Hochschule Esslingen
University of Applied Sciences

Fachbereich Informationstechnik

Studiengang Softwaretechnik

Bachelorarbeit

Konzeption und Implementierung eines Analyse-Werkzeugs zur Identifikation von Code Smells auf Basis der .NET-Compiler-Plattform Roslyn

---

Vorgelegt von	Simon Birk
am	15. Januar 2016
Erstprüfer	Prof. Dr.-Ing. Andreas Rößler
Zweitprüfer	Prof. Dr.-Ing. Reinhard Schmidt

IT DESIGNERS GRUPPE IT DESIGNERS

IT-Designers GmbH

Esslingen

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.
Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Esslingen, 15. Januar 2016 _____

Simon Birk

Danksagung

An dieser Stelle möchte ich mich bei allen Personen bedanken, die mich bei der Erstellung dieser Arbeit unterstützt haben.

Mein besonderer Dank gilt Herrn Prof. Dr.-Ing. Andreas Rößler für die Ermöglichung und Betreuung dieser Arbeit. Darüber hinaus gilt mein Dank der IT-Designers GmbH, die mir einen Arbeitsplatz zur Verfassung dieser Arbeit zur Verfügung stellte. Explizit möchte ich mich bei Herrn Kevin Erath für seine tatkräftige Unterstützung bei diesem Projekt bedanken.

Zusammenfassung

Im Rahmen dieser Arbeit wurde auf Basis der .NET-Compiler-Plattform Roslyn von Microsoft ein Softwarepaket zur Analyse von C#-Code konzipiert und entwickelt. Die prototypische Analysesuite ist in der Lage, einige der von Robert C. Martin in dem Buch Clean Code beschriebenen Code Smells in Quelltext zu identifizieren. Dazu gehören unter anderem Verstöße gegen das Law of Demeter, Ringabhängigkeiten in Vererbungshierarchien und auskommentierter Code.

Mit Hilfe der implementierten Programme wurde eine Reihe von frei zugänglichen Open Source-Projekten der Plattform GitHub auf Code Smells untersucht. Anhand der Analyseergebnisse wurden Schwellenwerte für die relative Häufigkeit des Auftretens bestimmter Smells in dem untersuchten Code aufgestellt. Diese wurden in einer Bewertungsmatrix zusammengefasst. Anhand der bestimmten Vergleichswerte kann auf die Qualität eines zu untersuchenden Projekts geschlossen werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

CCD	Clean Code Developer
SRP	Single-Responsibility-Prinzip
TDD	Test-driven development
CC	Cyclomatic Complexity
LOC	Lines Of Code
NOM	Number Of Methods
NOI	Number Of Invocations
NOCOM	Number Of Comments
NOD	Number Of Declarations
LCOM	Lack Of Cohesion Of Methods
CBO	Coupling Between Objects
ggT	größten gemeinsamen Teiler
LINQ	Language Integrated Query
IDE	Integrated Development Environment
XP	Extreme Programming
CMS	Content-Management-System
LoD	Law of Demeter

Kapitel 1

Einleitung

> We know about as much about software quality problems as they knew about the Black Plague in the 1600s. We've seen the victims' agonies and helped burn the corpses. We don't know what causes it; we don't really know if there is only one disease. We just suffer – and keep pouring our sewage into our water supply.
>
> *Tom Van Vleck*

1.1 Motivation

„Clean Code" ist derzeit eines der meist diskutierten und einflussreichsten Bücher über Codequalität. [3, 30] Darin formuliert dessen Autor Robert C. Martin zahlreiche Regeln für sauberen Code. Das Buch identifiziert und beschreibt anhand von Fallstudien sogenannte Code Smells[1]. An Beispielen wird gezeigt, wie schlechter in guten Code transformiert werden kann. Der Autor möchte ein Bewusstsein für die Problematiken schaffen, die durch schlechten Code entstehen: Unsaubere Programme können mit der Weiterentwicklung zunehmend verwahrlosen und werden so immer anfälliger für Fehler. Der Entwicklungsprozess erreicht einen Punkt, an dem die Implementierung neuer Features extrem zeitaufwändig wird, da sich die Codebasis nur noch unter großem Aufwand erweitern lässt. Das Projekt kommt zum Erliegen und eine Neuentwicklung der Software erweist sich aus wirtschaftlicher Sicht als die bessere Alternative zur Weiterentwicklung.

Um sauberen Code zu schreiben, kann ein Softwareentwickler auf die Unterstützung von Codeanalysetools zurückgreifen. In der einfachsten Form wird der Quellcode textuell analysiert und nach Schlüsselwörtern der entsprechenden Programmierspra-

[1] Martin Fowler definiert Code Smells als ein oberflächliches Symptom einer (meist) tiefer liegenden Problematik. Code Smells sind in der Regel leicht zu identifizieren. [7]

che gesucht. Mit dieser Methode ist es nur schwer möglich, den zu untersuchenden Code wirklich zu verstehen und tiefgreifende Analysen durchzuführen.

Andere, wesentlich komplexere Hilfsmittel, bauen eine Art eigenen Compiler zum Übersetzen des Codes. Diese Methode ermöglicht eine tiefgründigere Analyse des Quelltextes. Unter anderem können so Refaktorisierungen und umfangreiche Untersuchungen durchgeführt werden. Allerdings erfordert die Entwicklung eines solchen Programmes umfassende Kenntnisse im Compilerbau sowie der zu untersuchenden Programmiersprache und ist somit meist sehr aufwändig.

Mit der Veröffentlichung der Open Source .NET-Compiler-Plattform Roslyn schlägt Microsoft einen neuen Weg ein: Der Compiler bietet umfassende Möglichkeiten um Quellcode zu analysieren, zu generieren und zu verstehen. Bisherige Compiler sind geschlossene Systeme: Sie übersetzen Quelltext in ausführbare Programme, ohne dass Entwickler Zugriff auf den Funktionsumfang einzelner Compilermodule haben. Roslyn bietet Schnittstellen, mit deren Hilfe Quelltext mit vergleichsweise überschaubarem Aufwand auf syntaktische und semantische Aspekte untersucht werden kann. Entwicklern werden somit einige Hürden für die Implementierung eigener Werkzeuge zur Codeanalyse und Refaktorisierung genommen.

1.2 Vorgehensweise

Im Rahmen dieser Arbeit werden geeignete Praktiken und Regeln von Martin herausgegriffen und erläutert. Auf Basis von Roslyn werden prototypisch Analyse-Tools entwickelt, mit deren Hilfe Quellcode auf die Einhaltung der vorgestellten Regeln und Praktiken überprüft werden kann. Diese Werkzeuge werden in einem Programmpaket zusammengefasst. Damit kann Code auf verschiedene Kriterien überprüft werden, um so potentielle Schwachstellen und Makel aufzudecken. Die Funktionalität des Paketes soll anhand von frei zugänglichen Open Source-Projekten validiert werden. Außerdem sollen die Analyseergebnisse der untersuchten Projekte zusammengefasst werden, um so Vergleichswerte für die Bewertung von Quellcode zu erhalten.

1.3 Aufbau der Arbeit

In Kapitel 2 wird erläutert, wie schlechter Code entsteht und wie sauberer Code aussehen sollte. Außerdem wird die Clean Code Developer Initiative beschrieben, die sich Martins Buch als Manifest nahm und daraus ein Wertesystem für Softwareentwickler ableitete.

Kapitel 3 befasst sich mit der Compiler-Plattform Roslyn. Es werden die verschiedenen Schichten der API, die abstrakte Darstellung von Quellcode in einem Syntaxbaum sowie das semantische Modell erläutert.

Im darauf folgenden Kapitel 4 werden verschiedene Arten von Analysewerkzeugen und deren Funktionsweise vorgestellt. Außerdem werden einige wichtige Metriken und deren Einsatz im Softwareentwicklungsprozess beschrieben.

Kapitel 5 widmet sich der Analyse verschiedener Bereiche von Quellcode. Es wird erläutert, warum sauberer Code im jeweiligen Bereich wichtig ist, welche Auswirkungen schlechter Code auf das gesamte Projekt haben kann und welche Maßnahmen dagegen ergriffen werden können. Anschließend wird der Entwurf und die Implementierung eines Werkzeugs beschrieben, das die Codebasis auf die geschilderten Schwächen untersucht.

Anhand des Quelltextes frei zugänglicher Open Source-Projekte wird die Funktionalität der Tools demonstriert. Es wird exemplarisch geprüft, inwieweit die vorgestellten Regeln und Leitsätze bei den untersuchten Programmen eingehalten werden.

Kapitel 6 bildet den Abschluss der Bachelorarbeit. Es beinhaltet ein Fazit, in dem über die Arbeit und deren Implementierung reflektiert wird. Außerdem enthält das Kapitel einen Ausblick, in dem eine mögliche Erweiterung und Fortsetzung des Projekts diskutiert wird.

Clean code is simple and direct. Clean code reads like well-written prose. Clean code never obscures the designers' intent but rather is full of crisp abstractions and straightforward lines of control.

Grady Booch

Kapitel 2

Clean Code

Robert Cecil Martin, auch unter dem Pseudonym Uncle Bob bekannt, ist seit den 1970er Jahren als professioneller Softwareentwickler tätig. In den letzten 40 Jahren hat er nach eigener Aussage an hunderten von Software-Projekten mitgewirkt.

Martin ist eine wichtige Instanz im Bereich der agilen Softwareentwicklung: Er war an der Entwicklung des agilen Manifests[1] beteiligt und ist Vorsitzender der Agile Alliance[2]. Martin tritt regelmäßig als Redner auf internationalen Konferenzen und Handelsmessen auf. [16]

Seit 1990 arbeitet Martin außerdem als Software-Berater. Die von ihm gegründete Firma Object Mentor ist ein Zusammenschluss aus internationalen, erfahrenen Softwareentwicklern und Managern, die andere Firmen bei ihren Projekten unterstützt. Object Metor hält weltweit Schulungen, unter anderem in den Bereichen Verfahrenverbesserung und objektorientierte Softwareentwicklung.
[18, vgl. S. xxix f.]

---

1 Das Agile Manifest umfasst Leitsätze und Prinzipien, die die Grundsätze der agilen Softwareentwicklung beschreiben. [2]

2 Die Agile Alliance ist eine Non-Profit-Organisation mit Mitgliedern auf der ganzen Welt. Die Organisation ist darin bemüht, die Praktiken und Prinzipien der agilen Softwareentwicklung voranzutreiben und zu verbreiten. [1]

Martin ist Autor zahlreicher Publikationen und Bücher, darunter auch:

- Pattern Languages of Program Design 3
- Extreme Programming in Practice
- Agile Software Development: Principles Patterns, and Practices
- Clean Code

Das Buch Clean Code erschien erstmals in englischer Sprachausgabe im Jahr 2009 unter dem Verlag Prentice Hall. Durch den einschlagenden Erfolg des Buches etablierte sich bereits kurz nach der ersten Veröffentlichung des Buches der Begriff „Clean Code“ als eine „saubere“ Form der Softwareentwicklung.

Martin sieht das Programmieren als eine Handwerkskunst. Um diese Kunst zu meistern, muss sich der Programmierer Könnerschaft aneignen. Dies geschieht durch Wissen und Arbeit. Der Entwickler muss Prinzipien, Patterns, Techniken und Heuristiken kennen um sein Handwerk zu beherrschen. Allerdings reicht Wissen alleine nicht aus. Erst durch harte Arbeit wird das Gelernte verinnerlicht. Durch kontinuierliche Verbesserung der eigenen Programmierkünste, dem Lernen aus Fehlern und durch umfangreiches Fachwissen kann der Programmierer zum Meister seines Fachs werden. [17, vgl. S. 21 ff.]

2.1 Schlechter Code – und die Folgen

Mang Jeder Softwareentwickler macht im Laufe seines Berufslebens Erfahrungen mit schlechtem Code. Für Aufgaben, die in wenigen Minuten erledigt werden könnten, werden mehrere Stunden benötigt. Arbeiten, die einfach sein sollten, sind unnötig komplex. Uncle Bob bezeichnet diese Behinderung als ein Waten durch einen Morast von schlechtem Code. Die Gründe für eine mangelnde Qualität der Codebasis sind vielfältig und unterscheiden sich je nach Projekt: Der Entwickler ist unerfahren und weiß nicht, was er tut. Oder viel schlimmer, ihm ist es schlicht egal. Häufig haben Entwickler auch gar nicht die Zeit über die Auswirkungen und

Folgen ihres Programmierstils nachzudenken. Laut Martin ist einer der Gründe für eine schlechte Codebasis der Druck, dem die Entwickler ausgesetzt sind. Die Deadline hängt im Nacken und das Programm muss unbedingt fertig werden; es fehlt scheinbar die Zeit, guten Code zu schreiben oder den bereits geschriebenen Code aufzuräumen. Oft hat der Entwickler einfach keine Lust mehr, sich noch mehr als unbedingt nötig mit dem Projekt herumzuschlagen. [17, vgl. S. 27 f.]

All das führt häufig zu schlampigen Implementierungen und Fixes, die Chaos im Code verursachen. Da momentan allerdings keine Zeit vorhanden ist, wird beschlossen, das Chaos später zu bereinigen. Doch das Gesetz von LeBlanc besagt: „Später gleich niemals“. [17, vgl. S. 28]

Bei einem neuen Projekt kommen die meisten Teams anfangs sehr schnell voran. Der Fortschritt ist solange zügig, bis sie durch chaotischen Code ausgebremst werden und ein Vorankommen nur noch schleichend möglich ist. Wird Code an einer Stelle geändert, treten gleich an mehreren anderen Stellen Fehler und unerwünschte Nebeneffekte auf. Scheinbar kleine Anpassungen stellen sich als komplex und umfangreich heraus. Der Code verkommt immer mehr zu einem Geflecht aus Verzweigungen, Varianten und Knoten, die immer mehr wuchern, bis sie kaum noch verstanden werden können. Eine wirtschaftliche Bereinigung des Codes ist kaum noch möglich.

Der Code wird immer chaotischer und die Produktivität des Entwicklerteams sinkt weiterhin. Das Projekt ist im Begriff zu scheitern und das Management startet einen Rettungsversuch, indem es mehr Personal zuweist. Da die neuen Teammitglieder unter dem Druck stehen, das Projekt zu retten und die Produktivität nochmals zu steigern, verursachen sie nur noch mehr Chaos und erreichen somit das Gegenteil: Die Produktivität sinkt abermals. Außerdem müssen die neuen Entwickler sich erst einmal mit dem Projekt vertraut machen und belegen damit zusätzlich Resourcen, denn sie werden von den alten Mitgliedern des Teams bei der Einarbeitung unterstützt. [17, vgl. S. 28 f.]

Das Team kommt nicht mehr voran und wendet sich an das Management – ein Redesign wird gefordert. Nun steckt die Führung in einer Zwickmühle: Einerseits

können nicht alle verfügbaren Kapazitäten in das Redesign gesteckt werden. Andererseits ist die Produktivität inakzeptabel. Es muss etwas geschehen. Die Leistungsträger des alten Teams bilden eine neue Arbeitsgruppe, die mit dem Redesign des alten Projekts beauftragt wird. Die restlichen Mitglieder des alten Teams kümmern sich um die Wartung und Weiterentwicklung des bisherigen Systems.

Es kommt zum Wettlauf zwischen den beiden Teams: das neue Team muss alle Methoden des alten Programms in ein neues System übernehmen. Außerdem müssen sämtliche Änderungen und Erweiterungen des bestehenden Systems in das neue übernommen werden. Schließlich kann das Management nicht das alte System durch die Neuentwicklung ersetzen, solange nicht alle Methoden übernommen wurden. Dieser Prozess kann sich laut Martin bis zu zehn Jahre hinziehen. Somit schließt sich der Kreis: Die meisten der ursprünglichen Mitglieder des neuen Teams sind nicht mehr da und die neuen Mitglieder fordern wiederholt ein Redesign, da das neue System abermals zum Chaos verkommen ist. [17, vgl. S. 29]

Viele Projekte verlaufen ähnlich Martins Beschreibung. Und einer der Hauptauslöser ist schlecht geschriebener Code.

2.2 Wie sieht sauberer Code aus?

Es gibt zahlreiche Definitionen davon, wie sauberer Code auszusehen hat, Beispiele folgen in diesem Abschnitt. Die verschiedenen Interpretationen beruhen auf unterschiedlichen Denkschulen erfahrener Entwickler. Mit gesammelter Erfahrung entwickeln sich bei jedem Programmierer eigene Vorstellungen davon, wie sauberer Code auszusehen hat. Dabei gibt es nicht die einzig „richtige“ Definition. Martin befragte diverse namhafte und erfahrene Entwickler, wie ihrer Meinung nach sauberer Code aussehen sollte. [17, vgl. S. 40]

Im Folgenden werden die Aussagen dreier renommierter Softwareentwickler vorgestellt:

Bjarne Stroustrup Erfinder von C++

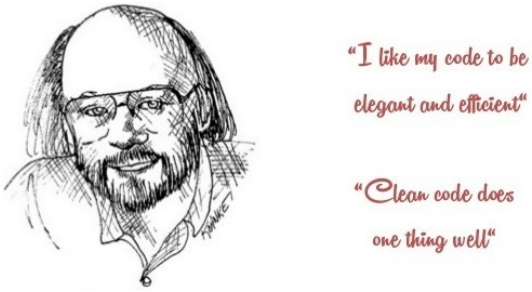

Abbildung 2.1: Bjarne Stroustrup [17, vgl. S.32]

„Mein Code sollte möglichst elegant und effizient sein. Die Logik sollte geradlinig sein, damit sich Bugs nur schwer verstecken können, die Abhängigkeiten sollten minimal sein, um die Wartung zu vereinfachen, das Fehler-Handling sollte vollständig gemäß einer vordefinierten Strategie erfolgen, und das Leistungsverhalten sollte dem Optimum so nahe wie möglich kommen, damit der Entwickler nicht versucht ist, den Code durch Ad-hoc-Optimierungen zu verunstalten. Sauberer Code erledigt eine Aufgabe gut." [17, S. 32]

Nach Stroustrup muss sauberer Code elegant und effizient sein. Elegante Dinge sind angenehm anzusehen, der Blick verweilt gerne darauf. Elegant geschriebener Code ist folglich angenehm zu lesen. Damit kann beispielsweise die Namensgebung der Variablen, Methoden und Klassen gemeint sein. Diese sollte passend gewählt sein, um die Absichten des Codes möglichst präzise zu beschreiben. Mit Effizienz kann zum einen die Ausführungsgeschwindigkeit, zum anderen der Umfang des Codes gemeint sein. Eine effiziente Methode erledigt ihre Aufgabe mit möglichst wenigen Zeilen Code. Wenige Verzweigungen und eine geringe Verschachtelungstiefe resultieren zwangsläufig in geradliniger Logik. Ebenfalls von essenzieller Wichtigkeit ist die Fehlerbehandlung. Wird diese vernachlässigt, können daraus Speicherlecks, Race-Bedingungen und letztlich eine fragile Anwendung hervorgehen. Interessant ist die Annahme Stroustrups, dass suboptimaler Code andere Entwickler dazu ver-

leitet, „Verbesserungen“ vorzunehmen, die den Code noch mehr verschlechtern. Als letzten Punkt führt Stroustrup auf, dass sauberer Code **eine** Aufgabe gut erledigen soll – die Definition des Single-Responsibility-Prinzips (SRP). [17, vgl. S. 33]

Michael Feathers Autor von *Working Effectively with Legacy Code*

Abbildung 2.2: Michael Feathers [17, vgl. S.36]

„Ich könnte alle Eigenschaften auflisten, die mir bei sauberem Code auffallen; aber es gibt eine übergreifende Qualität, die alle anderen überragt: Sauberer Code sieht immer so aus, als wäre er von jemandem geschrieben worden, dem dies wirklich wichtig war. Es fällt nichts ins Auge, wie man den Code verbessern könnte. Alle diese Dinge hat der Autor bereits selbst durchdacht; und wenn Sie versuchen, sich Verbesserungen vorzustellen, landen Sie wieder an der Stelle, an der Sie gerade sind: Sie sitzen einfach da und bewundern den Code, den Ihnen jemand hinterlassen hat – jemand, der sein ganzes Können in sorgfältige Arbeit gesteckt hat.“ [17, S. 36]

Michael Feathers listet bei seiner Definition von sauberem Code weniger einzelne Kriterien auf, die zu erfüllen sind. Vielmehr stellt er einen persönlichen Bezug zwischen dem Entwickler und seinem Code her: „Sauberer Code sieht immer so aus, als wäre er von jemandem geschrieben worden, dem dies wirklich wichtig war.“ Er beschreibt damit die sorgfältige Vorgehensweise von professionellen Entwicklern, die sich mit ihrem Code auseinandersetzen. Sie sind nicht nur an der Lösung, sondern auch an der bestmöglichen Umsetzung interessiert. Auch nach Feathers muss sauberer Code elegant sein, wenngleich er es etwas anders formuliert: „Sie sitzen einfach da und bewundern den Code, den Ihnen jemand hinterlassen hat.“

Dave Thomas Gründer der Object Technology International[3]

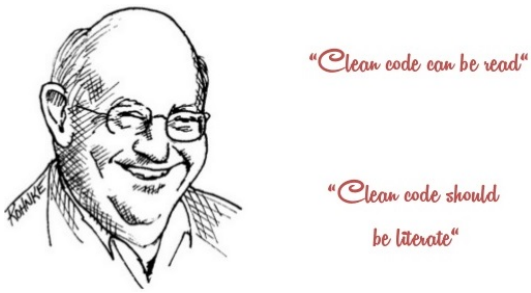

Abbildung 2.3: Dave Thomas [17, vgl. S.35]

„Sauberer Code kann von anderen Entwicklern gelesen und verbessert werden. Er verfügt über Unit- und Acceptance-Tests. Er enthält bedeutungsvolle Namen. Er stellt zur Lösung von Aufgaben nicht mehrere, sondern eine Lösung zur Verfügung. Er enthält minimale Abhängigkeiten, die ausdrücklich definiert sind, und stellt ein klares und minimales API zur Verfügung. Code sollte »literate« sein, da je nach Sprache nicht alle erforderlichen Informationen allein im Code klar ausgedrückt werden können." [17, S. 35]

Wie auch für Stroustrup ist für Thomas eine aussagekräftige Namensgebung und somit gute Lesbarkeit äußerst wichtig. Damit soll es Entwicklern leichter fallen, mit einer fremden Codebasis zu arbeiten. Auch die Anwesenheit von Tests sind für die Sauberkeit essenziell. Egal wie gut und lesbar Code geschrieben ist, ohne Tests kann die Funktionalität eines Programmes nach einer Refaktorisierung nicht gewährleistet werden. Außerdem sollte der Code »literate« sein, das heißt, es sollen an Passagen Kommentare verwendet werden, deren Aussagekraft alleine durch den Code nicht mehr gewährleistet werden kann. Der Code soll eine Einheit mit den Kommentaren bilden. Ebenso gilt: Weniger ist mehr! Eine minimale API ist einer umfangreichen Programmierschnittstelle vorzuziehen, vorausgesetzt mit beiden lässt sich dasselbe Ergebnis erzielen. [17, vgl. S. 35 f.]

Im Kern sind sich die meisten Entwickler darüber einig, wie sauberer Code auszusehen hat. Allerdings ist es nicht ausreichend, sauberen Code lediglich zu erkennen.

[3] OTI wurde 1988 in Ottawa gegründet und 1996 von IBM aufgekauft. Die Firma entwickelte unter anderem Eclipse. [26]

Viel wichtiger ist die Frage: Wie kann sauberer Code geschrieben werden? In Kapitel 5 werden einzelne Kriterien, die sauberen Code ausmachen, herausgegriffen und genauer erläutert. Es wird beschrieben, mit welchen Maßnahmen die Codequalitätität verbessert werden kann.

2.3 Clean Code Developer Initiative

Mit Martins „Clean Code“ als Fundament wurde kurz nach dessen Veröffentlichung die Clean Code Developer (CCD) Initiative von Ralf Westphal und Stefan Lieser gegründet. Sie waren begeistert von Martins praxisorientiertem Ansatz, sich auf die Codequalität zu konzentrieren. Allerdings fürchteten sie, dass die im Buch beschriebenen Prinzipien und Praktiken sich ohne weitere Unterstützung nicht schnell genug verbreiten würden. Außerdem wollten sie auf Martins Buch ein didaktisches Konzept aufbauen, das helfen soll, dessen Inhalte zu verinnerlichen und auch anzuwenden zu können. Also erstellten Westphal und Lieser ein Wiki, in dem sie die im Buch beschriebenen Prinzipien und Praktiken neben weiteren in ein Wertesystem gliedern (siehe Kapitel 2.3.1). Denn Prinzipien und Praktiken sind nur Mittel zum Zweck – sie müssen einem höheren Ziel dienen, um eine Existenzberechtigung zu haben. Westphal und Lieser glauben, in den von ihnen definierten Werten dieses höhere Ziel gefunden zu haben. Anhand dieser Werte lassen sich auch eventuelle neue Methodiken bewerten. Neben dem Wertesystem führt die CCD Initiative auch Tugenden (siehe Kapitel 2.3.2) ein, die den Entwickler dabei unterstützen sollen, das Gelernte auch konsequent umzusetzen.
[33, vgl. S.10 f.]

2.3.1 Die Werte

Das Wertesystem der CCD Initiative stützt sich auf die folgenden vier Werte, die für alle Clean Code Developer gelten sollten:

- Evolvierbarkeit
- Korrektheit
- Produktionseffizienz
- Kontinuierliche Verbesserung

Evolvierbarkeit

Um Software verändern zu können, muss diese eine innere Struktur haben, die dies begünstigt. Diese Eigenschaft der Weiterentwicklungsfähigkeit wird als Evolvierbarkeit bezeichnet. Zumeist wird Software über einen längeren Zeitraum eingesetzt. Über diese Zeitspanne können sich die Rahmenbedingungen und Anforderungen ändern – die Software muss angepasst werden.

Im Optimalfall ist es egal, zu welchem Zeitpunkt Änderungen implementiert werden sollen – die Kosten sind immer dieselben. Die Praxis allerdings zeigt, dass der Aufwand mit der Zeit exponentiell ansteigt. Anfangs können neue Features noch mit relativ geringem Aufwand implementiert werden und die steigenden Kosten werden zumeist kaum wahrgenommen. Es kommt allerdings der Punkt, an dem sich der steigende Aufwand immer mehr bemerkbar macht. Am Ende ist es schließlich kaum mehr möglich, die Software zu ändern. Der Code ist zu komplex geworden und muss neu entwickelt oder mit großem Aufwand umgestaltet werden.

Software muss auf Evolvierbarkeit ausgelegt sein, denn sie kann nachträglich nur schwer erhalten werden. Einer der Gründe für schlechte Weiterentwicklungsfähigkeit von Software ist die enge Kopplung verschiedener Komponenten aneinander.

Nach dem Single-Responsibility-Prinzip (SRP) hat eine Klasse genau eine Verantwortlichkeit und damit eine Aufgabe. Und folglich gibt es auch nur einen Grund,

diese zu ändern. Hat eine Klasse mehr als nur eine Verantwortlichkeit, steigt damit auch ihre Komplexität. Um eine Klasse ändern zu können, muss sie erst verstanden werden, was mit zunehmender Komplexität immer schwerer wird. Außerdem steigt die Kopplung – plötzlich sind Klassen voneinander abhängig, die in Teilen nichts miteinander zu tun haben sollten. Klassen müssen eine klar definierte Verantwortlichkeit haben und der Grad der Kopplung muss immer im Blick gehalten werden. Nur so kann verhindert werden, dass der Code verwildert und Evolvierbarkeit gewährleistet werden kann. [24]

Korrektheit

Software muss in aller erster Linie korrekt sein und die Funktionalität muss gewährleistet werden. Dazu gehört auch ein sparsamer Gebrauch mit den zur Verfügung stehenden Ressourcen. Die Antwortzeiten müssen ebenfalls in einem vertretbaren Rahmen liegen.

Korrektheit fängt bei der Entwicklung an – unklar definierte Anforderungen müssen geklärt werden. Die Abnahmekriterien für ein Feature müssen bekannt sein. Es ist nicht ausreichend, nach der Entwicklungsphase die Software nur von der Testabteilung prüfen zu lassen. Automatisierte Unit Tests unterstützten Entwickler bereits während des Entwicklungsprozesses dabei, die Korrektheit des Programmes zu gewährleisten. [24]

Produktionseffizienz

Mit steigender Ineffizienz der Softwareproduktion steigen auch Preis und Entwicklungszeit. Aus diesem Grund sollten bei der Entwicklung, wann immer es möglich ist, Vorgänge automatisiert werden. Die Produktionseffizienz und die Fehlerquote einer Software gehen ineinander über: Wenn bei einem Programm häufig nachgebessert werden muss, ist der Produktionsprozess nicht effizient. Außerdem bedeutet Produktionseffizienz, dass Software über einen längeren Zeitraum weiterentwickelt werden kann, ohne dass das Entwicklerteam von vorne beginnen muss.

Mit der Produktionseffizienz können die anderen Werte ebenfalls in Relation gestellt werden. Wird unendlich viel Zeit in die Evolvierbarkeit von Software gesteckt, läuft etwas aus dem Ruder. [24]

Kontinuierliche Verbesserung

Ohne Selbstreflexion ist keine Weiterentwicklung möglich. Um sich selbst zu verbessern, muss über abgeschlossene Projekte nachgedacht werden. War der eingeschlagene Weg der richtige? Oder gab es etwa eine andere Lösung, die auf einfachere Weise zum Ziel geführt hätte? Bei der Softwareentwicklung sollte ständig reflektiert werden: Beim gemeinsamen Pair Programming, beim Code Review oder in der Retrospektive nach jeder Iteration. [24]

2.3.2 Die Tugenden

Um die im in Kapitel 2.3.1 beschriebenen Werte erfüllen zu können, sollte sich ein Clean Code Developer verschiedene Tugenden aneignen:

Abbildung 2.4: Die CCD-Tugenden [33, S.14]

Die elf im Bild dargestellten Tugenden (grau) machen die Werte (blau) fassbar, da sie als konkrete Handlungen und Haltungen formuliert sind. Durch diese Tugenden wird ein Entwickler beim Erreichen der Werte unterstützt. [33, vgl. S. 14]

2.3.3 Die Grade

Für alle Lernwillige, die Clean Code praktizieren möchten, führt die Initiative sieben Grade ein. Dabei ist jedem Grad eine Farbe zugeordnet – ähnlich der Gürtelfarbe des Dans[4] einiger asiatischen Kampfsportarten. Ein Schüler kann durch ein Armband der entsprechenden Farbe zeigen, an welchem Grad er gerade arbeitet. Der Unterschied zu dem Dan-System liegt allerdings darin, dass der Lernende mit dem Erreichen des „höchsten" Grades noch lange kein Großmeister ist. Vielmehr wird das Grade-System als Zyklus verstanden. Der Schüler arbeitet sich durch die verschiedenen Grade und nach Abschluss des letzten Grades beginnt er wieder beim ersten. Der Wechsel in den nächsten Grad erfolgt, wenn der Entwickler 21 Tage lang erfolgreich den aktuellen Grad praktiziert hat.

Das folgende Bild skizziert die verschiedenen Grade und die jeweils zu erlernenden Prinzipien und Praktiken:

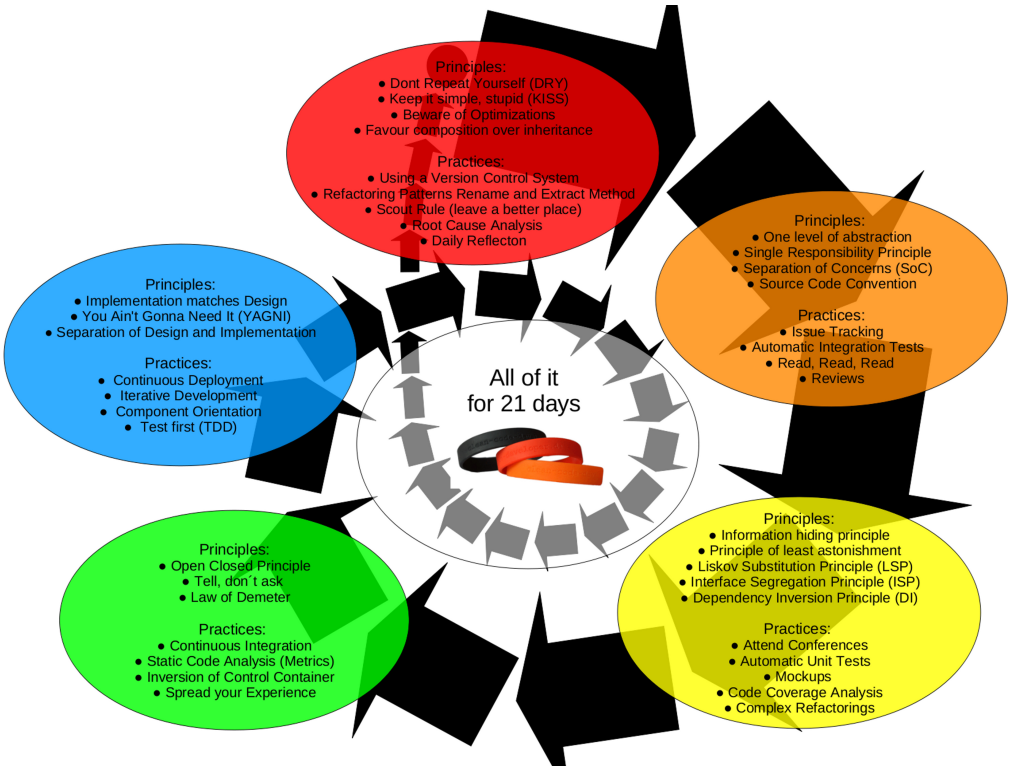

Abbildung 2.5: Der Clean Code Development-Zyklus [12]

4 Der Dan bezeichnet den erreichten Meistergrad eines Kampfsportlers. [4]

Bei jedem Grad arbeitet der Schüler an verschiedenen Prinzipien und Praktiken, die wiederum den in Kapitel 2.3.1 beschriebenen Werten zuarbeiten. In der Grafik nicht dargestellt ist der nullte, schwarze Grad. Jeder Entwickler, der prinzipiell an CCD interessiert ist, befindet sich automatisch im schwarzen Grad. Dieser symbolisiert lediglich das Interesse an der Materie. Die eigentliche Arbeit beginnt mit dem ersten, roten Grad. Die darin zu erlernenden Prinzipien und Praktiken sind so gewählt, dass ein Interessent mit minimalem Aufwand einsteigen kann. Die folgenden Grade erfordern zunehmend mehr Engagement und Willen für Veränderung. Schließlich fasst der letzte, weiße Grad alle vorhergegangen Grade zusammen. Nachdem auch der weiße Grad erfolgreich gemeistert wurde, beginnt der nächste Iterationszyklus und der Entwickler richtet seinen Fokus wieder vermehrt auf die Prinzipien und Praktiken der vorangegangenen Grade. [23]

2.4 Code-Katas und Coding-Dojos

Ebenfalls von Stefan Lieser und Ralf Westphal stammt das Konzept der CCD School. Auf der Webseite der Schule finden sich zahlreiche Katas, die in einem Coding Dojo angeboten werden. Auch hier wird die Analogie zum Kampfsport hergestellt: Der Begriff Kata bezeichnet eine Übungsform des Karates. Dabei werden vordefinierte Bewegungsabläufe, Schläge und Tritte nacheinander ausgeführt. Diese Übungen finden in der Regel in einem Dojo statt, der Übungshalle verschiedener japanischer Kampfkünste. [11, 5]

Die Begriffe der Code-Katas und Coding-Dojos gehen auf Laurent Bossavit und Emmanuel Gaillot zurück. Sie leiteten im Jahr 2005 eine Sitzung mit dem Namen Coding Dojo auf der XP2005[5] Conference in Sheffield. Im Rahmen der Sitzung wurde gemeinsam mit den Teilnehmern eine Übung programmiert – das erste Code-Kata. Auf ihrer Webseite[6] haben sie für Lernwillige zahlreiche Katas bereitgestellt. [18, vgl. S. 89 f.]

---

5 Extreme Programming (XP)

6 http://codingdojo.org/

Die Idee ist allerdings „Pragmatic“ Dave Thomas zuzuschreiben: Er vergleicht den Prozess der Softwareentwicklung mit Musik und Kunst. Um bekannter Musiker zu werden, ist es hilfreich, die Theorie und die Funktionsweise des zu spielenden Instrumentes zu kennen. Profisportler hingegen benötigen in der Regel körperliche Fitness und Talent. Wahre Größe wird allerdings nur durch die praktische Anwendung erreicht. Tägliche Übung ist vonnöten, um als Sportler, Musiker oder Softwareentwickler wirklich erfolgreich zu werden. Thomas fordert, dass diese Übung getrennt vom Berufsalltag erfolgen muss: „In software we do our practicing on the job, and that's why we make mistakes on the job. We need to find ways of splitting the practice from the profession. We need practice sessions.“ [29]

Im Rahmen der Clean Code Developer School werden zahlreiche Code-Katas (Übungen) angeboten, deren Ausmaß stark variiert. Von kleinen Function Katas, dessen Umfang sich auf einen Algorithmus innerhalb einer Methode beschränkt, bis hin zu Architecture Katas, die eine verteilte Implementierung erfordern. Letztere sind zum Teil so umfangreich, dass eine Implementierung der Übung von einer einzelnen Person unrealistisch ist – vielmehr geht es um den Architekturentwurf. Die Lösungen von Library Katas sind zumeist das Erstellung und Zusammenfassen von mehreren Klassen und Methoden in eine Bibliothek. Bei Application Katas geht es um die Entwicklung einer kompletten Anwendung – von der Benutzerschnittstelle bis hin zum Ressourcenzugriff.

Die Problemstellungen der Katas sind unterschiedlich. In der Regel wird eine Aufgabe in Form von fachlichen Anforderungen gestellt. Was letztlich damit trainiert wird, bleibt den Übenden überlassen. Vom Erlernen methodischer Vorgehensweisen wie Test-driven development (TDD) oder dem Verwenden eines Versionskontrollsystems, der praktischen Anwendung der CCD Prinzipien bis hin zu Dependency Injection können verschiedene Aspekte des Clean Code Developments gezielt trainiert werden. [25]

They told me computers could only do arithmetic.

Grace Cooper

Kapitel 3

Roslyn

Das Visual Studio von Microsoft gehört zu den verbreitetsten und komfortabelsten Entwicklungsplattformen. Zahlreiche Features wie die automatische Vervollständigung von Quellcode mittels *IntelliSense*, die intelligente Umbenennung von Methoden und Klassen oder die Navigationsfunktionen *Find all References* und *Go To Definition/Go To Declaration* erleichtern Entwicklern den Arbeitsalltag. So selbstverständlich diese Methoden für den Anwender erscheinen mögen, so erfordert ihre Entwicklung doch ein tiefgreifendes Verständnis der verwendeten Programmiersprache sowie der dazugehörigen Projektstruktur.

Der von Visual Studio verwendete .NET-Compiler `csc.exe` verfügt intern über die notwendigen Funktionalitäten um die oben genannten Features umzusetzen, allerdings verhindert die Black Box-Implementierung eine Wiederverwendung des Codes. In Folge dessen musste Microsoft eine zweite Compilerinfrastruktur für die in Visual Studio angebotenen Analysetools nachbauen. Diese Problematik beschränkte sich nicht alleine auf die Tools von Microsoft: sollten Tools zur Codeanalyse oder für automatische Refaktorisierungen entwickelt werden, mussten zumindest Teile des C#-Compilers nachgebaut werden. Dieser Aufwand konnte zumeist nur von wenigen großen und spezialisierten Teams betrieben werden.

Dies war mitunter Grund für eine Neuentwicklung des C#-Compilers. Ein weiteres Ziel war es, eine modulare Compiler-Klassenbibliothek zu schaffen, die von IDEs

und von Entwicklern entworfenen Werkzeugen genutzt werden kann, um Codeanalyse und Refaktorisierung zu betreiben. Das bisher unter dem Codenamen „Roslyn“ bekannte Projekt wurde 2015 aus dem Beta-Status gehoben und löst in Visual Studio 2015 den bisherigen Compiler ab. [28, vgl. S. 16 f.]

3.1 Aufbau

Die Compiler-Plattform ist in drei Schichten unterteilt. Das folgende Bild zeigt deren schematischen Aufbau:

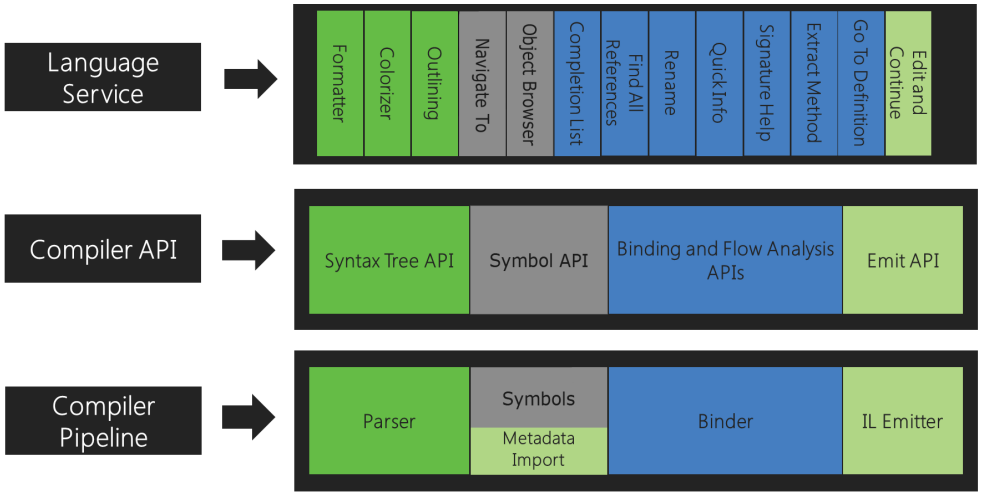

Abbildung 3.1: Systemarchitektur von Roslyn [9]

Grundlage ist eine Compiler-Pipeline, die in vier separate Komponenten unterteilt ist. Die Funktionalität jeder Komponente bildet jeweils eine Phase des Kompiliervorgangs. Auf die jeweiligen Komponenten kann mit den entsprechenden APIs zugegriffen werden. Auf Basis der APIs wurden verschiedene Language Services implementiert, wie zum Beispiel das Einfärben von Schlüsselwörtern im Code (*Colorizer*) oder das *Extract Method* Feature. Das Ergebnis jeder Phase ist ein Modell, das Zugriff auf den jeweiligen Informationsgewinn bietet. Im Folgenden werden die einzelnen Phasen kurz beschrieben:

1. In der ersten Phase zerlegt der Parser den C#-Quellcode in die einzelnen Bestandteile (siehe Kapitel 3.2.1-3.2.3) der Sprache. Diese werden daraufhin

in einem Syntaxbaum (siehe Kapitel 3.2) zusammengesetzt, der den Code vollständig repräsentiert. Der Syntaxbaum ist das Ergebnis der ersten Phase.

2. Anschließend werden in Phase zwei Informationen aus den Deklarationen des Quelltextes extrahiert: Alle benannten Symbole werden in einer hierarchischen Symboltabelle[1] gelistet, die das Ergebnis dieser Phase bildet.

3. Der Binder verknüpft in Phase 3 die im Syntaxbaum gespeicherten Identifier mit den Einträgen der Symboltabelle. Das Ergebnis ist ein semantisches Modell des Codes.

4. In der letzten Phase werden aus allen gewonnenen Informationen vom Emitter fertige Assemblies erstellt.

Die API des Compilers hat keinerlei Abhängigkeiten zu Visual Studio und kann folglich auch außerhalb der Entwicklungsumgebung verwendet werden. [9]

3.2 Syntax Trees

Ein Syntaxbaum ist eine strukturierte Repräsentation des Quelltextes, der drei maßgebliche Eigenschaften aufweist:

Unveränderbarkeit: Syntaxbäume sind unveränderlich und damit Thread-sicher. Ein Syntaxbaum ist folglich immer die Momentaufnahme des momentanen Zustandes des Codes. Durch die Unveränderbarkeit können mehrere Threads gleichzeitig an dem selben Syntaxbaum arbeiten, ohne dass der Baum gesperrt oder dupliziert werden müsste. Soll ein Syntaxbaum geändert werden, kann dies nur auf einer Kopie des Baumes erfolgen. [9]

[1] Symboltabellen sind Datenstrukturen, die hauptsächlich im Compilerbau verwendet werden. Darin werden Informationen über das Vorkommen von Variablen– und Klassennamen, Objekten, Klassen, Interfaces etc. verwaltet. [31]

Rückführbarkeit: Durch den Umstand, dass sämtliche Informationen des Quelltextes im Syntaxbaum vorhanden sind, ist dieser wieder in Source-Code überführbar. Es ist möglich, von jedem Knoten des Baumes aus die textuelle Repräsentation des darunterliegenden Baumes zu erhalten. Auf diese Weise kann Code generiert und editiert werden: Durch das Erstellen eines Syntaxbaumes wird implizit der äquivalente Code erstellt. Wird ein Syntaxbaum geändert und in einem neuen Baum abgespeichert, entspricht das einer Änderung des Quelltextes. [9]

Vollständigkeit: Alle Informationen des Codes werden in ihrer Ganzheit in einem Syntaxbaum abgelegt. Von grammatikalischen Konstrukten, lexikalischen Tokens bis hin zu Leerzeichen, Kommentaren und Präprozessor-Anweisungen wird alles strukturiert im Syntaxbaum gespeichert. Außerdem werden auch syntaktische Fehler des Source-Code im Syntaxbaum dargestellt, selbst wenn der Quelltext nicht vollständig ist oder sich aufgrund eines Tippfehlers nicht kompilieren lässt. [9]

Syntaxbäume sind Datenstrukturen, bei denen nicht-terminierende Elemente die Eltern weiterer Elemente sind. Ein Baum besteht aus Nodes, Tokens und Trivia. Diese Elemente kennen ihre eigene Position innerhalb des Baumes. So lässt sich zu einem Element beispielsweise die Position als Zeilennummer innerhalb der Quelltext-Datei ermitteln. [9]

Syntaxbäume werden aus verschiedenen Arten von Knoten aufgebaut. Die folgenden Kapitel zeigen die unterschiedlichen Elemente des Baumes.

3.2.1 Syntax Nodes

Syntax-Knoten sind die primären Elemente von Syntaxbäumen. Sie repräsentieren syntaktische Konstrukte wie Deklarationen, Expressions, Clauses und Statements. Die verschiedenen Kategorien werden je durch eine eigene, von $\mathrm{SyntaxNode}$ abgeleitete Klasse, vertreten. Dabei sind Syntax-Knoten nicht-terminierend – sie beherbergen selbst weitere Kind-Knoten. Auf die Eltern- und Kind-Knoten kann über die entsprechenden Parent und $\mathrm{ChildNodes}$ Properties zugegriffen werden. [9]

3.2.2 Syntax Tokens

Syntax Tokens sind die terminierenden Elemente eines Syntaxbaums: Sie haben keine weiteren Kind-Knoten. Zu den Syntax Tokens gehören Schlüsselworte, Bezeichner, Literale und Satzzeichen. Sie sind als Werttypen implementiert und besitzen alle dieselbe Struktur. Abhängig von der Art des Tokens werden die jeweiligen Properties unterschiedlich interpretiert. So werden beispielsweise numerische Werte als Integer Literal Token dargestellt. [9]

3.2.3 Syntax Trivia

Als Syntax Trivia werden die Teile des Quelltextes bezeichnet, die für den Compiler unerheblich sind. Dazu gehören Kommentare, Leerzeichen und Präprozessor-Anweisungen. Da Trivia nicht zu der regulären Sprachsyntax gehört und auch zwischen zwei Tokens Code stehen kann, wird sie nicht als Kind eines Knotens im Syntaxbaum dargestellt. Trotzdem müssen sie Teil des Baumes sein, damit der Quelltext in vollem Umfang dargestellt werden kann. Dazu wird Trivia beim Parsen des Quelltextes jeweils einem Token zugeordnet. Auf sie kann mittels der Properties LeadingTrivia und TrailingTrivia Properties eines Tokens zugegriffen werden. Auch die Syntax Trivia sind als Werttypen implementiert. [9]

3.3 Syntax Visualizer

Im SDK der .NET Compiler Platform ist das Syntax Visualizer Tool enthalten. Dabei handelt es sich um ein Werkzeug, das den Syntaxbaum der momentan geöffneten Quellcode-Datei grafisch als Tree View visualisiert. Wird der Code verändert oder erweitert, wird die Anzeige sofort automatisch aktualisiert. Das Tool ist somit sehr gut geeignet, um den Aufbau des Syntaxbaums zu verstehen.

Dabei wird die momentane Position des Cursors im Code repräsentativ im Syntaxbaum hervorgehoben. Dies erleichtert vor allem anfangs den Umgang mit der Roslyn-API, da mit dem Syntax Visualizer direkt überprüft werden kann, aus wel-

chen Elementen sich eine bestimmte Code-Zeile zusammensetzt. Außerdem kann zu jedem Knoten des Baumes ein Syntax Graph erstellt werden. Dieser ist eine grafische Repräsentation des ausgewählten Knotens mit allen dazugehörenden Kind-Knoten.

Der nachfolgender Screenshot zeigt den Tree View sowie den Syntax Graph eines einfachen HelloWorld Beispiels:

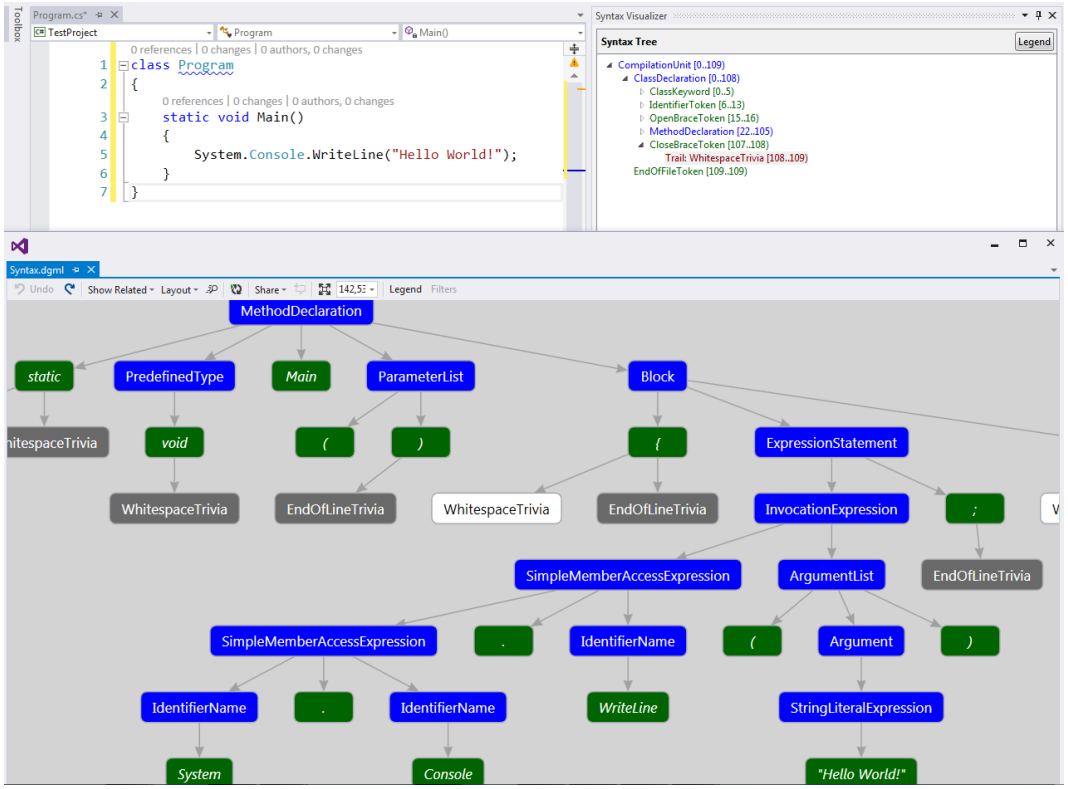

Abbildung 3.2: Syntax Visualizer - Tree View und Syntax Graph

Aus dem Graphen lässt sich beispielsweise ablesen, dass es sich um eine statische Methode handelt, die keine Übergabeparameter (leere ParameterList) und keinen Rückgabewert (PredefinedType void) hat. Innerhalb der Main-Methode befindet sich ein Code-Block, in dem ein Ausdruck (ExpressionStatement) steht. Dabei handelt es sich um die System.Console.WriteLine-Methode mit dem String-Argument "Hello World!".

3.4 Semantik

Syntaxbäume repräsentieren die lexikalische und syntaktische Struktur von Quellcode. Die darin enthaltenen Informationen sind ausreichend, um Deklarationen und Logik zu beschreiben. Syntax Nodes und Tokens können den Code vollständig abbilden. Sollen aber Informationen zu den Konstrukten gefunden werden, die hinter einem Identifier-Token stecken, kann der Syntaxbaum nicht weiterhelfen. Wo wurde die referenzierte Methode deklariert? Existiert die verwendete Variable wirklich, oder handelt es sich um einen Tippfehler? Das semantische Modell kann diese Fragen beantworten.

Code kann Elemente von zuvor kompilierten Bibliotheken referenzieren. Zu diesen ist kein Quellcode und folglich kein Syntaxbaum vorhanden. Trotzdem können diese Elemente verwendet werden – das semantische Modell verbindet den Quelltext mit den vorkompilierten Assemblies. [9]

3.4.1 Compilation

Eine Compilation ist eine Zusammenstellung von allem, was nötig ist, um ein Programm zu kompilieren. Dazu gehören Referenzen, Compileroptionen und Quellcode Dokumente.

In der Compilation werden alle Elementes des Quellcodes detailliert beschrieben. Zu jedem Typ, Attribut und jeder Variablen hält die Compilation ein Symbol (siehe Kapitel 3.4.2). Mit Hilfe der Compilation können Symbole miteinander verglichen und in Verbindung gebracht werden. Unabhängig davon, ob sie im Quelltext der zu untersuchenden Projektmappe oder in einer importierten Bibliothek deklariert wurden. [9]

3.4.2 Symbole

Ein Symbol ist ein einzigartiges Element, das im Quellcode deklariert oder aus einer Assembly importiert wird. Zu allem, was durch einen Namen referenziert werden

kann, existiert ein Symbol. Dazu gehören Namensräume, Typen, Methoden, Felder, Attribute, Events, Parameter und lokale Variablen.

Eine Compilation bietet Methoden, mit denen das Symbol zu einem im Code verwendeten Namen gefunden werden kann. Jedes Symbol enthält außerdem weitere Informationen, wie zum Beispiel andere referenzierte Symbole. Die verschiedenen Typen von Symbolen werden in Roslyn durch verschiedene, von ISymbol abgeleitete, Interfaces dargestellt. [9]

3.4.3 Semantisches Modell

In dem semantischen Modell werden alle semantischen Informationen einer einzelnen Quellcode Datei dargestellt. Das Modell beinhaltet Informationen zu den folgenden Punkten:

- Symbolen, die an einer bestimmten Stelle des Quelltexts referenziert werden
- Dem Ergebnistyp eines beliebigen Ausdrucks
- Allen Code-Diagnosen[2]
- Variablenfluss durch verschiedene Regionen von Code

Diese Fülle an Informationen hat ihren Preis: Es ist verhältnismäßig rechenintensiv, das semantische Modell anzufordern. Deswegen ist es empfehlenswert, eine Instanz des Modells wiederzuverwenden, sollte es mehrfach gebraucht werden. [9]

[2] Der Compiler erstellt einen Satz an Diagnosen, die zum Beispiel Zuweisungsfehler und verschiedene Warnungen in Bereichen Syntax und Semantik enthalten.

Nicht alles was zählt, kann gezählt werden, und nicht alles was gezählt werden kann, zählt!

Albert Einstein

Kapitel 4

Tools und Metriken zur Codeanalyse

Zahlreiche kommerzielle und auch kostenfreie Tools versprechen Entwicklern, durch ihren Einsatz eine bessere Codequalitätität zu erreichen. Viele dieser Werkzeuge basieren auf Metriken, die versuchen, verschiedene Elemente des Codes zu quantifizieren. Das folgende Kapitel stellt einige bekannte Metriken und Analysetools vor und beschreibt den Nutzen sowie die Risiken, die aus der Verwendung solcher Werkzeuge hervorgehen.

4.1 Statische und dynamische Codeanalyse

Prinzipiell wird zwischen zwei unterschiedlichen Analysetechniken unterschieden: der statischen und der dynamischen Codeanalyse.

Bei der statischen Codeanalyse wird der Code zur, beziehungsweise vor der Kompilierzeit nach vorgegebenen Regeln untersucht – der Code wird hierzu nicht ausgeführt. Es existieren auch statische Werkzeuge, die auf Basis der Binärdateien arbeiten, wie zum Beispiel NDepend[1] und Reflector[2]. Diese Programme arbeiten

[1] http://www.ndepend.com/

[2] www.red-gate.com/products/dotnet-development/reflector/

auf Basis des Zwischencodes mit dem Vorteil, unabhängig von der verwendeten Programmiersprache zu sein. Allerdings besteht keine eins zu eins Beziehung zwischen dem Quellcode und der Binärdatei – der exakte, ursprüngliche Code kann nicht wiederhergestellt werden. Auch Kommentare existieren in den Binärdateien nicht mehr.

Die Typprüfung der Übergabeparameter an eine Funktion sowie die Prüfung, ob veraltete oder unsichere Methoden verwendet werden, sind klassische Beispiele.

Alle statischen Analysen besitzen die folgenden, gemeinsamen Merkmale:

- Es gibt keine Ausführung der zu prüfenden Software.
- Alle statischen Analysen können prinzipiell ohne Computerunterstützung durchgeführt werden.
- Der Code einer Anwendung wird in seiner Ganzheit betrachtet.

Die nicht computergestützte Ausführung von statischen Analysen macht zum Beispiel bei der Validierung des Quelltextes im Code-Review durchaus Sinn. Bei den meisten statischen Analysen wäre die manuelle Durchführung jedoch zu zeitintensiv und deswegen nicht sinnvoll bzw. ökonomisch. [15, vgl. S. 40]

Im Gegensatz dazu wird bei der dynamischen Analyse das Laufzeitverhalten des Codes untersucht. Es werden zum Beispiel Ausführungszeiten von Prozeduren, Laufzeitwerte von Variablen oder Parametern sowie das Verhalten unter verschiedenen Laufzeitumgebungen überprüft. Der Nachteil hierbei ist, dass nicht der gesamte Code überprüft wird – sondern immer nur der Teil, der auch tatsächlich ausgeführt wird.

Alle dynamischen Analysemethoden haben folgende Merkmale gemein:

- Die übersetzte, ausführbare Software wird mit konkreten Eingabewerten versehen und ausgeführt.
- Es kann in der realen Betriebsumgebung getestet werden.

- Dynamische Testtechniken sind Momentaufnahmen, da immer nur der ausgeführte Code betrachtet wird.

In einfachster Form ist die dynamische Analyse das Ausführen der zu testenden Software mit den Eingaben eines Testers. Durch den stichprobenartigen Charakter kann allerdings die Korrektheit von Software nicht bewiesen werden. Da dynamische Analysen universell anwendbar sind, besitzen sie eine große praktische Bedeutung. [15, vgl. S. 35 f.]

Der Rahmen dieser Arbeit beschränkt sich auf die statischen Analysen und Werkzeuge. Im folgenden Abschnitt werden die Möglichkeiten und Einsatzgebiete verschiedener Metriken und Tools beschrieben.

4.2 Metriken

In Bezug auf Softwareentwicklung stehen Metriken für Maße, die Softwarequalität bewerten – es handelt sich um quantifizierte Beobachtungen, deren Ergebnis eine Maßzahl ist. Diese Maßzahlen werden verwendet, um verschiedenen Quellcode formal vergleichen und bewerten zu können. Dabei können die verschiedenen Metriken in der Regel auf das ganze Programm oder auch auf einzelne Klassen oder Methoden angewendet werden. Der Anwendungsraum wird als Softwareeinheit bezeichnet. Nachfolgend wird eine Auswahl gängiger Softwaremetriken vorgestellt. [32, vgl. S. 263]

4.2.1 Lines Of Code (LOC)

Das LOC-Maß ist eine der ersten und simpelsten Softwaremetriken: Es gibt an, aus wie vielen Codezeilen eine Softwareeinheit besteht. Trotzdem ist sie eine der beliebtesten Metriken, fast jedes Tool unterstützt sie. Auch Visual Studio verwendet sie zur Bewertung des Wartbarkeitsindex – einer Zahl zwischen 0 und 100, die besagt, wie wartbar Quellcode ist.

Die Zahl alleine gibt lediglich einen groben Überblick über den Umfang der Softwareeinheit. Erst im Vergleich mit anderen Softwareeinheiten können etwa Aussagen über die Effizienz getroffen werden. Dabei ist allerdings nicht definiert, wie Kommentare oder Leerzeilen behandelt werden. Neben dem einfachen LOC-Maß gibt es noch erweiterte Definitionen, die diese Fragen klären. [15, vgl. S. 235]

Es werden beispielsweise die LOC von zwei Methoden A und B gegenübergestellt, die beide den größten gemeinsamen Teiler (ggT) zweier ganzer Zahlen bestimmen sollen. Methode A bewältigt diese Aufgabe in acht Zeilen Code, Methode B hingegen benötigt dafür 23 Zeilen Code.

Es ist anzunehmen, dass Methode A effizienter arbeitet, da für dieselbe Aufgabe deutlich weniger LOC benötigt werden. Außerdem können die LOC je nach verwendeter Programmiersprache stark variieren. Es könnte auch sein, dass in Methode B mehr geschieht als nur das Bestimmen des ggT – was wiederum das SRP verletzen würde. Außerdem kann das Ergebnis verzerrt werden: So werden beispielsweise je nach LOC Implementierung verschiedene Anweisungen in einer Zeile als einzelne oder mehrere separate Zeilen interpretiert.

Aus dem Beispiel geht hervor, dass LOC, wie auch jede andere Metrik, mit Vorsicht zu verwenden ist. Es liefert lediglich grobe Vergleichswerte, die nicht überinterpretiert werden dürfen.

LOC kann auch mit anderen Maßen kombiniert werden: Möglich ist beispielsweise der Quotient Kommentarzeilen / LOC. Dieser gibt an, welcher Prozentsatz des Codes Kommentare sind.

4.2.2 Zyklomatische Komplexität (McCabe-Metrik oder CC)

Die zyklomatische Komplexität, auch bekannt als McCabe-Metrik, gibt an, wie viele linear unabhängige Programmpfade in einer Softwareeinheit zu finden sind. Demzufolge kann aus der Maßzahl direkt abgelesen werden, wie viele Testfälle für eine Softwareeinheit benötigt werden, um eine vollständige Testabdeckung für die Methode zu erreichen – ein Testfall für jeden Programmpfad.

Die zyklomatische Komplexität berechnet sich für eine Softwareeinheit wie folgt: Die Anzahl der Entscheidungen oder bedingten Anweisungen plus eins. Nachfolgend ein Codebeispiel zur Bestimmung des größten gemeinsamen Teilers: [15, vgl. S. 236]

```
1  public void GGT (int a, int b)
2  {
3    while (a > 0 && b > 0)
4    {
5      if (a > b)
6      {
7        a = a % b;
8      }
9      else
10     {
11       b = b % a;
12     }
13   }
14   return a + b;
15 }
```

Listing 4.1: Größter gemeinsamer Teiler [10]

Das Programm lässt sich auch als Kontrollflussgraph darstellen. Darin wird ersichtlich, welche Verzweigungen und Pfade es durch den Programmablauf gibt:

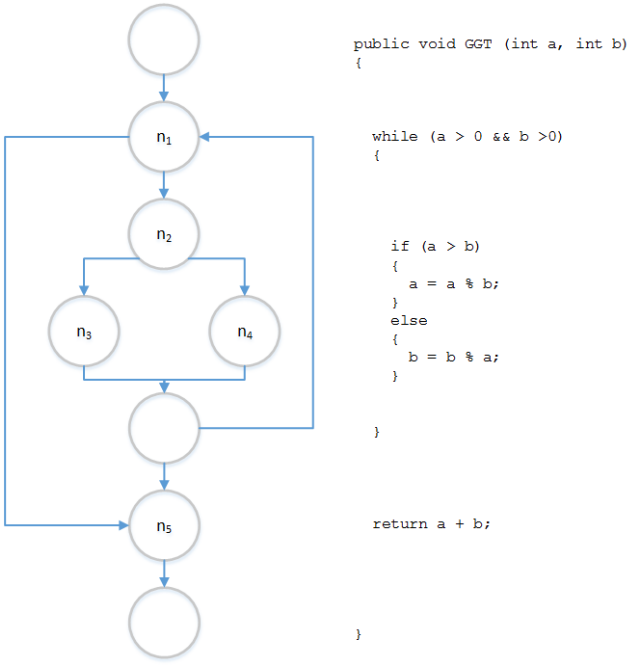

Abbildung 4.1: Kontrollflussgraph zu GGT-Programm

In der erste Zeile des Programmes GGT befindet sich eine abweisende Schleife mit zwei Konditionen: $a > 0$ und 6b > 0. Ab hier gibt es demzufolge schon drei mögliche Pfade durch das Programm: Ist a kleiner oder gleich null, wird die Schleife übersprungen. Ist a größer als null, wird die zweite Bedingung überprüft. Ist diese ebenfalls erfüllt, wird die Schleife betreten. In der Schleife selbst befindet sich eine if/else Kontrollstruktur. Wird die Schleife betreten, wird entweder der if- oder der else-Zweig ausgeführt. Es gibt demzufolge drei mögliche Pfade durch das Programm – die zyklomatische Komplexität ist folglich vier.

4.2.3 Lack Of Cohesion Of Methods (LCOM)

Mit der Kohäsion wird ausgedrückt, inwieweit die Methoden einer Klasse zusammengehören und arbeiten. Maximale Kohäsion innerhalb einer Klasse ist anzustreben, um Zusammengehörigkeiten zu kapseln. Die LCOM Metrik misst die Gleichartigkeit innerhalb einer Klasse. Diese kann unter anderem mit der folgenden Vorgehensweise bestimmt werden:

Für jedes Feld einer Klasse wird bestimmt, von welchem Prozentsatz der Klassen-Methoden es genutzt wird. Die ermittelten Prozente werden gemittelt und von 100% abgezogen. Je geringer das Ergebnis, desto höher ist die Kohäsion der Klasse.

Eine hohe Kohäsion lässt darauf schließen, dass die Klasseneinteilung gut gewählt ist. Wohingegen eine niedrige Kohäsion Hinweise darauf gibt, dass die betreffende Klasse eventuell besser aufgeteilt werden sollte. [27, vgl. S. 4]

4.2.4 Coupling Between Objects (CBO)

Die Kopplung drückt aus, wie stark Klassen voneinander abhängig sind. Stark gekoppelter Code ist weniger evolvier- und wiederverwertbar. CBO ist die Anzahl der Klassen, von denen die zu untersuchende Klasse abhängig ist. Dabei gelten zwei Klassen als voneinander abhängig, wenn Klasse A auf Methoden oder Attribute zugreift, die in Klasse B deklariert sind. Aus Vererbung resultierende Abhängigkeiten werden bei dieser Metrik nicht berücksichtigt. [27, vgl. S. 5]

4.2.5 Vergleichs- und Schwellenwerte für Metriken

Michele Lanza und Radu Marinescu verfolgen in ihrem Buch „Object-Oriented Metrics in Practice“ den Ansatz, Metriken zu kombinieren. Durch die Kombinationen werden neue Kennzahlen geschaffen, anhand derer Projekte qualitativ beurteilt und verglichen werden können. Dabei stützen sie sich im Wesentlichen auf drei Metriken[3]:

Die durchschnittliche Anzahl an Methoden pro Klasse

$$\frac{Number\ Of\ Methods\ (NOM)}{Number\ Of\ Classes\ (NOC)}$$

liefert Anzeichen über die Kohäsion von Klassen. Ein hoher Wert lässt vermuten, dass die betreffende Klasse zu viele Verantwortlichkeiten übernimmt.

Die durchschnittliche Anzahl von Codezeilen pro Methode

$$\frac{Lines\ Of\ Code\ (LOC)}{Number\ Of\ Methods\ (NOM)}$$

beschreibt den Umfang von Funktionen. Hohe Werte lassen darauf schließen, dass die Methoden der untersuchten Softwareeinheit zu groß sind. Normalerweise sollten Methoden klein sein, Martin sagt dazu: „Methoden sollten kaum jemals länger als 20 Zeilen sein.“ [17, S.64]

Die durchschnittliche zyklomatische Zahl pro Codezeile

$$\frac{Cyclomatic\ Complexity\ (CC)}{Lines\ Of\ Code\ (LOC)}$$

beschreibt die Komplexität eines Projektes. Je höher dieser Wert, desto höher ist die Dichte von Verzweigungspunkten im Code.

[3] Für die Berechnung der LOC wurden hier öffnende und schließende geschweifte Klammern, die in einer eigenen Zeile stehen, mitgezählt.

Diese drei Metriken wurden gewählt, da sie Kriterien erfüllen, die für eine statistische Auswertung wichtig sind:

1. Die Maßzahlen liefern Auskunft über den Umfang von Klassen und Methoden sowie die Komplexität eines Projekts.
2. Sie sind unabhängig von einander.
3. Sie sind unabhängig von der Größe des Projekts.

Um statistische Vergleichswerte zu haben, analysierten Lanza und Marinescu 45 Java und 37 C++ Projekte. Das Ergebnis ist in folgender Tabelle dargestellt:

	Java				C++			
Metric	Low	Average	High	Very High	Low	Average	High	Very High
CC/Line of Code	0,16	0,20	0,24	0,36	0,20	0,25	0,30	0,45
LOC/Method	7	10	13	19,5	5	10	16	24
NOM/Class	4	7	10	15	4	9	15	22,5

Tabelle 4.1: Schwellenwerte für Metriken in Java und C++ Projekten [19, vgl. S. 16]

Die Schwellenwerte „Low“, „Average“, „High“ und „Very High“ wurden dabei wie folgt ermittelt:

- Low: $\textit{Arithmetisches Mittel der Werte} - \textit{Standardabweichung}$
- Average: $\textit{Arithmetisches Mittel}$
- High: $\textit{Arithmetisches Mittel der Werte} + \textit{Standardabweichung}$
- Very High: $(\textit{Arithmetisches Mittel der Werte} + \textit{Standardabweichung}) * 1,5$

Anhand dieser Daten kann schnell ein Überblick über Softwareeinheiten gewonnen werden: Beinhaltet eine Java-Klasse vier Methoden, so ist das vergleichsweise „wenig“, 25 Zeilen Code in einer C++ Methode hingegen ist „sehr viel“.
[19, vgl. S.14 ff.]

Zum Thema dieser Arbeit gehört die Analyse und Bewertung von C#-Quelltext. In Ermangelung von C#-Referenzwerten für diese Metriken wurde im Rahmen dieser

Ausarbeitung ein Programm entwickelt, das diese Werte liefert. Das Tool ist in der Lage, mehrere Projekten einzulesen und für jedes Projekt die genannten Metriken zu berechnen.

Als Referenzprojekte werden populäre Open Source-Projekte[4] auf GitHub herangezogen. Darunter befinden sich unter anderem die CoreFX Libraries[5] der .NET Foundation. Diese beinhalten die Implementierungen der Kern-Komponenten von .NET, wie zum Beispiel System.Collections und System.IO. Insgesamt werden 31 C#-Projekte untersucht.

Von der Analyse ausgenommen sind Code-Dateien und Projekte mit Unit-Tests. Diese enthalten oft Methoden mit generiertem Code, die eine extreme Größe[6] erreichen können, wodurch das Ergebnis verfälscht werden kann.

Anders als bei den Auswertungen von Lanza und Marinescu werden bei dieser Implementierung öffnende und schließende geschweifte Klammern, die jeweils eine eigene Zeile belegen, nicht mitgezählt. Grund hierfür ist, dass die Codekonventionen für C# [20] vorsehen, dass geschweifte Klammern grundsätzlich in einer eigenen Zeile zu stehen haben. Vor allem bei Projekten mit kleinen Methoden fallen die beiden extra Zeilen stark ins Gewicht.

In der folgenden Tabelle sind die Ergebnisse der Analyse von 31 C#-Projekten dargestellt:

	C#			
Metric	Low	Average	High	Very High
CC/Line of Code	0,11	0,27	0,43	0,64
LOC/Method	1	8	15	23
NOM/Class	3	7	11	17

Tabelle 4.2: Schwellenwerte für Metriken in C#-Projekten

4 Eine vollständige Liste der untersuchten Projekte befindet sich im Anhang.

5 https://github.com/dotnet/corefx

6 Alleine die Initialisierung eines String-Arrays mit Testdaten in der Kern-Bibliothek System.Globalization.Extensions erstreckt sich beispielsweise über 18.000 Zeilen Code.

Am ehesten lassen sich die Ergebnisse mit den Analysen der Java-Projekte von Lanza und Marinescu vergleichen, da sich Java und C# sehr ähnlich sind. Die durchschnittliche Anzahl an Methoden je Klasse ist dieselbe, lediglich die unteren und oberen Schwellenwerte weichen ein wenig von einander ab.

Bei den Metriken, die das LOC-Maß verwenden, fällt der Unterschied wesentlich größer aus. Das liegt vor allem an der unterschiedlichen Berechnung der LOC-Zahl. Die LOC-Werte von Lanza und Marinescu beinhalten Zeilen, die je eine geschweifte Klammer enthalten. Nach Java-Konventionen stehen jeweils die schließenden geschweiften Klammern in einer eigenen Code-Zeile. Dadurch weicht die LOC-Zahl der Java-Projekte je Abfrage, Methode, Schleife etc. um eins von den C#-Projekten ab.

Um die Werte anzugleichen, kann die gesamte Zyklomatische Komplexität auf den LOC-Wert der C#-Projekte addiert werden. Da der Wert der Cyclomatic Complexity (CC) der Anzahl von Verzweigungen, überwiegend Schleifen, Methoden und Abfragen, entspricht, wird der LOC-Wert so gut angepasst. Lediglich Operatoren wie ||, && und :?, die ebenfalls die CC erhöhen, werden so nicht erfasst.

Mit den ermittelten Schwellenwerten kann schnell ein grober Überblick über den Zustand eines Projekts gewonnen werden. Werden die beschriebenen Metriken im eigenen Projekt errechnet und mit den Schwellenwerten verglichen, ist sofort erkennbar, ob und bei welchen Klassen und Methoden im Projekt Refaktorisierungsbedarf besteht.

4.3 Analysetools und deren Einsatzgebiet

Derzeit gibt es zahlreiche Werkzeuge mit denen Codeanalyse und Refaktorisierungen betrieben werden können. Die meisten setzten dabei auf unterschiedliche Schwerpunkte. CAT.NET[7] von Microsoft ist beispielsweise auf das Erkennen von Sicherheitslücken spezialisiert. NDepend hat über 80 Metriken im Repertoire, mit

[7] http://www.microsoft.com/en-us/download/details.aspx?id=19968

denen IL-Code analysiert werden kann. Die Ergebnisse werden in Grafiken aufbereitet. Andere Tools, wie zum Beispiel StyleCop[8], überprüfen, ob Richtlinien bezüglich des Kodierstils oder Styleguides eingehalten werden.

Viele Analysewerkzeuge lassen sich teilweise bis vollständig in die Entwicklungsumgebung integrieren. So bietet die Visual Studio Integration von ReSharper[9] beispielsweise eine on-the-fly Codeanalyse. Noch vor dem Kompiliervorgang werden Fehler und Code-Smells direkt im Editor angezeigt. Je nach Konfiguration werden diese als Hinweis, Warnung oder sogar als Fehler dargestellt – im letzten Fall kann der Code nicht mehr kompiliert werden. Das Tool meldet beispielsweise redundante Casts und listet nicht verwendete lokale Variablen. Das Tool ist auch zu tiefgreifenderen Analysen in der Lage: So werden zum Beispiel Schleifen erkannt, die sich eleganter und effizienter mit Language Integrated Query (LINQ)[10] formulieren lassen. Darauf aufbauende, umfangreiche Refaktorisierungsmöglichkeiten runden das Paket ab.

Auch bei der kontinuierlichen Integration spielen Analyse-Tools eine zunehmend größere Rolle. Neben der automatischen Ausführung der Unit Tests können beim Build auch automatisch statische Codeanalysen durchgeführt werden. SonarQube[11] lässt sich beispielsweise mit Plugins in gängige Buildsoftware wie Jenkins[12] einbinden. So können mit entsprechender Konfiguration nach jedem Build oder Commit Statistiken über die Codequalität generiert werden. Dabei ist die Zeitachse interessant: Negative Trends und Veränderungen können so rasch erkannt und Gegenmaßnahmen können ergriffen werden. [22]

---

8 stylecop.codeplex.com/

9 www.jetbrains.com/resharper/

10 LINQ ist eine in .NET integrierte Abfrage-Möglichkeit, die mit einer SQL-artigen Syntax Zugriff auf z.B. XML-Daten oder Elementen von Listen bietet. [6]

11 http://www.sonarqube.org/

12 jenkins-ci.org/

4.4 Nutzen und Risiken

Die beschriebenen Tools und Metriken können bei richtiger Verwendung von äußerstem Nutzen bei der täglichen Arbeit sein. So kann der Entwickler unmittelbar sehen, dass die veraltete Methode, die er gerade aufrufen möchte, Sicherheitsrisiken birgt. Wird festgestellt, dass der gewählte Klassenname nicht treffend ist, können mit Refaktorisierungstools ganz leicht sämtliche Vorkommnisse der betreffenden Klasse im Code umbenannt werden – inklusive Vorkommnisse in Kommentaren. Tools formatieren den geschriebenen Code automatisch so, dass der Quelltext mit den Stylguides des Auftraggebers konform ist. Die korrekte Anwendung kann in besserem Quellcode resultieren und Zeitersparnisse mit sich bringen.

Der Begriff Metrik stammt ursprünglich aus dem Griechischen und bedeutet übersetzt Kunst des Messens. In Bezug auf Softwaremetriken und Analysetools liegt die eigentliche Kunst nicht im Messvorgang selbst, sondern in der richtigen Interpretation der gemessen Ergebnisse und in der Umsetzung von Verbesserungsvorschlägen.

Werden Maße nicht korrekt interpretiert, sind sie bestenfalls nutzlos, schlimmstenfalls sogar schädlich – es wird an Stellen optimiert und verbessert, an denen kein Bedarf dafür besteht. Es ist schwer und auch nicht immer sinnvoll, einheitliche Ober- und Untergrenzen für ein bestimmtes Maß festzulegen. Oft kann mit Hilfe von Erfahrungswerten bestimmt werden, welcher Wertebereich für ein Maß gut oder gerade noch tolerierbar ist. Abweichungen von diesen Bereichen können, müssen allerdings nicht ein Problem sein. [15, vgl. S. 227]

Folgendes Beispiel zeigt, welche direkten Auswirkungen das Überbewerten von Metriken auf das Verhalten eines Entwicklerteam haben kann. Angenommen, die Testabdeckung für ein Projekt soll möglichst hoch sein. Diese Vorgabe wird mit Tools überwacht und fest in den Build-Prozess verdrahtet: sinkt die Abdeckung unter 90 Prozent, schlägt der Build fehl. Oft sind die Mitglieder des Teams wie besessen davon, diese Vorgaben einzuhalten – unabhängig davon, ob die daraus resultierenden Ergebnisse einen positiven oder negativen Einfluss auf das Projekt haben. Die bisherige Code Coverage liegt bei 91 Prozent. Nun wird ein großer Teil

des Projekts refaktorisiert und es werden einige Methoden entfernt, da sie nicht mehr benötigt werden. Allerdings existiert zu all diesen Methoden ein Test. Dadurch sinkt die Testabdeckung des Projekts auf 85 Prozent und das Tool schlägt Alarm. Daraus können drei Konsequenzen gezogen werden:

1. Die Änderungen werden rückgängig gemacht, sodass der ungenutzte Code im System bleibt. Damit werden auch die Tests nicht gelöscht und die Testabdeckung bleibt bei über 90 Prozent.

2. Es werden nachträglich Tests zu bisher ungetesteten Methoden geschrieben. Dabei kann es vorkommen, dass Test geschrieben werden, die keine sinnvolle Validierung vornehmen – Hauptsache, es werden 90 Prozent Code Coverage erreicht.

3. Die Anforderungen an die Vorgabe wird gesenkt.

Die ersten beiden Optionen sind Paradebeispiele dafür, dass Statistiken zu ernst genommen werden können. Bei der ersten Option wird bewusst auf eine Verbesserung des Projekts verzichtet, nur um die Vorgabe zu erreichen. Werden, wie in Option zwei beschrieben, Pseudotests für eine Methode geschrieben, besteht die akute Gefahr, dass bei Änderungen der besagten Methode potentielle Fehler nicht erkannt werden. Die Entwickler wiegen sich in trügerischer Sicherheit, denn alle Tests sind „grün“ und die betreffende Methode wird ja durch einen Test abgedeckt. Unter dem Standpunkt der Weiterentwicklung von Software ist es gut, nicht verwendeten Code aus dem Projekt zu entfernen. Dass die Testabdeckung sinkt, ist die logische Konsequenz daraus. Demzufolge ist Option drei die einzig vernünftige Vorgehensweise. [13, vgl. S. 126 f.]

Metriken und Analysetools können durchaus hilfreiche Werkzeuge sein, wenn es darum geht, die Qualität des Quelltextes zu verbessern. Werden diese jedoch falsch eingesetzt und deren Ergebnisse unangebracht interpretiert, kann dies schlimmstenfalls sogar negative Auswirkungen auf die Codequalität haben und einem Fortschritt im Wege stehen.

> Any fool can write code that a computer can understand. Good programmers write code that humans can understand.
>
> *Martin Fowler*

Kapitel 5

Konzeption und Realisierung von Code Smell Detektoren

Im folgenden Kapitel werden eine Reihe von Code Smells erläutert und die im Rahmen dieser Arbeit erstellten Analyse-Tools vorgestellt, mit denen der jeweilige Smells identifiziert werden kann. Dabei ist jedes Werkzeug in der Lage, Indizien für verschiedene Code Smells zu finden. Anschließend wird mit den Analyse-Tools eine Auswahl von Open Source Projekten, die auf der Plattform GitHub zur Verfügung stehen, auf Schwachstellen untersucht. Es folgt eine Zusammenfassung und Bewertung der Ergebnisse.

Die Implementierung der Tools ist als Proof of Concept zu betrachten. Die einwandfreie Implementierung solcher Programme ist zeitlich sehr aufwändig und würde über den Rahmen dieser Arbeit hinausgehen. Eine „einwandfreie Implementierung“ bedeutet in diesem Fall, dass die Funde sogenannter **False Positives**, bzw. **False Negatives**, möglichst gering ist. Identifiziert das Programm fälschlicherweise einen Code Smell, so handelt es sich dabei um einen False Positive. Ein False Negative liegt vor, wenn ein im untersuchten Code vorhandener Smell vom Tool nicht erkannt wird.

Die Analyseergebnisse sollen zur einfachen Auswertung mit Excel in eine CSV-Datei geschrieben werden.

5.1 Namen

Eine der am häufigsten anfallenden Aufgaben eines Softwareentwicklers ist es, Dingen einen Namen zu geben. Es fängt mit dem Namen einer neuen Projektmappe an, und zieht sich über die darin enthaltenen Projekte, Namensräume und Klassen. In den Klassen gilt es, Variablen, Methoden und deren Parameter zu benennen. Gute Namen zu finden kostet Zeit. [17, vgl. S.45 f.] Allerdings kostet es noch mehr Zeit, jedes Mal den Code einer Methode entziffern zu müssen, um zu wissen, was darin geschieht. Namen sollten sprechend, klar und ausdrucksvoll sein, und so die Absicht des Codes erklären. Martin sagt dazu: „Der Name einer Variablen, Methode oder Klasse sollte alle großen Fragen beantworten. Er sollte Ihnen sagen, warum er existiert, was er tut, und wie er benutzt wird.“ [17, S.46] Ob ein gewählter Name wirklich sprechend ist, lässt sich zum Beispiel in einem Codereview mit den Teammitgliedern überprüfen; eine Automatisierung der Überprüfung durch Tools ist nur teilweise möglich. Die Werkzeuge beschränken sich in der Regel auf die Einhaltung von Namenskonventionen.

5.1.1 Code Smells

Namen, an die Zahlenserien angehängt sind

Wie bereits in der Einleitung des Kapitels erwähnt, sollten Namen sprechend sein und somit eine Aussage über den Sinn und die Aufgabe der Variablen, Methode, Klasse etc. machen. Oft werden an Variablennamen gleichen Typs Zahlenserien angehängt. Martin führt als Beispiel folgende Methode auf:

```
1 public static void CopyChars(char[] a1, char[] a2)
2 {
3   for (int i = 0; i < a1.Length; i++)
4   {
5     a2[i] = a1[i];
6   }
7 }
```

Listing 5.1: CopyChars [17] vgl. S.49

In dem Beispiel werden die beiden Zeichen-Arrays der Parameterliste a1 und a2 genannt. Die gewählten Namen vermitteln allerdings keinerlei Informationen über ihren jeweiligen Zweck. Aus dem Funktionsnamen CopyChars lässt sich zwar ableiten, dass wohl der Inhalt des einen Arrays in das andere kopiert wird, allerdings ist erst durch genauere Betrachtung des Codes ersichtlich, dass der Inhalt von a1 in a2 kopiert wird. Bessere Namen der Parameter wären beispielsweise source und destination – in Verbindung mit dem Methodennamen ist dann sofort klar, wie die Methode funktioniert. [17, vgl. S.49]

Namen, die aus einem Buchstaben bestehen

Ein weiteres Problem ergibt sich bei Variablen, die nur aus einem Buchstaben bestehen: Der Name ist nicht suchbar. Wird eine Variable beispielsweise „e“ genannt, liefert eine entsprechende Suche logischerweise sehr viele unerwünschte Ergebnisse. [17, vgl. S.51 f.]

Aus den Beispielen geht hervor, dass Namen, die nur aus einem Buchstaben oder Buchstabe(n) plus Zahlenserien bestehen, im Allgemein nicht bezeichnend sind. In sauberem Code hat eine solche Namensgebung nichts verloren[1]. Der Code ist schlechter lesbar, es ist mehr Zeit erforderlich und die Einarbeitungszeit für Entwickler ist höher, als sie sein müsste.

5.1.2 Implementierung: NameInspector

Der NameInspector wurde im Rahmen dieser Arbeit mit dem Ziel entwickelt, die aus dem vorangegangenen Kapitel 5.1.1 beschriebenen Code Smells zu identifizieren. Es sollen Variablennamen, die aus einem Zeichen bestehen sowie Zahlenserien gefunden werden.

Dazu werden mit Hilfe der Roslyn-API sämtliche Deklarationen von Klassen, Funktionen, Variablen und Übergabeparameter aus den Quelltext-Dateien der zu un-

[1] Mit Ausnahme von Schleifenzählern und Variablen in Lambda-Ausdrücken.

tersuchenden Projektmappe gesammelt. Die Deklarationen werden von Roslyn als SyntaxNodes (siehe Kapitel 3.2.1) dargestellt. Jede Deklaration besitzt einen Identifier Kind-Knoten vom Typ SyntaxToken. Dieser stellt die Repräsentation des Bezeichners der jeweiligen Klasse, Methode oder Variable dar.

Mittels eines regulären Ausdruckes wird anschließend geprüft, ob der Identifier der jeweiligen Deklaration eine Zahlenserie enthält. Außerdem wird überprüft, ob der Identifier aus einem einzelnen Zeichen besteht.

Unter bestimmten Umständen wirken sich Variablenbezeichner, die lediglich aus einem Buchstaben bestehen, nicht negativ auf die Lesbarkeit des Quelltextes aus. Innerhalb von Kontrollstrukturen als Schleifenzähler (siehe Listing 5.2) oder als Eingabeparameter von Lambda-Ausdrücken (siehe Listing 5.3) ist diese Art der Variablenbenennung durchaus gängige Praxis. [17, vgl. S. 51 f.]

Folglich müssen Deklarationen, die einem der beiden oben genannten Kriterien genügen, noch entsprechend gefiltert werden. Dazu wird geprüft, ob Syntax-Knoten der Variablenerstellung einen Eltern-Knoten der folgenden Typen hat:

- LambdaExpressionSyntax
- WhileStatementSyntx
- ForStatementSyntx
- ForEachStatementSyntx

Nachfolgend Beispiele zur legitimen Verwendung von Variablennamen, die aus einem Buchstaben bestehen:

```
1 public void PrintOdds() {
2   for (var i = 0; i <= 100; i++)
3   {
4     if (i % 2 == 1)
5     {
6       Console.WriteLine(i);
7     }
8   }
9 }
```

Listing 5.2: Beispiel für die Verwendung eines Schleifenzählers

```
1 public IEnumerable<Person> FindOldPersons(IEnumerable<Person>
      persons)
2 {
3   return persons.Where(p => p.Age > 90);
4 }
```

Listing 5.3: Beispiel für die Verwendung eines Lambda-Ausdrucks

5.1.3 Analysefunde

Namen, die aus einem Buchstaben bestehen

Piranha[2] ist ein Open Source Framework zur Erstellung von CMS[3] basierten Webanwendungen. Bei der Untersuchung des Quelltextes durch das NameInspector-Tool wurde unter anderem auch folgender Codeausschnitt gefunden:

```
1 public override bool Save(IDbTransaction tx = null) {
2   [...]
3   IDbTransaction t = tx != null ? tx :
        Database.OpenConnection().BeginTransaction();
4   [...]
5 }
```

Listing 5.4: Ausszug aus dem Quelltext von Piranha

Dieser Ausschnitt ist einer Methode entnommen, die einen Datensatz in einer Datenbank speichern soll. Wird keine Transaktion übergeben (Nullreferenz), so wird für die Operation eine neue Transaktion erstellt.

Der Auszug ist gleich aus zwei Gründen problematisch: Zum einen sind die Namen für die Transaktion `tx` in der Parameterliste und die Transaktion `t` zu Beginn der Methode nicht sprechend. Die gewählten Namen geben keinen Hinweis auf die Art oder den Zweck der Objekte. Zum anderen wird das angehängte *x* des Parameters dazu verwendet, um zwischen den beiden Transaktionen unterscheiden zu können. Dies ist ebenfalls eine potentielle Fehlerquelle. Wird die Methode zukünftig geändert oder erweitert, so besteht die Gefahr, dass die beiden Transaktionen verwechselt

2 github.com/PiranhaCMS/Piranha

3 Content-Management-System (Redaktionssystem), mit dessen Hilfe der Inhalt z.B. von Websites verwaltet wird. [14]

werden. Beide sind vom gleichen Typ IDbTransaction und die Variablennamen unterscheiden sich nur durch ein Zeichen. Vor allem, wenn mehrere Entwickler an diesem Code arbeiten, besteht die Gefahr einer Verwechslung.

Aussagekräftiger wäre der Name rootTransaction. Für die Unterscheidung bietet es sich an, den Parameter beispielsweise actualTransaction zu nennen.

Namen, an die Zahlenserien angehängt sind

Bei NodejsTools[4] handelt es sich um ein Plugin, mit dem Visual Studio auch als Node.js Integrated Development Environment (IDE) verwendet werden kann. Folgender Codeausschnitt zeigt ein Vorkommen einer Zahlenserie in dessen Quelltext:

```
1 private static Exception ErrorOutOfRange(object p0, object p1)
2 {
3   return new ArgumentOutOfRangeException(string.Format("{0} must
        be greater than or equal to {1}", p0, p1));
4 }
```

Listing 5.5: Ausszug aus dem Quelltext von NodejsTools

Allein über den Methodennamen lässt sich kaum darauf schließen, wozu die Parameter p0 und p1 dienen. Ein Blick auf den Funktionsrumpf zeigt zwar, dass daraus die Nachricht einer ArgumentOutOfRangeException generiert wird, jedoch wird erst bei Betrachtung der Aufrufsyntax der Methode klar, wie der Vorgang im Detail funktioniert:

```
1  if (index < 0)
2  {
3    throw ErrorOutOfRange("index", 0);
4  }
5  if (line < 1)
6  {
7    throw ErrorOutOfRange("line", 1);
8  }
9  if (column < 1)
10 {
11   throw ErrorOutOfRange("column", 1);
12 }
```

Listing 5.6: Ausszug aus dem Quelltext von NodejsTools

[4] github.com/Microsoft/nodejstools

Ein index muss mindestens „0“ sein, eine line oder eine column muss mindestens den Wert „1“ haben, ansonsten wird eine ArgumentOutOfRangeException mit entsprechenden Fehlertext generiert und geworfen. Da es sich bei den drei gelisteten Aufrufen um die einzigen dieser Methode handelt, ist auch die Wahl des Typs object der Übergabeparameter fraglich. Dies soll allerdings nicht an dieser Stelle diskutiert werden, da es sich um eine andere Art von Smell handelt.

Eine bessere Wahl der Parameternamen wären numberingElement und minValue. Damit ist definiert, *was* außerhalb des Bereiches liegt und wie groß der kleinste Wert innerhalb des Bereiches mindestens sein muss.

Unter den vom Analysewerkzeug gelisteten Namensverstößen befinden sich Codestellen, an denen die gewählte Namensgebung durchaus gerechtfertigt ist. So ist es beispielsweise üblich, in den Parameterlisten von catch-Klauseln eines try-Blocks die gefangene Instanz einer Exception schlicht e zu nennen. Außerdem sind EventHandler zu nennen, in deren Parameterliste sich ein EventArgs e befindet. Der Kontext macht weitere Erklärungen überflüssig.

Bei ShareX[5] handelt es sich um ein Programm, mit dem von jedem beliebigen Ausschnitt des Bildschirms ein Screenshot erzeugt werden kann. Dieser kann anschließend mit einem Tastendruck auf verschiedene Zielplattformen wie Dropbox[6] oder Imgur[7] hochgeladen werden. Im Code des Programms wird häufig mit Koordinaten gearbeitet, wie folgender Funktionskopf zeigt:

```
1 private void UpdateColor(int x, int y)
```

Listing 5.7: Ausszug aus dem Quelltext von ShareX

Diese Methode stammt aus einem Farbmischer, mit dessen Hilfe der Anwender über einen Klick auf die Fläche einer Farbpalette eine Farbe auswählen kann. Aus dem Kontext ergibt sich, dass es sich bei den Parameter x und y um die Koordinaten innerhalb der Farbpalette handelt.

5 github.com/ShareX/ShareX

6 www.dropbox.com/

7 http://imgur.com/

Ebenfalls aus ShareX stammt folgende Funktionsdefinition:

```
1 public static bool operator == (ColorBgra c1, ColorBgra c2)
```

Listing 5.8: Ausszug aus dem Quelltext von ShareX

Hierbei handelt es sich um die Überladung des == Operators für die ColorBgra-Klasse[8]. Die Verwendung einer Zahlenserie als Namen für die beiden Übergabeparameter ist durchaus legitim, da sich deren Bedeutung leicht aus dem Kontext erkennen lässt.

5.1.4 Fazit

Mit der Roslyn-API ist es sehr einfach, alle Benennungen von Variablen, Parametern etc. zu finden. Über reguläre Ausdrücke und über die Länge des Namens können verschiedene Namenskonventionen überprüft werden. Weiterhin lässt sich durch den Syntaxbaum überprüfen, an welcher Stelle im Code der zu untersuchende Namen steht: Befindet er sich beispielsweise in einer Kontrollstruktur können der Überprüfung Ausnahmen hinzugefügt werden. Allerdings kann nicht geprüft werden, ob die Namen der gefundenen Smells im jeweiligen Kontext nicht doch in Ordnung sind. Die Implementierung eines Tools, das erkennt, ob gewählte Namen die Funktionalität einer Methode oder den Zweck einer Variablen treffend beschreiben, ist schwer bis unmöglich.

5.2 Methoden

Methoden gehören zu den primären Organisationseinheiten von Programmen. Clean Code zeichnet sich unter anderem dadurch aus, dass der Zweck von Methoden schnell ersichtlich ist. Durch das Vermischen von Abstraktionsebenen und tiefen Verschachtelungen von Kontrollstrukturen wird die Absicht von Methoden allzu oft verschleiert. [17, vgl. S. 61 ff.]

8 BRGA ist die Little-Endian Darstellung einer RGBA-Farbcodierung

5.2.1 Code Smells

Zu lange Methoden

„Die erste Regel für Methoden lautet: Methoden sollten klein sein. Die zweite Regel lautet: *Methoden sollten noch kleiner sein.*“[17, S.64] Laut Martin gehören große Methoden mit zu den Hauptgründen für unübersichtlichen Code. In den meisten Fällen lassen sich in langen Methoden Abschnitte ausmachen, die in andere Methoden ausgelagert werden sollten – in diesem Fall wird das SRP verletzt; die Methode übernimmt zu viele Aufgaben. Wie bereits in Kapitel 4.2.5 erwähnt, sollten Methoden im Allgemeinen nicht länger als 20 Zeilen sein. Je nach Programmiersprache variiert die empfohlene Zahl. Natürlich gibt es auch zu diesem Richtwert Ausnahmen. Diese sollten allerdings gut begründet werden können. Durch das Schreiben von kurzen Methoden wird erreicht, dass diese praktisch keine tief verschachtelten Strukturen beherbergen können und die Lesbarkeit wird ebenfalls erhöht. [17, vgl. S. 64 ff.]

Zu lange Parameterlisten

Je größer die Anzahl von Parametern für eine Funktion, desto schwerer ist es, diese zu verstehen. Funktionen, die keine Parameter benötigen, sind als ideal anzusehen, ein bis zwei Parameter sind noch vertretbar. Nach Möglichkeiten sollten Methoden mit drei Parametern vermieden werden. Methoden, die mehr als drei Parameter verwendete, erfordern eine spezielle Begründung.

Ein weiterer Grund für das Vermeiden langer Parameterlisten ist, dass Methoden mit vielen Parametern schwerer zu testen sind. Bei parameterlosen Methoden ist dies nahezu trivial. Mit steigender Parameterzahl steigen die unterschiedlichen Kombinationsmöglichkeiten der Parameter – und damit auch der Aufwand beim Schreiben geeigneter Tests.

Wenn Methoden mehr als zwei Parameter benötigen, ist es oft sinnvoller, diese in einer separaten Klasse zu kapseln. In vielen Fällen gehören die Parameter zu einem gemeinsamen Konzept, das einen eigenen Namen haben sollte. [17, vgl. S.71 ff.]

Flag-Argumente

Die Übergabe eines Schalter-Arguments an eine Methode ist eine unsaubere Technik, die vermieden werden sollte. Mit dem Parameter wird durch eine Kontrollstruktur bestimmt, welcher Pfad innerhalb der Methode ausgeführt wird. Damit ist auch klar, dass die Methode je nach Zustand des Flags eine andere Aufgabe erledigt. Außerdem verkompliziert diese Vorgehensweise die Signatur der Methode unnötigerweise. Stattdessen wäre es besser, auf das Flag zu verzichten und stellvertretend dafür zwei Methoden zu schreiben: eine Methode für den Zustand `true` und eine andere für den Zustand `false`. [17, vlg. S. 72]

Parameter vom Typ `Enum` können ebenfalls als Flag-Argument dienen. In einer `switch`-Anweisung wird anhand des Zustandes des Parameters der Ausführungspfad determiniert. Ist das der Fall, so stehen mindestens zwei alternative Ausführungspfade zur Wahl.

5.2.2 Implementierung: `FunctionInspector`

Der `FunctionInspector` hat die Aufgabe, die in den vorangegangen Kapitel beschriebenen Code Smells zu identifizieren. Das Modul ist in der Lage, Funktionen mit den folgenden Eigenschaften zu identifizieren:

- Der Umfang erstreckt sich über mehr als 20 Zeilen.
- Die Parameterliste fasst mehr als drei Parameter.
- Es werden Flag Argumente verwendet.

Dazu ist es hilfreich, die Darstellung einer Methodendeklaration im Syntaxbaum zu kennen. Die nachfolgende Grafik zeigt den schematische Aufbau der Deklaration von `public void ThisIsAFunction(int parameterA, int paramterB)` innerhalb des SytaxTrees:

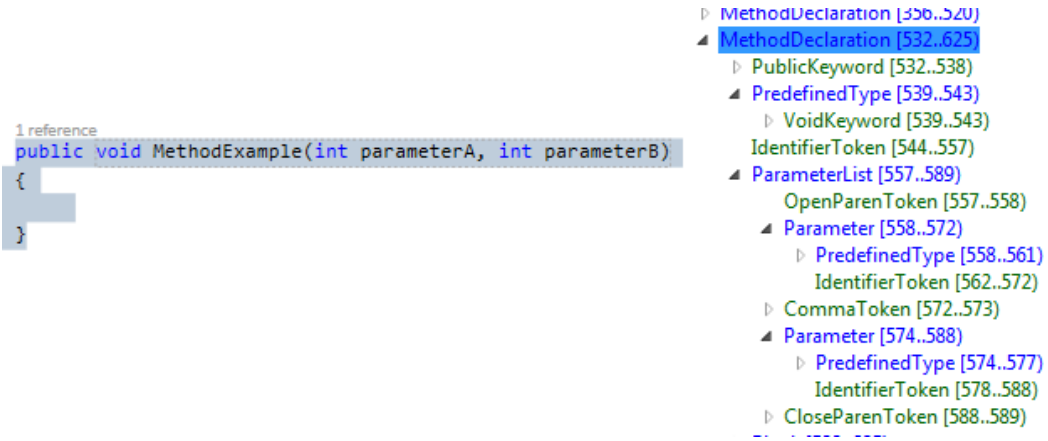

Abbildung 5.1: Darstellung einer Methodendeklaration im SyntaxTree

In dem Bild sind die SyntaxNodes blau und die SyntaxTokens grün dargestellt. Der Aufbau der Methode ist leicht auszumachen: Die Methode umfasst einen Rückgabewert (void) und eine Parameterlist mit den Parametern parameterA und parameterB, beide vom Typ int.

Um die oben genannten Code Smells zu identifizieren, werden aus dem Syntaxbaum des zu untersuchenden Dokuments zuerst alle SyntaxNodes vom Typ MethodDeclarationSyntax extrahiert. Unter Verwendung des für Kapitel 4.2.5 entwickelten MetricCalculator wird die Anzahl der LOC jeder Methode errechnet und Methoden mit einer Länge von über 20 Zeilen Code können aufgespürt werden.

Ein MethodDeclarationSyntax besitzt einen Kind-Knoten vom Typ ParameterListSyntax. Anhand der Anzahl der ParameterSyntax dieser Liste kann überprüft werden, ob die Methode zu viele Argumente benötigt.

Über das IdentifierToken des IdentifierNameSyntax eines jeden ParameterSyntax lässt sich dessen Datentyp bestimmen. Handelt es sich dabei um einen Parameter vom Typ Boolean oder Enum, ist ein potentieller Kandidat für ein Flag-Argument gefunden. Anschließend wird noch überprüft, ob dieser Parameter innerhalb der Methode in einer bedingten Anweisung (IfStatementSyntax, ConditionalExpressionSyntax oder SwitchStatementSyntax) verwendet wird. Ist dies der Fall, so handelt es sich um ein Flag-Argument, da mit dem Parameter ein Ausführungspfad determiniert wird.

5.2.3 Analysefunde

Zu lange Funktionen

Ebenfalls aus Piranha ist folgende Methode entnommen:

```
1  /// <summary>
2  /// Creates a command on the given connection and sql statement
       and fills it with the given parameters.
3  [...]
4  public static IDbCommand CreateCommand(IDbConnection conn,
       IDbTransaction tx, string sql, object[] args = null) {
5    // Convert all enum arguments to string
6    for (int n = 0; n < args.Length; n++)
7      if (args[n] != null &&
           typeof(Enum).IsAssignableFrom(args[n].GetType()))
8        args[n] = args[n].ToString();
9    // Create command
10   IDbCommand cmd = conn.CreateCommand();
11   if (tx != null)
12     cmd.Transaction = tx;
13   cmd.CommandText = sql;
14   if (args != null) {
15     int pos = args.Length > 0 && args[0] is IDbTransaction ? 1:0;
16     for (int n = 0 + pos; n < args.Length; n++) {
17       if (!(args[n] is Params)) {
18         IDbDataParameter p = cmd.CreateParameter();
19         p.ParameterName = String.Format("@{0}", n);
20         if (args[n] == null || (args[n] is DateTime &&
               ((DateTime)args[n]) == DateTime.MinValue)) {
21           p.Value = DBNull.Value;
22         } else if (args[n] is Guid) {
23           if (((Guid)args[n]) == Guid.Empty) {
24             p.Value = DBNull.Value;
25           } else {
26             p.Value = ((Guid)args[n]).ToString();
27             p.DbType = DbType.String;
28           }
29         } else {
30           p.Value = args[n];
31         }
32         cmd.Parameters.Add(p);
33       }
34     }
35   }
36   return cmd;
37 }
```

Listing 5.9: Ausszug aus dem Quelltext von Piranha

Diese Methode fällt vor allem durch ihre Größe und Unübersichtlichkeit auf. Sie besteht aus 26 LOC und besitzt eine zyklomatische Komplexität von 15. In der Methode sind sehr viele if/else-Verzweigungen. Aus dem Namen und der Dokumentation der Methode geht hervor, dass mit der übergebenen IDbConnection ein parametrisierbarer SQL-Befehl erstellt werden soll.

Bei genauerer Betrachtung fällt auf, dass die Methode mehr als nur diese eine Aufgabe übernimmt – eine klare Verletzung des SRP. Zuerst werden in den Zeilen 7-9 Enum-Werte der übergebenen Parameterliste in Strings konvertiert. Die String-Parameter werden anschließend noch geparst und es werden daraus IDbDataParameter erstellt. Es wird ein IDbCommand erstellt und mit den Parametern sowie dem SQL-Befehl und gegebenenfalls einer IDbTransaction initialisiert.

Auch wenn die gezeigte Methode noch weiteres Potential zur Refaktorisierung bietet, sollen für dieses Kapitel lediglich die Größe und die Komplexität betrachtet werden. Es wäre wesentlich sauberer und leichter verständlich, die im letzten Abschnitt identifizierten Aufgaben in separate Methoden auszulagern. Eine Refaktorisierung könnte so aussehen:

```
1  public static IDbCommand GetDbCommand(IDbConnection conn,
       IDbTransaction tx, string sql, object[] args = null)
2  {
3    var parameters = ConvertEnumParamsToString(args);
4    var dbParameters = CreateDbParameters(parameters);
5    return CreateAndInitializeCommand(conn, tx, sql, dbParameters);
6  }
7
8  private static object[] ConvertEnumParamsToString (object[]
       parameters) { ... }
9  private static IDbDataParameter[] CreateDbParameters(object[]
       parameters) {
10   List<IDbDataParameter> dbParameters = new
         List<IDbDataParameter>();
11   dbParameters.AddRange(CreateNullValueParameters(parameters));
12   dbParameters.AddRange(CreateGuidParameters(parameters));
13   dbParameters.AddRange(CreateRegularParameters(parameters));
14   return dbParameters.ToArray();
15 }
16 private static IDbDataParameter[] CreateNullValueParameters
       (object[] parameters) { ... }
17 private static IDbDataParameter[] CreateGuidParameters (object[]
       parameters) { ... }
```

```
18 private static IDbDataParameter[] CreateRegularParameters
       (object[] parameters) { ... }
19
20 private static IDbCommand
       CreateAndInitializeCommand(IDbConnection conn, IDbTransaction
       tx, string sql, IDbDataParameter[] parameters) { ... }
```

Listing 5.10: Refaktorisierung der CreateCommand Methode

Auf die Implementierung der Methodenrümpfe wurde aus Platzgründen bewusst verzichtet, die Auslagerung der Funktionalität in weitere Methoden ist entscheidend. Zuerst werden die Parameter konvertiert und anschließend die entsprechenden IDbDataParamter erstellt. Die CreateDbParameters(...)-Methode ruft weitere Methoden auf, um die verschiedenen Parametertypen zu erstellen. Letztlich wird ein neues IDbCommand Objekt erstellt und initialisiert. So kann der Umfang und die Komplexität der einzelnen Methoden gering gehalten und eine gute Lesbarkeit erreicht werden.

Unter den gefundenen Code Smells des ShareX-Projekts sind auch mehrere InitializeComponent–Funktionen. Dabei handelt es sich um automatisch generierte Funktionen, die eine Form initialisieren. Zu allen Benennungen und Positionierungen von Komponenten, die ein Entwickler über den in Visual Studio integrierten Form-Editor vorgenommen hat, wird Initialisierungs-Code generiert. Dieser Code darf nicht mehr verändert werden, sonst kann die Oberfläche nicht mehr korrekt erstellt werden.

In den meisten Fällen darf automatisch generierter Code nicht angepasst werden. Folglich ist es nicht sinnvoll, zu lange, automatisch generierte Methoden als Code Smell zu bewerten.

Zu lange Parameterliste

Folgende Methodendeklaration stammt aus dem Quelltext von Greenshot Image Editor[9], einem Tool mit dem leicht Screenshots erstellt und bearbeitet werden können:

[9] www.getgreenshot.org

```
1 public static int Distance2D(int x1, int y1, int x2, int y2)
```

Listing 5.11: Methodendeklaration aus Greenshot Image Editor

Diese Methode benötigt vier Parameter, um die Distanz zweier Punkte zu bestimmen. Beim Aufruf der Methode muss darauf geachtet werden, dass die Koordinaten in der richtigen Reihenfolge übergeben werden. Eine elegantere Lösung wäre es, die Übergabeparameter in einem Konstrukt zusammenfassen, der Klasse Punkt. Damit reduziert sich die Anzahl der Parameter von vier auf zwei, außerdem ist deren Reihenfolge nicht mehr relevant.

Flag-Argumente

Ebenfalls aus NodejsTools ist die folgende Methode entnommen:

```
1 internal static bool TryMakeUri(string path, bool isDirectory,
      UriKind kind, out Uri uri) {
2   if (isDirectory && !string.IsNullOrEmpty(path) &&
        !HasEndSeparator(path)) {
3     path += Path.DirectorySeparatorChar;
4   }
5   return Uri.TryCreate(path, kind, out uri);
6 }
```

Listing 5.12: Methode aus NodejsTools

Diese Methode generiert aus einem string-Pfad eine URI. Handelt es sich bei dem Pfad um ein Verzeichnis, wird vor der Erstellung der URI noch Backslash an den Pfad angehängt. Anstelle des Flag-Arguments hätten auch zwei Methoden erstellt werden können: TryMakeUriFromFile(...) und TryMakeUriFromDirectory(...).

5.2.4 Fazit

Mit Hilfe des FunctionInspectors können zuverlässig Methoden identifiziert werden, die eine vorgegebene Maximalgröße überschreiten. Allerdings kann nicht zwischen automatisch generiertem und von Entwicklern geschriebenem Code unterschieden werden. Auch hier liefert das Tool potentielle Kandidaten für eine Refaktorisierung, jedoch muss letztlich geprüft werden, ob die betrachtete Methode nicht automatisch

erstellt wurde. Dieser Prozess könnte zumindest teilweise automatisiert werden, indem generierter Code in definierten Regionen[10] steht, die von der Überprüfung ausgeschlossen werden.

Ähnlich verhält es sich bei einer langen Parameterliste und Flag-Argumenten: Die vom Tool gefundenen Codestellen sind in der Regel verbesserungswürdig. Es kann allerdings nicht mit Sicherheit bestimmt werden, ob es sich bei den Funden tatsächlich um Smells handelt, oder ob im Einzelfall die Verwendung einer zu langen Parameterliste oder eines Flag-Arguments gerechtfertigt ist. Letztlich muss der Anwender eines solchen Tools entscheiden, wie er mit den Funden umgeht, da diese nur Indizien für unsauberen Code sind.

5.3 Kommentare

Kommentare sind ein zweischneidiges Schwert: Auf der einen Seite kann ein gut platzierter Kommentar ungemein zum Verständnis eines Codeabschnitts beitragen. Auf der anderen Seite wird Code sehr schnell mit redundanten oder veralteten Kommentare überhäuft. Martin sagt zu diesem Thema: „[...] Kommentare [sind] bestenfalls ein erforderliches Übel. Wären unsere Programmiersprachen ausdrucksstark genug oder hätten wir genügend Talent, unsere Absichten in den vorhanden Sprachen immer klar genug auszudrücken, wir würden wohl kaum Kommentare brauchen.“ Code sollte, wenn möglich, selbstbeschreibend sein. Dies ist nicht immer umsetzbar und genau für diese Fälle können Kommentare sinnvoll eingesetzt werden. Allerdings werden Kommentare oft missbraucht, um schlecht und unverständlich geschriebenen Code zu erklären – obwohl dies durch eine vernünftige Namensgebung der verwendeten Variablen oder dem Auslagern von konkreten Funktionalitäten in eigene Methoden, nicht nötig wäre. [17, S.85]

---

[10] Mit #region kann ein Codeblock markiert werden, der erweitert oder reduziert werden kann, wenn das Gliederungsfeature von Visual Studio verwendet wird. [21]

5.3.1 Code Smells

Kommentarüberschriften

Kommentarüberschriften kommen meist in langen Methoden vor. Im Code werden Blöcke gebildet und mit einem Kommentar als Überschrift versehen. Dieser beschreibt die Funktionalität des Blocks, eine Auslagerung des betreffenden Codes in eine eigene Methode wäre die bessere Lösung.

Oft ist diese Form von Kommentaren auch redundant: Eine Zeile Code wird mit einem Kommentar versehen, welches die Zeile erklärt. Oft dauert es länger, den Kommentar zu lesen, als den Code, der dadurch erklärt werden soll. Diese Art von Kommentaren sind bestenfalls überflüssig. Im schlimmsten Fall sind Kommentare irreführender und schädlicher Ballast. Besonders, wenn Code weiterentwickelt wird und dadurch die Kommentare nicht mehr dem beschriebenen Block entsprechen. [17, vgl. S. 93 ff.]

Auskommentierter Code

Wenn ein Codeblock umgeschrieben wird, fertigen Entwickler oft eine Kopie des Quelltextes an, kommentieren das Original aus und ändern die Kopie. Nicht mehr benötigter Code wird häufig nicht gelöscht, sondern auskommentiert. Für den Fall, dass der Code nochmals gebraucht werden sollte, existiert noch die Sicherungskopie in Form eines Kommentars. Martin sagt dazu: „Nur wenige Techniken sind so widerlich wie auskommentierter Code. Lassen Sie es einfach!“ [17, S.102] Andere Entwickler, die auskommentierten Code sehen, werden nicht wissen, was sie damit anfangen sollen. Sie werden glauben, dass es einen Grund für den auskommentierten Code gibt. Sie werden glauben, dass der Code noch für eine andere Person Relevanz hat. Auskommentierter Code ist fast immer irrelevant und wird zunehmend belangloser – es werden Methoden aufgerufen und Variablen verwendet, die nicht mehr existieren. Auskommentierter Code sollte vor dem Einchecken in das Repository gelöscht werden. Und für den Fall, dass der Code tatsächlich noch-

mals gebraucht werden sollte, kann einfach eine alte Version des Projekts aus dem Versionskontrollsystem ausgecheckt werden. [17, vgl. S.102]

Dokumentation in nicht-öffentlichem Code

Gegen Dokumentation in öffentlichem Code ist nichts einzuwenden, sie ist meist sogar erforderlich und äußerst nützlich. Vorausgesetzt, sie ist gut geschrieben. Allerdings sollte laut Martin auf Dokumentation in nicht-öffentlichem Code verzichtet werden – sie ist im Allgemeinen nicht nützlich. Oft sogar erschwert sie die Arbeit mit dem Code. Wird der Quelltext geändert, kann dies Änderungen der Dokumentation nach sich ziehen. Wird die Dokumentation nicht mit dem Code gepflegt und gewartet, so häufen sich alte und nicht mehr korrekte Kommentare an. Dies führt zu einer Irreführung der Leser, da falsche oder nicht mehr aktuelle Informationen vermittelt werden. [17, vgl. S.105]

5.3.2 Implementierung: CommentInspector

Kommentare fallen, wie in Kapitel 3.2.3 bereits erwähnt, unter die Kategorie der Syntax Trivia. Die Trivia wird nicht durch eigene Knoten im Syntaxbaum repräsentiert, sondern geht Syntax Tokens voraus oder wird an diese angehängt. Folgender Screenshot soll dies verdeutlichen:

Abbildung 5.2: Kommentar und Repräsentation im Syntaxbaum

Der Screenshot zeigt eine Methode mit Kommentar, sowie den dazugehörigen Ausschnitt des Syntaxbaumes. Sowohl im Code, als auch im Syntaxbaum, ist das Kommentar selektiert. Das Syntax Token ForKeyword besitzt zwei Listen mit Lea-

dingTrivia und TrailingTrivia, worunter auch das Kommentar zu finden ist. Kommentare und Trivia werden immer Syntax Tokens zugeordnet.

Die Roslyn API unterscheidet zwischen drei verschiedenen Arten von Kommentaren:

- SingleLineCommentTrivia, eingeleitet durch //,
- MultiLineCommentTrivia, immer zwischen /* und */,
- DocumentationCommentTrivia, eingeleitet durch ///, wird als Dokumentation von Methoden und Klassen verwendet.

Um Kandidaten für die Code Smells Kommentarüberschriften und Auskommentierter Code zu finden, werden in einem ersten Schritt alle relevanten Kommentare aus dem zu untersuchenden Dokument extrahiert. Dazu wird der Syntaxbaum des Dokuments traversiert und Syntax Trivia der Typen SingleLineComment und MultiLineComment in einer Liste gespeichert.

Um potentielle Kandidaten für Kommentarüberschriften zu finden, wird anschließend nach folgendem Muster gesucht: Nach einem Kommentar kommt ein Codeblock der Länge 1-10 Zeilen, anschließend erneut ein Kommentar, gefolgt von 1-10 Zeilen Code.

Nicht alle Kommentar-Code-Muster dieses Typs gehören automatisch zur Kategorie Kommentar-Überschrift. Das Tool liefert auch Kommentare, die lediglich Code dokumentieren.

Um auskommentierten Code zu finden, werden die gefundenen Kommentare mit einem regulären Ausdruck verglichen: Befindet sich am Ende des Kommentars ein Semikolon, so handelt es sich möglicherweise um auskommentierten Code. Allerdings werden auskommentierte Deklarationen von Namensräumen, Klassen und Methoden auf diese Art nicht erkannt, da diese nicht mit einem Semikolon enden. Da die Codekonventionen für C# vorsehen, öffnende geschweifte Klammern in eine separate Zeile zu schreiben, kann diese nicht im regulären Ausdruck berücksichtigt werden.

Dies ist nicht von Relevanz, wenn anschließend eine weitere Zeile Code auskommentiert wurde, die mit einem Semikolon endet. Das Tool findet die folgende, auskommentierte Zeile und beim Review des Nutzers fällt auch die darüberliegende, auskommentierte Zeile auf. Nur, wenn die deklarierende Zeile einzeln auskommentiert ist, wird diese nicht bemerkt.

Um Dokumentation in nicht-öffentlichem Code zu finden, müssen erst alle Dokumentationskommentare gefunden werden. Anschließend werden alle Kommentar-Trivia herausgefiltert, die keinem PublicKeyword Token zugeordnet sind.

5.3.3 Analysefunde

Redundante Kommentare

Nochmals soll die Methode Distance2D als Beispiel dienen, diesmal um redundante Kommentarüberschriften zu zeigen:

```
1  public static int Distance2D(int x1, int y1, int x2, int y2)
2  {
3    //Our end result
4    int result = 0;
5    //Take x2-x1, then square it
6    double part1 = Math.Pow((x2 - x1), 2);
7    //Take y2-y1, then square it
8    double part2 = Math.Pow((y2 - y1), 2);
9    //Add both of the parts together
10   double underRadical = part1 + part2;
11   //Get the square root of the parts
12   result = (int)Math.Sqrt(underRadical);
13   //Return our result
14   return result;
15 }
```

Listing 5.13: Redundante Kommentare aus Greenshot

Jede einzelne Codezeile wird mithilfe eines Kommentars „erklärt“. Die Funktion, aus der diese Zeilen entnommen wurden, berechnet über den Satz des Pythagoras den Abstand zweier Punkte in der Ebene. Die Kommentare sind überflüssig und ziehen die Methode unnötig in die Länge. Falls sich die Implementierung ändert, muss die Dokumentation ebenfalls geändert werden, um Missverständnisse durch

falsche Kommentare zu vermeiden. Generell sollte auf solche Kommentare verzichtet werden, wenn der Code selbst sprechend genug geschrieben werden kann. Mit einem Kommentar könnte vor der Methode erwähnt werden, dass die Implementierung der Methode mittels dem Satz des Pythagoras erfolgt – damit ist der Code ausreichend erklärt. Alternativ könnte die Art der Implementierung auch in den Funktionsnamen einfließen: CalculateDistanceViaPythagoras.

Kommentarüberschriften

Nachfolgend ein Auszug der Methode Save() aus Piranha. Diese speichert den aktuellen Datensatz in der Datenbank.

```
1  public virtual bool Save(IDbTransaction tx = null) {
2    bool result = false;
3
4    // Check primary key
5    foreach (string key in PrimaryKeys)
6      if (Columns[key].GetValue(this, null) == null)
7        throw new ArgumentException("Property \"" +
             Columns[key].Name + "\" is marked as Primary Key and can
             not contain null");
8
9    // Execute command
10   using (IDbConnection conn = tx != null ? null :
         Database.OpenConnection()) {
11     if (IsNew) {
12       using (IDbCommand cmd = CreateInsertCommand(tx != null ?
             tx.Connection : conn, tx)) {
13         result = cmd.ExecuteNonQuery() > 0;
14         IsNew = !result;
15       }
16     } else {
17       using (IDbCommand cmd = CreateUpdateCommand(tx != null ?
             tx.Connection : conn, tx)) {
18         result = cmd.ExecuteNonQuery() > 0;
19       }
20     }
21   }
22   // Check for cache interface
23   if (this is ICacheRecord<T>)
24     ((ICacheRecord<T>)this).InvalidateRecord((T)((object)this));
25   return result;
26 }
```

Listing 5.14: Kommentarüberschriften aus Piranha

Die Methode ist durch Kommentare in drei Sektionen unterteilt: Zuerst wird geprüft, ob sich null-Values unter den Primary Keys befinden. Falls ja, wird eine ArgumentException geworfen (//Check primary key). Anschließend wird der Datensatz in die Datenbank geschrieben – entweder wird der bereits bestehende Datensatz aktualisiert oder ein neuer Satz angelegt (//Execute command). Schließlich wird geprüft, ob es sich um einen gepufferten Datensatz handelt, der gegebenenfalls als ungültig markiert wird (//Check for cache interface).

Jeder Kommentar dieser Methode beschreibt eine Funktionalität, die in eine separate Methode ausgelagert werden sollte:

```
1 public virtual bool Save(IDbTransaction tx = null) {
2   ValidatePrimaryKeys();
3   WriteInDatabase(tx);
4   InvalidateCachedRecord();
5 }
```

Listing 5.15: Kommentarüberschriften aus Piranha nach Refaktorisierung

Dokumentation in nicht-öffentlichem Code

Nancy[11] ist ein leichtgewichtiges Framework, um HTTP basierende Services für .Net zu entwickeln. Nachfolgend zwei Codeschnipsel mit dokumentiertem, nicht-öffentlichem Code:

```
1  /// <summary>
2  /// Returns whether the given filename is contained within the
       content folder
3  /// </summary>
4  /// <param name="contentRootPath">Content root path</param>
5  /// <param name="fileName">Filename requested</param>
6  /// <returns>True if contained within the content root, false
       otherwise</returns>
7  private static bool IsWithinContentFolder(string contentRootPath,
       string fileName)
8  [...]
9
10 /// <summary>
11 /// Creates a response from the compatible headers.
12 /// </summary>
```

[11] github.com/NancyFx/Nancy

```
13 /// <param name="compatibleHeaders">The compatible
       headers.</param>
14 /// <param name="negotiationContext">The negotiation
       context.</param>
15 /// <param name="context">The context.</param>
16 /// <returns>A <see cref="Response"/>.</returns>
17 private static Response CreateResponse(IList<CompatibleHeader>
       compatibleHeaders, NegotiationContext negotiationContext,
       NancyContext context)
18 [...]
```

Listing 5.16: Dokumentation in Nancy

Die Beispiele wurden aus zwei Gründen ausgewählt: Zum einen dokumentieren sie, wie bereits erwähnt, nicht-öffentlichen Quelltext. Zum anderen sind sie redundant. Sie fügen fast keine neuen Informationen zu den Funktionsköpfen hinzu – würde die zweite genannte Methode CreateResponseFromHeaders heißen, wäre alles Wichtige bereits genannt. Diese Kommentare sind überflüssig.

5.3.4 Fazit

Die Erkennung von dokumentiertem, nicht-öffentlichem Code funktioniert mit Hilfe des CommentInspector eindeutig. Auch das Erkennen von auskommentiertem Code funktioniert ebenfalls sehr gut. Lediglich einzelne auskommentierte Zeilen, die nicht auf ein Semikolon enden, werden übersehen (False Negative).

Im Gegensatz dazu werden bei der Suche nach Kommentarüberschriften überwiegend False Positives gefunden. Das in Kapitel 5.3.2 beschriebene Kriterium für Kommentarüberschriften ist deshalb nicht hinreichend.

5.4 Objekte und Datenstrukturen

In der objektorientierten Programmierung werden Variablen gekapselt. Sie werden durch das private-Schlüsselwort nach außen hin verborgen, damit sie nur innerhalb von Objekten manipuliert werden können. Dennoch missbrauchen viele Entwickler das **Getter/Setter**-Pattern. Durch das sorglose Implementieren von Getter- und

Setter-Methoden wird die Kapselung von Instanzvariablen zum Teil ausgehebelt, denn die im Grunde verborgenen Variablen lassen sich damit so verwenden, als wären sie öffentlich[12]. [17, vgl. S.129]

5.4.1 Code Smells

Hybridstrukturen

Martin definiert den Unterschied zwischen Objekten und Datenstrukturen wie folgt: „Objekte verbergen ihre Daten hinter Abstraktionen und enthüllen Funktionen, die mit diesen Daten arbeiten. Datenstrukturen enthüllen ihre Daten und haben keine Funktionen.“ [17, S.131]

Hybride sind Klassen, die halb Objekt, halb Datenstruktur sind. Sie haben öffentliche Funktionen, um Aufgaben zu erfüllen. Sie haben ebenfalls öffentliche Variablen, beziehungsweise öffentliche Accessoren (Getter) und Mutatoren (Setter), die private Variablen enthüllen. [17, vgl. S.135]

Diese Art der Programmierung resultiert in einem Code Smell, der von Martin Fowler auch als „Feature Envy“, oder „Funktionsneid“ genannt wird. Werden in einer Methode die Variablen (oder deren Accessoren / Mutatoren) einer anderen Klasse verwendet um deren Daten zu manipulieren, so beneidet sie diese Klasse um deren Variablen. Die Methode wäre gerne in der anderen Klasse untergebracht, um deren Daten direkt nutzen zu können. [17, vgl. S.346 f.]

Verstöße gegen das Law of Demeter (LoD)

Das LoD besagt, dass eine Softwareeinheit nichts über die innere Struktur von Objekten wissen soll, mit denen sie arbeitet. Durch die Getter- und Setter-Zugriffsfunktionen werden die Daten und somit die innere Strukturen von Objekten enthüllt.

---

[12] Mit der Einschränkung, das mit Getter und Setter Methoden eine Validierung vorgenommen werden, um z.B. ungültige Zustände zu vermeiden.

Das LoD besagt, dass eine Methode m einer Klasse K nur Methoden der folgenden Komponenten aufrufen darf:

- K,
- ein Objekt, das von m erstellt wird,
- ein Objekt, das als Argument an m übergeben wird,
- ein Objekt, das in einer Instanzvariablen von K enthalten ist.

Der nachfolgende Codeausschnitt skizziert nochmals die beschriebenen Aufrufkriterien des LoD:

```
1  public class K
2  {
3    private FooBar _aMemberClass;
4    private void AnInstanceFunction() { }
5    private void m(Foo aParameter)
6    {
7    // Methodenaufruf von ...
8      // K
9      this.AnInstanceFunction();
10     // einem Objekt, das von m erstellt wird
11     Bar c = new Bar();
12     c.Foo();
13     // einem Objekt, das als Argument an m uebergeben wird
14     aParameter.Bar();
15     // einem Objekt, das eine Instanzvariable von K ist
16     _aMemberClass.BarFoo();
17   }
18 }
```

Listing 5.17: Aufrufkriterien des LoD

Es dürfen auch keine Methoden von Objekten aufgerufen werden, die von einer der oben genannten Komponenten zurückgegeben wird. Deswegen wird das LoD auch wie folgt zusammengefasst: „Sprich nur zu Freunden, nicht zu Fremden!“

Handelt es sich nicht um Objekte, sondern Datenstrukturen, sind solche Arten des Zugriffs erlaubt, da diese ihre interne Struktur, im Gegensatz zu Objekten, enthüllen. [17, vgl. S. 133 f.]

5.4.2 Implementierung: ObjectAndDatastructureValidator

Der ObjectAndDatastructureValidator ist für das Aufdecken von Hybridstrukturen und Verstößen gegen das LoD verantwortlich. Die Implementierung der Funktionalitäten wird im Folgenden beschrieben.

Erkennen von Hybridstrukturen

Um Hybridstrukturen zu erkennen, werden aus dem zu untersuchenden Dokument die TypeDeclarationSyntax herausgefiltert. Dazu zählen Klassen, Strukturen und Interfaces. Letztere können für die weiteren Untersuchungen vernachlässigt werden.

Nachfolgend ein Ausschnitt aus dem Syntaxbaum, der die verschiedenen Typen von Instanzvariablen und deren Platzierung innerhalb einer ClassDeclarationSyntax zeigt:

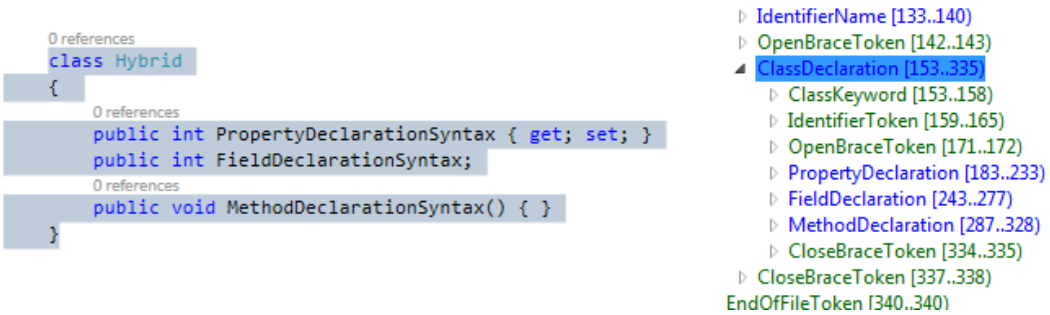

Abbildung 5.3: Aufbau einer ClassDeclarationSyntax

Die Klasse Hybrid besitzt zwei Instanzvariablen. Dabei wird zwischen PropertyDeclarationSyntax (mit Accessoren/Mutatoren) und FieldDeclarationSyntax (ohne Accessoren/Mutatoren) unterschieden. Außerdem beherbergt die Klasse noch eine Methode. Instanzvariablen und Methoden haben jeweils ein Token, das ihren Zugriffsmodifikatoren entspricht: public, private, protected oder internal.

Bei dem zu untersuchenden Typ wird zwischen den folgenden Fällen unterschieden:

- Sind öffentliche Variablen und keine Methodendeklarationen vorhanden, handelt es sich um eine Datenstruktur.

- Sind öffentliche Methoden und keine öffentlichen Variablen vorhanden, handelt es sich um ein Objekt.
- Treffen beiden der oben genannten Kriterien zu, handelt es sich um eine Hybridstruktur.

Erkennen von Verstößen gegen das Law of Demeter (LoD)

Um zu überprüfen, ob es sich bei einem gegebenen Methodenaufruf um einen Verstoß des LoD handelt, ist eine Auswertung des Syntaxbaumes alleine nicht ausreichend – das semantische Model (siehe Kapitel 3.4.3) muss ebenfalls zu Rate gezogen werden.

In einem ersten Schritt werden alle Methodenaufrufe aus dem zu untersuchenden Codedokument extrahiert. Roslyn stellt diese als $\mathrm{InvocationExpressionSyntax}$-Knoten im Syntaxbaum dar. Ausgehend eines jeden einzelnen Aufrufes a und dem korrespondierenden InvocationExpression-Knoten, wird der Baum in Richtung des Root-Knotens durchlaufen, um:

- die beinhaltende Methode m des Aufrufes a zu finden ($\mathrm{MethodDeclarationSyntax}$),
- die beherbergende Klasse K zu ermitteln ($\mathrm{TypeDeclarationSyntax}$).

Diese Informationen sind Ausgangspunkt für die Überprüfung des LoD. Nun folgen die Prüfungen der in Kapitel 5.4.1 genannten Kriterien und deren schematischer Ablauf:

Ist der Aufruf a Instanzmethode von K?

Mit Hilfe der bereits gefundenen $\mathrm{TypeDeclarationSyntax}$ von K können alle Kind-Knoten vom Typ $\mathrm{MethodDeclarationSyntax}$ aus dem Syntaxbaum gefiltert werden. Diese Knoten entsprechen den Instanzmethoden von K. Über das semantische Modell werden deren Symbole (siehe Kapitel 3.4.2) ermittelt. Befindet sich unter den Symbolen das Symbol des Aufrufes a, so ist dieses Kriterium erfüllt.

Ist der Aufruf a Instanzmethode eines Objekts, das in m erstellt wird?

1. Ausgehend der $\mathrm{MethodDeclarationSyntax}$ von m, werden alle Kind-Knoten vom Typ $\mathrm{LocalDeclarationStatementSyntax}$ aus dem Syntaxbaum gefiltert. Diese repräsentieren alle Objektinstanziierungen innerhalb von m.

2. Mit dem semantische Modell werden anschließend aus den Bezeichnern der Instanziierungen deren Symbole ermittelt.

3. Über die Symbole können die Instanzmethoden der in m erstellen Objekte gefunden werden. Diese werden als $\mathrm{MethodDeclarationSyntax}$ dargestellt.

4. Erneut werden über das semantische Model die Symbole der Deklarationen ermittelt. Ist das Symbol von a in den ermittelten Symbolen der Deklarationen, so ist dieses Kriterium erfüllt.

Ist der Aufruf a Instanzmethode eines Objekts, das an m übergeben wird?

1. Aus der gefundenen $\mathrm{MethodDeclarationSyntax}$ der Methode m werden, wie in Kapitel 5.2.2 beschrieben, die Parameter und deren Bezeichner herausgelesen.

2. Über das semantische Modell werden anschließend aus den Bezeichnern die korrespondierenden Symbole ermittelt.

3. Mit den Symbolen können die Instanzmethoden der Parameter-Typen in Form von $\mathrm{MethodDeclarationSyntax}$ ausfindig gemacht werden.

4. Das semantische Modell liefert zu den $\mathrm{MethodDeclarationSyntax}$ ebenfalls die entsprechenden Symbole. Wenn sich unter den Symbolen der Methodendeklarationen des Parametertyps das Symbol der zu untersuchenden $\mathrm{MethodInvocationSyntax}$ befindet, so handelt es sich um einen Methodenaufruf eines Objekts, das als Argument an die Methode übergeben wurde.

Ist der Aufruf a **Instanzmethode einer Instanzvariablen von** K**?**

Um dies zu überprüfen werden von der bereits gefundenen TypeDeclarationSyntax von K die Kind-Knoten vom Typ MemberDeclarationSyntax im Syntaxbaum gesucht. Anschließend können, wie bereits in den beiden letzten Überprüfungen beschrieben, über das semantische Modell und die Symbole der Instanzvariablen, deren Instanzmethoden gefunden werden. Befindet sich das Symbol des Aufrufes a unter den Symbolen der Instanzmethoden, so ist dieses Kriterium erfüllt.

Ist eines dieser Kriterien für Objekte und Hybrid-Typen erfüllt, so wird das LoD nicht verletzt.

5.4.3 Analysefunde

Hybridstrukturen

Das folgende Codeschnipsel stammt aus dem Nancy Projekt. Die dargestellte Klasse Response repräsentiert eine HTTP-Antwort:

```
1  public class Response: IDisposable
2  {
3    [...]
4    public IDictionary<string, string> Headers { get; set; }
5    public HttpStatusCode StatusCode { get; set; }
6    public string ReasonPhrase { get; set; }
7    public IList<INancyCookie> Cookies { get; set; }
8    [...]
9    public Response AddCookie(INancyCookie nancyCookie)
10   {
11     Cookies.Add(nancyCookie);
12     return this;
13   }
14   [...]
15 }
```

Listing 5.18: Hybrid-Klasse Response aus Nancy

Aus Platzgründen ist die Klasse nur in Auszügen dargestellt. Es hat den Anschein, als ob zu allen Instanzvariablen automatisch Getter und Setter Methoden[13] hin-

[13] inklusive redundanter Dokumentationskommentare, die aus Platzgründen ebenfalls nicht im Listing dargestellt sind

zugefügt wurden – selbst wenn diese nicht alle verwendet werden. So wird die Kapselung der Variablen umgangen, obwohl beispielsweise mit AddCookie eine Methode bereitgestellt wird, mit der gekapselt ein neues Cookie der Liste hinzuzufügen werden kann. Bei der gezeigten Klasse handelt es sich um einen Objekt-Datenstruktur-Hybriden, da die Instanzvariablen über Zugriffsfunktionen publik gemacht werden und die Klasse zudem über öffentliche Methoden verfügt.

Bei dem Projekt Orchard[14] handelt es sich um ein freies Content-Management-System (CMS). Die Hybridklasse WebHost hat die meisten ihrer Variablen als private und ohne Getter/Setter deklariert. Mit Ausnahme des folgenden Ausschnittes:

```
1 public class WebHost
2 {
3   [...]
4   public string Cookies { get; set; }
5   [...]
6 }
```

Listing 5.19: Hybridklasse Webhost aus Orchard

In dieser Variablen können mehrere Cookies, durch ein Semikolon getrennt, in Textform gespeichert werden. Wird der Zugriff auf diese Variable nicht gekapselt, setzt das voraus, dass andere Klassen, die mit den Cookies eines WebHost arbeiten, über deren Implementierung und Formatierung Bescheid wissen müssen. Sollen neue Cookies hinzugefügt, ersetzt oder gelöscht werden, sieht der entsprechende Code so aus:

```
1 public class WebHost
2 {
3   // remove
4   webHost.Cookies = Regex.Replace(webHost.Cookies,
        string.Format("{0}=[^;]*;?", cookieName), "");
5   // replace
6   webHost.Cookies = Regex.Replace(webHost.Cookies,
        string.Format("{0}=[^;]*(;?)", cookieName),
        string.Format("{0}$1", setCookie.Split(';')[0]));
7   // add
8   webHost.Cookies = (webHost.Cookies + ';' +
        setCookie.Split(';').FirstOrDefault()).Trim(';');
9 }
```

Listing 5.20: Verwendung von Cookies

[14] github.com/OrchardCMS/Orchard

Die Operationen sind kompliziert und nur schwer verständlich. Außerdem wird eine große Menge an Wissen über die interne Struktur der WebHost–Klasse vorausgesetzt. Eine Änderung der Implementierung zieht in diesem Fall auch eine Änderung der Aufrufe mit sich: Wird beispielsweise der string durch eine List ersetzt, hat das zur Folge, dass auch Code zum Manipulieren von Cookies außerhalb der Klasse WebHost angepasst werden muss.

Verstöße gegen das Law of Demeter

Hangfire[15] ist ein Task-Handler und Scheduler für ASP.NET Anwendungen. Aus dem Projekt ist folgende Codezeile entliehen:

```
1 configuration.Entry.QueueProviders.Add(provider, queues);
```

Listing 5.21: LoD-Verstoß in Hangfire

configuration ist ein Übergabeparameter an die Methode, aus der die Zeile entnommen wurde. Nach dem LoD ist es in Ordnung, wenn von configuration eine Instanzfunktion (Entry) aufgerufen wird (siehe Kapitel 5.4.1). Anschließend wird allerdings noch auf die QueueProviders-Collection und deren Add()-Methode zugegriffen. Die aufrufende Methode weiß folglich, dass der configuration-Parameter die Instanzvariable Entry vom Typ SqlServerStorage hat. Weiterhin weiß die Funktion, dass Entry eine Collection vom Typ PersistentJobQueueProviderCollection besitzt, welche wiederum die Add-Methode zur Verfügung stellt. Das ist sehr viel Wissen für eine Funktion. Die Funktion, in der diese Verkettung aufgerufen wird, weiß, wie sie zwischen all diesen Objekten navigieren kann. Ändert sich die Struktur oder die Implementierung einer der an dem verketteten Aufruf beteiligten Klassen, so muss auch die Aufruflogik überarbeitet werden.

5.4.4 Fazit

Hybridstrukturen und Verstöße gegen das LoD können mit einer hohen Zuverlässigkeit ermittelt werden. Allerdings ist die wörtliche Interpretation das Law of

[15] github.com/HangfireIO/Hangfire

Demeter als „Gesetz“ umstritten: Auf der einen Seite resultiert dessen konsequente Anwendung in gut gekapselten, nur schwach voneinander abhängigen Softwareeinheiten. Dies hat unmittelbar eine höhere Evolvierbarkeit des Codes zur Folge. Auf der anderen Seite steigt damit anfänglich auch der Mehraufwand bei der Entwicklung: Es müssen Wrapper-Methoden erstellt werden, die Methodenaufrufe weiterleiten. Manchmal ist es sehr unwahrscheinlich, dass sich die Struktur von Klassen ändert, der Vorteil der besseren Kapselung von Klassen ist somit obsolet. Es bleibt der Mehraufwand, den die Entwickler durch die Einhaltung des LoD haben. Eine strikte Anwendung des LoD ist deswegen nicht in allen Fällen sinnvoll und wird von manchen Entwicklern auch bewusst vermieden.

5.5 Fehlerbehandlung

Maßgeblich für die Qualität von Software ist deren Robustheit – die Fähigkeit, auftretende Fehler so zu behandeln, dass der Programmablauf möglichst wenig beeinträchtigt wird. Fehler treten in den besten Programmen auf: Die Eingabe des Benutzers hat das falsche Format oder der Zugriff auf einen Dateiserver ist nicht möglich, da das Netzwerk ausgefallen ist. Geht etwas schief, muss sichergestellt sein, dass die Software in dieser Situation richtig mit dem Fehler umgeht. Die Bedingungen für das Fortsetzen des regulären Programmablaufes müssen hergestellt werden.

Nach Martin werden viele Projekte von Fehlerbehandlungscode dominiert. Überall im Code verstreut sind Anweisungen, die der Fehlerbehandlung dienen. Dadurch wird die eigentliche Absicht des Quelltextes verdeckt. Er sagt: „Fehler-Handling ist wichtig, aber wenn es die Logik verschleiert, ist es falsch.“ [17, S.139]

5.5.1 Code Smells

Fehlercodes als Rückgabewert

Zu Zeiten, als es noch keine Exceptions gab, wurden zur Fehlerbehandlung oft Fehler-Flags und Rückgabewerte verwendet. Über das Flag wurde bestimmt, ob eine Methode erfolgreich war oder nicht, der codierte Rückgabewert hingegen lieferte Hinweise auf den aufgetretenen Fehler.

Das Problem dieser Technik ist, dass der Fehlerstatus unmittelbar nach Aufruf der Methode geprüft werden muss. So vermischt sich Code, welcher der Fehlerbehandlung dient, mit dem der Geschäftslogik. Der Code wird unübersichtlich und es werden verschiedene Belange vermischt. Außerdem passiert es schnell, dass die Prüfung vergessen wird – der Fehler bleibt unbehandelt.

Mittlerweile sind solche Techniken überflüssig. Im Fehlerfall sollte stattdessen eine Exception geworfen werden. So kann Geschäftslogik von Fehlerbehandlung besser getrennt werden. [17, vgl. S.139 ff.]

Null-Referenzen als Rückgabewerte

Ein $\mathrm{return\ null}$ wird oft verwendet, um zu zeigen, das etwas außergewöhnliches passiert ist oder ein Fehler aufgetreten ist. Quasi eine Vorstufe der Fehlerbehandlung. Dieser Art der Programmierung hat allerdings unangenehme Folgen: Im Fehlerfall ist es leicht und schnell, eine null-Referenz zurückzugeben. So werden die Probleme auf den Aufrufer abgewälzt. Jeder Rückgabewert könnte eine potentielle null-Referenz sein und muss entsprechend geprüft werden. So passiert es schnell, dass der aufrufende Code mit Prüfungen auf null-Referenzen überschwemmt wird. Die eigentliche Logik geht in verschachtelten $\mathrm{if\ (returnValue\ !=\ null)}$ Kontrollstrukturen unter.

Wird die Prüfung auf null vergessen, kann zur Laufzeit eine unbehandelte $\mathrm{NullReferenceException}$ ausgelöst werden. Oder die Exception wird noch auf oberster Ebene noch gefangen. Beides ist schlecht. Im ersten Fall stürzt das Programm ab, im

zweiten Fall kann kaum noch etwas getan werden, um den Fehler zu beheben, der in einer tieferen Ebene des Programms aufgetreten ist. [17, vgl. S.147 ff.]

Alternativen zu null-Rückgabewerten sind beispielsweise das Auslösen einer Exception oder die Rückgabe eines Special Case-Objekts. Das Special Case-Pattern sieht vor, anstelle von null eine eigene Null-Implementierung des Interfaces, das der Aufrufende als Rückgabe erwartet, zurückzugeben. [8]

Methoden aus den APIs von Drittanbietern, die null-Rückgabewerte haben können, sollten in einem Wrapper verpackt werden. Dieser wirft im Fehlerfall eine Exception oder gibt ein Special Case-Object zurück.

Mit diesen Techniken wird die Gefahr einer NullReferenceException minimiert. Außerdem wird die Aufruflogik und Fehlerbehandlung so übersichtlicher und sauberer. [17, vgl. S.147 f.]

Null-Referenzen als Übergabewerte

Manchmal werden null-Argumente als Defaultwert oder Hinweis für einen nicht vorhandenen Wert an eine Methode übergeben. Immer, wenn diese Technik angewandt wird, besteht die Gefahr, dass das Programm zur Laufzeit eine NullReferenceException auslöst. Wenn nicht gerade mit APIs gearbeitet wird, die null als Übergabewert für bestimmte Parameter erwartet, sollte darauf verzichtet werden.

Es besteht die Möglichkeit der defensiven Programmierung, also zu Beginn jeder Methode die Übergabeparameter auf null zu prüfen und eine InvalidArgumentException zu werfen. Diese benötigt allerdings einen Handler mit definiertem Verhalten. Was soll geschehen, wenn ein null-Parameter übergeben wurde? Mittels Zusicherungen (Asserts) können die Parameter ebenfalls verifiziert werden. Damit wird zwar das Programm dokumentiert, trotzdem wird noch ein Laufzeitfehler ausgelöst.

Es ist besser, gänzlich auf die Übergabe von null zu verzichten. Wird unter dieser Voraussetzung im Code ein Aufruf mit null-Argument entdeckt, weiß der Entwickler sofort, dass etwas nicht in Ordnung ist. [17, vgl. S.148 f.]

5.5.2 Implementierung: ErrorHandlingInspector

Erkennen von Fehlercodes und Null-Referenzen als Rückgabewerte

Um die in Kapitel 5.5.1 beschrieben Code Smells aufzuspüren, werden die Rückgabewerte von den Methoden des zu untersuchenden Code-Dokuments näher betrachtet. Der Aufbau einer return-Zeile einer Methode sieht im Syntaxbaum wie folgt aus:

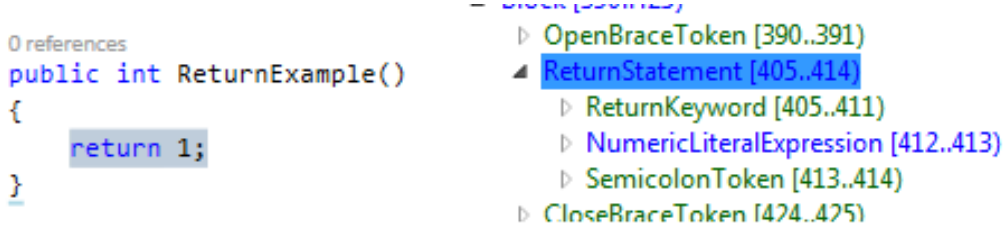

Abbildung 5.4: Darstellung eines Return-Ausdrucks im Syntaxbaum

Wie in der Abbildung zu sehen ist, wird die return-Anweisung durch ein ReturnStatementSyntax dargestellt. Dieses beinhaltet zuerst ein SyntaxToken für das return-Schlüsselwort, anschließend folgt der zurückzugebende Ausdruck, der variieren kann. In dem gezeigten Beispiel wird eine fest einprogrammierte „1“ zurückgeben – repräsentiert durch eine NumericLiteralExpression. Wenn eine Methode eine hart-codierte, ganze Zahl zurückgibt, ist dies ein Indiz für einen Fehlercode. Im Syntaxbaum werden postive Zahlen mit einer NumericLiteralExpression, negative mit einer PrefixUnaryExpressionSyntax dargestellt.

Wird eine NullLiteralExpresision zurückgegeben, handelt es sich um einen Fund des in Kapitel 5.5.1 beschriebenen Smells.

Null-Referenzen als Übergabewerte

Um null-Argumente zu finden, werden alle Methodenaufrufe näher unter die Lupe genommen. Das nachfolgende Bild zeigt die den Aufbau der Aufrufsyntax innerhalb des Syntaxbaums:

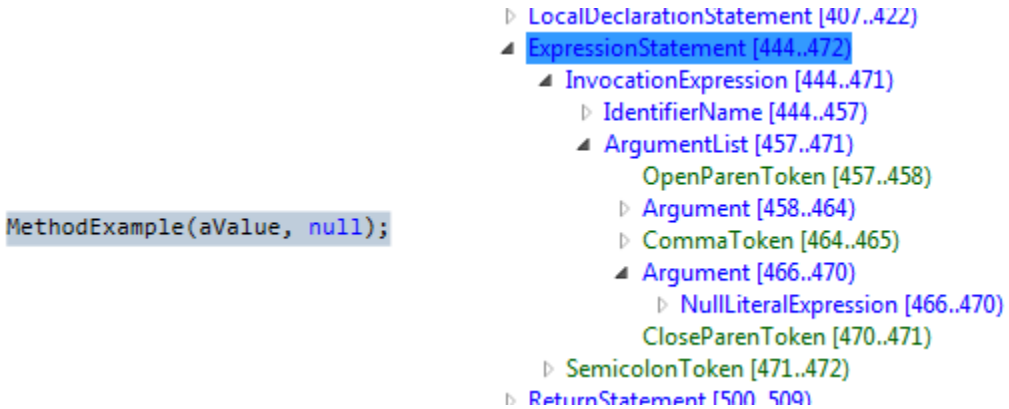

Abbildung 5.5: Darstellung eines Methoden-Aufrufes im Syntaxbaum

Der Aufruf wird durch eine InvocationExpressionSyntax dargestellt. Diese hat ein ArgumentListSyntax-Member, welches wiederum die tatsächlichen Argumente beherbergt. Auf der Suche nach null-Argumenten werden alle Argument-Listen nach NullLiteralExpressionSyntax-Argumenten durchsucht.

5.5.3 Analysefunde

Fehler-Rückgabewerte

Das folgende Beispiel stammt aus einer Utility-Klasse aus Nancy:

```
1  private static int GetInt(byte b)
2  {
3    char c = (char) b;
4    if (c >= '0' && c <= '9')
5      return c - '0';
6
7    if (c >= 'a' && c <= 'f')
8      return c - 'a' + 10;
9
10   if (c >= 'A' && c <= 'F')
11     return c - 'A' + 10;
12
13   return -1;
14 }
```

Listing 5.22: Fehler-Rückgabewerte

Mit Hilfe der gezeigten Methode wird ein einstelliger, als Byte codierter, hexadezimal Wert in einen Integer Wert konvertiert. Falls der Parameter kein gültiger Hex-Wert ist, so wird „-1“ zurückgegeben. Der Aufrufende muss also nach Erhalt des Rückgabewertes der GetInt-Methode prüfen, ob das Ergebnis valide ist. Wird die Prüfung vergessen, wird der Rückgabewert falsch interpretiert und weitere Berechnungen mit dem Wert liefern ebenfalls falsche Ergebnisse. Eine besser Lösung wäre es, z.B. eine NotAValidHexValueException zu werfen. Der Entwickler merkt sofort, dass etwas nicht in Ordnung ist und wird zur Behandlung des Fehlers gezwungen.

Null-Referenzen als Rückgabewerte

Orleans[16] ist ein Microsoft Research Projekt. Dabei handelt es sich um ein Framework, mit dessen Hilfe verteilte Systeme für rechenintensive Anwendungen entwickelt werden können. Die folgenden Codeausschnitte stammen aus diesem Projekt:

```
1
2  internal readonly RuntimeQueue<Message> PendingInboundMessages;
3  [...]
4  public Message WaitMessage(Message.Categories type,
       CancellationToken ct)
5  {
6    try
7    {
8      Message msg = PendingInboundMessages.Take();
9      [...]
10     return msg;
11   }
12   catch (ThreadAbortException tae)
13   {
14     [...]
15     return null;
16   }
17   catch (OperationCanceledException oce)
18   {
19     [...]
20     return null;
21   }
```

16 github.com/dotnet/orleans

```
22    catch (Exception ex)
23    {
24       [...]
25       return null;
26    }
27 }
```

Listing 5.23: Null-Rückgabewerte

Mit Hilfe der Methode WaitMessage wird auf eine eingehende Nachricht gewartet. Diese soll mittels der Methode Take der PendingInboundMessages-Queue entnommen werden. Dazu wird innerhalb der Methode Take in einer Endlosschleife geprüft, ob eine neue Nachricht in der Warteschlange eingetroffen ist. Dabei können etliche Ausnahmen auftreten, welche die Schleife und somit auch die Wartezeit terminieren. In jedem Fall wird anstelle der erwarteten Antwort eine null-Referenz zurückgegeben.

Stattdessen wäre es möglich, beide der in Kapitel 5.5.1 beschrieben Lösungsansätze zu implementieren: Die bereits gefangen Ausnahmen könnten weiter geworfen werden. Ist aus einem bestimmten Grund nicht erwünscht, dass der Aufrufer eine Exception erhält, besteht die Alternative darin, ein Special Case-Implementierung der Message-Klasse zurückzugeben, die kein, oder ein bestimmtes Verhalten für den Fehlerfall implementiert. Somit entfällt die null-Prüfung des Rückgabewerts und es muss keine Ausnahme behandelt werden.

Null-Referenzen als Übergabewerte

Nachfolgend ein Auszug eines Call-Stacks in ShareX:

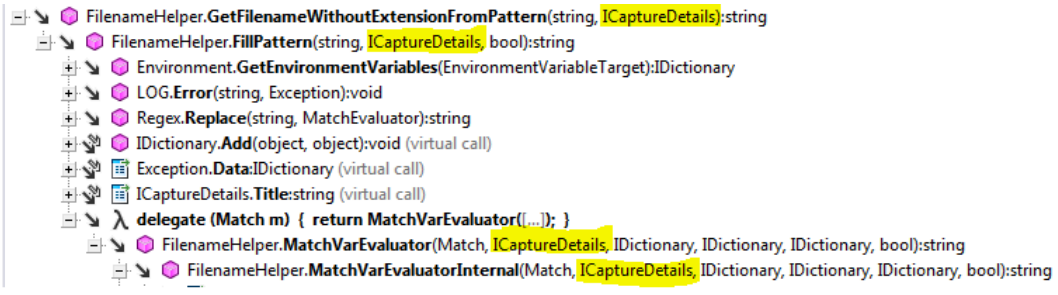

Abbildung 5.6: Übergabe einer null-Referenz als Argument in ShareX

Der Aufruf der obersten Methode des Stacks sieht wie folgt aus:

```
1 GetFilenameWithoutExtensionFromPattern(pattern, null);
```

Listing 5.24: null-Referenz als Argument in ShareX

Das ICaptureDetails-Argument wird als null-Referenz übergeben. Wie den hervorgehobenen Stellen des Callstacks zu entnehmen ist, wird ICaptureDetails durch die Methoden FillParameter und MatchVarEvaluator bis hin zu MatchVarEvaluatorInternal weitergereicht. Erst dort wird das Argument letztlich auf null geprüft und verwendet:

```
1 [...]
2 if (captureDetails != null && [...])
3 {
4   replaceValue = captureDetails.MetaData[variable];
5   [...]
6 }
```

Listing 5.25: null-Referenz als Argument in ShareX

Wird die Implementierung geändert, und das ICaptureDetails bereits vorher genutzt, wird schnell die notwendige Prüfung auf null vergessen und eine NullReferenceException fliegt. Zwar befinden sich die meisten Methodenaufrufe des gezeigten Call-Stacks in try / catch-Blöcken, diese implementieren allerdings nur den Default-Handler für die Basis-Klasse Exception.

5.5.4 Fazit

Die implementierten Detektoren können zuverlässig erkennen, wenn explizit null-Referenzen als Argumente übergeben bzw. als Rückgabewert zurückgegeben werden. Auch sogenannte Magic-Numbers oder Fehlerflags als Rückgabewert werden zuverlässig erkannt. Vorausgesetzt, die Werte sind nicht in einer Konstanten o.ä. gekapselt. Dies zu erkennen wäre auch möglich, allerdings wurde aus zeitlichen Gründen auf die Implementierung verzichtet. Dadurch werden nicht alle Vorkommen der beschriebenen Smells erkannt.

5.6 Sonstiges

5.6.1 Code Smells

Nicht gekapselte Grenzbedingungen

Die Verarbeitung von Grenzbedingungen sollte an einer Stelle im Code zusammengefasst sein. Wird sie im Code verteilt, macht das diesen anfällig für Flüchtigkeitsfehler. Folgendes Beispiel soll zeigen, was unter Grenzbedingungen verstanden wird:

```
1 public int GetValue(int index, int[] arr)
2 {
3   if (index >= arr.Length + 1)
4   {
5     throw new IndexOutOfRangeException
6       ("Index is greater than array size.");
7   }
8   return arr[index];
9 }
```

Listing 5.26: Beispiel für eine Grenzbedingung

In dem Beispiel wird geprüft, ob ein gegebener Index außerhalb der Reichweite des Array-Parameters liegt. Die Prüfung auf $\mathrm{array.Length} + 1$ wird als Grenzbedingung bezeichnet. Wird eine Grenzbedingung öfter als einmal verwendet, sollte diese mit einer separate Variable gekapselt werden. [17, vgl. S.359]

Ringabhängigkeiten in Vererbungshierarchien

Beim Entwurf objektorientierter Systeme werden oft Vererbungshierarchien entwickelt. Dabei ist es wichtig, dass die Basisklasse unabhängig von den abgeleiteten Klassen ist, damit das Konzept der Basisklasse von den Konzepten der abgeleiteten Klassen getrennt ist. Sind in einer Basisklasse Abhängigkeiten zu einer von ihr abgeleiteten Klasse vorhanden, so ist möglicherweise mit dem Code etwas nicht in Ordnung. Im Allgemeinen sollten Basisklassen nichts über ihre abgeleiteten Klassen wissen. [17, vgl. S. 344]

5.6.2 Implementierung: BoundaryConditionInspector

Um nicht gekapselte Grenzbedingungen zu finden, werden in einem ersten Schritt alle arithmetischen Operationen aus dem zu untersuchenden Dokument herausgefiltert. Folgender Screenshot soll kurz deren Aufbau verdeutlichen:

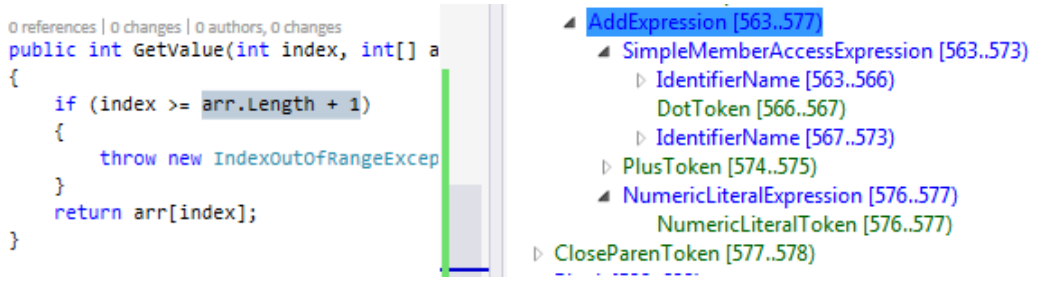

Abbildung 5.7: Darstellung einer arithmetischen Operation im Syntaxbaum

In dem Bild wird das oben bereits vorgestellte Beispiel nochmals aufgegriffen: Bei der arithmetischen Operation handelt es sich um eine Addition ($\mathrm{arr.Length} + 1$). Die im Syntaxbaum zu sehende AddExpressionSyntax ist eine Subklasse von BinaryExpressionSyntax. Diese wiederum umfasst unter anderem die vier Grundrechenarten. Eine BinaryExpressionSyntax hat drei direkte Kind-Knoten: zwei ExpressionSyntax, welche die Werte darstellen, mit denen gerechnet wird. Dabei kann es sich um fest einprogrammierte Werte, Variablen, Rückgabewerte von Methoden usw. handeln. Zwischen den beiden ExpressionSyntax befindet sich ein Token, das die auszuführende Rechenoperation determiniert; im Falle des Beispiels ein PlusToken.

Anschließend werden alle Operationen herausgefiltert, die mindestens zweimal vorkommen. Dazu muss geprüft werden, ob eine BinaryExpressionSyntax ein Äquivalent mit den selben Operanden und dem selben Operator in der Liste aller gefundenen BinaryExpressionSyntax hat. Handelt es sich bei einem Operanden um eine fest einprogrammierte Zahl (NumericLiteralExpression), so muss geprüft werden, ob die Werte identisch sind. Handelt es sich um eine ExpressionSyntax, wird mit Hilfe dem semantischen Modell überprüft, ob die Symbole der Operanden identisch sind.

5.6.3 Implementierung: InheritanceInspector

Um Ringabhängigkeiten in Vererbungshierarchien zu finden, muss eine Projekt mit allen dazugehörigen Quellcode-Dokumenten betrachtet werden. In einem ersten Schritt werden alle Basisklassen gesucht. Dazu werden alle Typdeklarationen, repräsentiert durch BaseTypeDeclarationSyntax, betrachtet. Leitet eine Klasse von einer anderen Klasse ab, so beinhaltet die BaseTypeDeclarationSyntax eine BaseListSyntax, in welcher die Basisklassen (SimpleBaseTypeSynax), sowie zu implementierende Interfaces gelistet sind. Mit Hilfe des Identifiers eines SimpleBaseTypeSyntax kann über das semantische Modell das Symbol der Basisklasse gefunden werden. Folgender Auszug aus dem Syntaxbaum verdeutlicht die beschriebene Struktur:

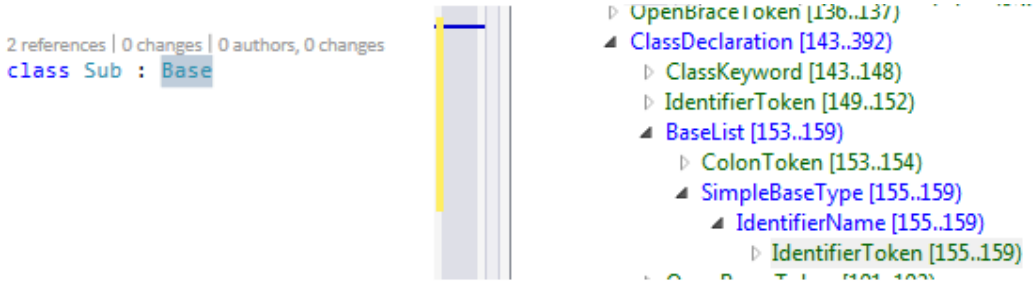

Abbildung 5.8: Darstellung einer Klasse Sub, die von Base ableitet

In einem nächsten Schritt werden zu den gefundenen Basisklassen alle von ihnen ableitenden Klassen zusammengetragen. Dazu wird erneut über alle BaseTypeDeclarationSyntax iteriert. Ist die momentan betrachtet Klasse eine abgeleitete Klasse, so wird sie dem passenden, im ersten Schritt gefundenen, Basistypen zugeordnet.

Nun werden die Symbole aller Objekterstellungen (ObjectCreationExpressionSyntax) und Methodenaufrufe (InvocationExpressionSyntax) jeder gefundenen Basisklasse genauer betrachtet: Ist das erstellte Objekt vom Typ einer abgeleiteten Klasse oder ist die aufgerufene Methode in einer abgeleiteten Klasse implementiert, so handelt es sich um eine Ringabhängigkeit.

5.6.4 Analysefunde

Nicht gekapselte Grenzbedingungen

Das folgende Beispiel ist aus ShareX entnommen:

```
protected override void Draw(Graphics g)
{
  if (points.Count > 2)
  {
    [...]
    g.DrawLine(borderPen, points[points.Count - 1], points[0]);
    g.DrawLine(borderDotPen, points[points.Count - 1], points[0]);
    [...]
  }
  [...]
}
```

Listing 5.27: Beispiel für nicht gekapselte Grenzbedingungen aus ShareX

Unter anderem zeichnet diese Methode eine gepunktete und eine durchgezogene Verbindungslinie zwischen dem ersten und letzten Punkt, die in einem Array gespeichert sind. Die Indexierung des letzten Punktes erfolgt für beide Linien über dieselbe Grenzbedingung: points.Count-1. Bessere wäre es, den Index des letzten Punktes zu kapseln:

```
protected override void Draw(Graphics g)
{
  [...]
  var lastIndex = points.Count - 1;
  g.DrawLine(borderPen, points[lastIndex], points[0]);
  g.DrawLine(borderDotPen, points[lastIndex], points[0]);
  [...]
}
```

Listing 5.28: Beispiel für Grenzbedingung nach Refaktorisierung

Ringabhängigkeiten in Vererbungshierarchien

Das folgende Codebeispiel wurde aus dem Greenshot Image Editor entnommen:

```
public abstract class FastBitmap : IFastBitmap
{
  [...]
  public static IFastBitmap Create(Bitmap source, Rectangle area)
```

```
5     {
6       switch (source.PixelFormat)
7       {
8         case PixelFormat.Format8bppIndexed:
9           return new FastChunkyBitmap(source, area);
10        case PixelFormat.Format24bppRgb:
11          return new Fast24RGBBitmap(source, area);
12        case PixelFormat.Format32bppRgb:
13          return new Fast32RGBBitmap(source, area);
14        case PixelFormat.Format32bppArgb:
15        case PixelFormat.Format32bppPArgb:
16          return new Fast32ARGBBitmap(source, area);
17        default:
18          throw new NotSupportedException(string.Format("Not
              supported Pixelformat {0}", source.PixelFormat));
19      }
20    }
21    [...]
22  }
```

Listing 5.29: Beispiel für Ringabhängigkeit in Vererbungshierarchien

Je nach Zustand des übergebenen Bitmap-Parameters erstellt die Methode Create der FastBitmap-Klasse eine spezielle Implementierung des IFastBitmap-Interfaces und gibt diese zurück.

Problematisch ist, dass alle möglichen Rückgabewerte von der FastBitmap Klasse ableiten:

```
1 public class FastChunkyBitmap : FastBitmap [...]
2 public class Fast24RGBBitmap : FastBitmap [...]
3 public class Fast32RGBBitmap : FastBitmap [...]
4 public class Fast32ARGBBitmap : FastBitmap [...]
```

Listing 5.30: Beispiel für Ringabhängigkeit in Vererbungshierarchien

Die Basisklasse FastBitmap muss alle von ihr abgeleiteten Klassen kennen, um diese erstellen zu können. Soll eine weiter Ableitung dazukommen, muss die Methode Create der Basisklasse geändert werden, um die neue Ableitung berücksichtigen können – die Basisklasse ist von ihren abgeleiteten Klassen abhängig.

Eine bessere Lösung wäre die Implementierung des Factory-Entwurfsmusters[17]. Dadurch bleibt die Basisklasse unabhängig von ihren Ableitungen.

[17] http://www.blackwasp.co.uk/AbstractFactory.aspx

5.6.5 Fazit

Der Code Smell einer nicht gekapselten Grenzbedingung kann ebenfalls sehr zuverlässig erkannt werden. Bei den Auswertungen der Analyseergebnisse ist aufgefallen, dass dieses Smell sehr häufig auftritt, obwohl dessen Vermeidung nur sehr wenig Aufwand erfordert.

Aufgrund eines Implementierungsfehlers werden viele False Positives bei der Suche nach Ringabhängigkeiten gefunden. Trotzdem ist die Zahl der Funde relativ gering. Folglich tritt der beschriebene Code Smell in der Praxis relativ selten auf.

5.7 Auswertung der Ergebnisse

Im Rahmen dieser Arbeit wurden in Summe 3.096.708 Zeilen Code, 390.168 Methoden und 54.366 Klassen in 31 verschiedenen Projekten untersucht. Darunter wurden in Summe 313.781 verschiedene Code Smells entdeckt. Die genannten Zahlen berücksichtigen nicht, dass sich unter den Funden auch False Positives befinden und nicht alle im untersuchten Code vorhandenen Smells auch tatsächlich gefunden wurden. Die Funde der in den Kapiteln 5.3.1 und 5.6.1 beschriebenen Code Smells „Kommentarüberschriften“ und „Ringabhängigkeiten in Vererbungshierarchien“ werden nicht in die oben genannten Zahlen mit eingerechnet, da diese aus überwiegend False Positives bestehen. Für weitere Auswertungen werden diese beiden Smells nicht berücksichtigt.

Nachfolgend werden die drei am häufigsten auftretenden Code Smells aufgezeigt:

- Verstöße gegen das LoD (112.373)
- Namen, die aus einem Buchstaben bestehen (54.716)
- Nicht gekapselte Grenzbedingungen (31.569)

Mit Abstand am häufigsten wird gegen das LoD verstoßen – knapp ein Drittel aller gefundenen Smells sind von diesem Typ. Die absoluten Zahlen sind allerdings

wenig aussagekräftig. In einer Zeile Code kann mehrfach gegen das LoD verstoßen werden, wohingegen eine Hybridklasse nur einmal je Klassendeklaration auftreten kann.

Um Quellcode qualitativ bewerten zu können, wird die Anzahl der Funde eines Smells unter anderem in Relation zu den in Kapitel 4.2 beschriebenen Metriken gesetzt. Noch nicht beschrieben wurden die folgenden Kennzahlen:

- Number Of Invocations (NOI): Die Anzahl aller Methodenaufrufe und Zugriffsfunktionen
- Number Of Comments (NOCOM): Die Anzahl aller Kommentarzeilen, ausgenommen Dokumentationskommentare
- Number Of Declarations (NOD): Die Anzahl aller Deklarationen von Namensräumen, Klassen, Methoden und Variablen

Die Metriken repräsentieren dabei jeweils den möglichen Raum, in dem das betrachtete Code Smell auftreten kann. So macht es beispielsweise Sinn, die Anzahl gefundener Hybridstrukturen in Relation zu der gesamten Anzahl von Klassen zu setzen. Die Relationen wurden so gewählt, dass diese unabhängig von der Größe des untersuchten Projekts sind.

Die getroffene Auswahl der Kennzahlen soll Erkenntnisse über die Codequalität in den Bereichen „Namen“, „Methoden“, „Kommentare“, „Objekte und Datenstrukturen“ sowie „Fehlerbehandlung“ liefern. Um die Qualität von Projekten in diesen Bereichen beurteilen zu können, werden für jedes in dieser Arbeit untersuchte Projekt die genannten Kennzahlen[18] errechnet. Die Ergebnisse werden, wie bereits in Kapitel 4.2.5 beschrieben, kumuliert und in einer Matrix mit den errechneten Schwellenwerten „Low“, „Average“, „High“ und „Very High“ aufgeführt:

[18] Starke ausreißende Werte wurden durch den Mittelwert des Datensatzes ersetzt.

	Low	Average	High	Very High
Hybride/NOC	0,0049	0,0187	0,0325	0,0487
Zu lange Methoden/NOM	0,0160	0,0348	0,0537	0,0805
Zu lange Parameterlisten/NOM	0,0143	0,0411	0,0679	0,1019
null-Ref. als Rückgabew./NOM	0,0072	0,0157	0,0243	0,0364
Fehlercodes als Rückgabewert/NOM	-0,0002	0,0063	0,0128	0,0192
LOD Verstöße/NOI	0,0189	0,0452	0,0714	0,1071
nicht gekapselte Grenzbed./LOC	0,0000	0,0018	0,0036	0,0054
Auskomm. Code/NOCOM	0,0029	0,0250	0,0471	0,0706
Zahlenserien in Namen/NOD	0,0000	0,0016	0,0033	0,0049
Ein-Buchstaben-Namen/NOD	0,0028	0,0226	0,0425	0,0637

Tabelle 5.1: Schwellenwerte für die Häufigkeit von Code Smells in C#-Projekten

Oft variieren die errechneten Kennzahlen in den untersuchten Projekten stark. Dies rührt daher, dass die Verteilung eines bestimmten Smells über die betrachteten Projekte hinweg nicht gleichmäßig ist. So tritt der Smell „nicht gekapselte Grenzbedingungen“ in dem Projekt SimpleFramework[19] durchschnittlich einmal in 298 Zeilen Code auf. Im Kontrast dazu tritt der selbe Smell im Projekt Unity[20] durchschnittlich nur in jeder 24.875sten Zeile auf. Die Spannweite der relativen Häufigkeit mancher Smells ist zum Teil so hoch, dass die Standardabweichung der Messewerte größer oder gleich wie deren arithmetisches Mittel ist. Dies resultiert in den teilweise negativen Werten der „Low“-Schwelle.

Infolgedessen sind die ermittelten Schwellenwerte der Code Smells

- Fehlercodes als Rückgabewert
- nicht gekapselte Grenzbedingungen
- Zahlenserien in Namen

nicht oder nur leidlich als Vergleichswerte geeignet.

Die Ergebnisse können optimiert werden, indem für den Bezugswert Number Of Methods (NOM) der beiden Smells „Fehlercodes als Rückgabewert“ und „null-

[19] github.com/ngallagher/simpleframework

[20] github.com/unitycontainer/unity

Referenzen als Rückgabewert“ Methoden ausgeschlossen werden, die keinen Rückgabewert (void) besitzen. So kann die Streuung der errechneten Häufigkeiten reduziert werden. Ferner können so auch aussagekräftigere Ergebnisse für die Schwellenwerte erreicht werden.

Außerdem ist es fraglich, ob der gewählte Bezugswert LOC für nicht gekapselte Grenzbedingungen sinnvoll ist. Dieser Smell tritt häufig in Verbindung mit Arrays und Schleifen auf. Die Häufigkeit der Verwendung dieser Konstrukte kann in verschiedenen Projekten stark variieren und so auch die Ergebnisse verzerren.

Ansonsten können die restlichen ermittelten Schwellenwerte als Anhaltspunkte für die Bewertung von Quellcode herangezogen werden. Allerdings sind die implementieren Detektoren noch nicht ausgereift und komfortabel genug, um sie als Analysewerkzeuge zu veröffentlichen. Damit deren Ergebnisse von Entwicklern ernst genommen werden, müssen die Tools noch zuverlässiger und einfacher zu bedienen sein. Dies konnte mit der prototypischen Implementierung dieser Arbeit nicht erreicht werden. Allerdings wurde das vorhandene Potential der verwendeten Plattform Roslyn demonstriert.

Kapitel 6

> Let us change our traditional attitude to the construction of programs. Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.
>
> *Donald Knuth*

Zusammenfassung und Ausblick

Das Ziel der vorliegenden Arbeit war es, auf Basis der in „Clean Code“ beschriebenen Theorien, ein Programmpaket an Analysewerkzeugen zu entwickeln. Die einzelnen Tools sollen in der Lage sein, einige von Martin beschriebene Code Smells in Quelltext aufzuspüren. Die Implementierung der Tools erfolgte mit Hilfe der Roslyn-API, um den Funktionsumfang und die Möglichkeiten der Compiler-Plattform zu demonstrieren.

Bei den Recherchen dieser Arbeit ergab sich, dass Tools und Metriken geeignete Werkzeuge sind, um die Qualität von Quellcode bewerten und sogar steigern zu können. Werden diese fest in den Entwicklungsprozess integriert, können schon während der Implementierung negative Tendenzen und Trends erkannt und geeignete Gegenmaßnahmen ergriffen werden.

Keinesfalls darf sich ein Entwickler ausschließlich auf die verwendeten Werkzeuge verlassen – deren Ergebnisse sind stets zu reflektieren und mit gesundem Menschenverstand zu hinterfragen. Eine Falsch- oder Überinterpretation der Ergebnisse kann im Zweifel auch zu einer Verschlechterung der Codequalität führen.

Mit den prototypischen Implementierungen der Analysewerkzeuge wurde gezeigt, mit welch relativ geringem Aufwand leistungsstarke Tools auf Basis von Roslyn entwickelt werden können. Die entwickelten Werkzeuge sind in der Lage, 14 verschiedene Code Smells zu identifizieren. Es wurde festgestellt, dass nicht alle in

Clean Code beschriebenen Smells zuverlässig in einem automatisierten Vorgang aufgespürt werden können. Teilweise liegt es im Ermessen des Entwicklers, zu entscheiden, wie mit den Funden verfahren werden soll. Für eine Veröffentlichung der Analysesuite sind die darin enthaltenen Werkzeuge noch nicht präzise genug. Auch die Auswertung der Ergebnisse erfolgt derzeit noch manuell und ist deswegen für den Entwicklungsalltag nicht praktikabel genug.

Die Genauigkeit der Tools kann in weiterführenden Arbeiten an dem Projekt noch gesteigert werden, indem die gefundenen False Positives der jeweiligen Smells weiter reduziert werden. Auch der Komfort der Analysewerkzeuge bietet Potential für Verbesserungen. Die Smell Detektoren können in die Entwicklungsumgebung integriert werden: In Kombination mit Visual Studio bietet Roslyn die Möglichkeit, bereits während des Schreibens Quellcode zu analysieren und etwaige Funde für Entwickler visuell hervorzuheben. Darüber hinaus können Hinweise für deren Behebung oder gar automatische Refaktorisierungen angeboten werden.

Im Rahmen dieser Arbeit wurden 31, auf GitHub veröffentlichte, Open Source-Projekte auf Code Smells untersucht. Die Anzahl der gefundenen Smells wurde für jedes untersuchte Projekt in Relation zu verschiedenen Metriken gesetzt. Diese repräsentieren dabei den möglichen Raum, in dem ein Code Smell auftreten kann. Daraus wurde eine Bewertungsmatrix mit Schwellenwerten für die relative Häufigkeit des Auftretens der einzelnen Smells erstellt. Anhand der Vergleichswerte dieser Matrix können erste Einschätzungen hinsichtlich der Qualität eines zu untersuchenden Projekts getroffen werden.

Es wurde festgestellt, dass nicht jede der ermittelten relativen Häufigkeiten taugliche Vergleichswerte liefert. Das kann mehrere Gründe haben: Nicht alle Smells sind für die Aufstellung von Vergleichswerten geeignet, da verschiedene Aufgabenstellungen und Absichten der Entwickler dazu beitragen können, dass manche Smells begünstigt und andere wiederum gehindert werden. Auch die Bezugswerte können noch schärfer definiert werden, um bessere Ergebnisse zu erzielen.

Durch die Untersuchung weiterer Projekte und einer damit verbundenen Ausdehnung der Datengrundlage können die Grenzwerte stabilisiert und verbessert werden.

Mit einer Vergrößerung des Datenfundaments sinkt der Einfluss einzelner Ausreißer auf die ermittelten Schwellenwerte und die Aussagekraft der Bewertungsmatrix steigt.

Analysewerkzeuge können ebenfalls dazu beitragen, die propagierten Werte der Clean Code Developer Bewegung zu verbreiten. Die sofortige Bewertung von Code lässt Entwickler direkt auf potentielle Probleme und Fehler aufmerksam werden. Das wiederum hat direkten Einfluss auf die Korrektheit des fertigen Produkts und die Produktionseffizienz der Entwickler. Ihr traditionelles Verhalten wird insofern beeinflusst, dass sie bereits während der Entwicklung über ihre Vorgehensweise reflektieren.

Ein allgemeines, durch Analysesoftware gestütztes, Verständnis der Notwendigkeit von Clean Code und eine gemeinsame Basis in Form von Werten und Normen kann die Art und Weise verändern, mit der Programme entwickelt werden. Dies kann einen Beitrag dazu leisten, dass Code künftig nicht mehr nur geschrieben wird, um eine bestimmte Aufgabe zu erledigen. Quellcode wird nicht mehr für die Maschine, sondern vielmehr für seine Leser geschrieben. In letzter Konsequenz entsteht dadurch eine höhere Evolvierbarkeit des Codes und damit eine Qualitätssteigerung der entwickelten Software.

Literaturverzeichnis

[1] agile alliance. http://www.agilealliance.org/. [letzter Zugriff: 01. Sep. 2015].

[2] agiles manifest. http://www.agilemanifesto.org/iso/de/. [letzter Zugriff: 01. Sep. 2015].

[3] Stackoverflow Community. What is the single most influential book every programmer should read? http://stackoverflow.com/questions/1711/what-is-the-single-most-influential-book-every-programmer-should-read. [letzter Zugriff: 11. 11. 2015].

[4] kungfuwebmag. http://www.kungfuwebmag.de/033ce29c5e022d80d.html. [letzter Zugriff: 04. Sep. 2015].

[5] Wikipedia. https://de.wikipedia.org/wiki/D%C5%8Dj%C5%8D. [letzter Zugriff: 07. Sep. 2015].

[6] Anders Hejlsberg Don Box. Microsoft msdn. https://msdn.microsoft.com/en-us/library/bb308959.aspx. [letzter Zugriff: 10. Sep. 2015].

[7] Martin Fowler. Codesmell. http://martinfowler.com/bliki/CodeSmell.html. [letzter Zugriff: 23. 11. 2015].

[8] Martin Fowler. Special case. http://martinfowler.com/eaaCatalog/specialCase.html. [letzter Zugriff: 15. 12. 2015].

[9] Neal Gafter. Microsoft roslyn github repository. https://github.com/dotnet/roslyn/wiki/Roslyn [letzter Zugriff: 25. Sep. 2015].

[10] Wikibooks. https://de.wikibooks.org/wiki/Algorithmensammlung:_Zahlentheorie:_Euklidischer_Algorithmus#Gr.C3.B6.C3.9Fter_gemeinsamer_Teiler_.28ggT.29. [letzter Zugriff: 09. Sep. 2015].

[11] Herbert Haß. karatekata. http://www.karatekata.de/Karatekata_Artikel.htm. [letzter Zugriff: 04. Sep. 2015].

[12] Michael Hoennig. Ccd zyklus. http://michael.hoennig.de/download/CCD-Poster-A2.png. [letzter Zugriff: 07. Sep. 2015].

[13] Donald Belcham Kyle Baley. *Brownfield Application Development in .NET*. Manning, Greenwich, 2010.

[14] Prof. Dr. Richard Lackes. Content management system (cms). http://wirtschaftslexikon.gabler.de/Definition/content-management-system-cms.html. [letzter Zugriff: 24. 11. 2015].

[15] Peter Liggesmeyer. *Software-Qualität - Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag GmbH, Heidelberg, Berlin, 2002.

[16] Robert C. Martin. clean coder. http://www.cleancoder.com/. [letzter Zugriff: 01. Sep. 2015].

[17] Robert C. Martin. *Clean Code - Refactoring, Patterns, Testen und Techniken für sauberen Code*. mitp, Heidelberg, München, Landsberg, Frechen, Hamburg, 2009.

[18] Robert C. Martin. *The Clean Coder*. Prentice Hall, Upper Saddle River, New Jersey, 2011.

[19] Radu Marinescu Michele Lanza. *Object-Oriented Metrics in Practice*. Springer, Berlin, Heidelberg, 2006.

[20] Microsoft MSDN. Microsoft msdn csharp codekonventionen. https://msdn.microsoft.com/de-de/library/ff926074. [letzter Zugriff: 15. 10. 2015].

[21] Mircosoft MSDN. region (csharp-referenz). https://msdn.microsoft.com/de-de/library/9a1ybwek.aspx. [letzter Zugriff: 11. Jan. 2016].

[22] Lasitha Ishan Petthawadu. Configure jenkins with sonarqube for static code analysis and integration. https://lasithapetthawadu.wordpress.com/2014/05/03/configure-jenkins-with-sonarqube-for-static-code-analysis-and-integration/. [letzter Zugriff: 10. Sep. 2015].

[23] Stefan Lieser Ralf Westphal. clean code developer - die grade. http://clean-code-developer.de/die-grade/. [letzter Zugriff: 04. Sep. 2015].

[24] Stefan Lieser Ralf Westphal. clean code developer - die werte. http://clean-code-developer.de/das-wertesystem/. [letzter Zugriff: 04. Sep. 2015].

[25] Stefan Lieser Ralf Westphal. clean code developer school - coding dojo. http://ccd-school.de/coding-dojo/#cd9. [letzter Zugriff: 07. Sep. 2015].

[26] Stefan Lieser Ralf Westphal. ibm oti. http://www-03.ibm.com/software/ca/en/ottawalab/roots.html. [letzter Zugriff: 02. Sep. 2015].

[27] Linda H Rosenberg and Lawrence E Hyatt. Software quality metrics for object-oriented environments. *Crosstalk journal*, 10(4), 1997.

[28] Rainer Stropek. Die neue .net compiler platform. *windows.developer*, pages 16–19, 2015.

[29] Dave Thomas. Codekata. http://codekata.com/. [letzter Zugriff: 02. 11. 2015].

[30] Christiaan Verwijs. Agile book essentials #1: Clean code by robert c. martin. http://blog.agilistic.nl/agile-book-essentials-1-clean-code-by-robert-c-martin1/. [letzter Zugriff: 11. 11. 2015].

[31] Axel Wagner. Arbeitsweise eines compilers. http://www.saar.de/~awa/compiler_ueberblick.html. [letzter Zugriff: 19. 10. 2015].

[32] Ernest Wallmüller. *Software-Qualitätsmanagement in der Praxis*. Hanser, München, Wien, 2001.

[33] Ralf Westphal. Auf zur nächsten stufe. *dotnetpro*, pages 10–16, 2014.

Anhang A

Anhang

A.1 Untersuchte Projekte

Projekt	URL
akka.net	github.com/akkadotnet/akka.net
aspnetboilerplate	github.com/aspnetboilerplate/aspnetboilerplate
nancy	github.com/NancyFx/Nancy
CefSharp	github.com/cefsharp/CefSharp
choco	github.com/chocolatey/choco
nodejstools	github.com/Microsoft/nodejstools
PnP	github.com/OfficeDev/PnP
automatic-graph-layout	github.com/Microsoft/automatic-graph-layout
dapper-dot-net	github.com/StackExchange/dapper-dot-net
Destroy-Windows-10-Spying	github.com/Nummer/Destroy-Windows-10-Spying
DotNetty-dev	github.com/Azure/DotNetty
enode	github.com/tangxuehua/enode
corefx	github.com/dotnet/corefx
Git-Credential-Manager	github.com/bargainloaf/credentialmanager
Hangfire	github.com/HangfireIO/Hangfire
framework-development	github.com/accord-net/framework
Hearthstone-Deck-Tracker	github.com/Epix37/Hearthstone-Deck-Tracker
Nancy	github.com/NancyFx/Nancy
OptiKey	github.com/OptiKey/OptiKey
Piranha	github.com/PiranhaCMS/Piranha
shadowsocks-csharp	github.com/shadowsocks/shadowsocks-windows
ShareX	github.com/ShareX/ShareX
SimpleFramework_NGUI	github.com/jarjin/SimpleFramework_NGUI
ui-for-aspnet-mvc-examples	github.com/telerik/ui-for-aspnet-mvc-examples
Umbraco-CMS-dev-v7	github.com/umbraco/Umbraco-CMS
unity	github.com/unitycontainer/unity
WebEssentials2015	github.com/madskristensen/WebEssentials2015
orleans	github.com/dotnet/orleans
OrchardCMS	github.com/OrchardCMS/
zxing	github.com/zxing/zxing
Greenshot Image Editor	www.getgreenshot.org

Tabelle A.1: Auflistung der untersuchten Projekte

Das Projekt zxing wird von shadowsocks verwendet und ist in dessen Sourcecode integriert. Die Bewertungsergebnisse von xzing sind deswegen in den Ergebnissen von shadowsocks enthalten. Gleich verhält es sich bei den Projekten Greenshot Image Editor und ShareX.

A.2 Quellcode der Implementierung dieser Arbeit

Der Code des im Rahmen dieser Arbeit entstandenen Programme ist unter der folgenden GitHub-URL frei zugänglich:

https://github.com/birksimon/CodeAnalysis

Unter der selben URL ist auch die Tabelle mit der Auswertung der Analyseergebnisse zu finden.