Klassen dieses Modells befinden sich im Namespace `andrena.Usus.net.Core.Reports`. Abbildung 8.2 zeigt, dass das zentrale Datenobjekt der `MetricsReport`-Typ ist. Wie in Abschnitt 8.1 beschrieben, wird dieses Objekt zunächst mit den Daten befüllt, die das `MetricsCollector`-Objekt sammelt. Anschließend werden im Rahmen der nachträglichen Bearbeitung weitere Berechnungen vorgenommen um auch die kontextabhängigen **Metriken** zu bestimmen. Die komplette **statische Code-Analyse** wird durch die statischen Methoden der `Analyze`-Klasse gestartet. Diese Methoden liefern das `MetricsReport`-Objekt als Ergebnis an den Aufrufer zurück. In diesem Abschnitt wird genau dieses Objekt genauer betrachtet. Abbildung 8.4 zeigt alle Klassen, aus denen dieser Bericht besteht.

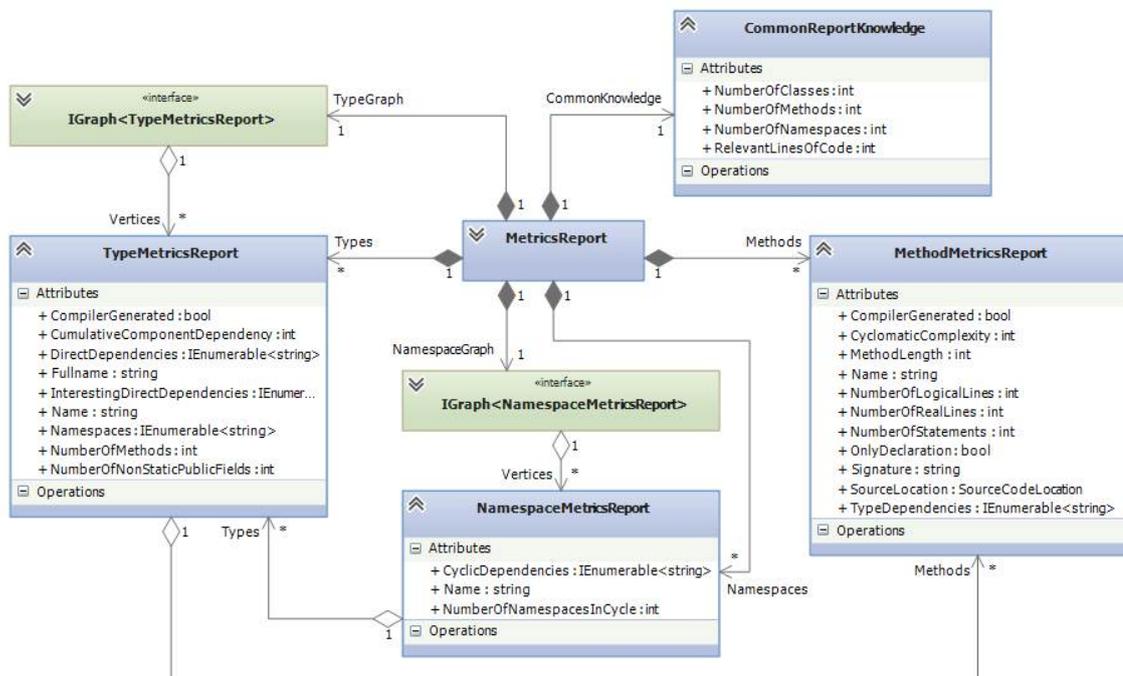


Abbildung 8.4: Objektmodell und Klassen des vollständigen Metrikberichts

8.2.1 Methodenberichte

Der Metrikbericht besteht aus mehreren Methodenberichten. Diese `MethodMetricsReport`-Objekte enthalten alle relevanten Daten und **Metriken** jeder im System gefundenen Methode. Die eindeutige Zuordnung von Methode zu Methodenbericht ist über die Eigenschaft `Signature` möglich. Die Signatur einer Methode macht sie im System eindeutig, da neben Methodennamen auch Typ- und Namespace-Namen, sowie vollständige Namen des Rückgabetypen und aller Parametertypen enthalten sind. Neben der Menge aller Methodenberichte kann der Bericht zu einer bestimmten Methode in `MetricsReport` auch

gesucht werden. Dazu existieren mehrere Extension Methods¹, die wie in Listing 8.14 dargestellt, benutzt werden können. Die Variable `metrics` enthält die Referenz zu dem `MetricsReport`-Objekt. Methodenberichte sind vom Typ `MethodMetricsReport` und lassen sich zum einen über die `Signature`-Eigenschaft der Methode finden, wie Zeile 2 zeigt. Wenn die gesuchte Methode im Scope ist, also aufgerufen werden könnte, kann sie wie in Zeile 4 auch über einen `Expression`-Ausdruck angegeben werden. Microsoft beschreibt dieses Konzept in Verbindung mit den in C# integrierten Lambda-Ausdrücken in einem eigenen Artikel in der MSDN Library [Mic10c]. Dadurch bleibt die Methodenreferenz, im Gegensatz zur Verwendung von Strings, Ziel von automatisierten Refactorings. Eine dritte Möglichkeit erlaubt das Suchen von Methodenberichten mit `MethodInfo`-Objekten aus der `System.Reflection`-API des .NET Framework.

Listing 8.14: Methoden- und Property-Suche in einem `MetricsReport`-Objekt

```

1 //Methoden
2 methodMetrics = metrics.ForMethod(
3     "System.Void Landrena.Usus.net.Console.Analyzer.AnalyzeFile(System.String)");
4 methodMetrics = metrics.ForMethod(() => AnalyzeFile(null));
5 //Properties
6 propertyMetrics = metrics.ForProperty(() => Name);
7 methodMetrics = metrics.ForMethod(
8     "System.String Landrena.Usus.net.Console.Analyzer.Name.get()");

```

Das gleiche gilt auch für Properties. Da der C#-Compiler den `get`- und `set`-Block eines Properties als Methoden kompiliert, werden diese ebenfalls auf `MethodMetricsReport`-Objekte abgebildet. Was der Compiler sonst noch als Methoden erzeugt, wurde bereits in Unterabschnitt 8.1.4 beschrieben. Wie in Zeile 6 in Listing 8.14 gezeigt, lassen sich Properties auch über einen `Expression`-Ausdruck suchen. Das resultierende Objekt vom Typ `PropertyMetricsReport` ist einfach eine Zusammenfassung zweier Objekte vom Typ `MethodMetricsReport`, nämlich eins für den `get`- und eins für den `set`-Teil des Properties. Diese `MethodMetricsReport`-Objekte der Properties lassen sich aber auch direkt über die Signatur finden, wie Zeile 7 zeigt.

8.2.2 Klassenberichte

Da die Methoden, zu denen `MetricsReport` die Methodenberichte enthält, in einer Klasse deklariert sind, existiert auch ein Bericht für eben diese Klasse. Die Abbildung von Klasse zu Methode ist im Usus.NET-Objektmodell als Abbildung von `TypeMetricsReport`-Objekt zu `MethodMetricsReport`-Objekten realisiert.

Listing 8.15: Klassen- und Methodensuche in einem `MetricsReport`-Objekt

```

1 typeMetrics = metrics.ForType<Analyzer>();
2 typeMetrics = metrics.ForType(typeof(Analyzer));
3 typeMetrics = metrics.ForType("Landrena.Usus.net.Console.Analyzer");
4 foreach (var methodMetrics in metrics.MethodsOfType(typeMetrics)) {...}

```

Der Codeausschnitt in Listing 8.15 zeigt in Zeile 4 wie mithilfe der `MethodsOfType`-Methode alle Methodenberichte, die zu einem Klassenbericht gehören, ermittelt werden

¹Mehr Informationen: „Extension Methods“ <http://msdn.microsoft.com/en-us/library/bb383977.aspx>

können. Die Variable `metrics` enthält wieder die Referenz zu dem `MetricsReport`-Objekt. Zeile 1 und 2 zeigen wie der Klassenbericht für einen Typen gesucht werden kann. Das `MetricsReport`-Objekt muss in diesem Fall für die `Assembly` erzeugt worden sein, in der sich auch der gesuchte Typ befindet. Der Klassenname kann damit weiterhin von automatisierten Refactorings erreicht werden. In Zeile 3 wird der Bericht anhand der `FullName`-Eigenschaft des Klassenberichts gesucht. Diese Eigenschaft enthält den Namespace, den Klassennamen sowie eventuelle generische Parameter und ist damit eindeutig.

Die **Metriken** auf Klassenebene, die im `TypeMetricsReport` enthalten sind, beinhalten auch die Menge aller direkten Abhängigkeiten. In Unterabschnitt 8.1.3 wurde beschrieben, dass daraus ein Abhängigkeitsgraph auf Klassenebene erstellt wird. Dieser Klassengraph steht anschließend in einem schreibgeschützten Format im `MetricsReport`-Objekt zur Verfügung. Nach außen ist der Graph als `IGraph<V>`-Objekt sichtbar, das Knoten vom Typ `TypeMetricsReport` enthält. Neben der Liste von Knoten besteht der Graph auch aus einer Liste von Kanten zwischen diesen Knoten. Eine Kante ist vom Typ `Tuple<TypeMetricsReport, TypeMetricsReport>`. Das erste Element in diesem Tuple stellt die Quelle, das zweite das Ziel der Verbindung dar. Beide Listen sind schreibgeschützt, sodass der Graph nicht ohne weiteres von außen verändert werden kann. Intern verwendet Usus.NET den Typ `MutableGraph<V>`, der das `IGraph<V>`-Interface implementiert. Dieser Typ ist nicht mehr schreibgeschützt und erlaubt einige weitere Operationen. Beispielsweise lassen sich Kreise ermitteln, Erreichbarkeitsmengen bestimmen und Knoten reduzieren. Der Graph ist Bestandteil des Metrikberichts um eine Abbildung der Beziehungen zwischen den Klassen zu haben.

8.2.3 Namespace-Berichte

Neben den Methoden- und Klassenberichte enthält das `MetricsReport`-Objekt auch Berichte über Namespaces. Diese Berichte sind vom Typ `NamespaceMetricsReport` und enthalten im Vergleich zu den Methoden- und Klassenberichten nur wenig Informationen. Doch auch Namespaces haben einen eindeutigen Namen. Diese `Name`-Eigenschaft des Berichts wird für die Suche verwendet, wie Listing 8.16 in Zeile 1 zeigt.

Listing 8.16: Namespace- und Klassensuche in einem `MetricsReport`-Objekt

```
1 namespaceMetrics = metrics.ForNamespace("andrena.Usus.net.Console");
2 foreach (var typeMetrics in metrics.TypesOfNamespace(namespaceMetrics)) {...}
```

Da Namespaces mehrere Klassen enthalten, lässt sich diese Zuordnung auch unter den Berichten auflösen. Die Schleife in Zeile 2 iteriert über alle Klassenberichte eines Namespace-Berichts, indem die `TypesOfNamespace`-Methode verwendet wird.

Neben dem Abhängigkeitsgraph auf Klassenebene steht auch der Abhängigkeitsgraph auf Namespace-Ebene zur Verfügung. In Unterabschnitt 8.1.6 wurde beschrieben, wie dieser Graph aus dem Klassengraph erzeugt wird. Der Namespace-Graph besteht aus Kanten zwischen Knoten vom Typ `NamespaceMetricsReport`. Im Gegensatz zum Graph auf Typebene verwendet Usus.NET intern keinen `MutableGraph<NamespaceMetricsReport>`-Graph, sondern einen Graph, dessen Knoten `NamespaceMetricsWithTypeMetrics`-Objekte sind. Dies ist der Fall, da der Namespace-Graph aus dem Graph mit Klassenberichten als Knoten erzeugt wird. Um trotzdem einen schreibgeschützten Graphen vom Typ

`IGraph<NamespaceMetricsReport>` zur Verfügung stellen zu können, ohne das ein neuer Graph erzeugt werden muss, verwendet Usus.NET eine Projektion des Knotentypen. Für diese Projektion wird ein `GraphSurrogate<V, R>`-Objekt verwendet, welches das Interface `IGraph<V>` implementiert und ein Objekt vom Typ `IGraph<R>` kapselt. Die Knoten- und Kantenliste in diesem Typ werden über die Projektion von $R \rightarrow V$ an den Zieltyp angepasst. Über diesen Adapter ist es möglich, einen Graph mit Knoten vom Typ `NamespaceMetricsWithTypeMetrics` als Graph mit Knoten vom einem anderen Typ zu behandeln. In diesem Fall mit Knoten vom Typ `NamespaceMetricsReport`. Dies ist allerdings nur möglich, wenn eine Abbildung von R nach V existiert. Da ein `NamespaceMetricsWithTypeMetrics`-Objekt den eigentlichen Namespace-Bericht sowie alle Klassenberichte in diesem Namespace enthält, ist die Abbildung einfach zu realisieren.

8.3 Metrikgewichtung

Nachdem in den vorherigen beiden Abschnitten beschrieben wurde, wie die Berechnung der [Metriken](#) funktioniert und in welcher Struktur das Ergebnis präsentiert wird, beschäftigt sich dieser Abschnitt mit der Weiterverarbeitung dieser Ergebnisse. Usus verwendet die [Metriken](#) beispielsweise für die in [Unterabschnitt 5.1.1](#) vorgestellte Cockpit-Ansicht und in der in [Unterabschnitt 5.1.3](#) betrachteten Hotspots-Ansicht. Für diese beiden Ansichten werden die ermittelten [Metriken](#) gewichtet, mit Schwellwerten verglichen und Mittelwerte gebildet. Wie dies geschieht wurde bereits in [Unterabschnitt 5.2.3](#) ausführlich beschrieben. Die im folgenden vorgestellten Klassen befinden sich alle im Namespace `andrena.Usus.net.Core.Hotspots`.

8.3.1 Hotspots

Hotspots sind Methoden, Klassen und Namespaces dessen [Metriken](#) über einer Schwelle liegen. Usus.NET kann diese Hotspots in Form eines `MetricsHotspots`-Objekts aus einem `MetricsReport`-Objekt bestimmen. [Listing 8.17](#) zeigt dies in Zeile 1. Die Variable `metrics` enthält wieder eine Referenz zu einem `MetricsReport`-Objekt. Anschließend können alle Berichte der Methoden, Klassen und Namespaces ermittelt werden, dessen [Metriken](#) über den Schwellen liegen, die in [Unterabschnitt 5.2.3](#) erwähnt wurden.

Listing 8.17: Methoden-, Klassen- und Namespace-Hotspots in `MetricsReport`

```
1 MetricsHotspots hotspots = metrics.Hotspots();
2 methodHotspots = hotspots.OfMethodLength();
3 classHotspots = hotspots.OfClassSize();
4 classHotspots = hotspots.OfCumulativeComponentDependency();
5 classHotspots = hotspots.OfNumberOfNonStaticPublicFields();
6 namespaceHotspots = hotspots.OfNamespacesInCycle();
```

Wenn, wie beispielsweise in Zeile 2, alle `MethodMetricsReport`-Objekte bestimmt werden, dessen `MethodLength`-Eigenschaft über der definierten Schwelle liegt, iteriert Usus.NET über alle Methoden im `MetricsReport`-Objekt. Dabei werden nur Berichte zurückgegeben, dessen [Methodenlängen](#) über dem Ergebnis der Schwellwertfunktion für [Methodenlängen](#) liegen. Die Schwellwerte der [Metriken](#) müssen nicht konstant sein. Der Schwellwert kann sich auch abhängig von der Projektgröße ändern, wie es beispielweise bei der [kumulierten Komponentenabhängigkeit](#) der Fall ist (siehe [Unterabschnitt 5.2.3](#)). Daher verwendet

Usus.NET das Konzept einer Schwellwertfunktion. Alle Schwellwertfunktionen können bei Bedarf auch zur Laufzeit geändert werden. Dafür lässt sich das `RatingFunctionLimits`-Objekt über die statische Eigenschaft `RatingFunctions.Limits` erreichen. In diesem Objekt sind alle Schwellwertfunktionen definiert. Listing 8.18 zeigt die Deklaration und Definition der Schwellwertfunktionen für die [kumulierte Komponentenabhängigkeit](#) und die [Methodenlänge](#) unter Verwendung von Lambda-Ausdrücken. Die Formel der Schwellwertfunktion `CumulativeComponentDependency` entspricht der bereits beschriebenen Formel 5.7.

Listing 8.18: Schwellwertfunktionen sind abhängig von `MetricsReport`-Daten

```

1 //Deklarationen
2 Func<CommonReportKnowledge, int> MethodLength { get; set; }
3 Func<CommonReportKnowledge, int> CumulativeComponentDependency { get; set; }
4 ...
5 //Definitionen
6 MethodLength = ck => 9;
7 CumulativeComponentDependency = ck => ck.NumberOfClasses *
8     (1.5 / Math.Pow(2, (Math.Log(ck.NumberOfClasses) / Math.Log(5))));
9 ...

```

Alle Schwellwertfunktionen sind Abbildungen von einem `CommonReportKnowledge`-Objekt auf eine Ganzzahl. `CommonReportKnowledge` ist Teil des Objektmodells des vollständigen Metrikberichts und wurde bereits in Abbildung 8.4 gezeigt. Dieses Objekt enthält einige allgemeine Daten der [statischen Code-Analyse](#), wie beispielsweise die Anzahl der analysierten Klassen und Namespaces.

8.3.2 Statistiken

Die Unterscheidung zwischen [Metrik](#) und Statistik der [Metrik](#) wurde bereits in Unterabschnitt 2.2.3 definiert. Die Theorie, also wie diese Statistiken in Usus entstehen und berechnet werden können, wurde in Unterabschnitt 5.2.3 vorgestellt. In Usus.NET lassen sich die gleichen Werte über ein `RatedMetrics`-Objekt bestimmen, welches wie in Listing 8.19 in Zeile 1 gezeigt, über die `Rate`-Methode erzeugt werden kann. Die Variable `metrics` enthält wieder eine Referenz eines `MetricsReport`-Objekts.

Listing 8.19: Statistiken aller Berichte in einem `MetricsReport`-Objekt

```

1 RatedMetrics statistics = metrics.Rate();
2 double acd = statistics.AverageComponentDependency;
3 double acs = statistics.AverageRatedClassSize;
4 double acc = statistics.AverageRatedCyclomaticComplexity;
5 double aml = statistics.AverageRatedMethodLength;
6 double anf = statistics.AverageRatedNumberOfNonStaticPublicFields;
7 double ncn = statistics.NamespacesWithCyclicDependencies;

```

Diese Statistiken entstehen durch eine Gewichtung der Methoden-, Klassen und Namespace-Berichte, die anschließend gezählt oder gemittelt werden. Die Berichte können auch unabhängig von einander gewichtet werden. Die `Rate`-Methode existiert für die einzelnen Berichte ebenfalls und erzeugt Objekte vom Typ `RatedMethodMetrics` für Methoden, `RatedTypeMetrics` für Klassen und `RatedNamespaceMetrics` für Namespaces. Tatsächlich ruft Usus.NET diese Methode für jeden Bericht auf und speichert die dadurch erzeugten

Objekte in einem `RatedMetrics`-Objekt. Dort können dann die Mittelwerte der einzelnen gewichteten **Metriken** als projektübergreifende Statistiken bestimmt werden. Sämtliche Gewichtungsfunktionen implementieren die Formeln aus Unterabschnitt 5.2.3 und sind in der Klasse `RatingFunctions` definiert.

Listing 8.20: Berechnung der Statistiken in einem `RatedMetrics`-Objekt

```

1 AverageRatedMethodLength = RatedMethods.AverageAny(m => m.RatedMethodLength);
2 AverageComponentDependency =
3     RatedTypes.AverageAny(m => m.CumulativeComponentDependency)
4     / metrics.CommonKnowledge.NumberOfClasses;
5 NamespacesWithCyclicDependencies =
6     RatedNamespaces.CountAny(m => m.IsInCycle)
7     / metrics.CommonKnowledge.NumberOfNamespaces;
```

Listing 8.20 zeigt die Berechnung der **durchschnittlichen Methodenlänge** in Zeile 1, der **durchschnittlichen Komponentenabhängigkeit** in den Zeilen 2 bis 4 und der Anzahl der **Namespaces mit zyklischen Abhängigkeiten** in den Zeilen 5 bis 7. Obwohl die Berechnungen einfach lesbar sind, ist der Aufwand zur Bestimmung jeder Statistik $O(n)$. Für jede Statistik müssen alle Methoden oder Klassen-Berichte, die die **Metrik** enthalten, berücksichtigt werden.

8.3.3 Ignorierbares

In Unterabschnitt 8.3.2 wurde beschrieben, dass alle Methodenberichte berücksichtigt werden müssen, wenn eine Statistik einer **Metrik** für Methoden bestimmt werden soll. Allerdings existieren in einer **Assembly** Methoden, die keine relevanten **Metriken** enthalten. Alle Methoden und Klassen, die vom Compiler erzeugt wurden, werden mit einem Attribut vom Typ `[CompilerGenerated]` gekennzeichnet. So gekennzeichnete Stellen ignoriert Usus.NET bei der Berechnung der Statistiken und der Hotspots. Da diese Methoden und Klassen im Quellcode nicht offensichtlich sind, würde ein Hotspot an einer solchen Stelle den Entwickler auf etwas nicht nachvollziehbares hinweisen.

Beispielsweise erzeugt der Compiler aus einem `yield return`-Statement eine Klasse, die den Zustand der Methode verwaltet. Die erzeugte Logik ist kein Hotspot und sollte die Statistik auch nicht beeinflussen, da ein daraus resultierender Komplexitätsanstieg nicht nachvollziehbar ist. Ein weiteres Beispiel sind anonyme Methoden. Die durch einen Lambda-Ausdruck erzeugte Klasse wird teilweise ignoriert. Lambda-Ausdrücke sind Funktionen, die in Form von `()=>{}` geschrieben und wie Objekte behandelt werden können. Microsoft beschreibt diese Funktionsobjekte in der MSDN Library [Mic10c]. In einem Lambda-Ausdruck können die Variablen, die sich zum Zeitpunkt der Definition im Scope befinden, verwendet werden. Wenn der Lambda-Ausdruck dann als Objekt weitergegeben wird, behält dieser weiterhin den Scope der Definition. Dies ist möglich, da der .NET-Compiler eine Klasse generiert, die eine generierte Methode enthält. Diese generierte Methode enthält die Logik des Lambda-Ausdrucks. Externe Variablen, die von diesem Ausdruck verwendet werden, werden vom Compiler ebenfalls in der generierten Klasse deklariert und nicht wie erwartet in der Klasse, die den Lambda-Ausdruck definiert. Diese erzeugt lediglich ein Objekt der generierten Klasse um die Variablen definieren zu können. Dank der in Unterabschnitt 7.2.2 beschriebenen Bibliothek **CCI Code and AST Components** ist Usus.NET in der Lage, die Logik des Lambda-Ausdrucks in der Methode zu

finden, die den Ausdruck definiert und nicht nur in der Methode, die ihn nach dem Kompilieren enthält. Dadurch werden die **zyklomatische Komplexität** und die **Methodenlänge** der definierenden Methode, unter Berücksichtigung der Logik des Lambda-Ausdrucks, korrekt berechnet. Die **Metriken** der erzeugten Klasse werden ignoriert, da diese wieder die Statistiken nicht nachvollziehbar beeinflussen würden.

Außerdem betrachtet Usus.NET nur Methoden, die einen nicht leeren Methodenrumpf besitzen. Abstrakte Methoden und Methodendeklarationen in Interfaces haben offensichtlich eine **Methodenlänge** und **zyklomatische Komplexität** von 0. Sie werden ebenfalls nicht in die Berechnung der Statistiken miteinbezogen, da sie das Ergebnis verfälschen. Eine Methodendeklaration ist weder gut noch schlecht, würde aber als sehr gute Methode die Statistik ungewollt verbessern. Um die ignorierbaren Methoden und Klassen zu finden, bieten die Methoden- und Klassenberichte im Usus.NET-Objektmodell aus Abschnitt 8.2 die beiden Eigenschaften **CompilerGenerated** und **OnlyDeclaration** an. Das in Abschnitt 5.1 vorgestellte Usus-Plugin für Eclipse nimmt diese Unterscheidung nicht vor und behandelt abstrakte Methoden und Interface-Deklarationen wie normale Methoden, was zu einer trügerischen Verbesserung der Statistiken führt, wenn viele Interfaces und abstrakte Methoden vorhanden sind.

8.4 Metrikverteilung

Im Namespace `andrena.Usus.net.Core.Math` existiert die Klasse `Distributions`. In dieser Klassen sind mehrere `Extensions Methods` definiert, die **Verteilungen** von Methoden- und **Klassenmetriken** bestimmen können. Listing 8.21 zeigt, wie diese Methoden verwendet werden können.

Listing 8.21: **Verteilungen** der Methoden- und Klassenberichte erstellen

```
1 IHistogram css = Metrics.TypeDistribution(t => t.ClassSize);
2 IHistogram ccs = Metrics.MethodDistribution(m => m.CyclomaticComplexity);
```

Die Variable `Metrics` enthält eine Referenz auf ein `MetricsReport`-Objekt. Die **Verteilung** besteht aus einem `Histogram`-Objekt und weiteren Verteilungsinformationen. Diese `Histogram`-Klasse bekommt über den Konstruktor eine Sequenz von `int`-Werten übergeben, aus denen sie das Histogramm erstellt. Das Histogramm ist eine einfache Abbildung von den diskreten Werten der Sequenz zu der absoluten Häufigkeit des Auftretens des entsprechenden Werts. Diese Zuordnung kann auch in einem Koordinatensystem angezeigt werden kann. Auf der x-Achse werden die diskreten Werte der Sequenz dargestellt, während die absoluten Häufigkeiten des Auftretens auf der y-Achse angezeigt werden. Dadurch wird ersichtlich, wie oft ein bestimmter Wert in der Sequenz enthalten ist. Listing 8.22 zeigt, wie ein Histogramm in Usus.NET erstellt wird.

Listing 8.22: Erzeugung eines Histogramms mit Math.NET

```
1 var maxValue = data.Max();
2 for (int i = 0; i <= maxValue; i++)
3     histogram.AddBucket(new Bucket(-0.5 + i, 0.5 + i));
4 histogram.AddData(data.Select(d => d * 1.0));
```

Die Variable `data` enthält die erwähnte Sequenz von Ganzzahlen (`IEnumerable<int>`), die über den Konstruktor übergeben wird. Um diese Sequenz auf die diskreten Werte

abzubilden, verwendet die `Histogram`-Klasse Funktionalität der öffentlichen Bibliothek `Math.NET`¹. `Usus.NET` unterstützt dabei nur Histogramme mit positiven Werten. Ein fertiges Histogramm kann in Form eines `IHistogram`-Objekts weitergegeben und analysiert werden. Dort können die Werte der x-Achse mithilfe der Eigenschaft `BinCount` von 0 bis `BinCount-1` bestimmt werden. Um den y-Wert, also die absolute Häufigkeit zu einem x-Wert zu bestimmen, kann die `IHistogram`-Methode `ElementsInBin(x)` mit dem x-Wert als Parameter aufgerufen werden.

`Usus.NET` nutzt Mechanismen um das Histogramm zu analysieren und speichert die Ergebnisse in Form von `IFittingReport`-Objekten ebenfalls in dem `Distribution`-Objekt. Das Interface `IFittingReport` definiert die beiden Properties `Parameter` und `Error`, welche von einer konkreten Verteilungsfunktion zur Verfügung gestellt werden können. In der aktuellen Version von `Usus.NET` wird nur die geometrische `Verteilung` unterstützt. Die Klasse `Distribution` hat dafür das `GeometricalFit`-Property. Dieses Property liefert ein `GeometricalDistributionFitting`-Objekt, welches als `Parameter` das λ der geometrischen `Verteilung` aus Formel 10.2 zurückgibt. Um diese Parameter der `Verteilungen` zu bestimmen, berechnet `Usus.NET` statistische Kennzahlen (beispielsweise den Mittelwert) der Sequenz des Histogramms mithilfe der `Math.NET`-Bibliothek. Wie diese Analyse des Histogramms und die Annäherung der Verteilungsfunktionen genau funktioniert, ist in Abschnitt 10.2 beschrieben. Sobald der Parameter einer `Verteilung` bestimmt wurde, kann auch der Fehler dieser `Verteilung` berechnet und über das `Error`-Property veröffentlicht werden. Anhand des Fehlers können unterschiedliche Approximationen des Histogramms verglichen werden. Da `Usus.NET` momentan nur eine `Verteilung` unterstützt, ist der Vergleich der Fehler noch nicht relevant. Wie der Fehler ermittelt werden könnte, wird in Abschnitt 10.3 behandelt.

8.5 Testen

Nachdem in den vorherigen Abschnitten die `statische Code-Analyse`, das Objektmodell sowie die Statistiken von `Usus.NET` erläutert wurden, beschäftigt sich dieser Abschnitt mit einer Möglichkeit `Metriken` mit Quellcode zu verbinden. `Usus.net.Core` stellt dafür eine Infrastruktur zur Verfügung, die in dieser Master-Thesis als *Verification Framework* bezeichnet wird. Das `Verification Framework` befindet sich im Unter-Namespace `Verification` und besteht aus einer Menge an Attributen, mit denen Methoden und Klassen dekoriert werden können.

Listing 8.23: Erwartungswerte für `Metriken` per Attribut definieren

```
1 [ExpectDirectDependency("System.Exception")]
2 class ClassWithOneMethod
3 {
4     [ExpectCyclomaticComplexity(1)]
5     public void MethodWithNothing(Exception e) {}
6 }
```

Diese Attribute beschreiben Erwartungen in Form von `Metriken` an die betroffene Methode oder Klasse, wie in Listing 8.23 gezeigt. Diese Erwartungen können mithilfe der

¹Download und mehr Informationen: „Math.NET Numerics“ <http://mathnetnumerics.codeplex.com/>

Methoden der Klasse `Verify` überprüft werden, was in Listing 8.24 zu sehen ist. Dazu ist ein `MetricReport`-Objekt erforderlich, welches auf einfache Weise mit `Analyse.Me()` ermittelt werden kann.

Listing 8.24: Verifikation der erwarteten `Metriken` für Methoden und Klassen

```

1 Verify.MethodsWith<ExpectCyclomaticComplexityAttribute>(metrics);
2 Verify.TypesWith<ExpectDirectDependencyAttribute>(metrics);

```

Bei der Verifikation werden dann alle Methoden und Klassen in der aktuellen `Assembly` ermittelt, die mit einem solchen Attribut dekoriert sind. Für diese werden anhand der Signatur oder dem Klassennamen die entsprechenden Berichte in `metrics` gesucht. Anschließend werden die Erwartungswerte, die über die Konstruktoren der Attribute angegeben werden, mit den tatsächlichen `Metriken` verglichen. Im Falle einer Abweichung wird eine `VerificationException` geworfen. Auf diese Weise können `Metriken` erzwungen werden. Dadurch kann erreicht werden, dass eine Methode oder Klasse Bedingungen an die eigene Implementierung stellt, die auf Basis von `Metriken` formuliert werden können. Eine unabsichtliche Verletzung dieser Bedingungen wird dann sofort offensichtlich. Außerdem sind die erwarteten `Metriken` bereits im Quellcode sichtbar.

Usus.NET nutzt das Verification Framework um die eigene Metrikberechnung und das Objektmodell zu testen. So bestehen die Integrationstests von `Usus.net.Core` aus beispielhaften Klassen, dessen tatsächliche `Metriken` künstlich erzeugt und mit den erwarteten Werten der Attribute verglichen werden. Sind die Werte nicht kongruent, schlagen die Testfälle fehl und ein eventueller Fehler in der Metrikberechnung wurde gefunden. Entsprechen die tatsächlichen `Metriken` den Werten in den Attributen, laufen die Tests erfolgreich durch und bestätigen so die Berechnungen der `Metriken`. Ein weiterer Anwendungsfall ist ein Konzept, das in dieser Master-Thesis als *Metrics Meta Testing* bezeichnet wird. Damit sind Testfälle gemeint, die die `Metriken` von Testfällen testen. Zum Beispiel schreibt Roy Osherove in seinem Buch [Osh09], dass Testmethoden keine Logik enthalten sollen. Keine Logik bedeutet keine entscheidungstreffenden Anweisungen, wie beispielsweise `if` oder `while` zu verwenden. Eine solche Anforderung an Tests kann einfach mit einem `[ExpectCyclomaticComplexity(1)]`-Attribut an allen Tests sichergestellt werden. Eine spezielle Testmethode erzeugt dann das `MetricsReport`-Objekt des Testprojekts mit `Analyse.Me()` und ruft die entsprechende Methode auf der `Verify`-Klasse auf. Dieser spezielle Testfall testet also, ob alle Testmethoden mit dem Attribut auch wirklich nur einen Ablaufpfad enthalten. Ein anderes Beispiel wäre die Sicherstellung, dass kein Testfall von einem bestimmten Objekt der Implementierung abhängig ist. Dafür kann die Testklasse um das Attribut `[ExpectNoDirectDependency("...")]` mit dem vollständigen Klassennamen erweitert werden. Eine spezielle Testmethode würde wieder die entsprechende Methode der `Verify`-Klasse aufrufen und fehlschlagen, sobald eine Methode in der Klasse von dem Typ aus dem Attribut abhängig ist. Das Verification Framework dient also dazu die `Metriken`, die eigentlich implizit im Code existieren, offen zu legen und einen Fehler zu erzeugen wenn die Implementierung nicht mit den erwarteten `Metriken` übereinstimmt. Der Entwickler bekommt so die Gelegenheit über ein eventuell fehlerhaftes Verhalten der Implementierung zu reflektieren und ein ansonsten verstecktes Problem schnell zu beheben.

Die Funktionen von Usus.NET, die für die Durchführung der [statischen Code-Analyse](#) zuständig sind, können automatisiert getestet werden. Die dafür erforderlichen Tests sind alle zusammen mit der Implementierung entstanden und können auf Knopfdruck ausgeführt werden. Neben den oben erwähnten Integrationstests existieren auch viele Unit Tests, die beispielsweise die korrekte Gewichtung der [Metriken](#) oder die Funktionsweise der Manipulation der Graphen sicherstellen. Eine Liste aller automatisierten Testfälle befindet sich im Anhang [A.3](#).

9 Usus.NET als Visual Studio-Erweiterung

Abbildung 8.1 zeigt die Verwendung der in Kapitel 8 beschriebenen Funktionen der Komponente `Usus.net.Core`. `Usus.net.Core` steht als Bibliothek in Form einer `Assembly` zur Verfügung und wird von der Bibliothek `Usus.net.View` genutzt. Diese Bibliothek enthält alle grafischen Oberflächen von Usus.NET und wird von der `Usus.net`-Komponente in Form einer Visual Studio-Erweiterung verwendet. Die Erweiterung stellt dann nur noch die Infrastruktur der Fenster für die Usus.NET-Oberflächen zur Verfügung. Usus.NET ist kein Addin im Sinne von Unterabschnitt 2.1.3, sondern eine Erweiterung. In diesem Kapitel werden zunächst die Steuerelemente von `Usus.net.View` vorgestellt. Anschließend wird auf die Integration in Visual Studio eingegangen.

9.1 Oberflächen

Die Oberfläche von Usus.NET besteht genau wie das Usus für Eclipse (Kapitel 5), aus mehreren grafischen Benutzeroberflächen. Alle diese Oberflächen bieten besondere Sichten auf eine gemeinsame Datenquelle. Der Metrikbericht aus Abschnitt 8.2 ist als Ergebnis der statischen Code-Analyse aus Abschnitt 8.1 das zentrale Objekt. Sobald dieses Objekt vom Typ `MetricsReport` zur Verfügung steht, müssen sich alle Sichten aktualisieren und die Daten des neuen Berichts verwenden. In diesem Abschnitt wird zunächst die Infrastruktur beschrieben, die es den Usus.NET-Oberflächen erlaubt, eine gemeinsame Datenquelle zu verwenden. Anschließend werden die Implementierungen der Ansichten beschrieben, die bereits für das Usus für Eclipse vorgestellt wurden.

9.1.1 Hubs

Usus.NET besteht aus mehreren Fenstern mit einer gemeinsamen Datenquelle. Jedes Fenster enthält ein Steuerelement vom Typ `UserControl`¹. Um alle diese Steuerelemente mit einem zentralen Objekt zu verknüpfen, bietet die Bibliothek `Usus.net.View` die Klasse `ViewHub` an, welche nur in Form einer einzigen Instanz existieren kann. Die Ansichten `Cockpit`, `Hotspots`, `Distribution` und `CleanCode` werden von der `ViewFactory` in Kombination mit den entsprechenden `ViewModels` erzeugt. Das `ViewHub`-Objekt wird dabei bei den `ViewModels` bekannt gemacht. Diese können sich dann für die Events `AnalysisStarted` und `MetricsReady` registrieren. Ein Aufruf der `TryStartAnalysis`-Methode, mit einer Liste an `Assembly`-Pfadern, startet die `statische Code-Analyse` und teilt allen registrierten Objekten das neue Objekt vom Typ `MetricsReport` mit. In Abbildung 9.1 sind die Beziehungen zwischen den Views und den `ViewModels` in Bezug auf die Hub-Infrastruktur verdeutlicht. Die Oberflächen besitzen eigene `ViewModels` und implementieren damit das MVVM-Muster, welches Josh Smith in seinem Artikel [Smi09b] ausführlich beschreibt. Nachdem die Steuerelemente erzeugt wurden, erreichen die Events

¹Mehr Informationen: „Windows Presentation Foundation“ <http://msdn.microsoft.com/de-de/library/ms754130.aspx>

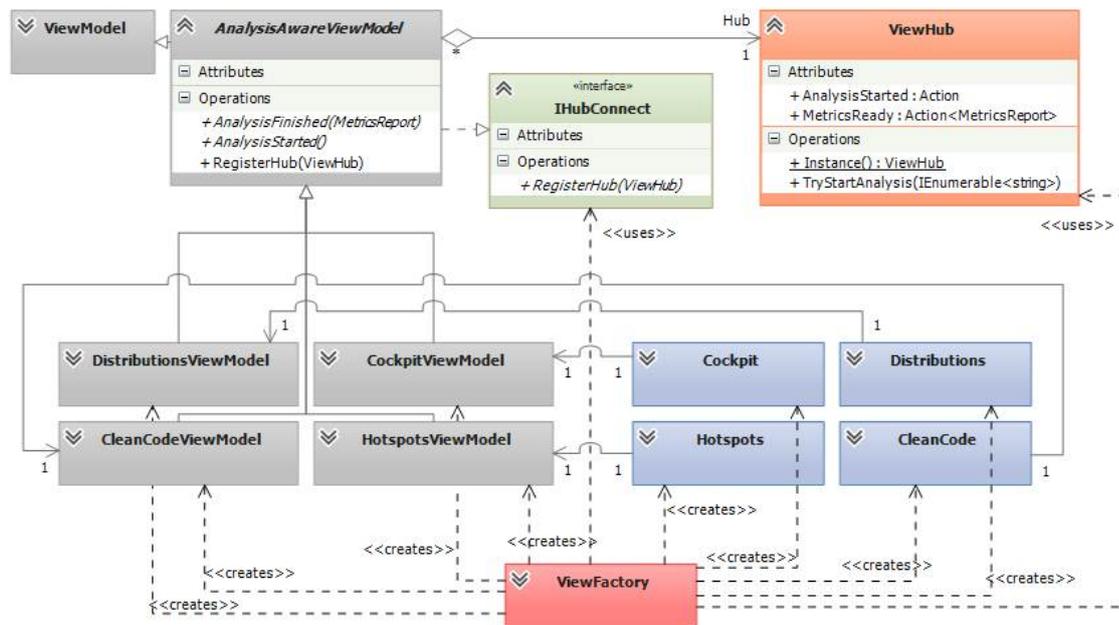


Abbildung 9.1: Oberflächen und ViewModels in Verbindung mit dem ViewHub-Objekt

`AnalysisStarted` und `MetricsReady` des einen `ViewHub`-Objekts, die ViewModels aller Oberflächen in `Usus.NET`. In der `TryStartAnalysis`-Methode wird sichergestellt, dass eine `Analyse` nur dann gestartet werden kann, wenn sie nicht schon läuft. Wie die `Analyse` dann gestartet wird, ist in Listing 9.1 zu sehen.

Listing 9.1: Start der statischen Code-Analyse in der `TryStartAnalysis`-Methode

```

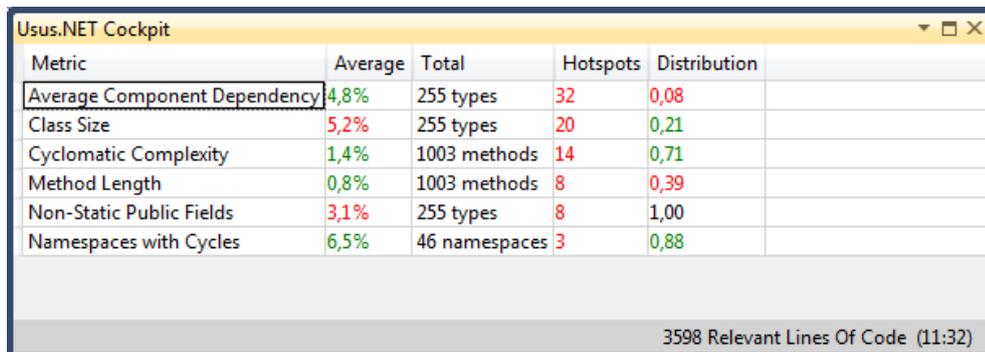
1 AnalysisStarted ();
2 ThreadPool.QueueUserWorkItem((c) =>
3 {
4     MetricsReport metrics = Analyze.PortableExecutable(files.ToArray());
5     MetricsReady(metrics);
6 });

```

Zu Beginn wird das Event `AnalysisStarted` gefeuert. Alles weitere findet asynchron in einem eigenen Thread statt, der über den `ThreadPool` erzeugt wird. Dadurch muss der Aufrufer der `TryStartAnalysis`-Methode nicht warten, da die `Analyse` im Hintergrund ausgeführt wird. Die Klasse `Analyse` ist bereits aus Abschnitt 8.1 bekannt. Sobald der Bericht vom Typ `MetricsReport` erzeugt wurde, wird dieser über das Event `MetricsReady` an alle ViewModels übergeben. Diese müssen sich beim Anzeigen der Daten des Berichts nur noch darum kümmern, dass sie die Daten mit dem Thread der Oberfläche synchronisieren. Die grafischen Steuerelemente und die Factory befinden sich im Namespace `andrena.Usus.net.View`. Im Namespace `andrena.Usus.net.View.Hub` befindet sich die Hub-Infrastruktur. Sämtliche ViewModels der Oberflächen sind im Namespace `andrena.Usus.net.View.ViewModels` zu finden.

9.1.2 Cockpit

Das Fenster mit den am weitesten aggregierte **Metriken** ist „Usus.NET Cockpit“. Nach dem Vorbild aus Unterabschnitt 5.1.1 enthält die Oberfläche eine einfache Tabelle und zeigt alle Statistiken an, die bereits in Unterabschnitt 8.3.2 vorgestellt wurden. Abbildung 9.2 zeigt das Cockpit-Steuererelement als Fenster in Visual Studio. Dieses Fenster verhält



Metric	Average	Total	Hotspots	Distribution
Average Component Dependency	4,8%	255 types	32	0,08
Class Size	5,2%	255 types	20	0,21
Cyclomatic Complexity	1,4%	1003 methods	14	0,71
Method Length	0,8%	1003 methods	8	0,39
Non-Static Public Fields	3,1%	255 types	8	1,00
Namespaces with Cycles	6,5%	46 namespaces	3	0,88

3598 Relevant Lines Of Code (11:32)

Abbildung 9.2: Usus.NET Cockpit-Fenster

sich wie alle anderen Fenster und kann in Visual Studio beliebig andockt werden. Das ViewModel reagiert auf einen verfügbaren Metrikbericht, indem es die Einträge der Tabelle durch neue Einträge für die Statistiken ersetzt. Wenn die neuen Werte kleiner sind, werden sie als Verbesserung grün angezeigt, während größere Werte als Verschlechterung rot dargestellt werden. Per Doppelklick auf die Tabelle lassen sich die Methoden anzeigen, die sich seit dem letzten Kompilieren geändert haben. Usus.NET erlaubt es dann auch zu eben diesen Methoden zu springen. Dafür wird der gleiche Mechanismus wie für die Hotspots aus Unterabschnitt 9.1.4 verwendet. Als Tabelle wird ein `DataGrid`-Objekt verwendet. Neben der Anzahl der Hotspots wird in der Spalte „Distribution“ zu jeder **Metrik** das λ der geometrischen Verteilung angezeigt, das ausführlicher in Abschnitt 10.1 beschrieben wird. Hier ist ein größerer Wert ein besserer Wert. Außerdem ist mit einem Blick in die Statusleiste ersichtlich, wie viele relevante Codezeilen die analysierte Codebasis enthält. Unterabschnitt 11.2.1 beschreibt wie sich dieser Wert zusammensetzt.

9.1.3 Info

Das nächste Usus.NET-Fenster entspricht dem Usus Info-Fenster aus Unterabschnitt 5.1.2. Dieses Fenster zeigt die **Metriken** einer Methode in Verbindung mit der Klasse an. Im Gegensatz zu der Eclipse Variante, lässt sich dieses Fenster nicht mit dem Shortcut `Ctrl-U` öffnen. Usus.NET platziert einen kleinen Button rechts neben jeder Methodendeklaration, wie in Abbildung 9.3 dargestellt. Ein einfacher Mausklick öffnet das Usus.NET Info-Fenster und zeigt die **Metriken** der entsprechenden Methode an. Die **Metriken** der Klasse sind ebenfalls sichtbar. Wie der Button im Code-Editor angezeigt wird, ist in Unterabschnitt 9.2.5 beschrieben. Der Button erscheint nur neben Methoden und nicht neben Properties oder Klassen. Daher lässt sich das Info-Fenster auch nur für Methoden öffnen. Dies ist ausreichend, da die **Klassenmetriken** ja ebenfalls angezeigt werden. Das Info-Fenster in der Eclipse Variante lässt sich zusätzlich auch für Klassen direkt öffnen. Wenn der Entwickler auf den Button klickt, werden die Zeilennummer sowie der Name der Datei an das ViewModel des Info-Fensters übertragen. Beide Informationen kommen dort in Form eines

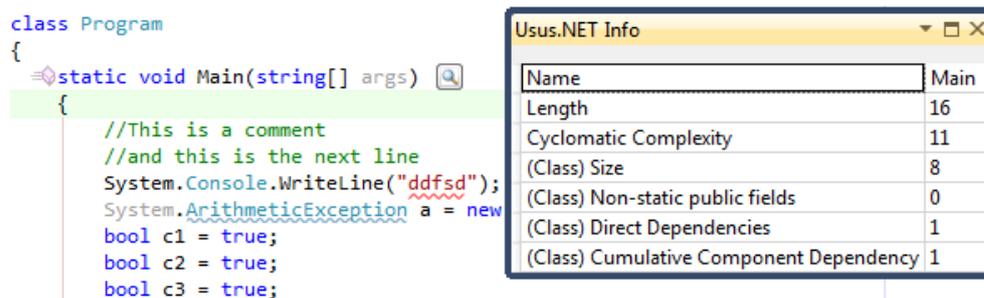


Abbildung 9.3: Usus.NET Info-Fenster

LineLocation-Objekts an. Obwohl die Editor-Buttons und das Fenster in separaten Erweiterungsprojekten implementiert sind, ist diese Übertragung über das in Unterabschnitt 9.2.3 beschriebene Event-System möglich. Um den Methoden- und Klassenbericht zu einer Datei und Zeile zu ermitteln, wird die Klasse `CurrentMethodInfoCalculator` verwendet. Jeder Methodenbericht erhält durch die Metrikberechnung aus Abschnitt 8.1 die Zeileninformationen der `pdb`-Datei in Form eines `SourceCodeLocation`-Objekts. Usus.NET kann dann anhand der Zeilennummer und dem Dateipfad des geklickten Button, den entsprechenden Methodenbericht finden und dessen **Metriken** anzeigen.

9.1.4 Hotspots

Abbildung 9.4 zeigt alle Hotspots in der Codebasis. Wie die Oberfläche in Unterabschnitt

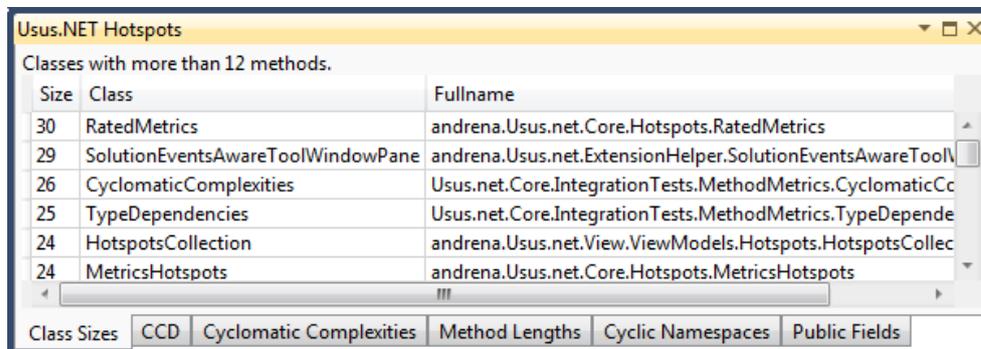


Abbildung 9.4: Usus.NET Hotspots-Fenster

5.1.3, lassen sich zu jeder **Metrik** genau die Methoden, Klassen und Namespaces anzeigen, dessen **Metriken** über den definierten Schwellen aus Abschnitt 5.2 liegen. Dazu wird die API verwendet, die bereits in Unterabschnitt 8.3.1 vorgestellt wurde. Diese Problemfälle sind nach **Metrik** gruppiert und lassen sich über die Registerkarte der entsprechenden **Metrik** anzeigen. Mit einem Doppelklick auf einen Methoden- oder Klassen-Hotspot springt Visual Studio an die Problemstelle im Quellcode. Dies ist möglich, da alle Methodenberichte ja ein `SourceCodeLocation`-Objekt beinhalten. Das ViewModel von „Usus.NET Hotspots“ verwendet das Interface `IJumpToSource`, um an beliebige Zeilen von beliebigen Dateien zu springen. Auf der Seite von Visual Studio implementiert das Fenster, das das grafische Hotspots-Element enthält, dieses Interface. Wie eine Zeile in einer Visual Studio-Erweiterung angesprungen werden kann, beschreibt Unterabschnitt 9.2.4. Listing 9.2 zeigt

den Sprung von der Seite des Hostspots-ViewModel.

Listing 9.2: Zu der Methode eines `MethodMetricsReport`-Objekt springen

```

1 if (Report.SourceLocation.IsAvailable)
2     jumper.JumpToFileLocation(
3         Report.SourceLocation.FileName,
4         Report.SourceLocation.Line, true);

```

Die Variable `Report` enthält eine Referenz auf das `MethodMetricsReport`-Objekt eines Hotspots. Der letzte Parameter des Methodenaufrufs gibt an, dass die entsprechende Zeile selektiert werden soll und damit schneller sichtbar wird. Die `pdb`-Datei einer `Assembly` enthält keine Zeileninformationen für Klassen. Um per Doppelklick zu einem Klassen-Hotspot zu springen, ermittelt das `ViewModel` die erste Methode dieser Klasse und springt zu dieser. Wenn ein Typ keine anspringbaren Methoden enthält, kann `Usus.NET` zu diesem Typen nicht springen. Ein Doppelklick auf einen Namespace-Zyklus-Hotspot öffnet den Dialog aus Abbildung 9.5. In der linken Spalte werden alle Namespaces in eben diesem

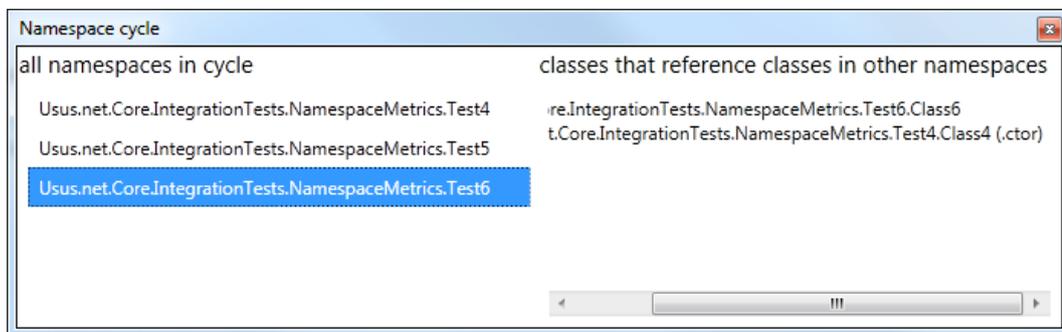


Abbildung 9.5: `Usus.NET` Namespace-Zyklus-Dialog

Zyklus anzeigt. Per Doppelklick auf einen dieser Namespaces werden in der rechten Spalte die Typen angezeigt, die einen Typen in einem der anderen Namespaces referenzieren. Zusätzlich wird der dort referenzierte Typ angegeben und von welcher Methode diese Referenz ausgeht. Auch hier kann mit einem Doppelklick auf einen der Typen in der rechten Spalte direkt zu der Quellcodedatei, die den Typen enthält, gesprungen werden. Dadurch können die Klassen, die den Zyklus erzeugen schnell gefunden werden.

9.1.5 Histogramm

Abbildung 9.6 zeigt die Häufigkeitsverteilungen aller Berichte des Objektmodells aus Abschnitt 8.2 in Bezug auf die `Metriken` an. Ähnlich wie das Hotspots-Fenster, ist auch das Verteilungs-Fenster mit Registerkarten unterteilt. Um die Histogramme zu bestimmen, werden die Funktionen verwendet, die bereits in Abschnitt 8.4 vorgestellt wurden. Diese `Verteilungen` können anschließend angezeigt werden. Dazu verwendet `Usus.NET` das Spaltendiagramm des WPF Toolkits¹. `Usus.NET` versucht die Histogramme zu interpretieren, was in Abschnitt 10.1 beschrieben wird. Das Ergebnis dieser Interpretation wird am rechten oberen Rand angezeigt.

¹Download und mehr Informationen: „WPF Toolkit - February 2010 Release“ <http://wpf.codeplex.com/releases/view/40535>

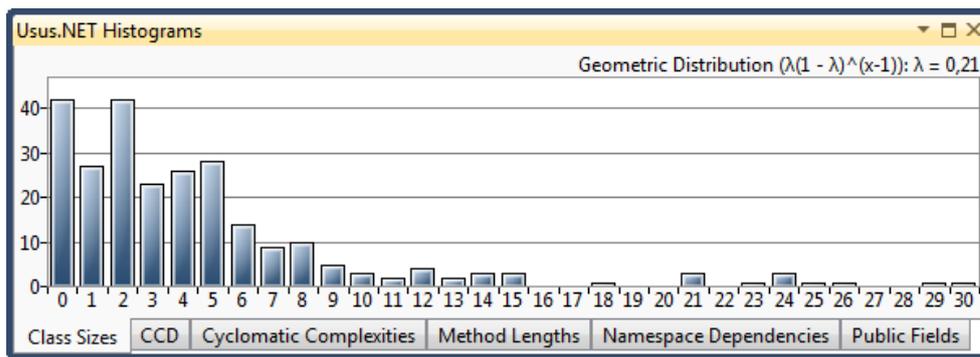


Abbildung 9.6: Usus.NET Histogramm-Fenster

9.1.6 Class Graph & Package Graph

In Unterabschnitt 5.1.5 wurde vorgestellt, wie Usus für Eclipse die Abhängigkeitsgraphen auf Klassen- und Paketebene visualisiert. Usus.NET erstellt diese Graphen ebenfalls und veröffentlicht sie als Teil des Objektmodells, wie in Abschnitt 8.2 beschrieben. Eine grafische Visualisierung dieser Graphen in Form eines Fensters in Visual Studio, wird von Usus.NET noch nicht unterstützt. Dieses Fenster kann im zeitlichen Rahmen dieser Master-Thesis nicht implementiert werden, da dafür eine zeitintensive Evaluierung der Darstellungsmöglichkeiten erforderlich wäre.

9.1.7 Clean Code-Grade

Ein Usus.NET-Fenster, das es nicht in der Eclipse-Version gibt, ist „Usus.NET Clean-Clode“. Damit ist Alexander Zeitler’s Idee die Clean Code-Grade von Ralf Westphal und Stefan Lieser [WL11] immer direkt in der IDE zu sehen, in Usus.NET integriert. Zeitlers Idee wurde bereits in Abschnitt 6.3 vorgestellt. Abbildung 9.7 zeigt die Liste, mit den verschiedenen Graden. Zu jedem Grad werden die Prinzipien (Unterabschnitt 6.3.1) und



Abbildung 9.7: Usus.NET Clean Code-Grade-Fenster

Parktiken (Unterabschnitt 6.3.2) als Hyperlinks aufgelistet. Über diese Hyperlinks lassen sich die Internetseiten von Westphal und Lieser öffnen, die ausführliche Beschreibungen der Grade enthalten.

9.1.8 SQI

Ein weiteres Usus.NET-Fenster, welches es in der Eclipse Version ebenfalls nicht gibt, ist „Usus.NET SQI“. Abbildung 9.8 zeigt die Parameter des **Softwarequalitätsindex (SQI)** in der Tabelle, sowie den berechneten **SQI** einer Beispielanwendung. Wie diese Berechnung funktioniert und was die Parameter bedeuten, wird in Kapitel 11 ausführlich beschrieben. Nach jedem Kompilervorgang trägt Usus.NET alle Werte automatisiert ein, bis auf die

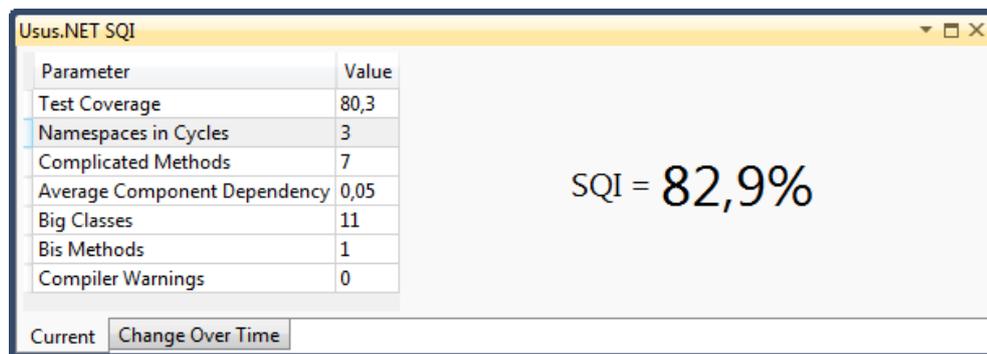


Abbildung 9.8: Usus.NET SQI-Fenster

Testabdeckung. Da Usus.NET die prozentuale Testabdeckung noch nicht automatisiert bestimmt, kann dieser Wert vom Entwickler manuell eingetragen werden. Der **SQI**-Wert wird daraufhin sofort aktualisiert und in groß in der Registerkarte „Current“ angezeigt. Die Registerkarte „Change Over Time“ zeigt an, wie sich der **SQI** über die letzten Build-Vorgänge verändert hat. Die Entwicklung des **SQI** wird für jede Solution separat verwaltet. Wenn Visual Studio eine neue Solution analysiert, werden deren **SQI**-Werte angezeigt. Wie diese Veränderungen gespeichert werden, wird in Abschnitt 11.3 ausführlicher beschrieben.

9.2 Visual Studio Integration

Nachdem alle Oberflächen von Usus.NET beschrieben wurden, beschäftigt sich dieser Abschnitt mit der tatsächlichen Integration der Views in Visual Studio. Jede Oberfläche erhält ein eigenes Visual Studio-Fenster, welches beliebig in der IDE verschoben, ange-dockt und angepasst werden kann. Diese Fenster sind alle in separaten Projekten vom Typ *VSPackage* implementiert. Die sechs Fensterprojekte sind *Usus.net.Cockpit* (1), *Usus.net.Current* (2), *Usus.net.Hotspots* (3) und *Usus.net.Distributions* (4). Dazu kommen *Usus.net.CleanCode* (5) und *Usus.net.Sqi* (6). Zusätzlich zeichnet das Info-Fenster (*Usus.net.Current*) Buttons direkt neben dem Quellcode im Visual Studio-Editor. Diese Buttons sind in dem *Editor Adornment*-Projekt *Usus.net.EditorAdornment* (7) definiert. Das Projekt *Usus.net.Menus* (8) ist ein weiteres *VSPackage*, welches das Usus.NET-Menü in die Menüleiste von Visual Studio integriert. Alle diese acht Erweiterungsprojekte werden vom dem *VSPackage*-Projekt *Usus.net* (0) zusammengefasst. Dieses Projekt enthält keinen Quellcode und erzeugt auch keine eigene Assembly. Das *Usus.net*-Projekt erzeugt die VSIX-Datei der kompletten Usus.NET-Erweiterung. Eine VSIX-Datei ist eine ZIP-Datei, welche alle benötigten *Assemblies* und das Erweiterungs-manifest enthält. Durch die Aufteilung von Usus.NET in separate Projekte, braucht Visual Studio nicht die komplette Erweiterung zu laden sondern lediglich die Fenster, die

tatsächlich benutzt werden. Ein weiterer Vorteil der getrennten Projekte ist, dass ein potenzieller Fehler in einem VSPackage sich nicht auf die anderen auswirkt. Nur das fehlerhafte VSPackage wird dann von Visual Studio beendet, während die anderen Fenster, die sich ja in anderen VSPackages befinden, weiter lauffähig bleiben.

Im Folgenden werden die allgemeinen Implementierungsdetails der Usus.NET-Fenster vorgestellt. Dabei wird beschrieben, wie diese Fenster auf das Build-Ereignis in Visual Studio reagieren können. Weiterhin wird die Integration aller Erweiterungsprojekte in einen einzigen Menüpunkt behandelt, der außerdem eine Kommunikation zwischen den Fenstern ermöglicht. Abschließend wird gezeigt wie Erweiterungen an beliebige Codezeilen springen und wie in diesen Codezeilen grafische Elemente dargestellt werden können.

9.2.1 Events

Um auf Ereignisse in Visual Studio zu reagieren, wird eine Referenz auf das DTE2-Objekt benötigt. Laut Microsoft ist dieses Objekt, als Bestandteil des `EnvDTE`-Namespace, Teil der COM¹-Bibliothek, die für die Automatisierung von Visual Studio verwendet werden kann [Mic10b]. Listing 9.3 zeigt, wie eine Referenz zu diesem Objekt erstellt werden kann. Dafür muss dieser Code innerhalb einer Klasse, die von `ToolWindowPane` erbt, ausgeführt werden, sodass sich `base` auf ein Objekt vom Typ `WindowPane` bezieht.

Listing 9.3: Zentrales Automatisierungsobjekt von Visual Studio referenzieren

```
1 EnvDTE80.DTE2 dt2 = base.GetService(typeof(SDTE)) as EnvDTE80.DTE2;
```

Sobald eine Referenz zu dem DTE2-Objekt verfügbar ist, kann die aktuelle Solution in der Visual Studio-Instanz mit `dt2.Solution` ermittelt werden. Über dieses `Solution`-Objekt können dann alle Projekte in dieser Solution bestimmt werden. Mit `dt2.Events` kann außerdem ein Objekt erhalten werden, über welches Eventhandler für sämtliche Ereignisse definiert werden können. Für Usus.NET ist das `OnBuildDone`-Ereignis des `dt2.Events.BuildEvents`-Objekts am wichtigsten. Nur wenn der Kompilervorgang erfolgreich beendet wird, soll die *statische Code-Analyse* gestartet werden. Ob der Build erfolgreich war, kann mit der `LastBuildInfo`-Eigenschaft ermittelt werden, die über das `dt2.Solution.SolutionBuild`-Objekt aufgerufen wird.

9.2.2 Fenster

Ein Visual Studio-Fenster ist eine Klasse, die von `ToolWindowPane` erbt. Alle Fenster von Usus.NET sollen auf das Build-Ereignis reagieren und alle Projekte der aktuellen Solution und deren erzeugte *Assemblies* bestimmen können. Um diese Logik für alle Usus.NET-Fenster wiederverwenden zu können, stellt Usus.NET einige Oberklassen zur Verfügung, die die benötigten Funktionen bereitstellen. Diese Klassen befinden sich im Namespace `andrena.Usus.net.ExtensionHelper`. Abbildung 9.9 zeigt `DtAwareToolWindow` als oberste Klasse der Hierarchie. Diese Klasse erbt von `ToolWindowPane` und stellt das in Unterabschnitt 9.2.1 beschriebene DT2-Objekt zur Verfügung. Zusätzlich erbt die Klasse `SolutionAwareToolWindowPane` von `DtAwareToolWindow` und nutzt das dort verfügbare Automatisierungsobjekt, um eine Referenz auf die aktuelle Solution, sowie alle Projekte dieser Solution zu ermitteln. Da die Projekte alle als COM-Objekte vorliegen, wer-

¹Mehr Informationen: „What is COM?“ <http://www.microsoft.com/com/default.msp>

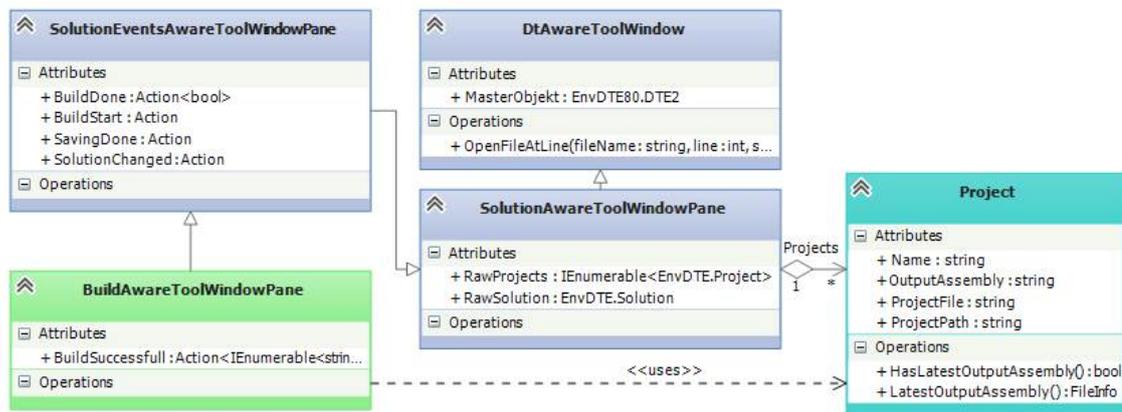


Abbildung 9.9: Oberklassen der Usus.NET-Fenster

den diese in einfache Objekte vom Typ `Project` konvertiert. Weiterhin erbt die Klasse `SolutionEventsAwareToolWindowPane` von `SolutionAwareToolWindowPane` und verwendet die COM-Objekte der Oberklassen um Ereignisse in Visual Studio verfügbar zu machen. Diese Ereignisse wurden bereits in Unterabschnitt 9.2.1 erwähnt. Alle Klassen die von dieser Klasse ableiten, können damit informiert werden, wenn der Entwickler einen Speicher- oder Kompilervorgang ausführt. Zusätzlich können alle betroffenen Projekte dieses Kompilervorgangs bestimmt werden. Um über die tatsächlich erzeugten `Assembly`-Dateien im Falle eines Builds benachrichtigt werden zu können, erbt die Klasse `BuildAwareToolWindowPane` abschließend von `SolutionEventsAwareToolWindowPane`. Diese Klasse stellt ein Ereignis zur Verfügung, welches nur nach einem erfolgreichen Kompilieren ausgeführt wird und welchem eine Liste an Dateipfaden übergeben wird. Diese Liste enthält zu jedem Projekt die aktuellste `Assembly`. Wenn ein Projekt keine `Assembly`-Datei erzeugt, wird es ignoriert. Die aktuellste `Assembly` wird von der Methode `LatestOutputAssembly` der Klasse `Project` gefunden. Auf diese Weise wird sichergestellt, dass immer der aktuellste Build berücksichtigt wird, unabhängig davon, ob die Dateien im build- oder debug-Verzeichnis erstellt werden.

Die in Abschnitt 9.1 vorgestellten Oberflächen von Usus.NET sind also Fensterklassen vom Typ `BuildAwareToolWindowPane`. Diese Fenster erzeugen in ihren Konstruktoren die entsprechenden ViewModels, registrieren sich für das `BuildSuccessfull`-Ereignis und leiten dessen Liste mit Dateipfaden an die `TryStartAnalysis`-Methode der Hub-Infrastruktur aus Unterabschnitt 9.1.1 weiter.

Listing 9.4: Initialisierung eines Usus.NET-Fenster

```

1 BuildSuccessfull += files => ViewHub.Instance.TryStartAnalysis(files);
2 base.Content = ViewFactory.CreateCockpit(ViewHub.Instance);

```

Alle Fenster werden also über die Build-Vorgänge informiert und benachrichtigen die `ViewHub`-Instanz. Listing 9.4 zeigt diese Initialisierung exemplarisch im Konstruktor des Cockpit-Fenster. Usus.NET ist als Erweiterung im Sinne von Unterabschnitt 2.1.3 implementiert. Dadurch können alle Usus.NET-Fenster als separate Projekte mit der Projektvorlage `VSPackage` realisiert werden. Das `ViewHub`-Objekt ist für alle diese Fenster aber

das gleiche. In einem speziellen Container-Projekt werden alle VSPackage-Projekte von Usus.NET dann zu einer VSIX-Datei zusammengefasst.

9.2.3 Menüpunkt

Standardmäßig werden alle selbsterstellten *Visual Studio Tool Windows* unter **View / Other Windows** aufgeführt. Für Usus.NET sollen die in Abschnitt 9.1 beschriebenen Fenster aber einen besonderen Stellenwert genießen. Microsoft beschreibt in der MSDN Library wie ein solcher Menüpunkt in Form eines VSPackage-Projekts implementiert werden kann [Mic10a]. Abbildung 9.10 zeigt diesen Menüeintrag direkt in Visual Studio. Um die ein-

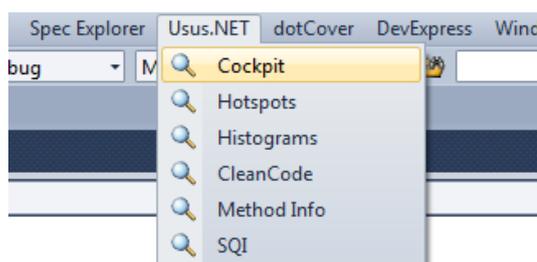


Abbildung 9.10: Usus.NET-Menüpunkt in der Menüleiste von Visual Studio

zelnen Fenster zu öffnen, muss das VSPackage des Fensters erst gestartet werden. Dies ist der Fall, da Visual Studio die Erweiterungen erst dann startet, wenn sie das erste Mal durch den Entwickler verwendet werden. Microsoft beschreibt in der MSDN Library wie ein VSPackage manuell gestartet werden kann [Mic10d]. Sobald das VSPackage eines Fensters gestartet wurde, kann dieses Fenster angezeigt werden. Um das Anzeigen von Usus.NET-Fenstern zu vereinfachen, steht die Klasse `UsusNetWindow` im Namespace `andrena.Usus.net.ExtensionHelper` zur Verfügung. Diese Klasse enthält für jedes Usus.NET-Fenster die eindeutige Identifikationsnummer des dazugehörigen VSPackage in Form einer GUID¹. Listing 9.5 zeigt wie das VSPackage anhand dieser Identifikationsnummer geladen werden kann.

Listing 9.5: Anzeigen eines Usus.NET-Fenster

```
1 Guid packageToBeLoaded = new Guid(guid);
2 shell.LoadPackage(ref packageToBeLoaded, out package);
3 GlobalEventManager.Instance.FireEvent(guid, parameter);
```

Nachdem das VSPackage des Fensters in Zeile 2 geladen wurde, kann das Fenster in Visual Studio geöffnet werden. Dazu verwendet Usus.NET ein einziges Objekt vom Typ `GlobalEventManager`. Sobald das VSPackage eines Fensters geladen wird, registriert es eine Ereignisbehandlung bei der einzigen Instanz der `GlobalEventManager`-Klasse. Es ist eine Konvention in Usus.NET, dass das Paket dafür seine GUID als Trigger verwendet. Die Ereignisbehandlung kann dann mit der GUID des VSPackage ausgeführt werden, was in Listing 9.5 in Zeile 3 zu sehen ist. Listing 9.6 zeigt exemplarisch, wie das VSPackage des Usus.NET Cockpit-Fenster die Ereignisbehandlung zum Anzeigen des Fensters registriert. Alle VSPackage-Projekte, die zur der einen VSIX-Datei zusammengefasst werden,

¹Mehr Informationen: „Guid Structure“ <http://msdn.microsoft.com/en-us/library/system.guid.aspx>

teilen sich also die gleiche Instanz des `GlobalEventManager`-Objekts. Fenster und Menüs können über Package Grenzen hinweg kommunizieren, indem Ereignisse definiert und ausgeführt werden. Dies ist die einzige Möglichkeit der Fenster-Menü-Kommunikation im Kontext von Usus.NET.

Listing 9.6: Registrierung eines Ereignisses beim `GlobalEventManager`-Objekt

```
1 GlobalEventManager.Instance.RegisterEvent(
2     UsusNetWindow.Cockpit,
3     p => ShowToolWindow(null, null));
```

9.2.4 Zeilensprünge

In Unterabschnitt 9.1.4 wurde das Hotspots-Fenster vorgestellt. Nach einem Doppelklick auf einen Hotspot soll Visual Studio zu der entsprechenden Methode oder Klasse springen. Listing 9.7 zeigt, wie Visual Studio mithilfe des in Unterabschnitt 9.2.1 vorgestellten `DTE2`-Objekts eine Datei öffnen und zu einer Zeile in dieser Datei springen kann. Das `DTE2`-Objekt wird durch das `MasterObjekt`-Property der Klasse `DtAwareToolWindow` ermittelt.

Listing 9.7: Datei öffnen und an Zeile springen

```
1 MasterObjekt.ItemOperations.OpenFile(fileName,
2     EnvDTE.Constants.vsViewKindTextView);
3 var file = MasterObjekt.ActiveDocument.Selection as EnvDTE.TextSelection;
4 file.GotoLine(fileLine, selectLine);
```

Dieser Code befindet sich in der `DtAwareToolWindow`-Klasse und lässt sich über die Methode `OpenFileAtLine` aufrufen. Die Variable `fileName` enthält den vollständigen Pfad der zu öffnenden Datei und die Variable `fileLine` die Zeilennummer der zu markierenden Zeile. Ob der Code in der Zeile tatsächlich markiert werden soll, kann über den Parameter `selectLine` festgelegt werden. Sollte Visual Studio das Dokument bereits geöffnet haben, bringt es dies bei Bedarf in den Vordergrund und springt nur zu der betroffenen Zeile.

9.2.5 Editor Adornment

In Unterabschnitt 9.1.3 wurde beschrieben, dass das Info-Fenster von Usus.NET über Buttons aufgerufen wird, die sich direkt neben den Methodensignaturen befinden. Das Erweiterungsmodell von Visual Studio unterstützt das Zeichnen von beliebigen WPF-Elementen im Code-Editor. Über die Projektvorlage *Editor Adornment* (Editor-Verzierung) kann eine Erweiterung erstellt werden, welche auf die aktuell sichtbaren Codezeilen reagieren kann. In dem von der Projektvorlage generierten Code existiert die Methode `CreateVisuals`, die einen Parameter vom Typ `ITextViewLine` übergeben bekommt. In dieser Methode kann die Logik implementiert werden, die ein grafisches Element in der übergebenen Zeile anzeigt. Im Fall von Usus.NET ist dies ein kleiner Button, der in der Klasse `CodeTag` implementiert ist. Dieser Button wird nur neben einer Methodensignatur dargestellt. Usus.NET stellt dies durch einen regulären Ausdruck sicher. Dieser Ausdruck besteht aus vier Komponenten. Die erste Komponente ist in Listing 9.8 zu sehen und stellt eine Form dar, die die anderen drei Komponenten mehrfach wiederverwendet.

Listing 9.8: Regulärer Ausdruck einer gültigen Methodensignatur in C#

```
1 ^{0}{1} {1}\\\((((out |ref |this |params )?{1} {2})?(,[ ]?(out |ref |params )?
2 {1} {2})*))\)\$
```

{0} ist der Platzhalter für die zweite Komponente, während {1} für die dritte und {2} für die vierte Komponente steht.

Listing 9.9: Regulärer Ausdruck einer gültigen Zugriffseinschränkung in C#

```
1 (private | protected | public | internal )?(static | virtual new | virtual | new
2 | override | override sealed )?
```

Listing 9.9 zeigt die zweite Komponente des regulären Ausdrucks, welche die Sichtbarkeit einer Methode beschreibt. Die dritte Komponente beschreibt die Syntax eines Typen und die vierte beschreibt einen Variablennamen. Beide Ausdrücke sind in Listing 9.10 abgebildet.

Listing 9.10: Regulärer Ausdruck eines gültigen Klassen- und Parameternamen in C#

```
1 Klassenname: ([_a-zA-Z]+[\\\. _a-zA-Z0-9\\[\]]*( <.*> )?)
2 Parametername: ([_a-zA-Z]+[_a-zA-Z0-9]*([ ]?=[ ]?[^\s,<> ]*)?)
```

Zusammengesetzt entsteht ein Ausdruck, der nahezu alle gültigen Methodensignaturen zuverlässig erkennen kann und gleichzeitig weitgehend immun gegenüber Fehlern bleibt. Eine 100%ige Korrektheit wurde nicht angestrebt und würde im zeitlichen Rahmen dieser Master-These auch nicht erreicht werden können. Listen, generische Typen, Standardwerte und automatische `params`-Arrays werden aber alle korrekt erkannt. Die Erkennung der Methoden wurde mit Unit Tests spezifiziert und getestet.

Wenn eine Methode als solche erkannt wurde, wird die Zeilennummer und der Pfad der Codedatei ermittelt und innerhalb des Buttons gespeichert. Beide Informationen sind erforderlich, um eine Methodensignatur in der Quellcodedatei mit einem Methodenbericht des Usus.NET-Objektmodells zu verbinden. Usus.NET kann diese Zuordnung noch nicht anhand der Signatur vornehmen. Die Signatur, die durch die [statische Code-Analyse der Assembly](#) mittels [CCI Metadata](#) bestimmt wird, besteht aus vollqualifizierten Namen und unterscheidet sich im schlimmsten Falle zu sehr von der Signatur die im Quelltext steht. Obwohl ein Ähnlichkeitsvergleich prinzipiell möglich ist, würde dies doch den zeitlichen Rahmen dieser Master-These sprengen. Idealerweise könnte dazu der reguläre Ausdruck aus Listing 9.8, 9.9 und 9.10 mit einer leichten Anpassung verwendet werden. Wenn eine pdb-Datei vorhanden ist, funktioniert der zeilenbasierte Ansatz allerdings auch sehr gut. Da die Buttons ja sowieso nur in Visual Studio zu sehen sind und Visual Studio auch die Kompilierung des Codes vornimmt, wird eine solche pdb-Datei auch immer erzeugt. Listing 9.11 zeigt, wie in einer Editor Adornment-Erweiterung die aktuelle Zeilennummer ermittelt werden kann.

Listing 9.11: Die tatsächlichen Zeilennummer eines `ITextViewLine`-Objekts

```
1 int lineNumber = _view.TextSnapshot
2     .GetLineNumberFromPosition(line.Start.Position) + 1;
```

Die Variable `line` enthält das Objekt vom Typ `ITextViewLine`. Die Variable `_view` enthält eine Referenz auf ein `IWpfTextView`-Objekt. Listing 9.12 zeigt wie zusätzlich noch der Pfad der aktuellen Datei ermittelt werden kann. Es ist zu beachten, dass dafür die Zeile nicht erforderlich ist. Visual Studio kann den Dateinamen allein anhand des `IWpfTextView`-Objekts bestimmen. Nachdem jede Methodensignatur eine Kombination aus Zeilennummer und Dateinamen zugewiesen bekommen hat, ist der Button bereit auf das Klickereignis zu reagieren.

Listing 9.12: Bestimmung des Datei-Pfads eines `IWpfTextView`-Objekts

```
1 ITextDocument document;  
2 string filePath = string.Empty;  
3 if (_view != null && _view.TextDataModel.DocumentBuffer.Properties  
4     .TryGetProperty(typeof(ITextDocument), out document))  
5     string filePath = document.FilePath;
```

Sobald der Entwickler auf den Button klickt, wird die Zeileninformation verpackt und über die Event-Infrastruktur aus Unterabschnitt 9.2.3 an das `VSPackage` geschickt, welches das Info-Fenster enthält und daraufhin anzeigt.

Listing 9.13: Öffnen des Info-Fensters mit einem `LineLocation`-Objekt

```
1 GlobalEventManager.Instance.FireEvent(UsusNetWindow.Current,  
2     new LineLocation { Line = lineNumber, File = filePath });
```

Listing 9.13 zeigt wie ein Methodensignatur-Button seine Daten an das Info-Fenster aus Unterabschnitt 9.1.3 sendet. `UsusNetWindow.Current` enthält die GUID des Info-Fensters, sodass standardmäßig die Ereignisbehandlung zum Öffnen des Fensters ausgeführt wird. Diese Behandlung wird auch von dem Menüpunkt verwendet. Im Gegensatz zu dem Menüpunkt, öffnet der Button das Fenster mit einem Parameter. Die Behandlung auf der Seite des Fensters muss also unabhängig davon funktionieren, ob eine Zeileninformation übergeben wird oder nicht.

10 Clean Code-Unterstützung

Eine Anforderung an Usus.NET ist die Unterstützung bei der Entwicklung von **Clean Code** (Ziel 3). Dafür wurde bereits das **Clean Code-Grade-Fenster** in Unterabschnitt 9.1.7 vorgestellt. Dieses Fenster erinnert den Entwickler kontinuierlich an die Prinzipien und Praktiken, die er am besten befolgen sollte. Um ein Maß für das Level an **Clean Code** zu haben, reichen die bisher vorgestellten **Metriken** nicht aus. Robert C. Martin beschreibt in seinem Blog-Post [Mar09b] die beiden **Clean Code-Metriken** *CRAP* und *The Braithwaite Correlation*, welche nicht Bestandteil dieser Ausarbeitung sind. In dieser Master-Thesis wird eine dritte **Metrik** betrachtet, die aus der Häufigkeitsverteilung der **Metriken** berechnet werden kann. Der Parameter der approximierten Verteilungsfunktion soll die **Clean Code-Metrik** von Usus.NET sein.

10.1 Kleine Metriken

In Abschnitt 8.4 wurde beschrieben, dass Usus.NET Histogramme für alle **Metriken** erstellen kann. Dieser Abschnitt beschäftigt sich mit der Analyse ebendieser Metrik-Histogramme und wie die **Metriken** tatsächlich verteilt sind. Nach einer ersten Überlegung folgen beispielsweise **Methodenlängen** einer **Exponentialverteilung**. „Kurz“¹ ist das Erste, was Robert C. Martin über Methoden in Bezug auf **Clean Code** schreibt [Mar09a]. In einem Softwaresystem, das mit einem Bewusstsein für **Clean Code** entwickelt wurde, werden also sehr viele sehr kurze Methoden existieren und wenig lange Methoden. Die **Methodenlänge** nimmt dabei aber nicht linear ab, sondern eher exponentiell. Diese Entwicklung sollte auch im Histogramm der **Methodenlängen** zu sehen sein. Die Funktion der **Exponentialverteilung** ist laut Eric W. Weisstein wie in Formel 10.1 definiert [Wei12b].

$$f_{\lambda}(x) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases} \quad (10.1)$$

Da alle **Metriken** ganze Zahlen sind (es gibt keine **Methodenlänge** mit 1,5 logischen Zeilen), enthalten auch die Metrik-Histogramme nur diskrete Werte auf der x-Achse. Für die analytische Annäherung einer Verteilungsfunktion empfiehlt sich daher auch die Verwendung einer diskreten Verteilung. Laut Eric W. Weisstein ist das diskrete Gegenstück der kontinuierlichen **Exponentialverteilung** die geometrische **Verteilung** [Wei12c]. Die Funktion der geometrischen **Verteilung** ist in Formel 10.2 zu sehen.

$$f_{\lambda}(x) = \begin{cases} \lambda(1 - \lambda)^x & x \geq 0 \wedge \lambda > 0 \wedge \lambda < 1 \\ 0 & \text{sonst} \end{cases} \quad (10.2)$$

Beide Funktionen sind von dem Parameter λ abhängig. Die eigentliche Histogramm-Analyse besteht darin, diesen Parameter so zu bestimmen, dass die Funktionen das Histogramm möglichst genau beschreibt.

¹Seite 34

10.2 Approximation

Eine Methode um das λ zu berechnen ist die Maximum Likelihood-Methode, die Weisstein ebenfalls in einer MathWorld-Veröffentlichung beschreibt [Wei12d]. Mit dieser Methode kann eine Schätzer-Funktion errechnet werden, die den Parameter λ in Abhängigkeit der Werte des Histogramms bestimmt. Sigbert Klinke, Patrick Lehmann, Bernd Rönz, Sibylle Schmerbach und Olga Zlatkin-Troitschanskaia haben den Schätzer für die [Exponentialverteilung](#) bereits bestimmt [KLR⁺06]. Der Schätzer der geometrischen [Verteilung](#) wurde von Künzer, Meister und Nebe berechnet [KMN06]. Beide Schätzer-Funktionen sind identisch und abhängig von der Histogramm-Sequenz. Formel 10.3 zeigt diese Rechenvorschrift zur Bestimmung des λ -Parameters beider [Verteilungen](#).

$$\lambda = \frac{n}{\sum_{i=1}^n x_i} = \frac{1}{\bar{x}} \quad (10.3)$$

Der Parameter λ entspricht also dem reziproken Mittelwert. Je mehr kurze Methoden der Quellcode enthält, desto niedriger ist der Mittelwert. Für ein großes λ ist der Mittelwert klein und umgekehrt. Auf das Histogramm bezogen bedeutet ein großes λ , dass die Exponentialfunktion der [Verteilung](#) schnell fällt (sehr viele kurze und sehr wenige lange Methoden). [Abbildung 10.1](#) zeigt die wünschenswerte [Verteilung](#) in rot und die weniger optimale in blau. Das Schaubild geht von einer Stichprobe mit 100 Werten aus, beispielsweise [Methodenlängen](#). Die Funktion der geometrischen [Verteilung](#) verhält sich genauso,

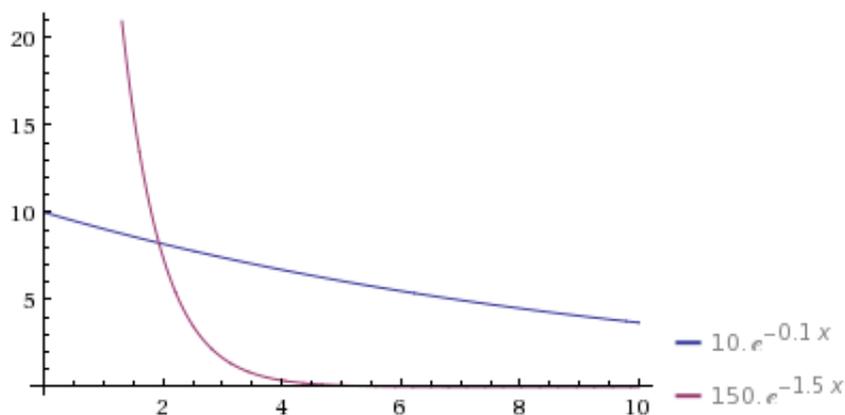
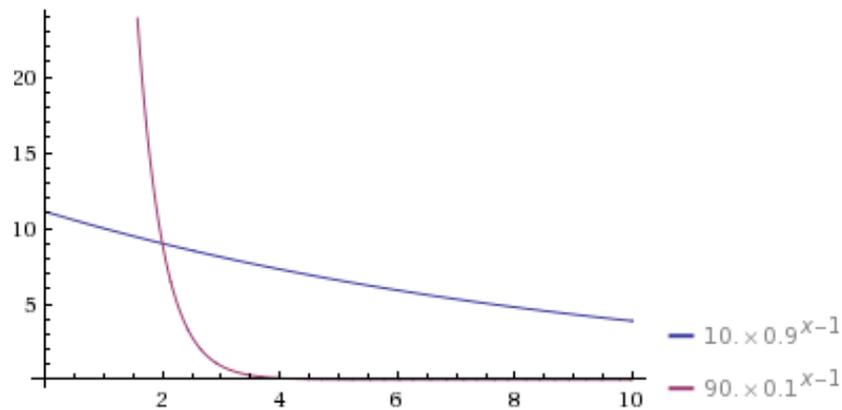


Abbildung 10.1: Exponentialverteilung für großes und kleines λ

da sie den gleichen Maximum Likelihood-Schätzer wie die [Exponentialverteilung](#) hat. [Abbildung 10.2](#) zeigt wieder die wünschenswerte [Verteilung](#) in rot und die weniger optimale in blau. Auch hier besteht die Stichprobe aus 100 Werten. Obwohl die [Exponentialverteilung](#) und die geometrische [Verteilung](#) nicht identisch sind, kann der Parameter, von dem beide abhängig sind, auf die gleiche Art berechnet werden. Das Ergebnis verhält sich in beiden [Verteilungen](#) vergleichbar. In der vorliegenden Master-Thesis wird das λ der geometrischen [Verteilung](#) einer [Metrik](#) als neue [Clean Code-Metrik](#) definiert.

Definition 20 (Lambda der geometrischen [Verteilung](#)) *Das λ der geometrischen [Verteilung](#) ist eine [Metrik](#), die per Approximation der Funktion 10.2 an ein [Metrik-Histogramm](#) ermittelt wird. Je größer das λ , desto eher entsprechen die Werte der betrachteten [Metrik](#) dem [Clean Code-Paradigma](#). Der Wert liegt im Intervall $]0, 1[$.*

Abbildung 10.2: Geometrische Verteilung für großes und kleines λ

Ein großes λ deutet darauf hin, dass die betrachtete Metrik dem Clean Code-Paradigma von Robert C. Martin folgt. Martin befürwortet sehr einfache und sehr kurze Methoden, sowie Klassen mit sehr wenig Abhängigkeiten [Mar09a].

Usus.NET berechnet den λ -Parameter in der Klasse `GeometricalDistributionFitting` nach Formel 10.3. Die Komponenten, die für diese Berechnung zuständig sind, wurden bereits in Abschnitt 8.4 vorgestellt. Das Ergebnis kann anschließend in der Oberfläche aus Unterabschnitt 9.1.5 angezeigt werden. Beispielsweise zeigt Abbildung 9.6 die Verteilung der Klassengrößen einer Beispielanwendung. In Kapitel 12 werden konkretere Beispiele betrachtet und mit Usus.NET analysiert.

10.3 Abweichung

Da das Histogramm von einer stetigen Funktion repräsentiert wird, ist es sehr wahrscheinlich, dass die Funktion mit dem bestimmten Parameter nicht exakt durch die Punkte des Histogramms läuft. Diese Abweichung kann laut Eric W. Weisstein mit den Chi-Squared-Test ermittelt werden [Wei12a]. Formel 10.4 zeigt dieses Maß der Abweichung.

$$\chi^2 = \sum_{i=1}^n \frac{(x_i - f(x_i))^2}{f(x_i)} \quad (10.4)$$

Da das λ der geometrischen Verteilung bekannt ist, kann die Funktion $f(x_i)$ durch die Funktion $f_\lambda(x_i)$ aus Formel 10.2 ersetzt werden. Das berechnete Ergebnis kann über den `Error`-Parameter des `GeometricalDistributionFitting`-Objekts zurückgegeben werden. Die Implementierung dieser Fehlerfunktion ist in der aktuellen Version von Usus.NET noch nicht enthalten. Sobald Usus.NET die Bestimmung von verschiedenen Verteilungen unterstützt, kann mithilfe des Fehlers die Verteilungsfunktion bestimmt werden, die das Histogramm am Besten (mit dem geringsten Fehler) beschreibt. Aufgrund der dadurch bestimmten wahrscheinlichsten Verteilung könnte eine akkuratere Aussage über die Metriken im betroffenen Softwaresystem getroffen werden. Diese weiteren Verteilungen könnten eventuell die Poisson-Verteilung und die Pareto-Verteilung sein. In Usus.NET ist aktuell nur die geometrische Verteilung implementiert.

10.4 Refactoring-Vorschläge

Da die **Metriken** durch die Verteilungsanalyse besser verstanden werden, könnten gezielt Refactorings vorgeschlagen werden, die eine **Verteilung** in eine andere **Verteilung** überführen. Zustände der inneren Softwarequalität können dadurch auf zwei Informationen reduziert werden. Zum Einen die wahrscheinlichste **Verteilung** und zum Anderen die Parameter dieser Verteilung. Anschließend könnten die gewünschten **Verteilungen** und deren Parameter als Endzustand definiert werden. Um diese **Verteilung** und deren Parameter zu erreichen, könnten von allen anderen **Verteilungen** Übergänge in Form von Refactorings definiert werden. Dadurch entsteht ein gerichteter Graph. Sobald Usus.NET die betrachtete Codebasis einem Zustand zuordnet, könnten die Übergänge (Refactorings) ermittelt werden, die zu einem Endzustand (gewünschte **Verteilung** und deren Parameter) führen.

Natürlich ist die **Verteilung** als Repräsentation der inneren Codequalität eine sehr allgemeine **Metrik**. Die damit ermittelten Refactorings müssten also auch sehr allgemein sein. Solche allgemeinen Refactorings könnten beispielsweise „Methode/Klasse X auf Methoden/Klassen X_1 und X_2 aufteilen“ oder „Namespace-Zyklus $X - Y - Z$ auflösen“ sein. Eine weiterführende theoretische Betrachtung oder eine rudimentäre Implementierung dieses Verteilung-Refactoring-Graph, kann im zeitlichen Rahmen dieser Master-Thesis nicht vorgenommen werden.

11 andrena-Softwarequalitätsindex

Dieses Kapitel beschäftigt sich mit dem Anwendungsfall *Software Qualität interpretieren* (Ziel 4). Wie bereits in Kapitel 10 beschrieben, interpretieren Entwickler kleinere **Metriken** in der Regel als bessere **Metriken** und das obwohl bessere **Metriken** nicht automatisch besseren Code beschreiben. In Bezug auf die **Methodenlänge** ist Robert C. Martin in seinem Buch sehr deutlich und verteidigt kurze Methoden [Mar09a]. Ein Softwaresystem, welches evolutionär gewachsen ist, enthält leider nicht nur kurze Methoden sondern auch zunehmend längere. Eine alleinige Interpretation über die Verteilung, wie in Abschnitt 10.1 beschrieben, ist dann schnell nicht mehr ausreichend. Die Gewichtung der **Metriken**, die in Usus für Eclipse und Usus.NET vorgenommen und in Abschnitt 5.2 beschrieben wird, versucht eine weitere Interpretation zu ermöglichen. Deswegen zeigen die Cockpit-Fenster der beiden Usus-Versionen auch nicht die ungewichteten **Metriken** an. Allerdings sind die Bedeutungen dieser Statistiken nicht sofort verständlich und erfordern anschließend viel manuelle Interpretation. Die Metrikgewichtung von Usus geht auf das andrena-interne Werkzeug Isis zurück. Isis erlaubt es ungewichtete **Metriken**, die von einem anderen Tool erstellt wurden, zu importieren und zu bewerten. Das Ergebnis ist der von andrena sogenannte *Softwarequalitätsindex* (SQI), welcher in dem Manuskript von Eberhard Kuhn ausführlich beschrieben wird [Kuh06] und sämtliche **Metriken** zu einem einzigen Wert aggregiert. Mit diesem Wert kann die innere Softwarequalität beschrieben werden, sodass weniger manuelle Interpretation erforderlich ist.

11.1 Parameter

Der von Isis berechnete *Softwarequalitätsindex* ist von mehreren **Metriken** abhängig. Im Folgenden werden diese **Metriken** in der vorliegenden Master-Thesis so definiert, wie sie Kuhn in seinem Manuskript ebenfalls verwendet.

Definition 21 (Testabdeckung) Die *Metrik Testabdeckung* ist der prozentuale Anteil allen Quellcodes, der durch automatisierte Tests abgedeckt wird.

Definition 22 (Pakete in Zyklen) Die *Metrik Pakete in Zyklen* ist die Anzahl aller Namespaces, die Teil eines Namespace-Zyklus sind.

Definition 23 (Komplizierte Methoden) Die *Metrik Komplizierte Methoden* ist die Anzahl aller Methoden, deren *zyklomatische Komplexität* den Schwellwert 5 übersteigen.

Definition 24 (Average Component Dependency) Die *Metrik ACD* ist der durchschnittliche *CCD*-Wert aller Klassen und gibt an, von wie viel Prozent aller Klassen jeder Klasse direkt oder indirekt abhängig ist. Die Berechnung wurde in Unterabschnitt 5.2.3 beschrieben.

Definition 25 (Große Klassen) Die *Metrik* Große Klassen ist die Anzahl aller Klassen, die mehr als 20 Methoden haben.

Definition 26 (Große Methoden) Die *Metrik* Große Methoden ist die Anzahl aller Methoden, die mehr als 15 Codezeilen enthalten.

Definition 27 (Compiler-Warnungen) Die *Metrik* Compiler-Warnungen ist die Anzahl der Warnungen, die der Compiler beim Kompilieren anzeigt.

Nachdem diese *Metriken* durch ein Tool bestimmt wurden, berechnet Isis für jede *Metrik* m , bis auf die Testabdeckung, das Softwarequalitätsniveau. Die Berechnungsvorschrift ist in Formel 11.1 dargestellt. Das Softwarequalitätsniveau der Testabdeckung ist die Testabdeckung selber.

$$SQNiveau(m) = \frac{100}{1,5 \left(\frac{RelativeGroesse(m)}{Zweidrittelkonstante(m)} \right)} \quad (11.1)$$

Isis rechnet nicht mit den Rohwerten der *Metriken*, sondern verwendet dessen relative Größe. Diese relative Größe wird zusätzlich noch mit einem Faktor gewichtet, den Kuhn „Zweidrittelkonstante“ nennt. Tabelle 11.1 zeigt diese Faktoren. Wie die Zweidrittelkon-

Metrik m	Relative Größe	Zweidrittelkonstante
Pakete in Zyklen	$\frac{m}{AnzahlPackages}$	$\frac{1}{15}$
Komplizierte Methoden	$\frac{m}{AnzahlMethoden}$	$\frac{1}{50}$
Average Component Dependency	m	$100 \times AnzahlKlassen^{-1/3}$
Große Klassen	$\frac{m}{AnzahlKlassen}$	$\frac{1}{25}$
Große Methoden	$\frac{m}{AnzahlMethoden}$	$\frac{1}{25}$
Compiler-Warnungen	$\frac{m}{AnzahlKlassen}$	$\frac{1}{50}$

Tabelle 11.1: Relative Größen und Zweidrittelkonstanten aller Isis-*Metriken*

stanten zustande kommen, beschreiben Andreas Leidig und Nicole Rauch in ihrem technischen Artikel [LR09]. Um das Softwarequalitätsniveau gegenüber monolithischen Konstruktionen robuster zu machen, kann der von Kuhn beschriebenen „Monolithkorrekturfaktor“ angewendet werden [Kuh06]. Als Beispiel beschreibt Kuhn, dass eine zweifelhafte Verbesserung der *Metriken* erreicht werden könnte, indem alle Klassen in einem einzigen Paket oder alle Methoden in einer einzigen Klasse platziert werden. Diese Fälle sollen durch den Monolithkorrekturfaktor berücksichtigt werden. Die Testabdeckung und die Compiler-Warnungen sind von dieser Korrektur nicht betroffen. Formel 11.2 zeigt wie dieser Faktor für jede *Metrik* m bestimmt werden kann.

$$f_{mk}(m) = \frac{1}{1,5 \left(\frac{MittlereGroesse(m)}{Zweidrittelkonstante_{f_{mk}}(m)} \right)^3} \quad (11.2)$$

Isis verwendet an dieser Stelle wieder eine Zweidrittelkonstante aus Tabelle 11.2, um die Gewichtung des Korrekturfaktors zu regulieren. Die Größe des Systems ist ebenfalls von Bedeutung und wird in Form der Anzahl aller relevanten Codezeilen (RLOC) verwendet. Wie Usus.NET diese Codezeilen ermitteln kann, ist in Unterabschnitt 11.2.1 beschrieben. Dieser Faktor f_{mk} wird, wie in Formel 11.3 gezeigt, mit dem Softwarequalitätsniveau

Metrik m	Mittlere Größe	Zweidrittelkonstante f_{mk}
Pakete in Zyklen	$\frac{RLOC}{AnzahlPackages}$	3000
Komplizierte Methoden	$\frac{RLOC}{AnzahlMethoden}$	15
Average Component Dependency	$\frac{RLOC}{AnzahlKlassen}$	200
Große Klassen	$\frac{RLOC}{AnzahlKlassen}$	200
Große Methoden	$\frac{RLOC}{AnzahlMethoden}$	15

Tabelle 11.2: Zweidrittelkonstanten f_{mk} der Isis-Monolithkorrekturfaktoren

multipliziert, um ein korrigiertes Softwarequalitätsniveau zu erhalten.

$$SQNiveau_{f_{mk}}(m) = SQNiveau(m) \times f_{mk}(m) \quad (11.3)$$

Nachdem die endgültigen Softwarequalitätsniveaus aller **Metriken** bekannt sind, können diese aufsummiert werden. Diese Summe ist in Formel 11.4 dargestellt.

$$SQI = \sum_{m \in M} SQNiveau_{f_{mk}}(m) \times Gewicht(m) \times 0,1 \quad (11.4)$$

Isis nimmt dabei eine abschließende Gewichtung anhand Tabelle 11.3 vor.

Metrik m	Gewicht
Testabdeckung	2,28
Pakete in Zyklen	1,93
Komplizierte Methoden	1,75
Average Component Dependency	1,58
Große Klassen	1,05
Große Methoden	1,05
Compiler-Warnungen	0,36

Tabelle 11.3: SQI-Gewichte

In diesem Abschnitt wurden die Parameter und deren Gewichtungen beschrieben, mit denen Isis den **Softwarequalitätsindex** berechnet. Die nachfolgenden Abschnitte beschäftigen sich damit, wie die Berechnung des **SQI** in Usus.NET durchgeführt wird.

11.2 Berechnung in Usus.NET

Nachdem in Abschnitt 11.1 die Parameter und die Berechnungsvorschriften des **Softwarequalitätsindex** beschrieben wurden, beschäftigt sich dieser Abschnitt mit der tatsächlichen

Berechnung in Usus.NET. Der Namespace `andrena.Usus.net.Core.SQI` enthält die Klasse `Calculate`, welche die Methode `SoftwareQualityIndex` enthält. Dieser Methode wird ein Objekt übergeben, das das `IParameterProvider`-Interface implementiert. Jenes Interface definiert `get`-Properties für alle Parameter, die für die Berechnung des `SQI` erforderlich sind. Listing 11.1 zeigt die beispielhafte Berechnung der *Großen Methoden*. Die Variable `parameters` enthält eine `IParameterProvider`-Referenz, über die die beiden Methoden `SqNiveau` und `SqNiveauCorrection` als Extension Methods aufgerufen werden können.

Listing 11.1: Berechnung des `SQI`-Werts für Großen Methoden

```

1 return parameters.SqNiveau(m => m.BigMethods,
2     Sqi.RelativeSizeBigMethods, Sqi.CalibrationBigMethods)
3     * parameters.SqNiveauCorrection(m => m.BigMethods,
4     Sqi.MiddleSizeBigMethods, Sqi.CorrectionBigMethods);

```

Die `Sqi`-Klasse enthält für jede `SQI-Metrik` je ein Objekt vom Typ `CorrectionCalibration`, `Calibration`, `RelativeSize` und `MiddleSize`. Diese Typen repräsentieren die Metrik-abhängigen Berechnungen und Gewichtungen aus Abschnitt 11.1. Mit diesen Berechnungen lassen sich das Softwarequalitätsniveau sowie der Monolithkorrekturfaktor jeder `SQI-Metrik` bestimmen und multiplizieren. Diese Produkte können anschließend aufsummiert werden um den `Softwarequalitätsindex` zu erhalten.

Die Berechnung des `SQI` kann für jedes Objekt stattfinden, dessen Klasse das Interface `IParameterProvider` implementiert. In Unterabschnitt 9.1.8 wurde das `SQI`-Fenster vorgestellt, welches das Ergebnis in Visual Studio präsentiert. Im ViewModel dieses Steuerelements, nämlich der Klasse `ViewModels.SQI`, wird für jeden Parameter ein `SqiParameter`-Objekt verwaltet. Diese Objekte werden direkt an die Oberfläche gebunden und in Form eines editierbaren `DataGrid`-Controls angezeigt. Eine Änderung eines Werts wird durch das Data Binding des ViewModels direkt in dem entsprechenden `SqiParameter`-Objekt vorgenommen. Die Parameterobjekte werden über ein Objekt vom Typ `SqiParameters` erzeugt. Dieses Objekt beobachtet alle erzeugten Parameter und berechnet im Falle einer Änderung (entweder durch neue `Metriken` oder manuelle Eingabe) den neuen `SQI`. Die `SqiParameters`-Klasse implementiert dafür das `IParameterProvider`-Interface. Über das Ereignis `SqiChanged` wird der neue `Softwarequalitätsindex` dann dem ViewModel und damit der View mitgeteilt.

Ein von Usus.NET erstellter Metrikbericht in Form eines `MetricsReport`-Objekt, enthält fast alle Daten, die für die Berechnung des `SQI` erforderlich sind. Im Folgenden wird die Berechnung der Parameter beschrieben, die im Rahmen der Usus.NET-Metrikberechnung aus Abschnitt 8.1 und der Metrikgewichtung aus Abschnitt 8.3 noch nicht beschrieben wurden.

11.2.1 Relevante Code-Zeilen

Der `Isis-SQI` verwendet eine einheitliche Größe für Projekte, nämlich die *Anzahl aller relevanten Codezeilen* (engl. Relevant Lines Of Code, RLOC). Dabei werden alle Codezeilen in allen Quellcode-dateien, ohne Kommentare und ohne Klammern, berücksichtigt. Der Wert wird in dem `CommonReportKnowledge`-Objekt des Metrikberichts gespeichert (siehe Abschnitt 8.2). `CCI Metadata` erlaubt es die Zeileninformationen der CIL-Anweisungen

innerhalb einer Methoden mithilfe der `PeReader`-Klasse zu bestimmen. In Unterabschnitt 8.1.2 wurde beschrieben, dass Usus.NET auf diese Weise auch die `Methodenlänge` einer Methode berechnet. Die Zeileninformationen von Methoden- und Klassendefinitionen können über das `PeReader`-Objekt nicht ermittelt werden. `CCI Metadata` kann die Anzahl der relevanten Codezeilen also nicht direkt aus einer `Assembly` bestimmen. Aus diesem Grund versucht Usus.NET den RLOC-Wert auszurechnen. Die `Methodenlänge` bildet dafür die Grundlage. Für jede Methode im `MetricsReport`-Objekt addiert Usus.NET die `Methodenlänge` sowie eine weitere Zeile (Methodendefinition) zu dem `RelevantLinesOfCode`-Property in dem `CommonReportKnowledge`-Objekt. Für jede Klasse wird dieses Property um zwei weitere Zeilen erhöht, welche für die Klassendefinition sowie die Namespace-Zugehörigkeit stehen. Mehrzeilige Methodensignaturen und Klassendefinitionen werden also trotzdem mit nur jeweils einer Zeile gerechnet. Auch für jedes Feld wird die Eigenschaft `RelevantLinesOfCode` um einen Zähler erhöht. Die `using`-Angaben am Anfang jeder Quellcodedatei werden ignoriert.

11.2.2 Compiler-Warnungen

Wenn beim Kompilieren Warnungen entstehen, werden diese ebenfalls in die Berechnung des Isis-SQI einbezogen. `Usus.net.Core` kann die Anzahl dieser Warnungen allerdings nicht bestimmen, da diese nicht in der kompilierten `Assembly` hinterlegt sind. Eine Visual Studio-Erweiterung in Form eines `VSPackage` kann auf die Warnungen und Fehler eines Build-Vorgangs zugreifen. Wie in Listing 11.2 gezeigt, ist dies über das `DTE2`-Objekt möglich, welches über das Property `MasterObjekt` der `DtAwareToolWindow`-Klasse veröffentlicht wird.

Listing 11.2: Anzahl der Compiler-Warnungen bestimmen

```

1 ErrorList list = MasterObjekt.ToolWindows.ErrorList;
2 for (long index = 1; index <= list.ErrorItems.Count; index++)
3 {
4     ErrorItem error = list.ErrorItems.Item(index);
5     if (error.ErrorLevel == vsBuildErrorLevel.vsBuildErrorLevelMedium
6         || error.ErrorLevel == vsBuildErrorLevel.vsBuildErrorLevelLow)
7         AddWarning();
8 }

```

Da das Usus.NET SQI-Fenster ebenfalls eine Unterklasse von `BuildAwareToolWindowPane` ist, können die Warnungen von dem Fenster bestimmt werden. Warnungen und Fehler werden von Visual Studio in dem Fenster der Fehlerliste angezeigt. Über alle diese Warnungen und Fehler kann iteriert werden, wobei die Unterscheidung zwischen Fehler und Warnung mithilfe der `ErrorLevel`-Eigenschaft vorgenommen werden kann. Das Fehlerlevel `vsBuildErrorLevelHigh` steht für einen Fehler, während `vsBuildErrorLevelMedium` oder `vsBuildErrorLevelLow` eine Warnung repräsentieren. Alle Warnungen können somit aufsummiert und an das `SqiParameters`-Objekt des `ViewModel` weitergegeben werden. Dazu implementiert das SQI-Fenster das Interface `IKnowSqiDetails`. Bei der Erzeugung des `ViewModels` registriert sich das Fenster bei dem `ViewModel`, sodass im Falle einer SQI-Berechnung auch die Werte, die nur vom `VSPackage` bestimmt werden können, ermittelt werden können. Durch das `IKnowSqiDetails`-Interface kann das Property `CompilerWarnings` des Fensters vom `ViewModel` aufgerufen werden, ohne das das `ViewModel` abhängig von dem `VSPackage` ist.

11.2.3 Testabdeckung

Usus.NET bezieht bei der Berechnung des Isis-SQI auch die prozentuale Testabdeckung mit ein. Dieser Wert wird von einem Test-Runner ermittelt, der sich in Visual Studio integriert. Das Ergebnis kann nicht aus einer *Assembly* allein bestimmt werden. Genau wie die Anzahl der Compiler-Warnungen muss die Testabdeckung von dem VSPackage des SQI-Fenster bestimmt werden. Anschließend kann der Wert wieder über das *IKnowSqiDetails*-Interface an das *SqiParameters*-Objekt des ViewModel übergeben werden. Da Visual Studio in der Professional-Version die Analyse der Testabdeckung nicht unterstützt, muss ein anderes Werkzeug genutzt werden. Usus.NET müsste also in der Lage sein, unabhängig vom verwendeten Werkzeug die Testabdeckung eines Projekts zu extrahieren. Dies ist nur möglich wenn Usus.NET die Formate, in denen die einzelnen Code Coverage Tools ihr Ergebnis speichern, analysieren kann. Im Rahmen dieser Master-Thesis wurden diese Analysen nicht implementiert. Die Testabdeckung kann daher im SQI-Fenster der aktuellen Version von Usus.NET von Hand eingegeben werden, nachdem sie mit Visual Studio oder einem anderen Werkzeug, wie beispielsweise NCover¹, ermittelt wurde.

11.3 Veränderung

Um die Veränderung der Softwarequalität zu beobachten, müssen die *Metriken* bei jedem Kompilervorgang gemerkt werden. Anschließend können die neusten *Metriken* mit den vorherigen *Metriken* verglichen werden. Auf diese Weise könnten Veränderung betrachtet und eine eventuelle Verbesserung der Qualität festgestellt werden. Bei den betrachteten *Metriken* handelt es sich um die Werte, die im Usus.NET Cockpit-Fenster angezeigt werden, sowie dem SQI-Wert. Die *Clean Code-Metrik* von Usus.NET wird ebenfalls berücksichtigt, da die Verteilungsparameter auch in der Cockpit-Ansicht angezeigt werden. Das Cockpit-Fenster, welches bereits in Unterabschnitt 9.1.2 vorgestellt wurde, visualisiert Veränderungen durch Farben. Ist der neue Wert weniger gut als der alte, wird er rot dargestellt. Ein besserer Wert wird grün angezeigt. Die Veränderung der Werte im Cockpit bezieht sich also immer nur auf den letzten und den vorletzten Build. Beim SQI dagegen, werden Veränderungen über mehrere Builds hinweg betrachtet. Diese zeitliche Entwicklung des SQI wird, wie Abbildung 11.1 zeigt, im SQI-Fenster dargestellt. Das SQI-Fenster

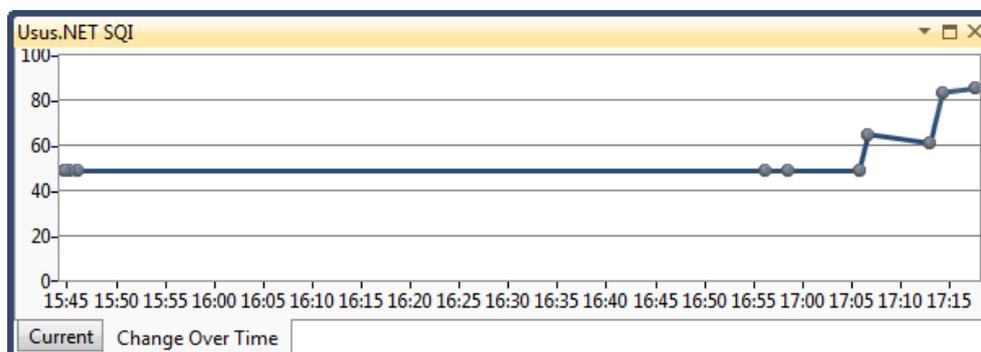


Abbildung 11.1: Zeitliche Entwicklung des SQI im Usus.NET SQI-Fenster

¹Download und mehr Informationen: „NCover: Code Coverage for .NET Developers“ <http://www.ncover.com/>

wurde bereits in Unterabschnitt 9.1.8 beschrieben. Das ViewModel verwendet dafür ein Objekt vom Typ `SqiHistory`, mit dessen `Now`-Methode neue **SQI**-Werte, die einer Solution zugeordnet sind, gespeichert werden können. Nach jedem Kompilierungsvorgang speichert Usus.NET den neuen **SQI**-Wert in Zusammenhang mit der aktuellen Zeit und dem aktuellen Datum in dem Unterverzeichnis `_sqi` des Solution-Verzeichnisses. Die erzeugten Dateien haben die Dateierweiterung `.sqi` und verwenden den Wert der `DateTime.Now.Ticks`-Eigenschaft als Dateinamen. Dadurch kann es keine Dateien mit doppeltem Namen geben und eine natürliche Sortierung ist implizit. Als Dateinhalt wird der **SQI**-Wert als `double` serialisiert. Um diese Gleitkommazahl in einem länderunabhängigen Format zu speichern, werden die Methoden des .NET Framework verwendet, die auch in Listing 11.3 angegeben sind. In dem `_sqi`-Verzeichnis der aktuellen Solution entsteht also pro Build eine `sqi`-Datei. Sobald eine andere Solution geladen und kompiliert wird, werden auch die letzten hundert **SQI**-Werte dieser Solution geladen und in dem Schaubild angezeigt. Auf diese Weise kann die zeitliche Entwicklung des **Softwarequalitätsindex** beobachtet werden.

Listing 11.3: Länderunabhängige Serialisierung von `double`-Werten

```
1 string sqiSerialized = Convert.ToString(sqi, CultureInfo.InvariantCulture);
2 ...
3 double sqi = Convert.ToDouble(sqiSerialized, CultureInfo.InvariantCulture);
```

Eventuell kann die Kombination aus **SQI**-Wert, Zeit und Datum auch ergänzt werden. Eine Serialisierung der **SQI**-Messpunkte kann dann auch mehr Werte als nur den **SQI** selber in den `sqi`-Dateien speichern.

12 Usus.NET-Evaluation

Nachdem Usus.NET in den vergangenen Kapiteln umfangreich beschrieben wurde, soll die Erweiterung in diesem Kapitel einmal tatsächlich eingesetzt werden. In diesem Zusammenhang werden unter anderem einige frei verfügbare Open-Source-Projekte analysiert, um die Performance von Usus.NET zu messen. Zusätzlich wird Usus.NET auch selber analysiert. Dabei werden die Größe dieser Programme mit der Laufzeit der [statischen Code-Analysen](#) in Verbindung gebracht. Abschließend wird eine Refactoring-Übung aus einem andrena-Kurs mit Usus.NET durchgeführt.

12.1 Laufzeitmessungen

Die [Metriken](#), die Usus.NET berechnet und gewichtet, werden in dieser Konstellation nur von dem Usus für Java berechnet. Ein fairer Vergleich mit anderen Werkzeugen wie Visual Studio Code Metrics Power Tool, Code Metrics Viewer und NDepend kann leider nicht durchgeführt werden, da diese Tools andere [Metriken](#) berechnen. Es hat den Anschein, dass NDepend die [Analysen](#) schneller durchführen kann als Usus.NET. Dies könnte daran liegen, dass NDepend eventuell nur [CCI Metadata](#) und nicht auch [CCI Code and AST Components](#), oder sogar eine optimierte Eigenentwicklung verwendet. [Abbildung 12.1](#) zeigt die Laufzeiten von Usus.NET bei der [Analyse](#) verschiedener Projekte. Die beiden CCI-Bibliotheken, sowie QuickGraph werden von der Usus.NET-Implementierung direkt verwendet. GraphSharp ist eine Bibliothek, die die mit QuickGraph erzeugten Graphen visualisieren kann. Zusätzlich liegt es nahe, Usus.NET selbst ebenfalls zu analysieren. Als weiteres Projekt wird ein von andrena entwickeltes System mit dem Namen „Mescor“ analysiert. Die Schaubilder zeigen deutlich, dass die [Analyse](#) größerer Projekte länger dauert. Als Projektgröße hat dabei die Anzahl der [Assemblies](#) einen größeren Einfluss auf die Laufzeit, als die Anzahl an relevanten Codezeilen (RLOC). Dieser Einfluss ist gut an der Laufzeit von QuickGraph zu erkennen. Obwohl diese eine Bibliothek im Vergleich zu GraphSharp mehr Zeilen Quellcode enthält, ist die [Analyse](#) schneller. Da GraphSharp aus zwei [Assemblies](#) mit insgesamt weniger Zeilen besteht, dauert die [Analyse](#) hier länger. Um die [Analyse](#) von großen Projekten zu beschleunigen, kann Usus.NET eine Menge von [Assemblies](#) auch parallelisiert analysieren. [Abbildung 12.1](#) zeigt daher auch für jedes Projekt einmal die Laufzeit der sequentiellen und einmal die der parallelen [Analyse](#). Je größer das Projekt, umso eher kann durch die Verwendung mehrerer Prozessorkerne eine Laufzeitreduktion gemessen werden. Im Beispiel konnte die Laufzeit des CciAst-Projekts ([CCI Code and AST Components](#)) auf einem System mit vier Kernen sogar fast halbiert werden. Leider konnte dieser Geschwindigkeitsgewinn nicht bei der [Analyse](#) des Mescor-Projekts erreicht werden. Das liegt vermutlich daran, dass die [Assemblies](#) alle sehr klein sind. Viele große [Assemblies](#) würden sich mit dieser Argumentation besonders effizient parallel analysieren lassen. Die tatsächlichen [Metriken](#) dieser Systems sind für die Performance-Messungen nicht von Bedeutung und befinden sich im [Anhang A.4](#).

In Abschnitt 8.1 wurde für die sequentielle *Analyse* die Klasse *Analyze* vorgestellt. Die Klasse *AnalyzeParallel* erlaubt es, die *Analyse* auf die einzelnen Prozessorkerne des Systems aufzuteilen. In der MSDN Library beschreibt Microsoft *Parallel LINQ* (PLINQ) als elegantes Mittel um Parallelität bei der Verarbeitung von Listen zu verwenden [Mic12b]. Da sich die zu analysierenden *Assemblies* in einer Liste befinden, kann PLINQ sehr einfach in Usus.NET verwendet werden. Listing 12.1 zeigt, dass nur durch den Methodenaufruf *AsParallel()* die Liste der *Assemblies* parallel verarbeitet wird.

Listing 12.1: Sequentielle und parallele *Analyse* von mehreren *Assemblies*

```
1 //sequentiell: Analyze.PortableExecutables(files)
2 from file in files select AnalyzeFile(file)
3
4 //parallel: AnalyzeParallel.PortableExecutables(files)
5 from file in files.AsParallel() select AnalyzeFile(file)
```

Leider hat die parallele *Analyse* gegenüber der höheren Geschwindigkeit auch einen Nachteil. Manchmal werden die *pdb-Dateien* von der *CCI* nicht korrekt geladen, wenn die *Analyse* auf mehrere Threads verteilt läuft. Dieses Problem äußert sich dadurch, dass beispielsweise Zeilenangaben für *Methodenlängen* nicht erfolgreich ermittelt werden können. Im Usus.NET Cockpit-Fenster entstehen dadurch sogenannte Phantomänderungen der *Methodenmetriken*, wenn die gleiche Solution zweimal hintereinander analysiert wird, ohne dass der Quellcode geändert wurde. Aus diesem Grund kann unter dem Menüeintrag *Tools / Options / Usus.NET* eingestellt werden, ob Usus.NET die *Analyse* sequentiell oder parallel durchführen soll. Generell scheint die *CCI* Schwierigkeiten mit dem Resource Management der *pdb-Dateien* zu haben. Wenn Usus.NET die *pdb-Dateien* nutzt und anschließend sofort wieder über die *CCI* freigibt, kann Visual Studio diese Dateien erst nach einer kurzen zeitlichen Verzögerung wieder ersetzen.

12.2 Fallbeispiel andrena-Kurs

Mit dem Kurs „Agile Software Engineering“¹ bietet andrena ein Training für Entwickler an, in dem moderne Softwareentwicklung vorgestellt und unter kontrollierten Rahmenbedingungen ausprobiert und geübt werden kann. Dabei wird unter anderem auch eine Übung zum Thema Refactoring durchgeführt. Die Teilnehmer bekommen dabei die Möglichkeit, eine bestehende Codebasis inklusive Tests selbstständig umzugestalten. Das Ausgangsprojekt wurde mit Usus.NET einmal analysiert. Abbildung 12.2 zeigt das Ergebnis dieser *Analyse*. Da das Projekt mit 96 relevanten Codezeilen sehr klein ist, gibt es nicht viele verbesserungswürdige Stellen. Das Hauptproblem ist die *GetMailMessage*-Methode der Klasse *MailComposer*. Mit 19 logischen Zeilen und einer *zyklomatischen Komplexität* von 7, ist diese Methode ein absoluter Hotspot und wird dementsprechend auch als solcher angezeigt. Ziel der Übung ist es also, diese Methode umzugestalten. Dank der 100%igen Testabdeckung ist dies ohne Schwierigkeiten möglich. Zu dieser Übung existiert auch eine „Musterlösung“ der Trainer, die ebenfalls mit Usus.NET analysiert wurde. Das Ergebnis ist in Abbildung 12.3 zu sehen. Zwischen diesen beiden Messungen hat also das Refactoring stattgefunden.

¹Mehr Informationen: „ASE - Agile Software Engineering“ <http://www.andrena.de/leistungen/ase-agile-software-engineering>

Der [Softwarequalitätsindex](#) konnte sehr verbessert werden. Dies konnte erreicht werden, indem die `GetMailMessage`-Methode auf 5 Zeilen und eine [zyklomatische Komplexität](#) von 1 reduziert wurde. Dafür wurden drei neue Typen erstellt, was dazu geführt hat, dass die `MailComposer`-Klasse mehr Abhängigkeiten bekommen hat. Tatsächlich hat diese Klasse jetzt so viele direkte und indirekte Abhängigkeiten, dass Usus.NET dies als Problem sieht und die Klasse in den Hotspots auflistet. Eine Interpretation dieses Hotspots ist, dass diese Klasse überdurchschnittlich viele Abhängigkeiten in dem betrachteten System hat. Da diese Klasse auch von der Testklasse `MailComposerTest` verwendet wird, die die gleichen indirekten Abhängigkeiten hat, wird auch diese Testklasse als Hotspot gesehen. Die [durchschnittliche Komponentenabhängigkeit](#) ist dagegen gefallen, was darauf zurückzuführen ist, dass die neuen Abhängigkeiten der `MailComposer`-Klasse selbst nicht sehr viele Abhängigkeiten haben und den Durchschnitt senken. In Bezug auf das [Clean Code](#)-Level der Codebasis hat sich das Hinzufügen der neuen Typen negativ auf die [Verteilung der Klassenmetriken](#) ausgewirkt. Die [Verteilungen der Klassenmetriken](#) entsprechen also geometrischen [Verteilungen](#), die weniger schnell fallen, als vor dem Refactoring. Diese Verschlechterung würde den Entwickler motivieren im nächsten Schritt die Abhängigkeiten der Klassen zu verändern, sodass auch die [Verteilungen der Klassenmetriken](#) auf [Clean Code](#) hindeuten. Was die [Methodenmetriken](#) angeht, so fallen die geometrischen [Verteilungen](#) schneller. Es gibt also mehr kleine [Methodenmetriken](#) als große, was nach Kapitel 10 ein sehr guter Indikator für [Clean Code](#) ist.

12.3 Ergebnis

Die oben beschriebene Refactoring-Übung wurde nicht selbst durchgeführt. Dank Usus.NET konnte allein durch die Betrachtung der Analyseergebnisse vor und nach dem Refactoring, genügend Informationen ermittelt werden, um die Codebasis und das Refactoring zu interpretieren (Ziel 4 erreicht). Der Quellcode müsste theoretisch nicht einmal mehr eingesehen werden um zu erkennen, dass die Methoden verbessert wurden, ohne die Typen sehr zu verschlechtern (Ziel 1 erreicht). Auch vor dem Refactoring hat Usus.NET die Problemfälle der Codebasis automatisch erkannt und so ein effizientes und zielgerichtetes Refactoring ermöglicht (Ziel 2 erreicht). Nach dem Refactoring existieren zwar immer noch einige Klassen-Hotspots, die beachtet werden müssen, aber der höhere [SQI](#) weist auf eine sehr positiv zu wertende Verbesserung hin. Diese Verbesserung ist auch im Kontext von [Clean Code](#) an dem λ der [geometrischen Verteilung der Methodenmetriken](#) zu erkennen (Ziel 3 erreicht). Ohne Usus.NET hätte die Problemstelle erst gesucht werden müssen. Eine anschließende Verbesserung wäre dann auch nur schwer messbar gewesen. Beispielsweise hätten viel gravierendere Probleme entstehen und unerkant bleiben können.

Usus.NET hat die [statische Code-Analyse](#) automatisch nach dem Kompilervorgang im Hintergrund durchgeführt, ohne dass der Entwickler irgendetwas einstellen musste. Während der [Analyse](#) kann der Entwickler weiterhin mit dem Quelltext arbeiten oder die Anwendung debuggen. Da die [statische Code-Analyse](#) in einem eigenen Thread im Hintergrund läuft, wird Visual Studio nicht blockiert oder merklich verlangsamt. Lediglich das Zeichnen der Usus.NET Info-Buttons aus Unterabschnitt 9.1.3 führt dazu, dass der Editor bei einer größeren Codebasis etwas langsamer wird. Wenn die Analyseergebnisse einmal erwartet werden, dauert die [Analyse](#) dank der parallelen Verarbeitung auch in sehr großen Projekten in der Regel nicht länger als 5 Minuten. Die umfangreiche Darstellung

der Ergebnisse unterstützt die Interpretation der Code-Qualität und nimmt diese durch das λ der geometrischen Verteilung und dem Softwarequalitätsindex teilweise sogar ab. Im dem Usus für Java sind genau diese Interpretationsfunktionen nicht enthalten, sodass Usus.NET an dieser Stelle eine umfangreichere Hilfestellung und Einsicht in den Code geben kann.

Abschließend ist interessant zu sehen, wie sich Usus.NET in der aktuellen Version selbst analysiert. Usus.NET hat sehr gute Clean Code-Metriken auf Methodenebene und zufriedenstellende Metriken auf Klassenebene. Ein Softwarequalitätsindex von 82,2% bewertet die Visual Studio-Erweiterung insgesamt positiv. Das ausführliche Ergebnis befindet sich in der Anlage A.4.

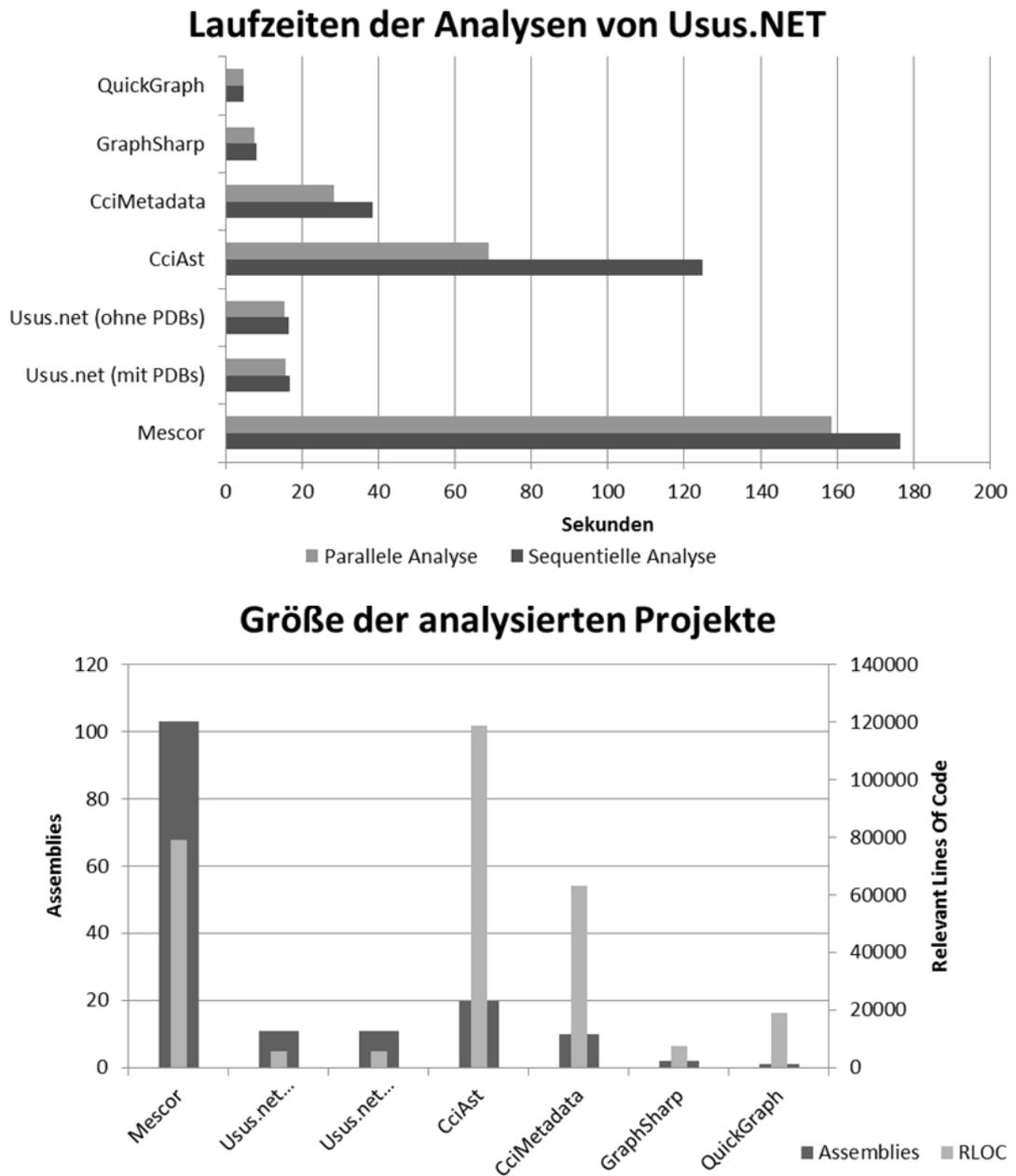


Abbildung 12.1: Laufzeiten der [statischen Code-Analyse](#) verschiedener Industrie- und Open-Source-Projekte

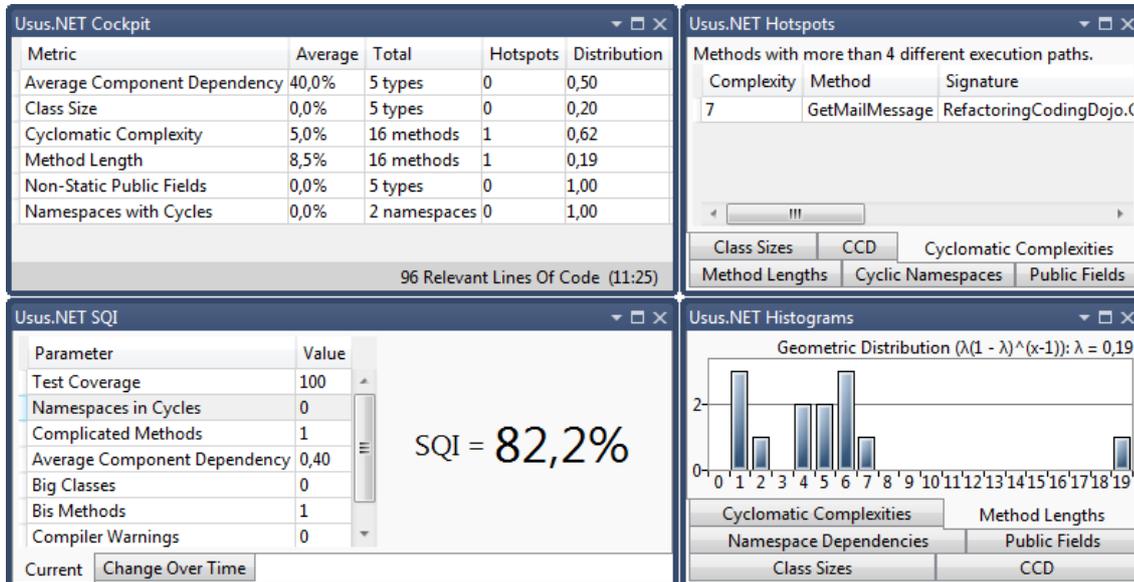


Abbildung 12.2: Vorher-Metriken des andrena Refactoring-Beispiels

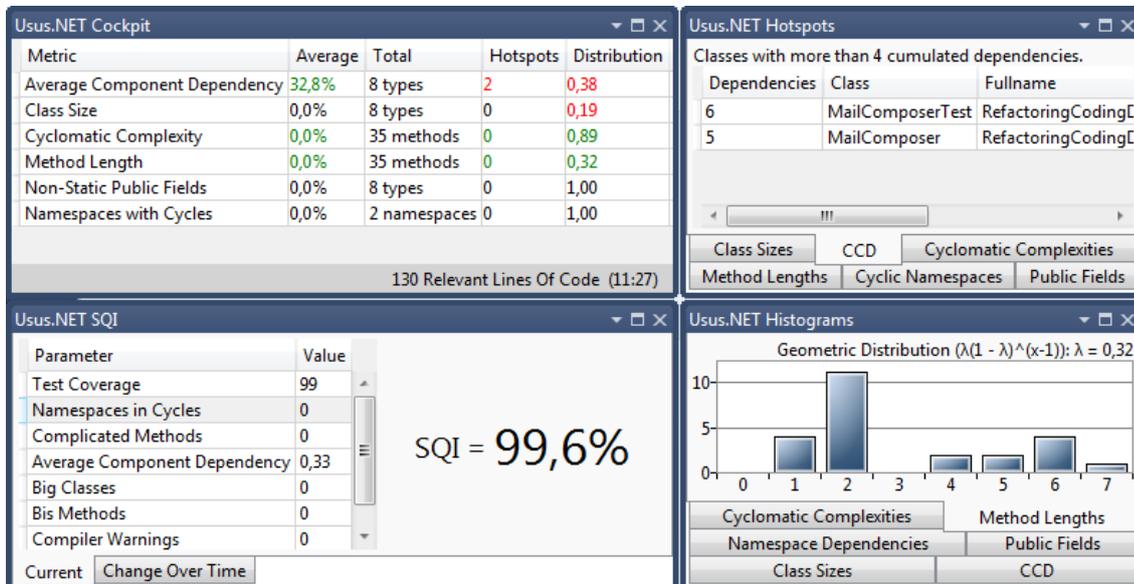


Abbildung 12.3: Nachher-Metriken des andrena Refactoring-Beispiels

13 Zusammenfassung

In dieser Master-Thesis wurde die Visual Studio-Erweiterung Usus.NET implementiert und dokumentiert. Bei der agilen Anforderungsanalyse wurden zunächst die Interessenvertreter und die Ziele der Erweiterung gefunden und in Kapitel 3 beschrieben. Usus.NET soll dem Entwickler Einsicht in die Codebasis geben (Ziel 1), problematische Stellen im Code erkennen (Ziel 2), die Entwicklung von **Clean Code** fördern (Ziel 3) und eine Interpretation der Softwarequalität ermöglichen und erleichtern (Ziel 4). Alle genannten Ziele können anhand von **Softwaremetriken** erreicht werden.

Dazu müssen die **Metriken** zuerst berechnet, gewichtet und bewertet werden, was auch von dem Eclipse-Plugin Usus für Java getan wird. Usus für Java wurde von andrena entwickelt und ermöglicht ebenfalls eine Einsicht in die Codebasis (Ziel 1) und erkennt auch Problemfälle (Ziel 2). Das Java-Usus ermöglicht dabei keine Interpretation der Softwarequalität anhand von **Clean Code-Metriken** oder dem **Softwarequalitätsindex**. Um die Funktionen des Usus für Java auch in Usus.NET zu realisieren, wurde das Java-Usus analysiert. Dabei wurden die **Metriken**, die dieses Werkzeug berechnet, dokumentiert und für die Implementierung von Usus.NET spezifiziert. Um Ideen für die Realisierung der anderen beiden Ziele, **Clean Code** fördern (Ziel 3) und Softwarequalität interpretieren (Ziel 4), zu bekommen, wurden auch einige andere bekannte Tools betrachtet. Dabei wurde festgestellt, dass diese Werkzeuge besagte Ziele ebenfalls nicht erreichen können.

Um eine geeignete Technologie zu finden, mit dessen Hilfe Usus.NET die **statische Code-Analyse** durchführen kann, wurden mehrere Frameworks untersucht und anhand von zuvor definierten Kriterien evaluiert. Dabei wurde festgestellt, dass das **Common Compiler Infrastructure**-Framework von Microsoft sich am ehesten für die Verwendung in Usus.NET eignet. Um die definierten Anforderungen zu erfüllen, wurde Usus.NET in fünf 10-tägigen Iterationen realisiert. Zwischen den Iterationen wurde dieses Dokument aktualisiert.

In den ersten beiden Iterationen wurde die hauptsächliche Funktionalität des Eclipse-Usus, also die **Metrik**-Berechnung, -Gewichtung und -Bewertung, für Usus.NET neu entwickelt und getestet. Um die Entwicklung durch Testfällen absichern zu können, wurde ein eigenes Framework für Integrationstests implementiert, welches ebenfalls Bestandteil von Usus.NET ist. Dieses Framework nutzt einen speziell dafür konzipierten Ansatz um **Methoden-** und **Klassenmetriken** anhand von Erwartungswerten zu verifizieren. In der dritten Iteration wurde die Visual Studio-Erweiterung implementiert, sodass die Fenster, die aus dem Usus für Eclipse bekannt sind, auch in Visual Studio vorhanden sind. Nach jedem Kompilervorgang berechnet Usus.NET die aktuellen **Metriken** der erzeugten **Assemblies**. Das Fenster des Klassen- und Paketgraph wurde aus zeitlichen Gründen nicht realisiert.

In der vierten Iteration wurden die **Metrik**-Histogramme analysiert. Dabei wurde festgestellt, dass **Metriken** in einem Softwaresystem tatsächlich der **Exponentialverteilung**, be-

ziehungsweise der geometrischen *Verteilung*, folgen. Wenn in einem Projekt die *Clean Code*-Prinzipien von Robert C. Martin, Ralf Westphal und Stefan Lieser berücksichtigt werden, ist der Parameter der Verteilungsfunktion höher. Je höher diese Parameter ist, desto schneller fällt die Verteilungsfunktion. Das bedeutet, dass es mehr kleine *Methodenlängen*, *Klassengrößen*, *zyklomatische Komplexitäten*, *Klassenabhängigkeiten* und ähnliches gibt, als große. Diese Tendenz entspricht eindeutig den Gedanken, die Robert C. Martin auch in seinem Buch „Clean Code“ beschreibt [Mar09a]. Mit dem λ der geometrischen *Verteilung* kann Usus.NET also eine *Clean Code-Metrik* berechnen. Diese *Metrik* wird ebenfalls im Usus.NET Cockpit-Fenster angezeigt, sodass der Entwickler sofort sieht, wie *clean* die *Metriken* seines Codes gerade sind. Wird dieser Wert besser, wird er grün hervorgehoben, wobei eine Verschlechterung rot dargestellt wird. Dadurch wird der Entwickler motiviert, die *Clean Code*-Prinzipien mehr zu beachten. Usus.NET fördert so aktiv die Erzeugung von *Clean Code* (Ziel 3).

In der vierten Iteration wurde zusätzlich die Berechnung des *Softwarequalitätsindex* implementiert. Um diesem Wert eine besondere Bedeutung beizumessen, wird er nicht im Usus.NET Cockpit, sondern in einem eigenen Fenster angezeigt. Der *SQI* gewichtet die *Metriken* ein wenig anders als Usus dies tut und kombiniert diese mit weiteren Details zu einem einzigen Wert. Die aktuelle Testabdeckung des betrachteten Softwaresystems ist ein solches Detail. Da die Testabdeckung von den verschiedenen Werkzeugen auf unterschiedliche Weise bestimmt wird, gibt es keine einheitliche Schnittstelle um diesen Wert auszulesen. In der aktuellen Version von Usus.NET wird dieser Wert daher noch nicht automatisiert ermittelt und kann manuell eingetragen werden. Sobald die Testabdeckung eingegeben wird, wird der *SQI* neu berechnet. Die Veränderung dieses Werts lässt sich über die Zeit beobachten und grafisch im Usus.NET *SQI*-Fenster darstellen. In Verbindung mit der *Clean Code-Metrik* und den anderen *Metriken* des Usus.NET Cockpit, erleichtert Usus.NET die manuelle Interpretation der Softwarequalität enorm (Ziel 4). Damit wurden alle in Abschnitt 3.2 genannten Ziele erreicht.

Dies wurde im Rahmen der fünften Iteration anhand der *Analyse* eines Refactoring-Beispiels aus einem andrena-Kurs verifiziert. Dennoch ersetzt Usus.NET keine manuellen Codereviews, sondern erleichtert diese. Zusätzlich wurde die Performance analysiert und optimiert. Bei der *Analyse* von einigen Industrie- und Open-Source-Projekten wurde festgestellt, dass die *statische Code-Analyse* langsamer wird, sobald ein Projekt aus vielen *Assemblies* besteht. Wenn diese *Assemblies* allerdings eine größere Anzahl an Code-Zeilen besitzen, kann die *Analyse* durch die Verwendung von mehreren Prozessorkernen wieder beschleunigt werden.

Die aktuelle Version von Usus.NET ist unter der LGPL¹ veröffentlicht. Der komplette Quellcode der Visual Studio-Erweiterung befindet sich auf github² und kann eingesehen und heruntergeladen werden. Zusätzlich kann die Integration beliebiger Weiterentwicklungen beantragt werden. Die aktuelle Installationsdatei von Usus.NET befindet sich im Anhang A.5.

¹Mehr Informationen: „GNU Lesser General Public License“ <http://www.gnu.org/copyleft/lesser.html>

²Download: „Repository of Usus.NET“ <https://github.com/usus/Usus.NET>

14 Fazit und Ausblick

Innerhalb von fünf Iterationen wurde die Visual Studio-Erweiterung Usus.NET entwickelt. Das von andrena erwartete Ziel einer lauffähigen Software wurde erreicht. Da einige User Stories, die für die fünfte Iteration geplant waren, bereits in der vierten Iteration erledigt wurden, konnten in der letzten Iteration einige Probleme behoben, sowie die Beispiel- und Performance-Messungen durchgeführt und beschrieben werden. Da diese Master-Thesis als Dokumentation der Entwicklung sehr wichtig ist, wurden zugunsten der Dokumentation wenige potenzielle Features von Usus.NET nicht implementiert. So wurde beispielsweise auf eine Visualisierung der Graphen komplett verzichtet, da eine notwendige Evaluierung verschiedener Darstellungstechnologien aus zeitlichen Gründen nicht durchgeführt werden konnte. Besonders sinnvoll hätte sich dieses Feature bei der Analyse von Hotspots in Form von Namespace-Zyklen einsetzen lassen. In der aktuellen Version werden die problematischen Klassen, also die Klassen, die Klassen in den anderen Namespaces im selben Zyklus referenzieren, in Listen angezeigt. Eine grafische Darstellung des Zyklus in Form eines Graphen würde die Suche nach der schuldigen Abhängigkeit erleichtern.

In einer Evaluierung wurde die [Common Compiler Infrastructure](#) als technisches Mittel gefunden, um eine [statische Code-Analyse](#) durchführen zu können. Usus.NET ermöglicht in der aktuellen Version auch einige Einstellungen vorzunehmen. So kann eingestellt werden, ob eine [Analyse](#) parallel oder sequentiell durchgeführt werden soll. Weitere Einstellungsmöglichkeiten, die die [statische Code-Analyse](#) beispielsweise komplett ausschalten, wären noch wünschenswert. Usus.NET analysiert in der aktuellen Version bei jedem Kompilervorgang automatisch alle Projekte der aktuellen Solution. Eine manuelle [Analyse](#) auf Knopfdruck könnte sich in manchen Szenarien als brauchbarer erweisen.

Aus Zeit-Gründen konnte eine automatisierte Bestimmung der aktuellen Testabdeckung nicht realisiert werden. Um den [Softwarequalitätsindex](#) in der aktuellen Usus.NET-Version vollständig zu bestimmen, kann dieser Wert manuell ermittelt und in das entsprechende Fenster eingetragen werden. Auch hier müsste wieder eine Evaluierung mehrerer Werkzeuge und Bibliotheken, die die Testabdeckung berechnen, durchgeführt werden.

Die oben beschriebenen nicht realisierten Features können problemlos im Anschluss an diese Master-Thesis implementiert werden. In den sechs Monaten konnte eine Grundlage gelegt werden, die eine Weiterentwicklung der Erweiterung als Open-Source-Projekt ermöglicht. Das bedeutet, dass sich auch andrena-externe Entwickler an der Fortführung des Projekts beteiligen dürfen und sollen. Nachdem die oben erwähnten Funktionen entwickelt wurden, kann Usus.NET um viele andere Funktionen ergänzt werden.

Beispielsweise wäre es spannend zu sehen, wie Namespace-Zyklus-Hotspots noch besser dargestellt werden können. Da Usus.NET in der aktuellen Version nur alle Namespaces in einem Zyklus auflistet, wäre eine Suche nach dem kleinsten Zyklus sehr interessant.

Ein solcher kleinster Zyklus, der das eigentliche Problem darstellt, könnte eine effizientere Auflösung ermöglichen. Außerdem wäre die Integration weiterer **Clean Code-Metriken** denkbar. Momentan berechnet Usus.NET zu jedem **Metrik-Histogramm** das λ der **geometrischen Verteilung** und verwendet diesen Wert als neuartige und alleinige **Clean Code-Metrik**. Sobald Usus.NET die automatisierte Bestimmung der Testabdeckung unterstützt, kann die in Kapitel 10 erwähnte **CRAP-Metrik** ebenfalls berechnet werden. Eventuell kann diese **Metrik** auch mit den Verteilungsinformationen kombiniert werden. Oder es können mehr **Verteilungen** betrachtet werden, sodass die **Metrik-Histogramme** besser interpretiert und Hinweise zu sinnvollen Refactorings gegeben werden können. Die Refactorings müssten im Vorfeld den Veränderungen der **Verteilungen** zugeordnet werden. Anschließend könnte eine Veränderung der **Verteilung** erreicht werden, indem ein Refactoring angewendet wird. So könnte Usus.NET den Entwickler Refactoring über Refactoring zu einer **Verteilung** führen, die mehr für **Clean Code** steht, beispielsweise die geometrische **Verteilung** mit einem großen λ . Eventuell könnten diese Refactorings auch automatisiert durchgeführt werden. Da Usus.NET die Refactorings kennen würde, die notwendig sind um ein System in Richtung **Clean Code** umzugestalten, könnte eine bestehende Codebasis eventuell auf Knopfdruck verbessert werden.

Aktuell sind noch keine Werkzeuge bekannt, die bestehenden Code *clean* machen. Wenn eine solcher Trend weiter verfolgt wird, wäre Usus.NET die richtige Grundlage, um diese Funktionen zu implementieren, auszuprobieren und zu testen.

Glossar

λ **der geometrischen Verteilung** ist eine [Metrik](#) eines [Metrik-Histogramms](#) und bezeichnet im Kontext von Usus.NET wie sehr die betrachtete [Metrik](#) dem [Clean Code-Paradigma](#) entspricht. Je größer das λ , desto besser. Das λ ist der Parameter der geometrischen [Verteilung](#), die an das Histogramm angenähert wird.

Abstrakter Syntaxbaum ist eine syntaktische Repräsentation des Codes und besonders der Methoden, die für jede Anweisung einen Knoten enthält.

Assembly ist eine Datei, die ein .NET-Compiler aus einem Visual Studio-Projekt erzeugt. Die Datei ist entweder eine exe- oder eine dll-Datei. Synonym wird eine Assembly auch als PE-Datei (Portable Executable) bezeichnet.

CCI Code and AST Components ist eine Komponente der von Microsoft veröffentlichten [Common Compiler Infrastructure](#), die es erlaubt Methoden zu dekompileieren und einen [abstrakten Syntaxbaum](#) zu erstellen.

CCI Metadata ist eine Komponente der von Microsoft veröffentlichten [Common Compiler Infrastructure](#), die es erlaubt [Assemblies](#) zu analysieren um Klassen-, Interface- und Methodendaten zu ermitteln.

Clean Code bezeichnet sauber strukturierten, wartbaren und leicht verständlichen Quellcode. Clean Code ist auch der Titel von Robert C. Martin's Buch, indem er Hilfestellungen zum Erzeugen von *clean* Code gibt.

Common Compiler Infrastructure ist eine umfangreiche Bibliothek die es ermöglicht [statische Code-Analysen](#) durchzuführen. Sie besteht aus den beiden Komponenten [CCI Metadata](#) und [CCI Code and AST Components](#).

Durchschnittliche Klassengröße ist eine [Metrik](#) einer Menge an Klassen und Interfaces und bezeichnet den Durchschnitt aller [Klassengrößen](#) der enthaltenen Klassen und Interfaces.

Durchschnittliche Komponentenabhängigkeit ist eine [Metrik](#) einer Menge an Typen und bezeichnet den Durchschnitt aller [kumulierter Komponentenabhängigkeiten](#) der enthaltenen Klassen und Interfaces.

Durchschnittliche Methodenlänge ist eine [Metrik](#) einer Menge an Klassen und Interfaces und bezeichnet den Durchschnitt aller [Methodenlängen](#) der Methoden aller enthaltenen Klassen und Interfaces.

Durchschnittliche Zyklomatische Komplexität ist eine [Metrik](#) einer Menge an Klassen und Interfaces und bezeichnet den Durchschnitt aller [zyklomatischer Komplexitäten](#) der Methoden aller enthaltenen Klassen und Interfaces.

Klassengröße ist eine [Metrik](#) von Klassen und Interfaces und bezeichnet im Kontext von Usus.NET die Anzahl der Methoden eines Typen.

Kumulierte Komponentenabhängigkeit ist eine [Metrik](#) von Klassen und Interfaces und bezeichnet die Anzahl anderer Klassen und Interfaces, von denen die Klasse oder das Interface direkt und indirekt abhängig ist.

Methodenlänge ist eine [Metrik](#) von Methoden und bezeichnet im Kontext von Usus.NET die Anzahl der Code-Zeilen die Logik enthalten.

Metrik ist eine Eigenschaft oder der Wert dieser Eigenschaft und wird für Methoden, Typen oder Namespaces bestimmt.

Namespaces mit zyklischen Abhängigkeiten ist eine [Metrik](#) einer Menge an Namespaces und bezeichnet im Kontext von Usus.NET die Anzahl aller Namespaces, die in einem Zyklus sind.

Nicht-statische öffentliche Felder ist eine [Metrik](#) einer Menge an Klassen und bezeichnet im Kontext von Usus.NET die Anzahl aller Klassen, die mindestens ein öffentliches Feld haben, das nicht statisch ist.

Program Database ist die pdb-Datei, die ein .NET-Compiler neben der [Assembly](#) erzeugt. Diese Datei enthält Dokument- und Zeileninformationen der Typen und Methoden in der entsprechenden [PE-Datei](#).

Softwarequalitätsindex ist eine [Metrik](#) eines Softwareprogramms, die verschiedene [Metriken](#) des Systems kombiniert und auf einen Wert reduziert. Je höher der Softwarequalitätsindex, desto höher ist auch die innere Qualität der Software.

Statische Code-Analyse bezeichnet die Analyse eines Softwareprogramms, ohne das dieses ausgeführt werden muss. Als Ergebnis wird ein Bericht, beispielsweise über [Metriken](#), erstellt.

Verteilung bezeichnet die statistische Häufigkeitsverteilung einer Wertesequenz. Sie kann durch eine approximierte Funktion beschrieben werden, dessen Parameter anhand eines Histogramms ermittelt werden.

Zyklomatische Komplexität ist eine [Metrik](#) von Methoden und bezeichnet die Anzahl der unabhängigen Möglichkeiten eine Methode zu durchlaufen.

Literaturverzeichnis

- [ao12] andrena objects. DIE Experten für agiles Software Engineering. <http://www.andrena.de/andrena-objects-die-experten-fuer-agiles-software-engineering>, 2012.
- [Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1999.
- [BFV12] Mike Barnett, Manuel Fahndrich, and Herman Venter. Common Compiler Infrastructure. <http://research.microsoft.com/cci/>, 2012.
- [Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 2nd edition, 1994.
- [CA08] Krzysztof Cwalina and Brad Abrams. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .Net Libraries*. Addison-Wesley, 2008.
- [CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.
- [Coh12] Mike Cohn. Agile Succeeds Three Times More Often Than Waterfall. <http://www.mountangoatsoftware.com/blog/agile-succeeds-three-times-more-often-than-waterfall>, 2012.
- [EDBS05] Christof Ebert, Reiner Dumke, Manfred Bundschuh, and Andreas Schmietendorf. *Best Practices in Software Measurement*. Springer, 2005.
- [Eva11] Jean-Baptiste Evain. Cecil - Mono. <https://github.com/jbevain/cecil>, 2011.
- [Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [FRS⁺10] Leif Frenzel, Nicole Rauch, Stefan Schürle, Marc Philipp, and Andreas Leidig. Usus Eclipse Plug-In. <http://code.google.com/p/projectusus/>, 2010.
- [GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [Gil11] Steve Gilham. Computing cyclomatic complexity with PowerShell and FxCop. <http://stevegilham.blogspot.de/2011/10/computing-cyclomatic-complexity-with.html>, 2011.

- [Gro02] Peter Grogono. Software Quality Control - Lecture Notes. <http://users.encs.concordia.ca/~grogono/CourseNotes/SOEN-345-Notes.pdf>, 2002.
- [Hag10] Stefan Hagen. CHAOS Report: Oft zitiert, aber was steckt dahinter? <http://pm-blog.com/2010/01/29/chaos-report-viel-zitiert-aber-was-steckt-dahinter/>, 2010.
- [HR02] Peter Hruschka and Chris Rupp. *Agile Softwareentwicklung für Embedded Real-Time Systems mit der UML*. Hanser, 2002.
- [ICS12] ICSsharpCode. Repository for NRefactory 5. <https://github.com/icsharpcode/NRefactory>, 2012.
- [KLR⁺06] Sigbert Klinke, Patrick Lehmann, Bernd Rönz, Sibylle Schmerbach, and Olga Zlatkin-Troitschanskaia. Exponentialverteilung. <http://mars.wiwi.hu-berlin.de/mediawiki/statwiki/index.php/Exponentialverteilung>, 2006.
- [KMN06] Künzer, Meister, and Nebe. geometrische Verteilung. <http://mo.mathematik.uni-stuttgart.de/inhalt/beispiel/beispiel461/>, 2006.
- [Kre08] Jason Kresowaty. FxCop and Code Analysis: Writing Your Own Custom Rules. <http://www.binarycoder.net/fxcop/pdf/fxcop.pdf>, 2008.
- [Kuh06] Eberhard Kuhn. Beschreibung der Berechnungsmethoden für Prozessqualitätsindex und Softwarequalitätsindex. Unveröffentlichtes Manuskript (siehe Anhang A.1), 2006.
- [LK94] Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics*. Prentice Hall, 1994.
- [LR09] Andreas Leidig and Nicole Rauch. Indicator-based Process and Software Quality Control in Agile Development Projects. Unveröffentlichter Technischer Artikel (siehe Anhang A.2), 2009.
- [Mar94] Robert C. Martin. OO Design Quality Metrics - An Analysis of Dependencies. <http://www.objectmentor.com/resources/articles/oodmetrc.pdf>, 1994.
- [Mar09a] Robert C. Martin. *Clean Code: A handbook of Agile Software Craftsmanship*. Prentice Hall, 2009.
- [Mar09b] Robert C. Martin. Metrics of Moment. <http://blog.objectmentor.com/articles/2009/06/08/metrics-of-moment>, 2009.
- [Mar10] Aaron Marten. Custom Extension Types with VSIX. <http://blogs.msdn.com/b/visualstudio/archive/2010/04/16/custom-extension-types-with-vsix.aspx>, 2010.
- [Mar11] Robert C. Martin. *The Clean Coder: A Code of Conduct for Professional Programmers*. Prentice Hall, 2011.

- [McC76] Thomas J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.
- [Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.
- [Mic10a] Microsoft. Adding a Menu to the Visual Studio Menu Bar. <http://msdn.microsoft.com/en-us/library/bb165473.aspx>, 2010.
- [Mic10b] Microsoft. DTE2 Interface. [http://msdn.microsoft.com/en-us/library/envdte80.dte2\(v=vs.100\).aspx](http://msdn.microsoft.com/en-us/library/envdte80.dte2(v=vs.100).aspx), 2010.
- [Mic10c] Microsoft. Expression Trees (C# and Visual Basic). <http://msdn.microsoft.com/en-us/library/bb397951.aspx>, 2010.
- [Mic10d] Microsoft. Force a VSPackage to Load. <http://msdn.microsoft.com/en-us/library/bb164612.aspx>, 2010.
- [Mic11] Microsoft. Program Database Files. <http://msdn.microsoft.com/en-us/library/ms241903.aspx>, 2011.
- [Mic12a] Microsoft. Common Language Infrastructure (CLI) ECMA-335. <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.pdf>, 2012.
- [Mic12b] Microsoft. Parallel LINQ (PLINQ). <http://msdn.microsoft.com/en-us/library/dd460688.aspx>, 2012.
- [Nab11] Zain Naboulsi. Code Metrics Maintainability Index. <http://blogs.msdn.com/b/zainnab/archive/2011/05/26/code-metrics-maintainability-index.aspx>, 2011.
- [New05] Philip Newcomb. Abstract Syntax Tree Metamodel Standard: ASTM Tutorial 1.0. http://www.omg.org/news/meetings/workshops/ADM_2005_Proceedings_FINAL/T-3_Newcomb.pdf, 2005.
- [NWGH11] Karen Ng, Matt Warren, Peter Golde, and Anders Hejlsberg. The Roslyn Project - Exposing the C# and VB compiler’s code analysis. <http://download.microsoft.com/download/E/A/D/EADDEC33E-FBA3-43BF-9226-427BDAC27610/Roslyn%20Project%200verview.docx>, 2011.
- [Oes01] Bernd Oestereich. *Objektorientierte Softwareentwicklung. Analyse und Design mit der Unified Modeling Language*. Oldenbourg, 5. edition, 2001.
- [O’R] O’REILLY. History of Programming Languages. http://oreilly.com/news/graphics/prog_lang_poster.pdf.
- [Ose11] Kirill Osenkov. Introducing the Microsoft ”Roslyn” CTP. <http://blogs.msdn.com/b/visualstudio/archive/2011/10/19/introducing-the-microsoft-roslyn-ctp.aspx>, 2011.

- [Osh09] Roy Osherove. *The Art of Unit Testing: With Examples in .Net*. Manning Publications Co., 2009.
- [PR10] Marc Philipp and Nicole Rauch. Einsicht und Handeln mit Usus. *eclipse magazin*, pages 37–39, 2010.
- [Sca01] Walt Scacchi. Process Models in Software Engineering. <http://www.ics.uci.edu/~wscacchi/Papers/SE-Encyc/Process-Models-SE-Encyc.pdf>, 2001.
- [Smi09a] Guy Smith. What is CCI Metadata? <http://ccimetadata.codeplex.com/wikipage?title=What%20is%20CCI%20Metadata>, 2009.
- [Smi09b] Josh Smith. WPF Apps With The Model-View-ViewModel Design Pattern. *MSDN Magazine*, 2009.
- [Smi10] Guy Smith. What is CCI Code? <http://cciast.codeplex.com/wikipage?title=What%20is%20CCI%20Code%3f>, 2010.
- [Tea12] Mono Team. Mono - Cross platform, open source .NET development framework. http://www.mono-project.com/Main_Page, 2012.
- [Veh07] Nadine Vehring. Software Design for Testability: Metriken und Darstellungsmöglichkeiten. <http://www.wi.uni-muenster.de/pi/lehre/ws0607/seminarQS/Ausarbeitungen/AusarbeitungNadineVehring.pdf>, 2007.
- [Wag] Artur Wagner. Einleitung und Einführung in das Thema Codeanalyse mit dem Schwerpunkt statische Codeanalyse. <http://www.ba-horb.de/fileadmin/media/it/studienprojekte/seminararbeiten/SWE3-071204/Codeanalyse.pdf>.
- [Wal01] Ernest Wallmüller. *Software-Qualitätsmanagement in der Praxis*. Hanser, 2001.
- [Wei61] Martin H. Weik. The ENIAC Story. <http://ftp.arl.mil/mike/comphist/eniac-story.html>, 1961.
- [Wei12a] Eric W. Weisstein. Chi-Squared Test. <http://mathworld.wolfram.com/Chi-SquaredTest.html>, 2012.
- [Wei12b] Eric W. Weisstein. Exponential Distribution. <http://mathworld.wolfram.com/ExponentialDistribution.html>, 2012.
- [Wei12c] Eric W. Weisstein. Geometric Distribution. <http://mathworld.wolfram.com/GeometricDistribution.html>, 2012.
- [Wei12d] Eric W. Weisstein. Maximum Likelihood. <http://mathworld.wolfram.com/MaximumLikelihood.html>, 2012.
- [WL11] Ralf Westphal and Stefan Lieser. Clean Code Developer. <http://www.clean-code-developer.de/MainPage.ashx>, 2011.
- [wor12] worldometers. How many computers are there in the world? <http://www.worldometers.info/computers/>, 2012.

Anhang

A.1 Manuskript über den Prozess- und Softwarequalitätsindex

Das unveröffentlichte Manuskript „Beschreibung der Berechnungsmethoden für Prozessqualitätsindex und Softwarequalitätsindex“ von Dr. Eberhard Kuhn befindet sich auf der beiliegenden CD in der Datei [Isis Beschreibung der Berechnungsmethoden.pdf](#).

A.2 Artikel über Isis

Der unveröffentlichte technische Artikel „Indicator-based Process and Software Quality Control in Agile Development Projects“ von Andreas Leidig und Nicole Rauch befindet sich auf der beiliegenden CD in der Datei [Isis.pdf](#).

A.3 Testpläne

Die Testpläne von Usus.NET befinden sich auf der beiliegenden CD in der Datei [Testpläne.xlsx](#).

A.4 Messergebnisse der Laufzeitmessungen

Die Ergebnisse der Beispielmessungen aus der Usus.NET-Evaluation befinden sich auf der beiliegenden CD im Verzeichnis [Messungen](#).

A.5 Visual Studio-Erweiterung Usus.NET

Die aktuelle Version der Visual Studio-Erweiterung Usus.NET befindet sich auf der beiliegenden CD in der Datei [Usus.NET.vsix](#). Eine Installation kann per Doppelklick durchgeführt werden.

