Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Sebastian Ptasik

Einsatzmöglichkeiten statischer Codeanalyse zur Unterstützung des Clean Code Development

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Sebastian Ptasik

Einsatzmöglichkeiten statischer Codeanalyse zur Unterstützung des Clean Code Development

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Olaf Zukunft

Eingereicht am: 2. Dezember 2011

Sebastian Ptasik

Thema der Arbeit

Einsatzmöglichkeiten statischer Codeanalyse zur Unterstützung des Clean Code Development

Stichworte

Clean Code Development, Statische Codeanalyse, Metrik, Code Smells, Innere Softwarequalität

Kurzzusammenfassung

Diese Bachelorarbeit befasst sich mit der Thematik des Clean Code Development. Es wird untersucht, in welchem Maße ausgewählte Code Smells und Paradigmen Einfluss auf die innere Qualität einer Software nehmen. Diese Untersuchung ist Grundlage für die Entwicklung von Codeanalyseregeln, die unter Verwendung des Toolkits FxCop umgesetzt werden.

Die implementierten Regeln dienen zur automatisierten Überprüfung der Interna einer beliebigen .NET Software. Design- und Architekturschwächen treten auf diese Weise schneller zu Tage und können so zeitnah behoben werden.

Sebastian Ptasik

Title of the paper

Possible applications of static code-analysis in support of Clean Code Development

Keywords

Clean Code Development, Static Code Analysis, Metric, Code Smells, Inner Software Quality

Abstract

This bachelor thesis deals with the topic of Clean Code Development. It analyses in which degree selected Code Smells and paradigms exert influence on the inner quality of a software. This analysis builds the base for the development of static code analysis rules, which will be realized with the FxCop toolkit.

These drafted rules are used for automated checks on the internal matter of a various .NET software. In this way design and architecture weaknesses can be detected faster and as a result they can be fixed promptly.

Inhaltsverzeichnis

1. Einleitung **1**
1.1. Motivation 1
1.2. Zielsetzung 2
1.3. Gliederung 2

2. Grundlagen **3**
2.1. Innere Softwarequalität 3
2.2. Qualitätsmerkmale 4
2.2.1. Testbarkeit 4
2.2.2. Wartbarkeit 4
2.3. Clean Code Development 5
2.4. Code Smells 6
2.4.1. Duplizierter Sourcecode 6
2.4.2. Übergroße Klassen 6
2.4.3. Übergroße Methoden 7
2.4.4. Viele Methodenparameter 7
2.4.5. Ungenutzter Sourcecode 8
2.4.6. Lokale Konstanten 8
2.4.7. Unbehandelte Ausnahmen 9
2.4.8. Veränderbare Datentypen 10
2.4.9. Vermischung von Konfiguration und Nutzung 11
2.4.10. Fehlende Codegestaltungsrichtlinien 14
2.5. Paradigmen 15
2.5.1. Law of Demeter 15
2.5.2. Program to an Interface, not to an Implementation 16
2.5.3. Interface Segregation Principle 16
2.5.4. Design by Contract 17
2.5.5. Liskov Substitution Principle 18
2.6. Statische Codeanalyse 21
2.6.1. Metriken 21

3. Analyse **22**
3.1. NimblePros Nitriq 1.0 23
3.1.1. Überblick 23
3.1.2. Bewertung 27

3.2. Microsoft FxCop 10.0 28
3.2.1. Überblick 28
3.2.2. Vordefinierte Regeln 32
3.2.3. Bewertung 33
3.3. StyleCop 4.6 34
3.3.1. Überblick 34
3.3.2. Bewertung 34
3.4. Microsoft Code Contracts 1.4 35
3.4.1. Überblick 35
3.4.2. Bewertung 37
3.5. Zusammenfassung 38

4. Realisierung **39**
4.1. Erkennung von übergroßen Klassen 40
4.2. Erkennung von übergroßen Methoden 40
4.3. Begrenzung der Anzahl von Methodenparametern 42
4.4. Vermeidung von ungenutztem Quellcode 42
4.5. Erkennung von lokalen Konstanten 46
4.6. Vermeidung von unbehandelten Ausnahmen 46
4.7. Erkennung veränderbarer Datentypen 48
4.8. Vermischung von Konfiguration und Nutzung 51
4.9. Überprüfung von Codegestaltungsrichlinien 54
4.10. Einhaltung des Gesetzes von Demeter 54
4.11. Verwendung von Schnittstellen anstelle konkreter Klassen 55
4.12. Trennung von Schnittstellen anhand ihrer Zuständigkeiten 56
4.13. Reflexion der Realisierungsphase 57

5. Auswertung **58**
5.1. Beschreibung des Prüflings 58
5.2. Testläufe 60
5.2.1. Erste Iteration 60
5.2.2. Zweite Iteration 63
5.3. Bewertung der Ergebnisse 67

6. Schluss **68**
6.1. Fazit 68
6.2. Ausblick 68

A. Inhalt der CD **69**

B. Glossar **70**

Abbildungsverzeichnis

3.1. Überblick Nitriq 23
3.2. CQL Ansicht 25
3.3. CQL Autovervollständigung 25
3.4. Ergebnisanzeige 26
3.5. Ergebnispräsentation in der Treemap 26
3.6. Überblick FxCop 28
3.7. FxCop Code-Modell 29

4.1. Ablaufdiagramm zur Erkennung übergroßer Methoden 41
4.2. Ablaufdiagramm zur Erkennung von ungenutzten Methoden 45
4.3. Gerichteter Beziehungsgraph 50
4.4. Gerichteter Beziehungsgraph mit Zyklen 50
4.5. Zustandsdiagramm zur Erkennung von vermischten Verantwortlichkeiten . . 53

5.1. Systemübersicht 58
5.2. Ergebnisse der ersten Iteration 60
5.3. Ergebnisse der zweiten Iteration 63

Listings

2.1. Beispiel: Spaghetticode . . . 5
2.2. Beispiel: Zu viele Methodenparameter . . . 7
2.3. Beispiel: Toter Sourcecode . . . 8
2.4. Beispiel: Magic Numbers . . . 9
2.5. Beispiel: Unsaubere Ausnahmebehandlung . . . 10
2.6. Beispiel: Seiteneffekt . . . 11
2.7. Beispiel: Vermischung von Konfiguration und Nutzung . . . 12
2.8. Beispiel: Dependency Injection per Konstruktor . . . 13
2.9. Beispiel: Verstoß gegen das Law of Demeter . . . 15
2.10. Beispiel: Vermischte Funktionalität . . . 17
2.11. Beispiel: Aufgetrennte Funktionalität . . . 17
2.12. Beispiel: Verletzung des LSP . . . 19
2.13. Beispiel: Ergebnis aus Verletzung des LSP . . . 20
2.14. Beispiel: Erweiterungen durch Verletzung des LSP . . . 20

3.1. FxCop: XML-Beschreibung des Regelwerkes . . . 30
3.2. FxCop: Implementierung einer beispielhaften Regel . . . 31
3.3. Code Contracts: Definition Invariante . . . 36
3.4. Code Contracts: Vererbungshierarchie . . . 37

4.1. InternalsVisibleTo . . . 43
4.2. Behandlungsroutine für unbehandelte UI Exceptions . . . 47
4.3. Behandlungsroutine für unbehandelte Thread Exceptions . . . 47
4.4. Eigenschaft ohne direkte Variablenbindung . . . 48
4.5. Datentypklasse . . . 49

5.1. Attribut zum Überspringen der Überprüfung . . . 61
5.2. Verletzung des Law Of Demeter . . . 62
5.3. Zugriff auf Elemente des .NET Frameworks . . . 62
5.4. Regelwidrige Konfiguration im Konstruktor . . . 65

1. Einleitung

1.1. Motivation

Der Begriff Qualität ist in der Softwareentwicklung allgegenwertig und ist heute wichtiger denn je. Vom Betrachtungsstandpunkt des Kunden stehen dabei Qualitätsmerkmale wie Performanz, Bedienbarkeit und Robustheit an vorderster Stelle der nicht-funktionalen Anforderungen. Diesen Merkmalen wird während der Design- und Realisierungsphase üblicherweise besonders viel Aufmerksamkeit gewidmet, da sie vom Kunden direkt wahrgenommen werden.

Neben den genannten Qualitätsmerkmalen gibt es weitere Merkmale, die auf den ersten Blick aus Sicht des Kunden keinen so hohen Stellenwert einnehmen. Damit ist neben der Testbarkeit auch die Erweiterbarkeit und Modularität gemeint. Diese Qualitätsmerkmale sind aus der Sicht des Softwareentwicklers extrem wichtig. So hat beispielsweise die Testbarkeit eines Systems direkten Einfluss auf weitere, für den Kunden essentielle Qualitätsmerkmale. Ist ein System nicht oder nur sehr eingeschränkt testbar, gestaltet sich die Integration von automatisierten Testfällen äußerst schwierig worunter die Robustheit des Systems leidet.

Die Wahrung der inneren Softwarequalität sollte demnach ebenfalls während der Konzeption und Implementierung einer Software beachtet und überprüft werden. Aufgrund verschiedenster Faktoren ist dies aber in den wenigsten Projekten der Fall. Zeitnot stellt dabei wohl einen der größten Einflussfaktoren dar. Änderungen und Erweiterungen werden oftmals unter Zeitdruck unsauber implementiert.

Um der stetig abnehmenden Codequalität entgegenzuwirken, gibt es seit geraumer Zeit den Ansatz der statischen Codeanalyse mit Hilfe von Metriken. Diese Analysemethode dient zur Vermessung einer Software anhand von vordefinierten Messgrößen, wie beispielsweise der **zyklomatischen Komplexität**. Über eine grafische Bedienoberfläche ist es dann möglich, die Analyseergebnisse auszuwerten. So können viele Schwachstellen und Unsauberkeiten im Quellcode lokalisiert und behoben werden. Tools wie NDepend und Nitriq bieten genau diese Funktionalität für die Sprache C#. Neben den beiden aufgeführten kommerziellen Produkten gibt es noch eine Reihe weiterer, frei verfügbarer Bibliotheken zur statischen Codeanalyse.

1.2. Zielsetzung

Hauptziel der Ausarbeitung ist die Untersuchung einer ausgewählten Menge an Paradigmen des Clean Code Development in Hinsicht auf die Übertragbarkeit in die statische Codeanalyse. Es sollen statische Analyseregeln verfasst werden, mit deren Hilfe eine in C# verfasste Software überprüft werden kann. Als Grundlage für diese Überprüfung dient die Beschreibung der Merkmale, die die innere Softwarequalität ausmachen.

1.3. Gliederung

Die Ausarbeitung bringt dem Leser zunächst das nötige grundlegende Wissen über innere Softwarequalität und das Clean Code Delevelopment nahe. Es wird beschrieben, welche der allgemein anerkannten Qualitätsmerkmale überhaupt von der inneren Qualität der Software betroffen sind und wie diese Qualitätsmerkmale zur Gesamtqualität der Software beitragen. Danach folgt die theoretische Einführung in die Thematik des Clean Code Development. Hier wird nach einer kurzen Erklärung des Begriffs der Bezug zu den im Voraus beschriebenen Qualitätsmerkmalen hergestellt. Ausgewählte Paradigmen werden vorgestellt und im Bezug auf ihren Einfluss auf die innere Softwarequalität bewertet. Anzumerken ist, dass es sich bei den in dieser Ausarbeitung aufgeführten Paradigmen nur um eine Teilmenge der in Martin (2009) vorgestellten Richtlinien handelt.

Im Anschluss an die theoretische Einführung folgt ein Überblick über verschiedene Analysetools, die am Markt verfügbar sind. Dabei wird im Schwerpunkt auf die gebotenen Fähigkeiten zum Verfassen eigener Analyseregeln sowie die Umsetzbarkeit der ausgewählten Paradigmen des Clean Code Development eingegangen. Weitere toolspezifische Eigenschaften werden nicht oder nur oberflächlich vorgestellt.

Auf die Evaluierung der verfügbaren Analysetools folgt die Übertragung der vorgestellten Paradigmen in eigenformulierte Analyseregeln. Dabei wird vorwiegend darauf eingegangen, wie Verstöße gegen die jeweiligen Paradigmen des Clean Code Development erkannt werden und welche Bewertungskriterien hierfür herangezogen werden.

Die umgesetzten Analyseregeln werden im darauffolgenden Kapitel unter Verwendung eines Prüflings validiert. Es wird nachgewiesen, dass die Analyseregeln Verstöße erkennen können. Zudem wird geklärt, ob die Analyseregeln uneingeschränkt oder nur mit Einschränkungen verwendbar sind. Außerdem wird ermittelt, ob und in welchem Umfang es zu fehlerhaften Verstoßmeldungen kommt. Dies ist erforderlich, um abschließend bewerten zu können, inwieweit die Nutzung der Analyseregeln die tägiche Arbeit eines Softwareentwicklers unterstützt.

2. Grundlagen

2.1. Innere Softwarequalität

> Qualität ist die Gesamtheit von Merkmalen einer Einheit bezüglich ihrer Eignung, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen.[1]

Qualität ist seit jeher ein Begiff, der je nach Betrachtungsweise unterschiedlich ausgelegt werden kann. Dies ist auch für den Bereich der Softwarequalität nicht anders. Auftraggeber haben andere Qualitätsansprüche an eine Software als die Entwickler, die dieses realisieren. Der Fokus eines Softwareprojekts liegt dabei grundsätzlich auf den Qualitätsaspekten, die dem Auftraggeber besonders wichtig sind. Die Qualitätsmerkmale einer Software, die vom Anwender wahrgenommen werden, bezeichnet man auch als äußere Softwarequalität.

Im Gegensatz dazu werden unter dem Begriff der inneren Softwarequalität, Aspekte wie die Wartbarkeit, Erweiterbarkeit und Testbarkeit verstanden. Auch die Verständlichkeit und Lesbarkeit auf Ebene des Quellcodes zählen zur inneren Softwarequalität hinzu. Diese Qualitätsmerkmale werden in der Regel nicht direkt vom Anwender wahrgenommen, sondern sind nur für den Entwickler ersichtlich.

Einfluss der inneren Softwarequalität

Die Frage ist nun, warum der inneren Softwarequalität besondere Aufmerksamkeit gewidmet werden sollte, wo der Anwender diese nicht direkt wahrnimmt. Die Frage ist leicht zu beantworten. Herrscht eine schlechte innere Softwarequalität vor, steigen die Kosten für Erweiterungen und Fehlerbehebungen. Je länger die innere Qualität einer Software vernachlässigt wird, umso mehr sinkt auch die Produktivität des Entwicklerteams [2]. Um die Produktivität wieder zu steigern, werden dem Projekt weitere Entwickler hinzugefügt. Das Problem hierbei ist, dass die neuen Entwickler häufig kein Detailwissen über das Design der Software besitzen wodurch die innere Qualität der Software weiter abnimmt.

[1] Laut ISO 8402.
[2] vgl. Martin (2009)

2.2. Qualitätsmerkmale

2.2.1. Testbarkeit

> Das Ausmaß, in dem ein System das Erstellen von Testbedingungen sowie die Durchführung von Tests erleichtert.[3]

Die Testbarkeit eines Systems hängt immer von verschiedenen Einflussfaktoren ab. So werden Qualitätsanforderungen und -ziele in Zusammenarbeit mit dem Kunden definiert.[4] Die innere Softwarequalität trägt entscheidend zur Verbesserung der Testbarkeit bei. Speziell im Bezug auf automatisierte Unit-Tests hat der innere Aufbau einer Software maßgeblichen Einfluss darauf, wie die Testfälle gestaltet werden können. Sind die Bestandteile einer Software extrem stark aneinander gekoppelt, ist ein isolierter Test einzelner Elemente nahezu unmöglich. Diese Tatsache steigert den Arbeitsaufwand für die Erstellung von Unit-Tests deutlich. Zudem wird die Lokalisierung auftretender Fehler erschwert, je mehr Elemente implizit mitgetestet werden. Als Konsequenz daraus wird oftmals bewusst auf diese Art des Testens verzichtet. Dies hat zur Folge, dass Fehler erst sehr spät entdeckt werden und, je nach Schweregrad, massiven Einfluss auf den weiteren Entwicklungsprozess einer Software nehmen können.

2.2.2. Wartbarkeit

> Gute Erweiterbarkeit und Wartbarkeit eines Softwaresystems bedeutet, Änderungen und Erweiterungen einfach und kostengünstig durchführen zu können.[5]

Wartbarkeit setzt aus Sicht des Softwareentwicklers mehrere Dinge voraus. Neben einer guten technischen Dokumentation spielt die innere Softwarequalität auch für diesen Qualitätsaspekt eine entscheidende Rolle. Verhält sich ein Softwaremodul anders als es zu erwarten ist, steigt der Wartungsaufwand. Die Abläufe hinter dem Modul müssen nun umständlich analysiert werden, um sie zu verstehen.

Auch die Lesbarkeit des Sourcecodes ist von enormer Bedeutung für die Wartbarkeit. Das typsiche Negativbeispiel für diesen Aspekt ist der sogenannte Spaghetticode. Hauptmerkmal hiervon sind Sprunganweisungen, die unter bestimmten Bedingungen ausgeführt werden und das Programm an einer vollkommen anderen Stelle im Sourcecode fortsetzen.

[3] Laut IEEE 610.12-1990
[4] vgl. Schneider (2007)
[5] Jürgen Dunkel (2003)

Listing 2.1: **Beispiel: Spaghetticode**

```
int value = 0;
SWITCH:
switch (value) {
    case 0: goto ADD1;
    case 1: goto MULTIPLY3;
    default: goto SUBTRACT1;
}

MULTIPLY3: value *= 3; goto SWITCH;
ADD1: value += 1; goto SWITCH;
SUBTRACT1: value -= 1; Console.WriteLine(value);
```

Das aufgeführte Beispiel ist dem Artikel Spaghetticode auf Wikipedia entnommen. Es verdeutlicht, wie die Lesbarkeit des Sourcecodes durch die verschiedenen Sprungbefehle leidet.

2.3. Clean Code Development

Die Initiative Clean Code Development[6] hat sich zum Ziel gesetzt, Softwareentwicklern Hilfestellung für die Steigerung der inneren Softwarequalität zu geben. Grundlage für das Clean Code Development sind allgemein anerkannte Programmierrichtlinien, die die Qualität des Sourcecodes steigern. Das Clean Code Development befasst sich dabei nicht mit den funktionalen Anforderungen einer Software oder mit der Programmiersprache, in der diese verfasst ist. Bei den propagierten Kodierrichtlinien handelt es sich vielmehr um allgemeine Hinweise und Pattern, die vom Programmierer auf die jeweilige Software- und Programmierumgebung adaptiert werden müssen.

Dieser Umstand stellt die eigentliche Herausforderung des Clean Code Development dar. „Guter“ Programmierstil wird in vielen Fachbüchern über simple Beispiele veranschaulicht. Ein tieferes Verständnis der diversen Richtlinien muss sich der Entwickler selbstständig im jeweiligen Anwendungsumfeld erarbeiten und deren Einhaltung kontinuierlich überprüfen.

[6] u.A. www.clean-code-developer.de

2.4. Code Smells

Der Begriff Code Smells[7] wird von vielen Autoren als Synonym für Sourcecodefragmente verwendet, die einen schlechten Programmierstil wiederspiegeln. Viele dieser sogenannten Smells spiegeln genau die Art von Sourcecodefragmenten wieder, welche durch die Richtlinien des Clean Code Development vermieden werden sollen. Im Folgenden werden auszugsweise wichtige Code Smells vorgestellt.

2.4.1. Duplizierter Sourcecode

Laut Fowler (1999) und Martin (2009) stellt das mehrfache Vorkommen des gleichen oder gleichartigen Sourcecodes in einer Software den schlechtmöglichsten Programmierstil dar. Duplizierter Sourcecode wirkt gleich zwei sehr wichtigen Qualitätsmerkmalen entgegen:

- Testbarkeit
- Wartbarkeit

Die Wartbarkeit leidet, da Änderungen, die nachträglich in die Software eingepflegt werden sollen, üblicherweise in **alle** duplizierten Sourcecodefragmente integriert werden müssen.[8] Die Wahrscheinlichkeit, eine oder sogar mehrere dieser Stellen zu vergessen, ist hoch.

Zudem leitet auch die Testbarkeit, denn jedes Duplikat eines Codefragments muss auch separat getestet werden. Genau wie bei der o.g. Wartbarkeit ist auch hier die Wahrscheinlichkeit hoch, dass eine oder mehrere duplizierte Stellen nicht von Testfällen abgedeckt werden.

2.4.2. Übergroße Klassen

Als übergroß werden Klassen bezeichnet, die viele Verantwortlichkeiten besitzen. Zudem hat auch die Anzahl von Methoden und Feldern Einfluss auf diesen Faktor. Aufgrund der vielen Verantwortlichkeiten, die solche Klassen im Regelfall inne haben, verschlechtert sich die Lesbarkeit. Je größer die Klasse ist, um schwerer gestaltet es sich für den Entwickler, die Zusammenhänge zu verstehen und Fehlerbehebungen oder Erweiterungen vorzunehmen.

[7] Auch Bad Smells.
[8] vgl. Jürgen Dunkel (2003)

2.4.3. Übergroße Methoden

Ebenso wie Klassen sollten auch Methoden so klein wie möglich gehalten werden. Lange Methoden sind ein Hinweis darauf, dass eine Methode für mehr als eine Aufgabe zuständig ist. Fowler (1999) nennt das Problem beim Namen, weist aber nicht direkt darauf hin, dass Methoden nur für einen Sachverhalt zuständig sein sollten. Martin (2009) dagegen sieht Methoden, die mehr als eine Funktion ausüben bereits als Code Smell an.

2.4.4. Viele Methodenparameter

Besitzt eine Methode viele Parameter, erhöht sich nicht nur der Schwierigkeitsgrad hinsichtlich der Lesbarkeit und Verständlichkeit, auch die Testbarkeit leidet. Durch die Vielzahl an möglichen Eingabewerten erhöht sich die Anzahl der erforderlichen Testfälle deutlich. Als Beispiel für eine schlecht konzipierte Methode mit vielen Parametern wird die Methode `CreateWindow` aus der Windows API genutzt.

Listing 2.2: **Beispiel: Zu viele Methodenparameter**

```
HWND WINAPI CreateWindow(
  __in_opt  LPCTSTR lpClassName,
  __in_opt  LPCTSTR lpWindowName,
  __in      DWORD dwStyle,
  __in      int x,
  __in      int y,
  __in      int nWidth,
  __in      int nHeight,
  __in_opt  HWND hWndParent,
  __in_opt  HMENU hMenu,
  __in_opt  HINSTANCE hInstance,
  __in_opt  LPVOID lpParam
);
```

In Martin (2009) wird vom Autor ausdrücklich empfohlen, Methoden mit so wenig Parametern, wie möglich zu konzipieren. Parameterlose Methoden sind laut Martin (2009) ideal. Es wird bereits von der Verwendung von drei Parametern ausdrücklich abgeraten.

2.4.5. Ungenutzter Sourcecode

Hierunter ist Sourcecode zu verstehen, der innerhalb der Software niemals aufgerufen wird. Nicht genutzte Codefragmente erhöhen unnötigerweise die Komplexität der Software und stellen Entwickler oftmals vor die Frage, ob nicht genutzte Methoden entfernt werden können. Diese Unsicherheit wird zusätzlich verstärkt, wenn die anscheinend nicht genutzte Methode an einer beliebigen Stelle im Sourcecode aufgerufen wird, dieser Aufruf aber auskommentiert ist.

Listing 2.3: **Beispiel: Toter Sourcecode**

```
public void DoSomething() {
    ...
    //this.DoTheMagic();
    ...
}

protected void DoTheMagic() {
    ...
}
```

Je länger solche nicht genutzten Codefragmente als Bestandteil des Systems verbleiben, umso mehr tragen sie zur Komplexität bei. Häufig werden Fehlerbehebungen nicht vom ursprünglichen Verfasser des Sourcecodes durchgeführt. Stößt nun ein Entwickler bei der Fehlersuche auf ein Codefragment, dass dem o.g. Beispiel ähnelt, so muss sich der Entwickler fragen, ob der auskommentierte Code **absichtlich** auskommentiert wurde. Der Aufwand für die Fehlerbehebung steigt durch solche Codefragmente unnötigerweise an.

2.4.6. Lokale Konstanten

Mit diesem Begriff werden konstante Werte innerhalb von Methoden bezeichnet[9]. Auf den ersten Blick scheinen sich daraus keine Probleme zu ergeben. Bei genauerer Betrachtung wird aber schnell klar, dass eine solche Verteilung konstanter Werte massive Nachteile mit sich bringt.

Oftmals werden Konstanten innerhalb einer Klasse mehrfach verwendet. Muss der Wert der Konstante aufgrund neuer Anforderungen geändert werden, bezieht sich diese Änderung

[9] Ugs. Magic Numbers

immer auf mehrere Stellen im Sourcecode. Wie schon bei 2.4.1 beschrieben, ist die Wahrscheinlichkeit hoch, dass eine oder mehrere Stellen nicht in die Änderung miteinbezogen werden.

Der nächste Nachteil bezieht sich weniger auf die Funktionalität der Software, als auf die Lesbarkeit des Sourcecodes. Konstante Werte, die innerhalb einer Methode verwendet werden, sind an keine Variable gebunden. Dem Leser erschließt sich die Bedeutung der Konstante also nicht aus dem Namen der ihr zugeordneten Variable.

Listing 2.4: **Beispiel: Magic Numbers**

```
public double CalculateArea(double r) {
    return r * r * 3.14d;
}
```

Das aufgeführte Beispiel verdeutlicht anschaulich, dass der Leser erst durch die Benennung der Methode Rückschlüsse auf die Bedeutung der Konstante **3.14** ziehen kann.

2.4.7. Unbehandelte Ausnahmen

Bei Ausnahmen handelt es sich um einen Mechanismus zur Fehlerberichterstattung innerhalb des Programmflusses. Ausnahmen können an beliebigen Stellen im Quellcode mit Hilfe der `throw` Anweisung ausgelöst werden, um jedweden Fehler zu melden und eine entsprechende Behandlung der aufgetretenen Ausnahme zu veranlassen. Die Behandlung von Ausnahmen findet üblicherweise innerhalb eines sogenannten `try-catch` Blocks statt. Innerhalb des `try` Blocks werden die Anweisungen aufgelistet, die Ausnahmen auslösen können. Der `catch` Block beinhaltet die Behandlungsroutine für möglicherweise auftretende Ausnahmen.

Da C# nur sogenannte **Unchecked Exceptions** unterstützt, ist eine Behandlung auftretender Ausnahmen nicht zwangsläufig gegeben. Ungeprüfte Ausnahmen müssen, im Gegensatz zu den aus Java bekannten geprüften Ausnahmen, nicht vom Aufrufer behandelt werden. Es obligt also dem Entwickler sicherzustellen, dass möglicherweise auftretende Ausnahmen behandelt werden.

Werden Ausnahmen nicht behandelt, führt dies zwangsläufig zu einem Absturz der Applikation ohne weitere fehlerbezogene Hinweise. Derartige Programmabstürze führen oft dazu, dass nicht gespeicherte Arbeiten verloren gehen oder das System in einem inkonsistenten Zustand hinterlassen wird.

Unsaubere Ausnahmebehandlung

Um die oben genannte Fehlerquelle zu umgehen, bedienen sich viele Entwickler der Vererbungshierarchie in .NET. Alle im .NET Framework vorhandenen Ausnahmen erben von der Basisklasse `Exception`. Dies gilt im Übrigen auch für Ausnahmen, die vom Entwickler erstellt oder zusammen mit einer Bibliothek geliefert wurden. Aufgrund der Vererbungshierarchie bietet sich für den Entwickler die Möglichkeit, innerhalb eines `try-catch` Blocks alle Arten von auftretenden Ausnahmen zu behandeln.

Listing 2.5: **Beispiel: Unsaubere Ausnahmebehandlung**

```
try {
    //...
}
catch (Exception e) {
    Console.WriteLine(e.Message);
}
```

Solch ein Konstrukt sollte äußerst kritisch betrachtet werden. In Anbetracht der Programmstabilität mag diese Art der Ausnahmebehandlung zwar durchaus vorteilhaft erscheinen, denn ein Programmabsturz durch eine nicht behandelte Ausnahme wird vermieden. Allerdings werden **alle** auftretenden Ausnahmen identisch behandelt, was die Wahrscheinlichkeit von Folgefehlern erhöht. Zudem kann aufgrund der Verallgemeinerung der Ausnahme keine Unterscheidung zwischen technisch bedingter Fehler und Fehlern, die durch eine falsche Benutzereingabe aufgetreten sind, unterschieden werden.

2.4.8. Veränderbare Datentypen

Die Implementierung neuer Datentypen ist in vielen Anwendungen erforderlich und auch durchaus wünschenswert. Sie dienen zur Repräsentation fachspezifischer Werte und werden zur anwendungsinternen Kommunikation genutzt. Für die Realisierung solcher Datentypen geben verschiedene Autoren nahezu die selben Richtlinien vor. Datentypen sollten laut Siedersleben (2004) immer so implementiert werden, dass ihre internen Attribute nach deren Initialisierung über den Konstruktor nicht mehr geändert werden können[10]. Wird diese Vorgabe nicht befolgt, ergeben sich einige Nachteile. Nutzer solcher Datentypen können die internen Werte nach Belieben anpassen und so (ungewollt) Schaden im System vornehmen. Durch die gegebene Veränderbarkeit des Objektes lassen sich beispielsweise die Mehrfachausführung von

[10] Sogenannte immutable-objects.

Schreiboperationen auf dem Objekt nicht verhindern. Zudem kann nicht kontrolliert werden, welche Softwareelemente überhaupt befugt sind, Änderungen am Objekt vorzunehmen.

Desweiteren wird das Vorkommen von Seiteneffekten bei der Realisierung von Operationen gefördert. Verdeutlicht wird dies anhand des folgenden Beispiels[11]:

Listing 2.6: **Beispiel: Seiteneffekt**

```
public class Point2D {
    public double X { get; private set; }
    public double Y { get; private set; }

    public void Add(Size2D other) {
        X += other.Height;
        Y += other.Width;
    }
}
```

Das Beispiel verdeutlicht, welchen Schaden eine seiteneffektbehaftete Programmierung bei der Realisierung von Datentypen haben kann. Obwohl die Attribute der Klasse `Point2D` nicht explizit außerhalb der Klasse geändert werden können, lassen sich die internen Werte durch einen Aufufe der Methode `Add()` manipulieren. Die implizite Manipulation der internen Werte ist deutlich schwerwiegender, als die zuvor genannten öffenlichen Methoden zur direkten Manipulation von Attributen. Durch die seiteneffektbefaftete Implementierung der Methode `Add()` ist mit einer deutlich höheren Fehlerquote innerhalb der Software zu rechnen, da die implementierte Funktionalität der Methode dem intuitiven Verständnis des Entwicklers entgegensteht.

Erschwerend kommt zu den genannten Nachteilen hinzu, dass es sich bei einem solchen Datentyp um einen Referenztyp handelt. Daraus ergibt sich, dass sich Änderungen an dem Objekt auf **alle** weiteren Objekte auswirken, die eine Referenz darauf speichern.

2.4.9. Vermischung von Konfiguration und Nutzung

Die klare Trennung von Zuständigkeiten innerhalb einer Software ist ein äußerst wichtiges Paradigma des Clean Code Development. Durch die Aufteilung der Zuständigkeiten wird der Grad an Kohäsion[12] deutlich erhöht, da jede Klasse nur einen wohldefinierten Aufgabenbereich

[11] Quelle: http://codebetter.com
[12] Maß für den Verwandtschaftsgrad zwischen den Bestandteilen einer Klasse.

abdeckt. Zusätzlich wird die Komplexität nicht mehr von einigen wenigen Klassen gebündelt, sondern logisch aufgeteilt.

Eine spezielle Ausprägung dieses Paradigmas ist die Trennung von Nutzung und Konfiguration eines Systems. In der Regel benötigen die einzelnen Komponenten und Klassen eines Systems Zugriff auf andere Systembestandteile, damit sie ihre Funktionalität bereitstellen können. Die Problematik, die sich aus dieser Tatsache ergibt, bezieht sich unmittelbar auf die Testbarkeit eines Systems. Ist eine Einheit sowohl für seine Konfiguration, als auch für die Nutzung zuständig, steigert dies den Aufwand zur Erstellung automatisierter Testfälle.

Modul- und Komponententests beziehen sich idealerweise nur auf genau ein Modul oder eine Komponente. Bedingt durch die Vermischung von Konfiguration und Nutzung ist der Verfasser der Testfälle aber gezwungen, auch die vom jeweiligen Modul bzw. von der jeweiligen Komponente referenzierten Einheiten zu nutzen.

Listing 2.7: **Beispiel: Vermischung von Konfiguration und Nutzung**

```
public class OrderManagement {
    private readonly ICustomerManagement mCustomerManagement;
    private readonly IPersistence mPersistence;

    public OrderManagement() {
        mCustomerManagement = new CustomerManagement();
        mPersistence = new Persistence();
    }

    public IList<Order> GetOrders() {
        // ...
    }
}
```

Dieses Beispiel verdeutlicht, wie eine Vermischung von Konfiguration und Nutzung erzeugt wird. Der Konstruktor der Klasse `OrderManagement` übernimmt die Konfiguration und instanziiert Objekte anderer Klassen. Soll nun ein Test für die Methode `GetOrders()` erstellt werden, hat der Verfasser des Tests keine Möglichkeit, die intern konfigurierten Objekte zu ersetzen und ist somit gezwungen, sowohl die Klasse `CustomerManagement`, als auch die Persistenzverwaltung zu benutzen. Dies wiederum kann zur Folge haben, dass der Test nur mit einer bestimmten Datenbankkonfiguration durchführbar ist. Der Test kann also nicht isoliert von anderen Komponenten und Modulen durchgeführt werden. Neben der Fehleranfälligkeit des

Tests durch Verwendung externer Elemente erhöht sich auch der Aufwand, der zur Durchführung des Tests betrieben werden muss. Um den Test durchführen zu können, müssen vorab Konfigurationsarbeiten erfolgen, die die Grundlage für den Test bilden.

Durch den Aufruf der Konstruktoren `CustomerManagement` und `Persistence` entsteht eine unnötig hohe Kopplung. Die Klasse `OrderManagement` ist durch die direkte Verwendung der anderen beiden Klassen stark an diese gekoppelt. Wird die Signatur eines der Konstruktoren geändert, hat diese Änderung unmittelbare Auswirkungen auf die Konfiguration der Klasse `OrderManagement` und würde eine Anpassung an dieser und an allen weiteren Stellen, an denen die geänderte Klasse instanziiert wird, erforderlich machen.

Zur Auflösung dieser Problematiken empfiehlt Martin (2009), das **Dependency Injection** Paradigma anzuwenden. Dieses Paradigma besagt, dass Abhängigkeiten, die ein Element einer Software besitzt, von einer externen Instanz injeziert werden. Die Injektion der Abhängigkeiten kann auf unterschiedlichste Weise erfolgen, wobei die gebräuchlichste und naheliegenste Variante wohl die Injektion per Übergabeparameter an den Konstruktor ist. Externe Abhängigkeiten werden dabei beim Aufruf des Konstruktors an das neu erzeugte Objekt übergeben.

Listing 2.8: **Beispiel: Dependency Injection per Konstruktor**

```
public class OrderManagement {
    private readonly ICustomerManagement mCustomerManagement;
    private readonly IPersistence mPersistence;

    public OrderManagement(ICustomerManagement ↩
        customerManagement, IPersistence persistence) {
         mCustomerManagement = customerManagement;
         mPersistence = persistence;
    }

    public IList<Order> GetOrders() {
         // ...
    }
}
```

Das Beispiel verdeutlicht anschaulich, wie durch die Anwendung des **Dependency Injection** Paradigmas gleich beide der oben genannten Schwachstellen beseitigt werden. Zum einen wird die Klasse `OrderManagement` von den beiden anderen Klassen entkoppelt. Es werden nun keine Kenntnisse mehr über den Aufbau der eigentlichen Implementierung benötigt.

Die Kommunikation erfolgt aussschließlich über die Schnittstellen `ICustomerManagement` und `IPersistence`. Änderungen der konkreten Implementierungen haben nun keinen Einfluss mehr auf die Klasse `OrderManagement`. Lediglich eine Änderung der öffentlichen Schnittstellen würde eine Änderung der Klasse `OrderManagement` nach sich ziehen.

Durch die Entkopplung der Klasse sowie die Auslagerung der Konfiguration ergeben sich hinsichtlich der Testbarkeit große Vorteile, da die Klasse `OrderManagement` nun isoliert getestet werden kann. Die externen Abhängigkeiten lassen sich über Mock-Objekte simulieren, deren Verhalten auf den Testfall abgestimmt werden kann. Innerhalb der Mock-Objekte können zudem auch Datenstrukturen spezifiziert werden, die zur Durchführung des Testfalles erforderlich sind. Hervorzuheben ist dabei, dass der Autor des Testfalles keine Kenntnis über die in der Software verwendete Persistenztechnik benötigt, sondern unabhängig von der verwendeten Technik gegen definierte Schnittstellen getestet werden kann.

2.4.10. Fehlende Codegestaltungsrichtlinien

Standardkonventionen für die Gestaltung des Sourcecodes haben auf den ersten Blick keinen entscheidenden Einfluss auf die Funktionalität einer Software. So haben beispielsweise die Benennung von Variablen oder deren Anordnung innerhalb von Methoden keinen Einfluss auf die Performanz einer Software. Dennoch handelt es sich hierbei um einen Code Smell, der keinesfalls vernachlässigt werden sollte. Durch eine inkonsistente Gestaltung des Sourcecodes leidet dessen Lesbarkeit und Verständlichkeit. Entwickler haben es durch die Vermischung verschiedener Codegestaltungen deutlich schwerer, den Sourcecode zu verstehen und zu warten.

2.5. Paradigmen

2.5.1. Law of Demeter

Das Gesetz von Demeter ist ein recht einfach gehaltenes Paradigma, welches sich mit den Beziehungen zwischen Klassen auseinander setzt. Es stellt klare Regeln darüber auf, welche Beziehungen zwischen Klassen zulässig sind und welche nicht. Primär soll die Kommunikation zwischen Klassen minimiert, sowie die Nutzung von verketteten Aufrufen verhindert werden.

Das Gesetz von Demeter stellt hierfür einige klare Regeln auf:

1. Eine Klasse darf auf eigene Methoden zugreifen.
2. Eine Methode darf auf Methoden von Eingabeparametern zugreifen.
3. Eine Klasse darf auf Methoden assoziierter Klassen zugreifen.
4. Eine Klasse darf auf die Methoden selbst erzeugter Objekte zugreifen.

Im folgenden Beispiel soll verdeutlicht werden, wie gegen das Gesetz von Demeter verstoßen wird:

Listing 2.9: **Beispiel: Verstoß gegen das Law of Demeter**

```
public class Example {
    public void DoSomething(X x) {
        x.GetInnerObject().DoMagic();
    }
}
```

In diesem Beispiel wird zunächst Zugriff auf die Methode `GetInnerObject()` des übergebenen Parameters genommen. Dies allein stellt noch keinen Verstoß gegen die Vorgaben dar. Allerdings wird ein weiterer Aufruf auf den Rückgabewert von `GetInnerObject()` durchgeführt und genau hier findet der Verstoß statt.

Durch die Nichtbeachtung dieses Paradigmas wird der Kopplungsgrad unnötig erhöht. Bedingt durch die Verkettung von Aufrufen werden zusätzliche Abhändigkeiten erzeugt und je länger eine solche Verkettung ist, umso mehr Abhängigkeiten kommen hinzu.

Durch die Restriktionen, die das Gesetz von Demeter vorgibt, werden solche Anhängigkeiten aktiv unterbunden. Klassen dürfen nur mit ihren unmittelbaren Nachbarn Informationen austauschen und sind somit auch nur von deren Aufbau abhängig.

2.5.2. Program to an Interface, not to an Implementation

Bereits im Jahre 1995 wurde dieses Prinzip in Vlissides (2000) beschrieben. Es handelt sich um eines der Grundprinzipien des objektorientierten Designs. Das Paradigma besagt, dass konkrete Implementierungen untereinander nicht voneinander anhängig sein sollten. Stattdessen sollten die Verknüpfungen untereinander über Schnittstellen oder abstrakte Klassen abgebildet werden.

Daraus ergeben sich zwei wichtige Vorteile. Zum einen wird durch die Verwendung von Schnittstellen die Austauschbarkeit der Elemente untereinander gewährleistet, denn solange die Schnittstelle bestand hat, ändert sich für den Nutzer der Schnittstelle nichts, auch wenn die konkrete Implementierung hinter der Schnittstelle geändert wird. Als zweiter wichtiger Vorteil ist die Entkopplung von technischen Details anzusehen. Oftmals müssen konkrete Implementierungen weitere Klassen oder Bibliotheken referenzieren, um ihre Funktionalität bereit zu stellen. Bei direkten Abhängigkeiten zwischen den einzelnen Implementierungen müssten solche technischen Details auch dem Nutzer der Implementierung bekannt sein.

Durch die Implementierung gegen Schnittstelle steigert der Entwickler aktiv die Testbarkeit des Systems, denn durch die Anwendung dieses Prinzips werden voneinander isolierte Modul- und Komponententests erst ermöglicht. Abhängigkeiten zu anderen Modulen oder Komponenten lassen sich durch die Verwendung von Schnittstellen und dem in 2.4.9 beschriebenen Verfahren nachbilden. Modul- und Komponententests können so isoliert und ohne Miteinbeziehnung der Funktionalität externer Codeelemente durchgeführt werden.

2.5.3. Interface Segregation Principle

> Clients should not be forced to depend upon interfaces that they do not use. Martin (1996b)

Das Interface Segregation Principle besagt, dass Clients nicht von Schnittstellen abhängig sein sollten, deren Funktionalität gar nicht benötigt wird. Das Paradigma bezieht sich auf Schnittstellen, die mehrere Funktionsgruppen in sich vereinen. Clients, die die Schnittstelle benutzen sind dadurch gezwungen, auch auf Funktionalität Zugriff zu haben, die eigentlich nicht benötigt wird.

Die folgenden beiden Beispiele stammen aus Martin (1996b) und wurden vom Autor in die Sprache C# übertragen.

Listing 2.10: **Beispiel: Vermischte Funktionalität**

```
public interface IDoor {
    bool IsOpen { get; }
    event EventHandler<EventArgs> TimerElapsed;
    void Open();
    void Close();
}
```

Bei der Konzeption von Schnittstellen sollte daher immer darauf geachtet werden, nur eng zusammengehörende Funktionalität in ein und der selben Schnittstelle vorzusehen.

Listing 2.11: **Beispiel: Aufgetrennte Funktionalität**

```
public interface IDoor {
    bool IsOpen { get; }
    void Open();
    void Close();
}

public interface ITimedDoor : IDoor {
    event EventHandler<EventArgs> TimerElapsed;
}
```

Die Auftrennung der Funktionalitäten trägt entscheidend dazu bei, die Abhängigkeit zwischen Clients und Schnittstelle zu minimieren. Durch Änderungen der einen Schnittstelle sind nun nur die Clients betroffen, die diese Schnittstelle auch nutzen.

2.5.4. Design by Contract

Das **Design by Contract** Prinzip zielt auf die Kollaboration zwischen Softwarelementen ab. Durch sogenannte Verträge werden Interaktionen zwischen Modulen durch die vorherige Festlegung von Bedingungen abgesichert. Durch die Spezifikation von Vorbedingungen kann für Methoden festgelegt werden, wie diese aufgerufen werden müssen. Sind gewisse Eingabewerte nicht zugelassen, können diese schon im Vorwege durch die Vorbedingung ausgeschlossen werden.

Zusätzlich zur Vorbedingung, die zur Absicherung eines Methodenaufrufes vorgesehen ist, können auch Nachbedingungen spezifiziert werden. Durch die Spezifikation von Nachbedingungen hat der Aufrufer der Methode genaue Informationen darüber, in welchem Zustand sich das Modul, dessen Methode ausgeführt wurde, nun befindet.

Als Ergänzung zu den beiden o.g. Bedingungen existiert die sogenannte Invariante. Durch die Invariante können allgemeine Rahmenbedingungen festgelegt werden, die zu fast jeder Zeit für ein Modul gelten müssen. Nur bei Ausnahmen oder während eines Methodenaufrufes besteht die Möglichkeit, dass eine spezifizierte Invariante verletzt ist.[13]

2.5.5. Liskov Substitution Principle

> Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.Martin (1996c)

Das Liskov'sche Substitutionsprinzip steht in enger Beziehung zu dem oben beschriebenen Verfahren **Design By Contract**. Es wurde von Barbara Liskov im Jahre 1987 eingeführt und mehrere Jahre später genauer spezifiziert. Thematisch befasst sich das Prinzip mit Vererbungshierarchien in Programmen. Speziell das Verhalten (beim Aufruf von Methoden und Ändern von Eigenschaften) verwandter Typen ist von Interesse.

Das Prinzip besagt, dass sich alle Untertypen eines Basistypen genauso verhalten müssen, wie der Basistyp. Dies bezieht sich auf Methodenaufrufe, das Ändern von Eigenschaften sowie auf Vergleiche (z.B. per `equals`). So müssen beispielsweise Methoden, die einen Basistypen als Parameter entgegen nehmen, ein identisches Verhalten sowohl für den Basistypen als auch für den Subtypen aufweisen. Ist dies nicht der Fall, müsste die Methode Kenntnis über alle Untertypen und deren spezielle Eigenarten besitzen[14]. Dies erzeugt nicht nur bei der initialen Erstellung der Methode, sondern ggf. auch bei späteren Erweiterungen zusätzlichen Aufwand. Wird ein neuer Untertyp zu dem Basistypen erstellt, dessen Verhalten von dem des Basistypen abweicht, so müssten auch alle Methoden, die den Basistypen nutzen, um das spezifische Wissen des neuen Untertypen erweitert werden.

Die Einhaltung des Liskov'schen Substitutionsprinzips ist aufwendig und erfordert bereits in der Konzeptionsphase erhöhte Wachsamkeit. Anhand des folgenden Codebeispiels soll verdeutlicht werden, welche Probleme bei Vererbungshierarien auftreten können, die für den Betrachter auf den ersten Blick als zulässig eingestuft werden. Das folgende Beispiel stammt aus Martin (1996c).

[13] vgl. Siedersleben (2004)
[14] vgl. Martin (1996c)

Listing 2.12: **Beispiel: Verletzung des LSP**

```
public class Rectangle {
    public virtual double Width { get; set; }
    public virtual double Height { get; set; }
    public double Area { get { return Width * Height; } }
}

public class Square : Rectangle {
    private double width;
    private double height;

    public override double Width {
        get { return width; }
        set { width = value; height = value; }
    }

    public override double Height {
        get { return height; }
        set { width = value; height = value; }
    }
}
```

Ein Quadrat ist eine spezielle Form eines Rechtecks mit der Ausnahme, dass die Länge aller Kanten bei einem Quadrat identisch sind. Auf den ersten Blick ist die oben aufgeführte Vererbungshierarchie demzufolge zulässig. Im Detail ergeben sich aus dieser Hierarchie aber massive Probleme, denn durch die Tatsache, dass die Klasse `Square` strengere Rahmenbedingungen formuliert (Breite == Höhe), können Methode, die ein Objekt des Typs `Rectangle` entgegennehmen, nicht mehr uneingeschränkt mit diesem Objekt arbeiten. Dies wird durch die folgende Methodenimplementierung verdeutlicht:

Listing 2.13: **Beispiel: Ergebnis aus Verletzung des LSP**

```
public void SomeMethod(Rectangle rect) {
    int width = 5;
    int height = 4;

    rect.Width = width;
    rect.Height = height;

    if (rect.Area != (width * height))
        throw new Exception();
}
```

Die aufgeführte Methode kann kann sowohl mit einem Objekt des Typs `Rectangle`, als auch mit einem Objekt des Typs `Square` aufgerufen werden. Allerdings ist das Verhalten der Methode für die beiden Typen nicht identisch. Durch die Tatsache, dass sowohl die Breite, als auch die Höhe des übergebenen Objektes modifiziert werden, werden im Falle eines übergebenen `Square` Objektes implizit beide Eigenschaften angepasst.

Die Verletzung des Liskov'schen Substitutionsprinzip wäre, wie in fast allen Fällen, für den Ersteller der `Rectangle` Vererbungshierarchie nicht von Bedeutung. Die Verletzung des Prinzips kommt erst in einem größeren Kontext zum tragen, denn erst wenn die Typen `Rectangle` und `Square` benutzt werden, ergeben sich bedingt durch das unterschiedliche Verhalten der Typen Schwierigkeiten. Die obige Methodenimplementierung müsste zur Herstellung der korrekten Funktionalität um zusätzliches Detailwissen über beide Typen erweitert werden:

Listing 2.14: **Beispiel: Erweiterungen durch Verletzung des LSP**

```
public void SomeMethod(Rectangle rect) {
    int width = 5, height = 4;

    rect.Width = width;
    rect.Height = height;

    if ((rect is Square) && (rect.Area != (height * height)))
            throw new Exception();
    else if (rect.Area != (width * height))
            throw new Exception();
}
```

Es bleibt festzuhalten, dass die Gestaltung einer Vererbungshierarchie eines Programms nicht ausschließlich nach den logischen Beziehungen zwischen verschiedenen Typen erstellt werden darf. Es muss besondere Rücksicht auf das Verhalten der einzelnen Typen innerhalb der Hierarchie gelegt werden. Untertypen eines Basistypen müssen sich immer genauso verhalten, wie der Basistyp. Einschränkungen, wie im oben aufgeführten Beispiel, sind nicht zulässig.

2.6. Statische Codeanalyse

Die statische Analyse von Sourcecode dient zur nachträglichen Prüfung bereits realisierter Module. Über die statische Codeanalyse kann a posteriori sichergestellt werden, dass eine Software zuvor festgelegten Regeln genügt.[15] Statisch bedeutet, dass die Software zur Analyse nicht ausgeführt werden muss. Die Überprüfung der Software erfolgt ohne deren Ausführung auf Ebene des Sourcecodes. Fehler und Auffälligkeiten werden von der Analysesoftware grundsätzlich nicht automatisch behoben. Es werden lediglich Hinweise auf mögliche Schwachstellen im Sourcecode geliefert.

2.6.1. Metriken

Eine Metrik dient zur Vermessung einer Software. Eine Metrik bildet eine Softwareeinheit in einen Zahlenwert ab und ermöglicht so die Messung des Erfüllungsgrades eines Qualitätsziels.[16] Durch die wiederholte Anwendung von Metriken auf eine Software lassen sich Trends erkennen und Qualitätsmängel frühzeitig ausmachen.

Als Beispiel sei an dieser Stelle die Messung der „**Lines of Code**“ genannt. Diese sehr simple Metrik dient zur Messung der Zeilen eines Programms. Weitere bekannte Metriken sind die Zyklomatische Komplexität nach McCabe sowie die Halstead Metriken.

[15] vgl. Hoffmann (2008)
[16] vgl. Schneider (2007)

3. Analyse

Das folgende Kapitel beinhaltet die Evaluierung verschiedener Tool-Lösungen, die für die Umsetzung der Analyseregeln in Frage kommen. Bei der Evaluierung wird das Hauptaugenmerk auf die Umsetzbarkeit der Code Smells und Paradigmen gelegt, die im Kapitel 2 vorgestellt wurden. Da es sich dabei nur um eine Teilmenge der bekannten Code Smells und Paradigen handelt, kann die Evaluierung der Tools keine verlässliche Auskunft darüber geben, ob sie auch für die Umsetzung anderer Analyseregeln geeignet sind.

Der im Vorwege vorgestellte Code Smell „**Duplizierter Sourcecode**“ wird bei der Toolanalyse nicht beachtet. Die Erkennung von Duplikaten innerhalb des Sourcecodes gestaltet sich äußerst kompliziert, weswegen hierfür spezielle Tools, wie z.B. Atomiq eingesetzt werden.

3.1. NimblePros Nitriq 1.0

3.1.1. Überblick

Bei Nitriq handelt es sich um ein Tool zur Durchführung einer statischen Codeanalyse für .NET Assemblies. Das Tool wird in zwei verschiedenen Ausführungen angeboten. Zum einen in der Version **Pro Developer Edition** und zum anderen in der sogenannten **Console Edition**. Die **Pro Developer Edition** zeichnet sich durch eine grafische Benutzeroberfläche aus und ist auch für den kommerziellen Gebrauch kostenlos. Bei der Console Edition handelt es sich um eine rein kommandozeilenbasierte Anwendung, die nicht frei verfügbar ist. Die Console Edition wird im Rahmen dieser Arbeit nicht weiter behandelt.

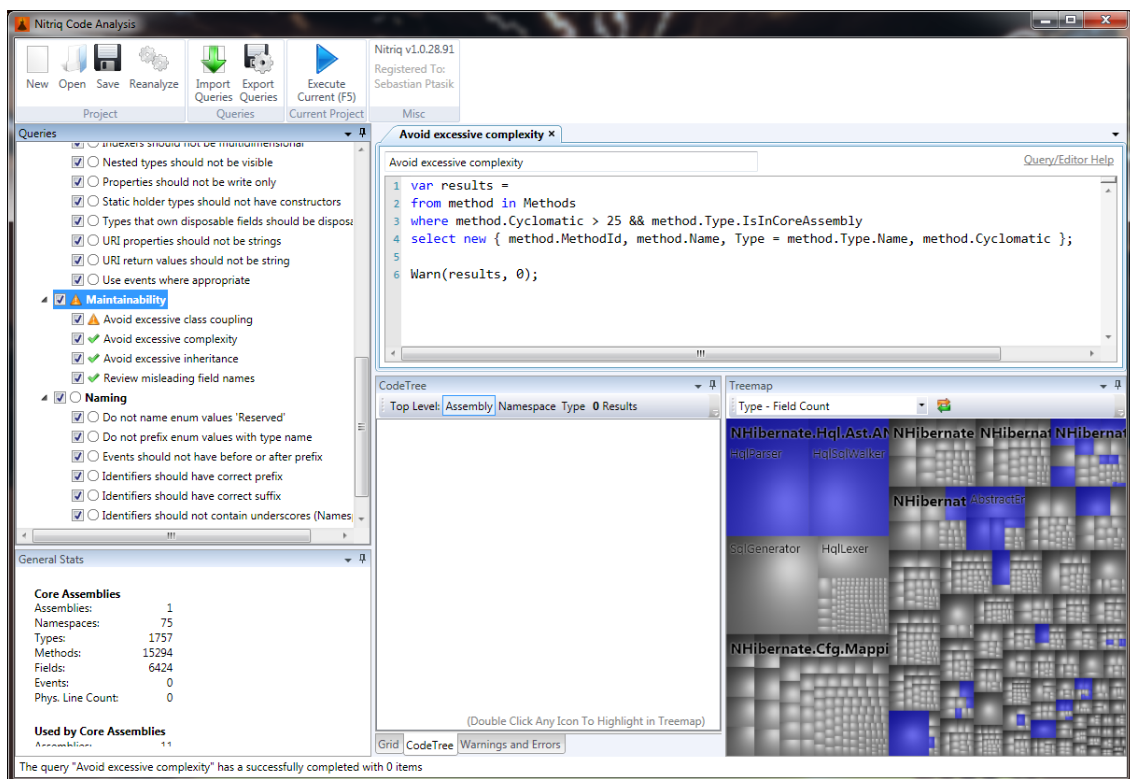

Abbildung 3.1.: Überblick Nitriq

Nach der Installation von Nitriq kann das Programm verwendet werden. Es besteht die Möglichkeit ein oder auch mehrere Assemblies in die Analyse mit ein zu beziehen. Nitriq bringt von Haus aus eine große Sammlung von Analyseregeln mit, die vom Benutzer nach Bedarf modifiziert werden können. Zur Formulierung der Analyseregeln verwendet Nitriq die

sogenannte **Code Query Language**[1]. Hierbei handelt es sich um eine spezielle Abfragesprache, die stark an LINQ[2] angelehnt ist und die Abfrage verschiedener Attribute des Quellcodes zulässt. Alle Analyseregeln, die mit Nitriq mitgeliefert werden, können vom Benutzer eingesehen und auch modifiziert werden. So hat der Benutzer z.B. die Möglichkeit, Maximalwerte für Regelverletzungen zu verkleinern oder zu vergrößern oder weitere Restriktionen zu bestehenden Regeln hinzuzufügen.

Die Bedienoberfläche der Anwendung gliedert sich in die folgenden fünf verschiedene Bereiche:

- Abfragenübersicht
- Allgemeine Statistikwerte
- Abfrage in CQL
- Ergebnisfenster
- Grafische Anzeige von Regelverstößen

Die Abfrageübersicht stellt alle verfügbaren Analyseregeln in einer Baumansicht dar. Dazu zählen sowohl die vom Hersteller mitgelieferten, als auch die vom Benutzer erstellten Analyseregeln. Die Regeln werden innerhalb der Baumansicht zur besseren Übersicht in Kategorien unterteilt. Jede Kategorie kann weitere Unterkategorien enthalten. Besonders hilfreich sind die Kategorien bei der Durchführung der Codeanalyse. Es besteht die Möglichkeit, neben der Durchführung einer kompletten Prüfung auch nur einzelne Regeln oder alle Regeln einer Kategorie überprüfen zu lassen.

Wesentlich interessanter ist die Anzeige der in CQL formulierten Abfragen. Die Abfragen werden entsprechend formatiert dargestellt und können innerhalb des Fensters vom Benutzer modifiziert werden.

[1] Kurz: CQL
[2] Language Integrated Query

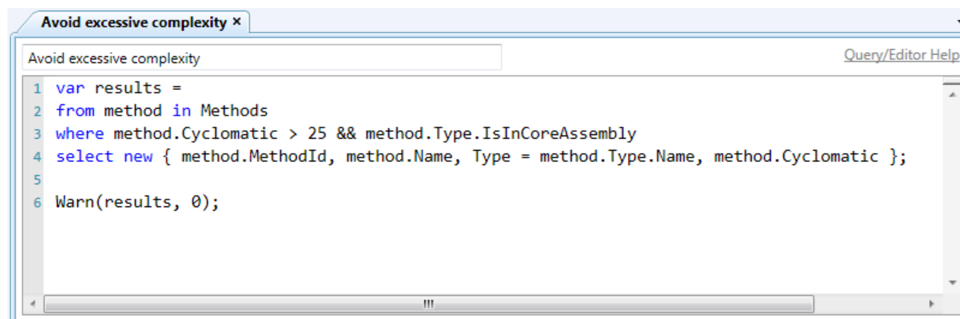

Abbildung 3.2.: CQL Ansicht

Bei der Modifikation bestehender Regeln und während der Erstellung neuer Analyseregeln wird der Benutzer von Nitriq unterstützt. Es existiert eine automatische Vervollständigung, die dem Benutzer Aufschluss über die Attribute gibt, die aktuell abgefragt werden können.

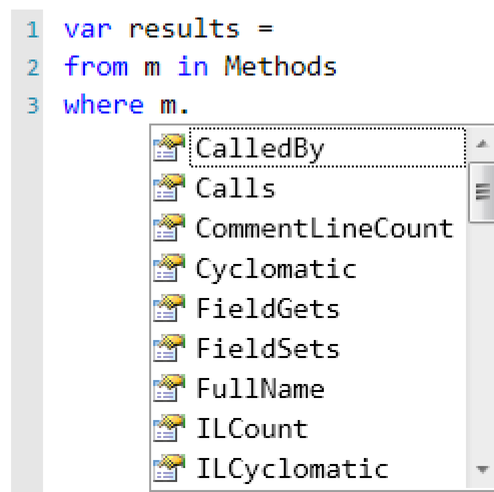

Abbildung 3.3.: CQL Autovervollständigung

Im Ergebnisfenster werden nach der Durchführung einer Abfrage die Resultate dargestellt. Werden Regelverstöße registriert, werden die entsprechenden Klassen, Methoden oder Felder in der Ansicht dargestellt. Innerhalb des Fensters können verschiedene Kategoriesierungen aktiviert werden. So können die Ergebnisse nach Assemblies, Namensräumen oder Klassen kategorisiert werden. Über das Ergebnisfenster werden zwar Regelverstöße dargestellt, es ist aber nicht möglich, die betroffene Stelle im Quellcode anzuzeigen.

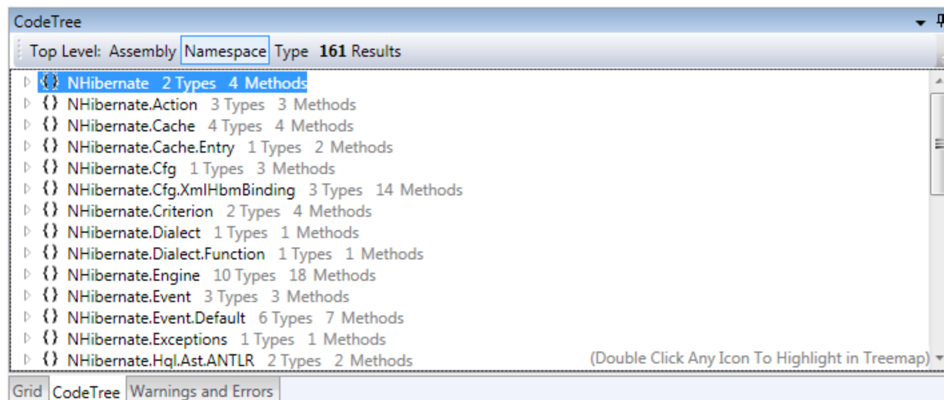

Abbildung 3.4.: Ergebnisanzeige

Bei der sogenannten **Treemap** handelt es sich um eine alternative Form der Visualisierung von Regelverstößen. Über eine farbliche Hervorhebung in blau werden die Regelverstöße dargestellt. Elemente, die nicht gegen die Regel verstoßen, werden grau dargestellt. Über eine Auswahlbox kann ausgewählt werden, welches Attribut innerhalb der Grafik visualisiert werden soll. Analysiert eine Regel beispielsweise die Anzahl der Parameter der vorhandenen Methoden, kann das Ergebnis der Abfrage durch Auswahl der Option „**Method - Parameter Count**“ angezeigt werden. Zur Visualisierung der Ergebnisse ist es also erforderlich, dass die ausgewählte Option in Zusammenhang mit der innerhalb der Analyseregel betrachteten Kategorie übereinstimmt. Wird innerhalb der Regel eine Anfrage auf Ebene von Methoden durchgeführt, werden innerhalb der Treemap keine Ergebnisse darstellt, wenn nicht eines der in der Analyseregel verwendeten Attribute ausgewählt wird.

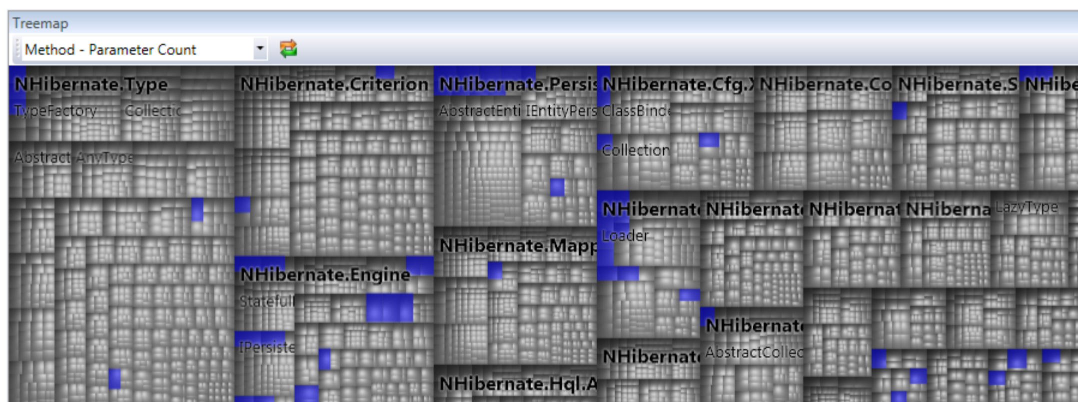

Abbildung 3.5.: Ergebnispräsentation in der Treemap

3.1.2. Bewertung

Als einziges der hier aufgeführten Analysetools bietet Nitriq die Möglichkeit der grafischen Auswertung von durchgeführten Analysen. Die „Treemap“ visualisiert Regelverstöße und ermöglicht dem Entwickler so, Regelverstöße schell zu erkennen. Allerdings bietet dieses Verfahren kaum Vorteile gegenüber der regulären textuellen Anzeige von betroffenen Klassen, so das dieses Feature vernachlässigt werden kann. Eine wahre Bereicherung hingegen ist die integrierte Abfragesprache CQL, über die einzelne Attribute verschiedener Codeelemente abgefragt werden können. Selbst aufwändige Analyseregeln lassen sich so einfach und kompakt formulieren. Leider ist der Informationsumfang dieser Attribute leicht eingeschränkt. So liefert die Abfragesprache beispielsweise keine Informationen über den Inhalt von Methodenrümpfen. Die Anweisungen innerhalb von Methodenrümpfen können also nicht direkt analysiert werden.

3.2. Microsoft FxCop 10.0

3.2.1. Überblick

Das Tool FxCop stammt von Microsoft. Ebenso wie das in 3.1 beschriebene Tool Nitriq handelt es sich bei FxCop um ein Werkzeug zur Durchführung einer statischen Codeanalyse mit Hilfe eines vordefinierten Regelwerks. FxCop ist Bestandteil des Windows SDK und ist sowohl für private, als auch für kommerzielle Zwecke frei verfügbar. FxCop bietet im direkten Vergleich mit Nitriq eine deutlich größere Anzahl an vordefinierten Regeln. Besonderer Vorteil dieses Tools ist die Tatsache, dass die von Microsoft verwendeten Guidelines zur Gestaltung von .NET Anwendungen und Bibliotheken bereits als vordefinierte Regeln vorhanden sind und somit ohne großen Aufwand genutzt werden können. Wie Nitriq verfügt auch FxCop über eine grafische Bedienoberfläche, mit deren Hilfe eine Analyse durchgeführt werden kann.

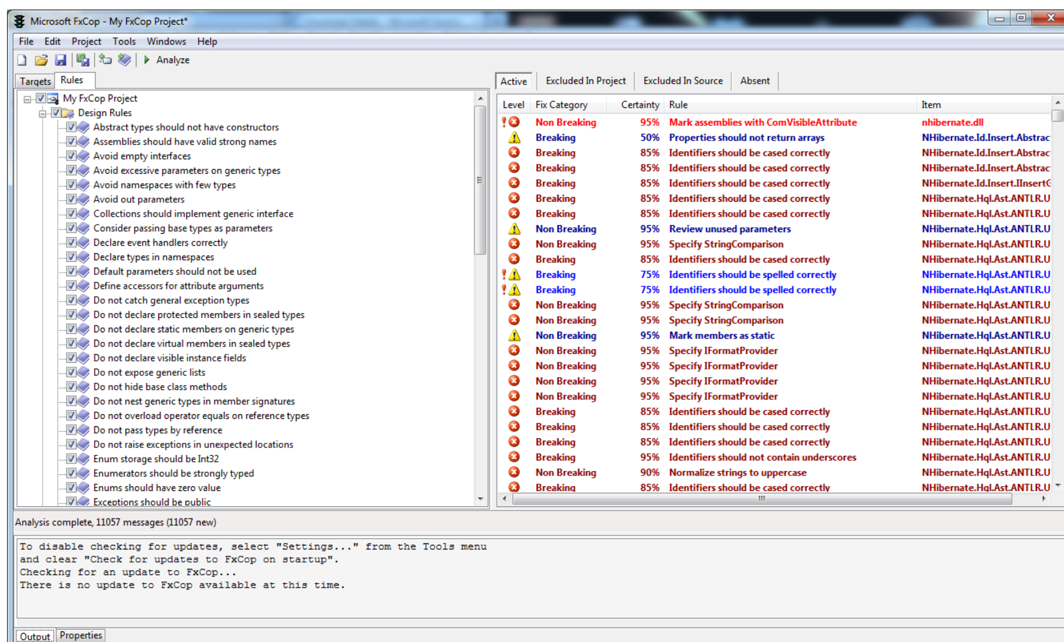

Abbildung 3.6.: Überblick FxCop

Neben der Analyse einer oder mehrerer Assemblies über die grafische Bedienoberfläche ist auch die automatisierte Durchführung der Analyse über die Kommandozeile Bestandteil von FxCop. So lässt sich die Analyse bequem in einen Continuous Integration Prozess integrieren.

Das ohnehin schon umfangreiche Regelwerk, dass mit FxCop geliefert wird, kann vom Benutzer beliebig erweitert werden. Allerdings gibt es hierbei zwei massive Einschränkungen im Vergleich zu Nitriq. Zum einen ist der Benutzer nicht in der Lage, bestehende Regeln

nach seinen Wünschen anzupassen. Sollen bestehende Regeln geändert werden, müssen diese vom Benutzer komplett neu erstellt werden. Die weitaus gravierendere Einschränkung, die FxCop mit sich bringt, erschwert den Erstellungsprozess von Analyseregeln erheblich. FxCop bietet, im Gegensatz zu Nitriq keine integrierte Abfragesprache für Sourcecodeelemente. Analyseregeln können nicht in CQL verfasst werden, sondern müssen umständlich durch Anlegen einer neuen Analysebibliothek und durch Erben von vorgefertigten Analyseklassen erstellt werden. Zusätzlich dazu muss für jede Analyseregel auch eine XML-Datei nach einem vordefinierten Schema erstellt werden, die für die Ausgabe von Fehlertexten etc. benötigt wird. Der Ablauf zur Erstellung benutzerdefinierter Analyseregeln wird im weiteren Verlauf dieses Abschnitts erklärt.

Bevor allerdings geklärt wird, wie benutzerdefinierte Analyseregeln für FxCop erstellt werden, muss geklärt werden, welche Art von Abfragemöglichkeiten geboten werden. Dazu ist ein Blick auf das Modell, mit dem FxCop arbeitet, erforderlich. Die folgende Grafik[3] zeigt die grundlegenden Elemente auf, die im Rahmen einer Analyse abgefragt werden können. Das Modell[4] gliedert die Sprachelemente des .NET Frameworks in eine Hierarchie ein und gibt zugleich vor, welche Beziehungen zwischen den einzelnen Knoten zulässig sind.

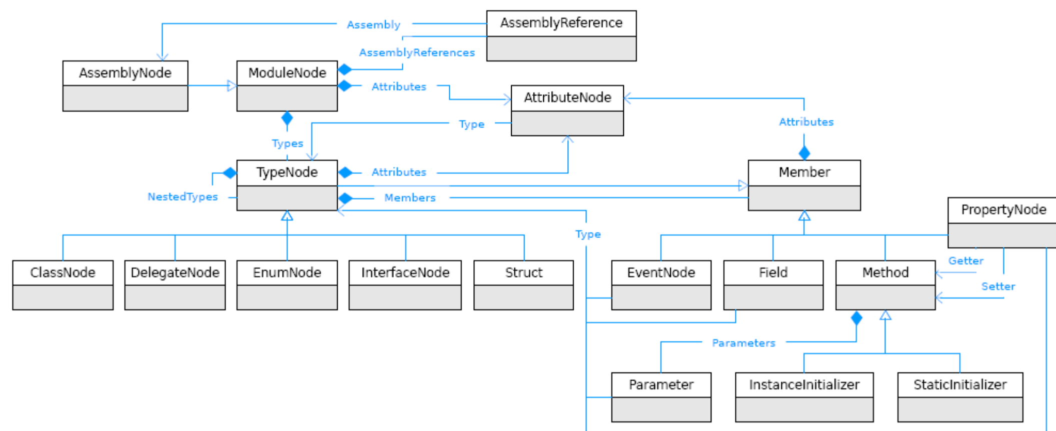

Abbildung 3.7.: FxCop Code-Modell

[3]Quelle: Kresowaty (2008)
[4]Auch Introspection Model genannt.

Im Rahmen einer Codeanalyse über FxCop kommt genau dieses Modell zum Einsatz. Beim Verfassen von Regeln wird auf die Elemente dieses Modells zugegriffen, um Regelverstöße zu erkennen. Die umständliche Erstellung eines eigenen Regelwerks über den Quellcode erscheint auf den ersten Blick sehr kompliziert. Allerdings wird hierdurch die größtmögliche Flexibilität erreicht. Besonders hervorzuheben ist hier, dass beispielsweise Hilfsklassen erstellt werden können, die bei der Realisierung von Regeln benötigt werden.

Für die Erstellung eines eigenen Regelwerkes wird eine Entwicklungsumbegung für C# oder Visual Basic benötigt. Zur Erstellung eines neuen Regelwerkes ist es erforderlich, die Regeln innerhalb einer neuen Klassenbibliothek anzuzlegen. Jede Regel besteht dabei aus zwei Komponenten. Zum einen ist es erforderlich, für jede neue Regel eine XML-Datei[5] anzulegen. Diese Datei beinhaltet regelspezifische Einstellungen, wie z.B. die Stufe (Warnung, Fehler), sowie die Festlegung regelspezifischer Texte, die an den Benutzer ausgegeben werden. Die XML-Datei muss dabei den folgenden Aufbau aufweisen, da das Regelwerk ansonsten nicht geladen werden kann.

Listing 3.1: **FxCop: XML-Beschreibung des Regelwerkes**

```
<Rules FriendlyName="CleanCodeRules">
    <Rule TypeName="InterfaceNamePrefixRule" Category="↩
       CleanCodeRules.CleanCodeRules" CheckId="CC0001">
      <Name>
        Schnittstellen sollten mit dem Prefix 'I' beginnen.
      </Name>
      <Description>
        Der Name einer Schnittstelle sollte mit 'I' beginnen.
      </Description>
      <Resolution Name="Prefix1">
        Fügen Sie an den Name von '{0}' das Prefix 'I' an.
      </Resolution>
      <Resolution Name="Prefix2">
        Der Buchstabe nach 'I' sollte groß geschrieben werden.
      </Resolution>
      <MessageLevel Certainty="95">Error</MessageLevel>
      <FixCategories>Breaking</FixCategories>
    </Rule>
</Rules>
```

[5]XML - Extendable Markup Language

Zusätzlich wird eine Klasse benötigt, die die Implementierung der Regel beinhaltet. Bei dieser Klasse muss es sich um eine Spezialisierung der Klasse `BaseIntrospectionRule` handeln, welche vom FxCop Framework bereit gestellt wird. Diese Klasse wird über die Festlegung diverser Attribute im Rahmen ihrer Instanziierung mit der eben genannten XML-Datei verknüpft.

Listing 3.2: **FxCop: Implementierung einer beispielhaften Regel**

```
class InterfaceNamePrefixRule : BaseIntrospectionRule {
    public InterfaceNamePrefixRule()
        : base("InterfaceNamePrefixRule",
               "CleanCodeRules.CleanCodeRules",
               typeof(InterfaceNamePrefixRule).Assembly)
    { }

    public override ProblemCollection Check(TypeNode type) {
        if (type.NodeType == NodeType.Interface) {
            if (!type.Name.Name.StartsWith("I")) {
                Problems.Add(new Problem(GetNamedResolution("↩
                    Prefix1", type.Name.Name)));
            }
            else {
                var secondCharacter = type.Name.Name.ElementAt↩
                    (1);

                if ((secondCharacter >= 0x61) && (↩
                    secondCharacter <= 0x7A))
                     Problems.Add(new Problem(GetNamedResolution↩
                        ("Prefix2", type.Name.Name)));
            }
        }

        return this.Problems;
    }
}
```

Soll das Regelwerk getestet werden, muss die Bibliothek händisch in ein spezielles Unterverzeichnis von FxCop kopiert werden. Dies ist erforderlich da FxCop sonst nicht in der Lage ist, die Regeln beim Starten einzulesen.

3.2.2. Vordefinierte Regeln

In diesem Abschnitt werden Analyseregeln vorgestellt, mit deren Hilfe FxCop von Haus aus das Clean Code Develpment unterstützt. Speziell die Kategorien „Design" und „Performance" beinhalten einige vordefinierte Analyseregeln, die im Rahmen dieser Arbeit Verwendung finden können. Hervorzuheben ist, dass bereits viele Analyseregeln zur Erkennung von nicht genutztem Sourcecode mitgeliefert werden. Die Analysemöglichkeiten gehen dabei über die Erkennung der in 2.4.5 vorgestellten Fehlerfälle hinaus. FxCop bietet über die Erkennung von nicht genutzten geschützten Methoden auch die Möglichkeit, nicht genutzte eingebettete Typen und Felder zu erkennen.

Do not catch general exception types

Diese vordefinierte Analyseregel geht auf den in 2.4.7 vorgestellt Code Smell ein. Es werden `catch`-Blöcke erkannt, in denen der Typ `Exception` abgefangen wird. Die Autoren der Regel folgen dabei der in 2.4.7 aufgeführten Argumentation.

Do not raise exceptions in unexpected locations

Hierbei handelt es sich um eine Analyseregel, die Bezug auf das Liskov'sche Substitutionsprinzip nimmt. Die Regel definiert, dass in vordefinierten Methoden, wie `ToString` oder `Equals` keine Ausnahmen ausgelöst werden dürfen, da sich dadurch das Verhalten der jeweiligen Methode drastisch ändert.

Avoid uncalled private code

Diese Regel nimmt Bezug auf den in 2.4.5 vorgestellten Code Smell. Es wird nach Methoden gesucht, die innerhalb der Assemby sichtbar sind und nicht explizit aufgerufen werden. Aufrufe mit Hilfe der .NET Reflection API werden nicht erkannt und verursachen fehlerhafte Ergebnisse. Die Analyseregel differenziert allerdings nicht zwischen den verschiedenen Arten von Assemblies. So ist es durchaus korrekt, dass es innerhalb von Bibliotheken viele als `public` delarierte Methoden gibt, die nicht genutzt werden. Allerdings sollte dies in ausführbaren Assemblies nicht der Fall sein. Diese Art von Assemblies wird überlicherweise nicht innerhalb von anderen Assemblies eingebunden.

Avoid uninstantiated internal classes

Neben einem Erkennungsmechanismus für ungenutzte interne Methoden wird auch eine Analyseregel für nicht genutzte interne Klassen geboten. Die Regel prüft, ob intern sichtbare

Klassen instanziiert werden. Ist dies nicht der Fall, wird von einem Regelverstoß ausgegangen. Die Regel ist so konzipiert, dass Objekte, die über die .NET Reflection API erzeugt werden, **nicht** erkannt werden. Wird ein solches Verfahren zur Erstellung von Objekten angewendet, würde die Analyseregel fehlerhafte Ergebnisse liefern.

3.2.3. Bewertung

FxCop bietet dem Entwickler im Gegensatz zu Nitriq keine grafische Auswertung von Analyseregeln. Ergebnisse einer Analyse werden in textueller Form dargestellt. Vorteil dieser Darstellung ist die Möglichkeit der Kategorisierung von Regelverstößen nach Schweregrad. So können Regeln, die keine vollständig zuverlässige Fehlererkennung zulassen, in der Kategorie „Warnung“ geführt werden.

Die umständliche Art und Weise der Regelerstellung stellt ein klares Argument gegen die Nutzung von FxCop dar. Allerdings wird dieser Kritikpunkt dadurch relativiert, dass FxCop Zugriff auf alle erdenklichen Programmelemente bietet. Dies geht soweit, dass selbst die Bestandteile einzelner Anweisungen analysieren werden können. Diese Fähigkeit wird nicht für alle Analyseregeln benötigt, allerdings ist dies ein Indikator dafür, dass sich viele weitere Analyseregeln durch dieses Tool realisieren lassen.

Abschließend sei gesagt, dass es sich bei FxCop um ein von Microsoft angebotenes Tool handelt. Es wird kontinuierlich weitergepflegt und neue Spracherweiterungen werden zeitnah eingepflegt.

3.3. StyleCop 4.6

3.3.1. Überblick

Bei StyleCop handelt es sich um ein Tool, das nach einem ähnlichen Prinzip wie FxCop funktioniert. Die Analyseregeln werden ebenfalls per Quellcode und XML-Datei definiert. Allerdings gibt es einige fundamentale Unterschiede. StyleCop arbeitet, anders als FxCop, nicht auf bereits kompilierten Assemblies, sondern auf Basis von Quellcode. Es wird daher nur die Sprache C# unterstützt. Eine Unterstützung von Visual Basic ist nicht vorgesehen. Da StyleCop nicht mehr von Microsoft weiterentwickelt wird, sondern von der „Community“ gepflegt wird, ist nicht gewährleistet, dass zu jeder Zeit alle Sprachelemente von C# unterstützt werden. Diese wesentliche Einschränkung gegenüber FxCop führt dazu, dass StyleCop für diese Arbeit nicht verwendet wird.

3.3.2. Bewertung

StyleCop wird im weiteren Verlauf dieser Arbeit **nicht** verwendet, da nicht sichergestellt werden kann, dass alle Sprachelemente des .NET Frameworks unterstützt werden. Da der Funktionsumfang von StyleCop den von FxCop nicht übersteigt, ist mit keinen Einschränkungen bei der Realisierung der Analyseregeln zu rechnen.

3.4. Microsoft Code Contracts 1.4

3.4.1. Überblick

Code Contracts von Microsoft[6] ist, wie die anderen vorgestellten Tools, ein Werkzeug zur Überprüfung von vordefinierten Regeln. Das Tool wird allerdings vorwiegend zur dynamischen Überprüfung, d.h. während der Laufzeit, eingesetzt. Dabei zielt Code Contacts bewusst nicht auf die Einhaltung von Codegestaltungsrichtlinien o.Ä. ab, sondern vielmehr auf die Einhaltung von Kommunikationsverträgen zwischen Softwarelementen. Es handelt sich dabei um eine mögliche Form zur Realisierung des **Design By Contract** Prinzips, dass unter 2.5.4 beschrieben wurde. Die Bedingungen, die für die Kommunikation verschiedener Softwareelemente gelten, werden dabei mit Hilfe des `System.Diagnostics.Contract` Namensraumen direkt im Quellcode spezifiziert. Im weiteren Verlauf dieses Abschnitts wird die Funktionsweise von Code Contracts erläutert, wobei sich die Beschreibung auf die drei wichtigsten Regularien (Vor- und Nachbedingung sowie Invarianten) beschränkt.

Definierte Vorbedingungen sichern den Aufgerufenen vor fehlerhaften Eingaben durch den Aufrufer ab. Durch diese Absicherung entsteht für beide Seiten zusätzliche Sicherheit. Der Aufrufer kann sich darauf verlassen, dass alle zugelassenen Eingaben auch zu einem gültigen Ergebnis führen. Zusätzlich wird durch die Vorbedingungen sichergetellt, dass sich der Aufgerufene bei Fehleingaben weiterhin in einem zulässigen Zustand befindet, da die eigentliche Funktionalität von Methoden in diesem Fall nicht ausgeführt werden würde. Die Spezifikation einer Vorbedingung erfolgt über eine simple Codezeile, die sich **am Anfang** der Methode befinden muss. Über den Befehl `Contract.Requires(x != null)` wird eine solche Vorbedingung eingefügt, die besagt, dass der Wert `null` für die Variable `x` unzulässig ist.

Nachbedingungen sichern dem Aufrufer bei Eingabe zulässiger Werte in eine Methode bestimmte Ausgabewerte zu. Dies ist nicht nur für den Aufrufer hilfreich, sondern auch zur Sicherstellung der korrekten Methodenimplementierung. Neben Nachbedingungen, die direkt für den Aufrufer wahrnehmbar sind (z.B. der Rückgabewert), können auch noch weitere Nachbedingungen spezifiziert werden, die nur klassenintern gelten und damit nicht unmittelbar vom Aufrufer wahrgenommen werden. Die Definition von Nachbedingungen erfolgt auf ähnlich einfache Weise, wie die eben beschriebenen Vorbedingungen. Ebenso wie die Vorbedingungen müssen auch die Nachbedingungen **am Anfang** der betreffenden Methode platziert werden. Über den Befehl `Contract.Ensures(x > 0)` wird eine Nachbedingung definiert. Diese aufgeführte Nachbedingung stellt sicher, dass der Wert der Variable `x` größer als `0` ist.

[6] http://research.microsoft.com/en-us/projects/contracts/

Zusätzlich zu den beschriebenen Vor- und Nachbedingungen können auch Invarianten für Klassen definiert werden. Invarianten beinhalten eine Ansammlung von Regularien, die ein Objekt einer Klasse zu **jederzeit** erfüllen muss. Ausgenommen davon sind lediglich Objektzustände, die während der Ausführung einer Methode auftreten können. Die Definition einer Invariante gestaltet sich kaum schwieriger, als die Definition von Vor- und Nachbedingungen. Zur Definition der Invariante einer Klasse muss eine neue Methode eingeführt und mit einem speziellen Attribut versehen werden. Das folgende Beispiel ist Microsoft (2011) entnommen und zeigt die Definition einer solchen Invariante.

Listing 3.3: **Code Contracts: Definition Invariante**

```
[ContractInvariantMethod()]
private void ClassInvariant() {
    Contract.Invariant(this.y >= 0);
    Contract.Invariant(this.x > this.y);
    //...
}
```

Im Hinblick auf das **Design By Contract** Prinzip, sowie das Liskov'sche Substitutionsprinzip muss geklärt werden, in wie weit sich Vor- und Nachbedingungen sowie Invarianten für Vererbungshierachien unter Verwendung von Code Contracts definieren lassen. Da Schnittstellen nur die Deklaration aber nicht die Definition von öffentlichen Eigenschaften und Methoden beinhalten, können Bedingungen für diese auch nicht direkt innerhalb der Schnittstelle festgelegt werden. Um dieses Problem zu umgehen, kann für eine Schnittstelle eine konkrete Klasse spezifiziert werden, welche die Implementierung der Bedingungen enthält. Diese Bedingungen gelten dann für **alle** Subtypen der Schnittstelle. In 2.5.5 wurde beschrieben, dass Vorbedingungen einer Methode niemals weiter eingeschränkt aber durchaus erweitert werden können. Dies ist unter Verwendung von Code Contracts nicht möglich. Einmal definierte Vorbedingungen können nur auf oberster Ebene und somit für alle Subtypen geändert werden. Laut Microsoft (2011) überwiegen die Nachteile durch die Fähigkeit, Vorbedingungen für Subtypen zu ändern, weswegen das Framework diese Möglichkeit nicht anbietet.

Zur Realisierung eines solchen Regelwerkes für eine Vererbungshierarche sind mehrere Schritte notwendig. Nach der Festlegung der Schnittstelle kann bereits mit der Implementierung der Bedingungen begonnen werden. Zur Implementierung der Bedingungen muss zunächst eine neue Klasse erstellt werden, welche die Eigenschaften und Methoden der Schnittstelle realisiert. Innerhalb dieser Methoden können dann die Bedingungen definiert, sowie eine Invariante für die gesamte Vererbungshierarchie festgelegt werden.

Listing 3.4: **Code Contracts: Vererbungshierarchie**

```
[ContractClass(typeof(ListContract))]
public interface IList {
    int Size { get; }
    void Clear();
}

[ContractClassFor(typeof(IList))]
class ListContract : IList {
    public int Size { get; private set; }

    public void Clear() {
        Contract.Ensures(Size == 0);
    }

    [ContractInvariantMethod()]
    private void ClassInvariant() {
        Contract.Invariant(Size >= 0);
    }
}
```

Wie in diesem Beispiel erkennbar ist, müssen sowohl die Schnittstelle, als auch die Implementierung der Bedingungen einen Verweis auf den jeweiligen Partner besitzen. So wird der Schnittstelle über das Attribut `ContractClass(typeof(ListContract))` mitgeteilt, dass die Klasse `ListContract` die Vor- und Nachbedingungen sowie die Invariante für alle Subtypen von `IList` bereitstellt. Umgekehrt muss auch die Klasse `ListContract` einen Verweis auf die Schnittstelle erhalten. Dies erfolgt über das Attribut `ContractClassFor(typeof(IList))`. Bei einer solchen Art der Definition von Bedingungen werden diese automatisch für alle Subtypen der Schnittstelle `IList` angewendet.

3.4.2. Bewertung

Bei Code Contracts handelt es sich um ein Tool zur Unterstützung des **Design By Contract** Prinzips (siehe 2.5.4), sowie des **Liskov'schen Substitutionsprinzips** (siehe 2.5.5) handelt. Zusätzliche Fähigkeiten, die über die Spezifizierung und Überprüfung von Bedingungen für die Kommunikation von Softwareelementen hinausgehen, werden nicht angeboten.

3.5. Zusammenfassung

Die vorgestellten Tools Nitriq, FxCop und StyleCop bieten allesamt eine Vielzahl an Möglichkeiten zur Erstellung benutzerspezifischer Analyseregeln zur Verbesserung der Codequalität. Als Spitzenreiter in Sachen Flexibilität und Zuverlässigkeit hat sich FxCop herausgestellt. Die umständliche Erstellung von Analyseregeln wirkt im ersten Moment abschreckend. Allerdings wird dadurch die größtmögliche Flexibilität hinsichtlich der Gestaltungsmöglichkeiten der Analyseregeln geboten. Auch bietet FxCop das von Microsoft verwendete Regelwerk als vordefinierte Analyseregeln an. Leider beinhaltet das Tool keine integrierte Abfragesprache, wie z.B. Nitriq. Durch die in Nitriq integrierte CQL gestaltet sich die Erstellung von Analyseregeln deutlich einfacher.

Code Contracts verfolgt einen anderen Ansatz, als die weiteren vorgestellten Tools. Das Tool füllt eine Lücke, die alle weiteren Tools offen gelassen haben. Die Überprüfung von Kommunikationsvertragen leistet einen fundamentalen Beitrag zur Qualität des Quellcodes und hilft bei der Realisierung der Kommunikation zwischen verschiedener Komponenten. Durch die Festlegung der Vor- und Nachbedingungen werden die Entwickler dahingehend entlastet, dass das Verhalten von Klassen und Methoden vorhersehbar ist. Sauber definierte Vor- und Nachbedingungen schaffen Klarheit bei der Benutzung von Schnittstellen und tragen aktiv dazu bei, dass der Benutzer einer Schnittstelle kein Wissen über die tatsächliche Implementierung benötigt.

Zur Zeit ist kein Werkzeug auf dem Markt verfügbar, dass die Fähigkeiten aller hier vorgestellten Tools in sich vereint. Allerdings existieren weitere Tools, die weitaus mächtiger sind, als die hier Vorgestellten. Als besonders praktisch hat sich das Tool **NDepend**[7] herausgestellt, das allerdings nur kommerziell vertrieben wird. Es erweitert die Fähigkeiten von Nitriq deutlich und bietet neben vieler zusätzlicher Funktionen auch eine vollständige Integration in die Entwicklungsumgebung **Microsoft Visual Studio** an.

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

4. Realisierung

Im Rahmen der Realisierung wird eine Auswahl der im Vorwege vorgestellten Code Smells und Paradigmen mit Hilfe des FxCop Frameworks in statische Codeanalyseregeln umgesetzt. Es werden sowohl einfache metrikbasierte Analyseregeln, als auch umfangreiche und komplexe Erkennungsmechanismen umgesetzt. Hierdurch soll ein möglichst großes Komplexitätsspektrum der Analyseregeln abgedeckt werden. Die während der Realisierungsphase gesammtelten Erkenntnisse sollen aufzeigen, mit welchem Arbeitsaufwand für die Erstellung weiterer Analyseregeln zu rechnen ist. Außerdem sollen die Erkenntnisse im Rahmen der Abschlussbetrachtung für eine Gegenüberstellungen von Aufwand und tatsächlichem Nutzen dienen.

Zur Realisierung der Regeln wird das Tool FxCop verwendet. Da sich in Kapitel 3 herausgestellt hat, dass dieses Tool nicht für die Überprüfung des **Design By Contract** Paradigmas und des **Liskov'schen Substitutionsprinzips** geeignet ist, werden diese Paradigmen von der Realisierung ausgeschlossen.

4.1. Erkennung von übergroßen Klassen

Die Anzahl der Verantwortlichkeiten, die eine übergroße Klasse inne hat, lassen sich nur schwer bis gar nicht ausmachen. Anhand des Sourcecodes können keine ausssagekräftigen semantischen Informationen gewonnen werden. Es ist zwar denkbar, die Verantwortlichkeiten einer Klasse anhand der Schnittstellen, die diese implementiert, zu bestimmen. Dieser Ansatz würde aber schnell zu fehlerhaften Ergebnissen führen, da es beispielsweise absolut zulässig ist, dass eine Klasse sowohl das Interface `IComparable<E>`, sowie das Interface `IClonable` implementiert. Aus diesem Grund wird im Rahmen dieser Arbeit der Ansatz verfolgt, die Anzahl der Methoden und Klasseninstanzvariablen auszuwerten. Der Autor folgt damit den Ausführungen in Fowler (1999) und Martin (2009). Für die Anzahl der Methoden und Klassenobjektvariablen werden dazu Obergrenzen definiert. Bei Überschreitung dieser Grenzen wird angenommen, dass es sich um eine übergroße Klasse handelt. Die vordefinierten Werte für die Obergrenzen werden in eine separate Konfigurationsdatei ausgelagert und können so nachträglich angepasst werden, ohne das eigentliche Regelwerk neu kompilieren zu müssen.

Allerdings garantiert auch dieser Ansatz keine fehlerfreie Erkennung, sondern liefert dem Softwareentwickler lediglich Hinweise auf Klassen, die potentiell zu mächtig sind und einer Überarbeitung bedürfen. Aus diesem Grund werden Verstöße gegen diese Regel auch lediglich als Warnung und nicht als Fehler ausgegeben.

4.2. Erkennung von übergroßen Methoden

Ähnlich wie die im vorangegangenen Abschnitt „Übergroße Klassen“ wird auch zur Erkennung dieses Code-Smells eine einfache Metrik verwendet. Die Anzahl der Codezeilen einer Methode gibt im begrenzten Maße Auskunft darüber, ob eine Methode zu viel Verantwortlichkeit besitzt bzw. zu viele Aufgaben erledigt.

Die Anzahl der Kommentarzeilen einer Methode können ein Hinweis auf eine komplexe Methode sein, haben aber beispielsweise keinen Einfluss auf Testfälle. Jede neue Codezeile innerhalb einer Methode kann dagegen Einfluss auf die Testfälle haben. Aus diesem Grund wird bei der Auswertung der Codezeilen einer Methode darauf geachtet, dass die Kommentarzeilen nicht mit in die Bewertung einfließen. Würden die Kommentarzeilen mit in die Bewertung einfließen, ließe sich die Auswertung dieser Regel manipulieren. So könnten zum einen große Methoden durch das Weglassen von Kommentaren künstlich geschrumpft und kurze Methoden unbeabsichtigt aufgebläht werden.

Genau wie bei der voherigen Regel lässt sich auch die vordefinierte Obergrenze für diese Analyseregel nachträglich über eine Konfigurationsdatei anpassen.

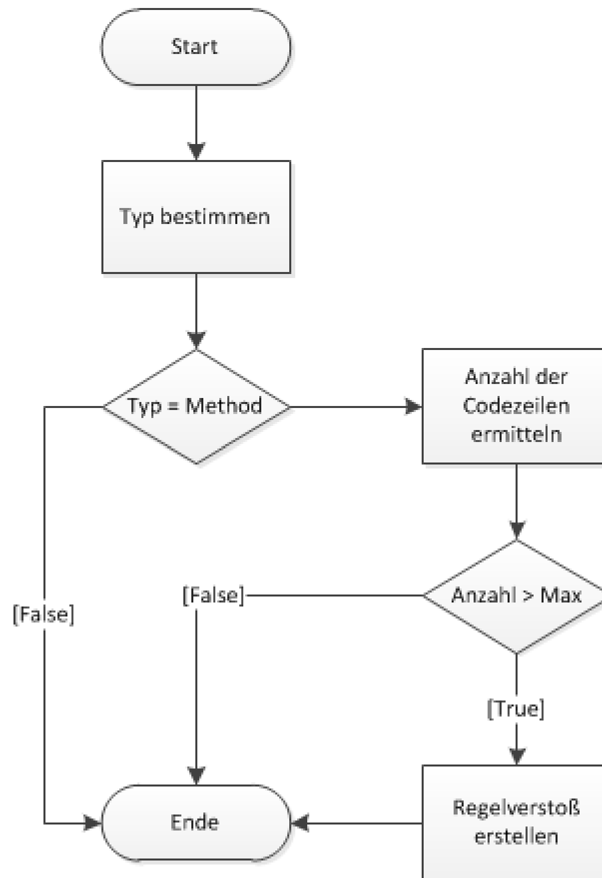

Abbildung 4.1.: Ablaufdiagramm zur Erkennung übergroßer Methoden

Das obige Ablaufdiagramm zeigt den Analyseablauf für **ein** Element eines Typs. Elemente eines Typs können Felder, Ereignisse, Eigenschaften sowie Methoden sein. Im Rahmen dieser Regel sollen nur Methoden betrachtet werden, weshalb alle anderen Elementtypen außer `NodeType.Method` ignoriert werden. Handelt es sich bei dem aktuell betrachteten Element allerdings um eine Methode, wird die Anzahl der Zeilen der Methode bestimmt und mit dem voreingestellten Schwellwert (**Max**) verglichen. Wird dieser Schwellwert überschritten, wird eine Regelverletzung generiert.

4.3. Begrenzung der Anzahl von Methodenparametern

Auf den ersten Blick handelt es sich hierbei abermals um einen Code-Smell, der durch simples Zählen erkannt werden kann. Das ist allerdings nur teilweise korrekt, denn Methodenparameter können nicht nur aus simplen Datentypen, wie `string` oder `int`, bestehen. Auch komplexe Datentypen in Form von Klasseninstanzen sind in der Regel in jeder Software als Übergabeparameter anzutreffen. Durch diesen Umstand gestaltet sich die Umsetzung der Regel zur Erkennung dieses Code-Smells deutlich schwieriger. So ist es beispielsweise denkbar, dass eine Methode nur einen einzigen Parameter besitzt, dieser aber ein komplexer Typ mit vielen Eigenschaften ist, welche ganz oder in Teilen innerhalb der Methode verwendet werden. Die interne Nutzung des übergebenen Objektes ist somit nach Außen nicht ersichtlich. Auch die Nutzung von Mengen als Übergabeparameter ist für diese Regel problematisch, denn streng genommen stellt jedes Element der Menge einen zusätzlichen Parameter dar.

Im Rahmen dieser Arbeit wird lediglich das Zählen der Methodenparameter realisiert. Die Überprüfung auf zusammengesetzte Übergabeparameter wird auch deswegen vernachlässigt, weil sowohl Martin (2009), als auch Fowler (1999) empfehlen, bei einer Häufung von Eingabeparametern, diese zu einer zusammenhängenden Datenstruktur zusammenzufassen.

Der Ablauf dieser Analyseregel deckt sich weitestgehend mit dem in der Abbildung 4.1 beschriebenen Ablauf. Der einzige Unterschied ist, dass anstelle der Zeilen der Methode nun die Parameter gezählt werden.

4.4. Vermeidung von ungenutztem Quellcode

Diese Regel kann sehr einfach realisiert werden und ist zumindest in Teilen auch in den bekannten Entwicklungsumgebungen integriert. Als Bespiel sei hier das Microsoft Visual Studio 2010 genannt, welches explizit darauf hinweist, wenn ein als `private` deklariertes Element nicht genutzt wird. Allerdings werden Elemente mit einer höheren Sichtbarkeit (`protected`, `internal`, `public`) außer Acht gelassen.

Die Grundlage der Analyseregel stellt das Zählen der Methodenaufrufe dar. Ist die Zahl der Aufrufe gleich 0, wird die Methode nicht genutzt. Im Detail ergeben sich aber noch einige Sonderfälle, die zur Vermeidung von Fehlerkennungen beachtet werden müssen. Wird die Analyseregel auf eine einzelne Programmbibliothek angewendet, ist es valide, wenn die öffentlichen Methoden, die die Bibliothek bereitstellt, nicht aufgerufen werden. Erst wenn die Bibliothek von einer anderen Bibliothek oder einem ausführbaren Programm referenziert wird, kann eine solche Analyse vorgenommen werden. Daraus ergibt sich eine weitere Einschränkung: Wird eine Programmbibliothek referenziert, so kann daraus nicht automatisch

geschlossen werden, dass alle öffentlichen Methoden der referenzierten Bibliothek genutzt werden.

Um eine möglichst fehlerfreie Erkennung von nicht genutztem Sourcecode zu gewährleisten, beschränkt sich die Suche auf eine einzelne Assembly. Speziell in Hinsicht auf `internal↩` -Elemente muss bei der Analyse zusätzlich beachtet werden, ob eine Assembly mit dem Attribut `InternalsVisibleTo` versehen wurde. Ist dies der Fall, werden alle als `internal` deklarieren Elemente des Assemblies behandelt, als seien sie als `public` deklariert. Dies gilt allerdings nur für die Assembly, welche innerhalb des Attributes angegeben wurde.

Listing 4.1: **InternalsVisibleTo**

```
...
[assembly: InternalsVisibleTo("AssemblyB")]
...
```

Durch das aufgeführte Attribut sind alle `internal`-Elemente der entsprechenden Assembly für „AssemblyB" sichtbar. Es kann daher nicht ausgeschlossen werden, dass Assembly-intern nicht genutzte und als `internal` deklarierte Elemente außerhalb der Assembly genutzt werden. Die Verwendung dieses Attributes zählt allerdings eher zur Ausnahme, als zur Regel und wird aus diesem Grund bei der Analyse vernachlässigt. Sollte das Attribut für die zu analysierende Assembly gesetzt sein, werden trotzdem alle nicht genutzten `internal`-Objekte als Regelverletzungen angesehen.

Eine weitere Einschränkung der Erkennungsrichtlinie stellen sogenannte Eventhandler dar. Hierbei handelt es sich um Methoden, die als Rückruffunktion für ausgelöste Ereignisse dienen. Der Weg zwischen der Auslösung des Ereignisses und der Abarbeitung im Eventhandler wird dabei von der Laufzeitumgebung verwaltet. Somit wird auch der Aufruf der Methode, die als Eventhandler eingerichtet ist, nicht direkt im Sourcecode erledigt, sondern von der Laufzeitumgebung im Hintergrund. Somit würden Eventhandler unabhängig von ihrem Sichtbarkeitslevel als nicht genutzter Sourcecode angesehen werden.

Eventhandler lassen sich zum Einen durch ihre Registrierung auf ein Ereignis als solche identifizieren. Zum Anderen kann eine Behandlungsroutine auch durch ihre Signatur identifiziert werden. Laut .NET Konvention sollten **alle** Evenhandler eine gleichartige Signatur aufweisen. Der erste Parameter ist dabei immer vom Typ `object`, der zweite Parameter ist vom Typ `EventArgs` oder einem Subtypen.

Zählen von Methodenaufrufen

Das Zählen von Methodenaufrufen wird mit Hilfe der Klasse `CallGraph` realisiert, die von FxCop bereitgestellt wird. Diese Klasse bietet die Methode `GetCallers` an, die alle Aufrufer der übergebenen Methode liefert. Ist die zurückgelieferte Liste leer, ist sichergestellt, dass die angegebene Methode nicht explizit aufgerufen wird. Ein Aufruf über die .NET Reflection API kann nicht erkannt werden.

Erkennung von Ereignisbehandlungsroutinen

Zur Erkennung von Ereignisbehandlungsroutinen wird das Verfahren genutzt, dass FxCop von Haus aus mitliefert. Über `RuleUtilities.IsEventHandling` kann überprüft werden, ob es sich bei der übergebenen Methode um eine solche Routine handelt oder nicht. Von weiteren Überprüfungen zur Erkennung von Ereignisbehandlungsroutinen wird abgesehen. Hierfür ergeben sich zu viele Sonderfälle. Hierfür bietet sich eine separate Analyseregel an, die die Verwendung dieses speziellen Methodentyps überprüft.

Unterscheidung von Bibliotheken und ausführbaren Dateien

Damit festgelegt werden kann, welche Methodensichtbarkeiten bei der Analyse beachtet werden müssen, muss der Typ der zu analysienden Assembly ermittelt werden. Hierzu bietet FxCop bereits eine integrierte Unterstützung an. Zwar kann nicht direkt ermittelt werden, ob es sich um eine ausführbare Assembly handelt. Allerdings kann der Typ einer Assembly abgefragt werden. Der Typ der Assembly lässt sich über die Eigenschaft `Kind` abfragen. Ist diese Eigenschaft entweder auf den Wert `ModuleKindFlags.ConsoleApplication` oder `ModuleKindFlags.WindowsApplication` gesetzt, handelt es sich um eine ausführbare Assembly.

Ablauf der Analyseregel

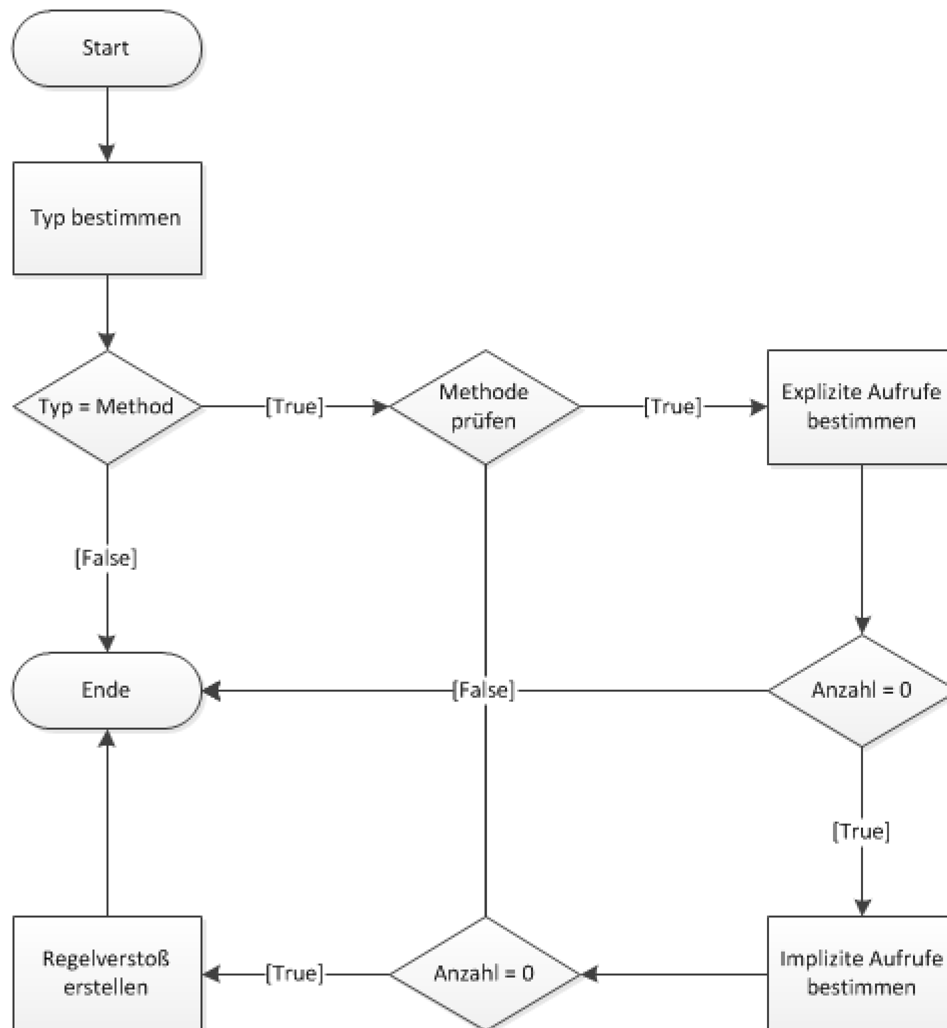

Abbildung 4.2.: Ablaufdiagramm zur Erkennung von ungenutzten Methoden

Das Ablaufdiagramm bezieht sich auf **ein** Element eines Typs. Da sich diese Regel lediglich auf Methoden bezieht, wird zu Beginn der Analyse eine Typüberprüfung durchgeführt. Ergibt die Typprüfung, dass es sich um eine Methode handelt, wird überprüft, ob die Methode weiter analysiert werden muss. In einigen Spezialfällen ist keine Überprüfung erforderlich bzw. sinnvoll. Handelt es sich beispielsweile um eine `public`-Methode innerhalb einer Bibliothek, soll keine Analyse durchgeführt werden.

Handelt es sich um eine Methode, die überprüft werden soll, werden zunächst die expliziten Aufrufe gezählt. Hierbei handelt es sich um Aufrufe, die direkt über den Klassentypen durchgeführt werden, der die Methode bereitstellt. Ergibt diese Überprüfung, dass die Methode nicht explizit aufgerufen wird, werden die implizierten Aufrufe ermittelt. Hierbei handelt es sich um Aufrufe, die über eine abstrakte Basisklasse oder einen Schnittstellentypen durchgeführt werden. Stellt auch diese Überprüfung fest, dass die Methode nicht aufgerufen wird, wird ein neuer Regelverstoß erzeugt.

4.5. Erkennung von lokalen Konstanten

Wie in 2.4.6 beschrieben, handelt es sich bei diesem Code Smell um konstante Werte innerhalb von Methoden. Dies umfasst sowohl konstante lokale Variablen, als auch konstante Werte innerhalb von Anweisungen.

Es hat sich herausgestellt, dass FxCop nicht in der Lage ist, lokale von globalen Konstanten zu unterscheiden. Üblicherweise werden Konstanten über das Schlüsselwort `const` als solche deklariert. Im Zuge der Kompilierung eines Programms oder einer Programmbibliothek werden allerdings alle Zugriffe auf eine globale konstante Variable durch deren Wert ersetzt. Ein Wertvergleich zwischen globalen Konstanten und methodenlokalen Literalen zur Umgehung dieses Defizits ist nicht sinnvoll, da hierdurch nicht sichergestellt werden kann, dass innerhalb der Methode tatsächlich die globale Konstante genutzt wurde.

4.6. Vermeidung von unbehandelten Ausnahmen

Die Sicherstellung der Behandlung aller innerhalb einer Anwendung möglichen Ausnahmen kann garantiert werden. Der hierfür erforderliche Aufwand steht aber in keinem Nutzen zu dem zu erwartenden Qualitätsgewinn. Für die lückenlose Erkennung wäre es erforderlich, die Methodenrümpfe **aller** innerhalb der Anwendung verwendeten Methoden dahingehend zu analysieren, ob eine Ausnahme erzeugt wird. Ist dies der Fall, müssten alle direkten Aufrufer der Methoden auf eine Behandlungsroutine überprüft werden. Stellen die dirkten Aufrufer keine solche Behandlungsroutine bereit, müssten auch die indirekten Aufrufer analysiert werden. Die Analyse dürfte sich außerdem nicht nur auf die eigentliche Anwendung beschränken, sondern müsste auch alle referenzierten externen Bibliotheken in die Analyse miteinbeziehen.

Um eine solch aufwendige Analyse zu vermeiden, bietet es sich an, die vom .NET Framework gelieferten Bordmittel zum Abfangen nicht behandelter Ausnahmen zu nutzen. So wird zwar

nicht verhindert, dass unbehandelte Ausnahmen auftreten, es kann aber effektiv verhindert werden, dass eine unbehandelte Ausnahme zum Absturz der Anwendung führt.

Das .NET Framework differenziert zwei Arten von nicht behandelten Ausnahmen. Bei der sogenannten UI Exception handelt es sich um eine Ausnahme, die innerhalb des Kontextes des Hauptthreads der Anwendung erzeugt aber nicht behandelt wurde. Zum Abfangen einer solchen Ausnahme bietet das .NET Framework den folgenden Mechanismus an:

Listing 4.2: **Behandlungsroutine für unbehandelte UI Exceptions**

```
AppDomain.CurrentDomain.UnhandledException +=
    new UnhandledExceptionEventHandler(OnUIException);

void OnUIException(object o, UnhandledExceptionEventArgs e) {
    //...
}
```

Als Gegenstück dazu existieren die Thread Exceptions. Hierbei handelt es sich um bisher unbehandelte Ausnahmen, die innerhalb des Kontextes eines Hintergrundthreads aufgetreten sind. Der Mechanismus zum Abfangen solcher Ausnahmen ist nahezu identisch mit dem bereits vorgestellten Mechanismus für UI Exceptions:

Listing 4.3: **Behandlungsroutine für unbehandelte Thread Exceptions**

```
Application.ThreadException +=
    new ThreadExceptionEventHandler(OnThreadException);

void OnThreadException(object o, ThreadExceptionEventArgs e) {
    //...
}
```

Um sicherzustellen, dass alle unbehandelten Ausnahmen abgefangen werden, sollten daher bei jeder Anwendung solche Behandlungsroutinen vorgesehen werden. Auch wenn die Anwendung beim Auftreten einer solchen Ausnahme nicht mehr ausgeführt werden kann, so kann zum einen sichergestellt werden, dass die aufgetretene Ausnahme im Rahmen einer Fehlerprotokollierung aufgezeichnet wird. Zum Anderen kann sichergestellt werden, dass die Anwendung kontrolliert heruntergefahren wird und es zu keinem Datenverlust kommt.

4.7. Erkennung veränderbarer Datentypen

Die nachträgliche Veränderung von internen Daten eines Datentyps lässt sich am einfachsten mit dem Attribut `readonly` verhindern. Dieses Attribut bewirkt, dass die damit gekennzeichneten Klasseninstanzvariablen innerhalb des Konstruktors initialisiert werden müssen und nachträglich nicht mehr geändert werden können. Einen Spezialfall stellen alle Arten von Behälterklassen (wie z.B. `List<E>`) dar. Zwar wird durch die Verwendung des `readonly`↩ Attributes die Liste als solche als nicht veränderbar deklariert. Der Inhalt der Liste ist aber weiterhin veränderbar. Es können also nach Belieben Elemente hinzugefügt oder entfernt werden.

Neben der traditionellen Abbildung von Variablen innerhalb einer Klasse bietet C# einen weiteren Mechanismus an. Mit Hilfe der sogenannten `Properties` können Klasseninstanzvariablen ohne direkte Deklaration abgebildet werden. Über die Eigenschaften werden dann lediglich die `get` und `set` Methoden abgebildet. Im Gegensatz zu den klassischen Instanzvariablen können Eigenschaften nicht mit dem Attribut `readonly` versehen werden. Es kann also allein durch deren Attributierung nicht gewährleistet werden, dass deren Inhalt zu einem späteren Zeitpunkt verändert wird. Allerdings kann durch eine entsprechende Attributierung der `set` Methode gewährleistet werden, dass die Eigenschaft nur klassenintern geändert werden kann. Hierzu wird die Eigenschaft mit dem Attribut `private` bzw. `protected` versehen.

Listing 4.4: **Eigenschaft ohne direkte Variablenbindung**

```
public string Name { get; private set; }
```

Heuristik zur Identifizierung von Datenklassen

Für diese und weitere Analyseregeln wird eine Heurisik benötigt, mit deren Hilfe Datenklassen von Kontrollklassen unterschieden werden können. Die Informationen, die zur Umsetzung der Heuristik genutzt werden sind:

- Referenzierung anderer Klassen
- Typen der Instanzvariablen und -eigenschaften

Referenziert eine Klasse lediglich weitere, als Datenklasse eingestufte Typen, so ist dies ein Indiz dafür, dass es sich bei der referenzierenden Klasse ebenfalls um eine Datenklasse handelt. Die Anzahl der Instanzvariaben- und Eigenschaften gibt keinen Aufschluss über den Typ der umschließenden Klasse. Vielmehr sind die Typen der internen Variablen von Interesse.

Die Heuristik stuft die folgenden Typen als Datentypen ein:

- Zeichenketten (`string`) und Zeichen (`char`)
- Numerische Werte (`int`, `float`, `byte`, etc.)
- Wahrheitswerte (`bool`)
- Aufzählungen (`enum`)

Neben diesen Elementen werden auch noch Strukturen (`struct`), sowie Arrays und generische Listen zu Datentypen gezählt, Bei Letzteren handelt es sich um einen Typus, der weder zu Kontroll-, noch zu Datentypenklassen gezählt werden kann. Den Ausschlag gibt hier der Typ der Elemente, die das Array bzw. die Liste beinhaltet.

Zur Identifizierung von Datentypklassen nach dem beschriebenen Verfahren ist es erforderlich, einen gerichteten Graphen zu erstellen. Mit Hilfe dieses Graphen kann für jeden potentiellen Datentypen eine Analyse erstellt werden. Zur Verdeutlichung des Aufbaus folgt ein kurzes Beispiel.

Listing 4.5: **Datentypklasse**

```
public class DataTypeA {
    public int IntValue { get; private set; }
    public DataTypeB TypeB { get; private set; }
    ...
}

public class DataTypeB {
    public string StringValue { get; private set; }
    ...
}
```

Die folgende Abbildung zeigt den Graphen, der sich aus dem o.g. Codebeispiel ergibt. Die Blätter sind die Codeelemente, die als Datentypen eingestuft werden und nicht aus weiteren Datentypen bestehen.

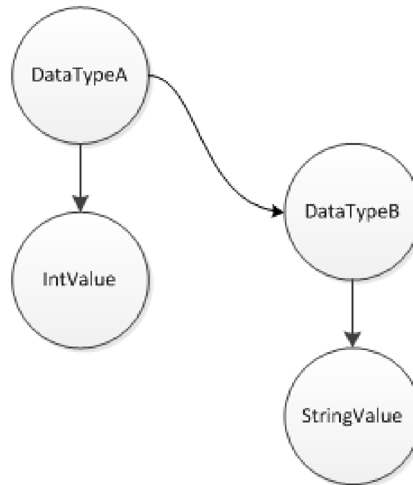

Abbildung 4.3.: Gerichteter Beziehungsgraph

Die Zyklenfreiheit innerhalb eines Graphen kann nicht gewährleistet werden. Beinhaltet eine Klasse Instanzvariablen ihres eigenen Typs, ergibt sich ein Zyklus. Ebenso ist es denkbar, das obige Beispiel so abzuändern, dass die Klasse `DataTypeB` eine Referenz auf `DataTypeA` besitzt.

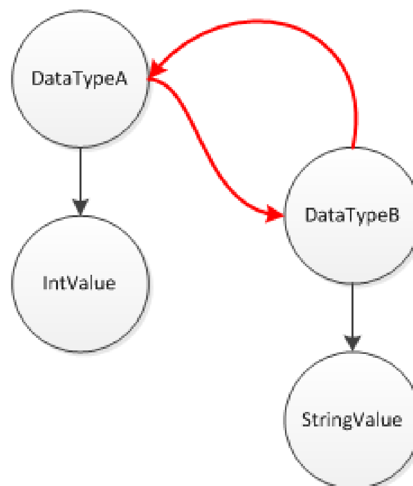

Abbildung 4.4.: Gerichteter Beziehungsgraph mit Zyklen

Zur Vermeidung einer Endlosrekursion bei der Erstellung des Graphen ist die Erkennung

und Behandlung solcher Zyklen unbedingt erforderlich. Wird ein Zyklus erkannt, wird die entsprechende Instanzvariable von der Überprüfung ausgeschlossen.

Überprüfung von Klasseninstanzvariablen

Für Instanzvariablen einer Datenklasse gilt im Rahmen dieser Regelung, dass sie entweder als `readonly` oder als `const` deklariert sein müssen. Für Listen werden keine speziellen Regeln vorgesehen, sie werden wie reguläre Variablen behandelt.

Überprüfung von Klasseneigenschaften

Alle Eigenschaften, die eine öffentlich zugängliche `set` Methode anbieten, werden als Regelverstoß angesehen. Dabei ist es unerheblich, ob die Eigenschaft eine Instanzvariable kapselt oder ob es sich um eine Eigenschaft ohne direkte Variablenbindung handelt.

4.8. Vermischung von Konfiguration und Nutzung

Die Vermischung von Konfiguration und Nutzung einer Klasse wurde bereits in 2.4.9 beschrieben. Aus dieser Beschreibung gehen auch die Indikatoren hervor, die zur Realisierung der Analyseregel genutzt werden sollen. Instanziiert eine Kontrollinstanz weitere Kontrollinstanzen zur internen Nutzung, so ist dies ein klarer Hinweis auf eine Regelverletzung. Grundvoraussetung für die Umsetzung einer solchen Klasse ist die im vorangegangenen Abschnitt genannte Heuristik zur Erkennung von Datentypen. Diese Heuristik findet bei dieser Regel ebenfalls Verwendung. Besitzt eine Klasse interne Instanzvariablen oder -Eigenschaften und handelt es sich dabei **nicht** um einen Datentypen, so sollten diese Elemente nicht klassenintern konfiguriert werden, sondern entweder durch Eingabeparameter des Konstruktors oder durch `set` Methoden initialisiert werden.

Ermittelung referenzierter Kontrollklassen

Zur Durchführung der eigentlichen Analyse ist es erforderlich, alle intern verwendeten Kontrollklassen ermitteln zu können. Die Ermittlung der Kontrollklassen wird mit Hilfe der Heuristik, die im Abschnitt 4.7 beschrieben wurde, ermöglicht. Über die Heuristik werden Daten- von Kontrollklassen unterschieden. Alle als Datentyp eingestuften Klassen werden von der Analyse ausgeschlossen.

Neben Datentypen müssen auch Typen, die während der Kompilierung erzeugt wurden, ignoriert werden. Der Entwickler kann nur bedingt Einfluss auf die Generierung und Nutzung

dieser Typen nehmen und hat somit im Falle eines erkannten Regelverstoßes kaum eine Möglichkeit, diesen zu beseitigen.

Überprüfung der Initialisierung

Um die Analyse durchführen zu können, ist es erforderlich, alle Wertzuweisungen an ein Feld zu ermitteln. Zur Realisierung dieser Anforderung muss eine spezielle Suchroutine erstellt werden. Die Routine wird mit Hilfe der Klasse `BinaryReadOnlyVisitor` realisiert, die von der FxCop Bibliothek bereitgestellt wird. Diese Klasse bietet die Möglichkeit, durch Überschreiben der Methode `VisitAssignmentStatement` alle Wertzuweisungen an eine Variable zu erkennen.

Ablauf der Analyseregel

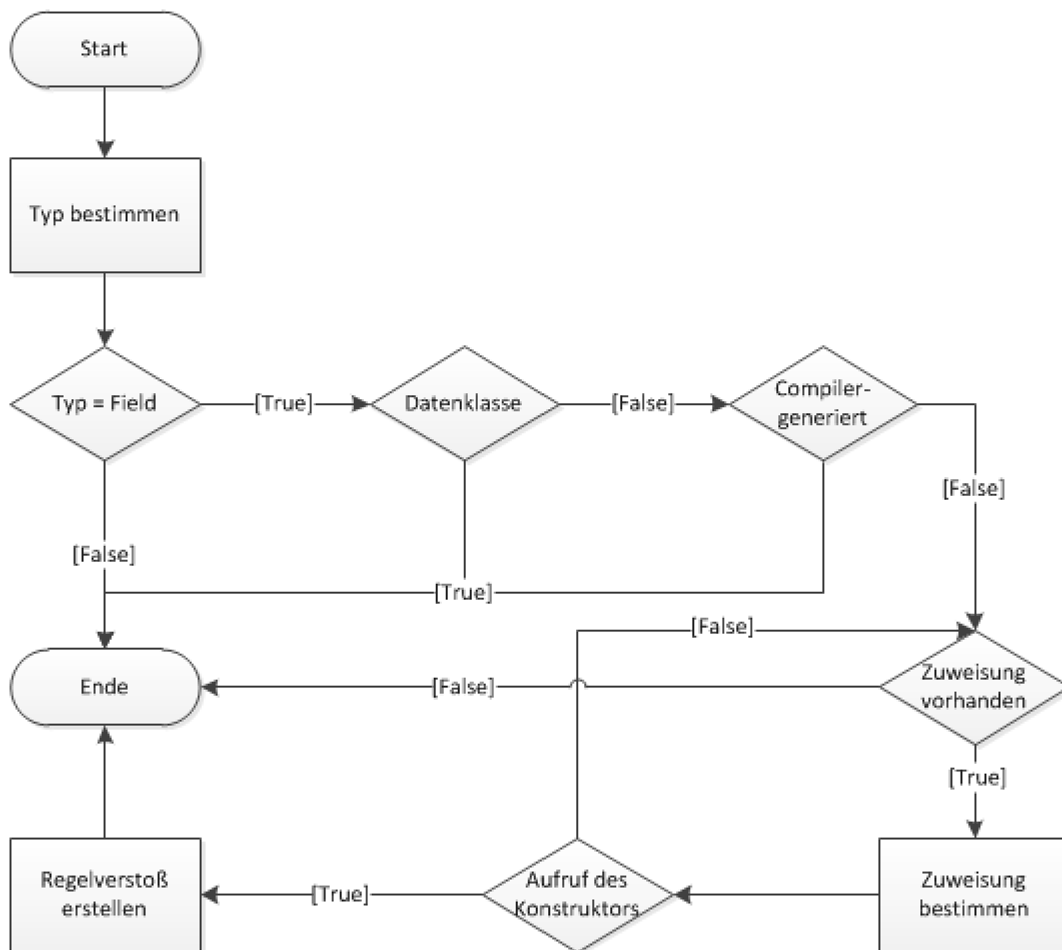

Abbildung 4.5.: Zustandsdiagramm zur Erkennung von vermischten Verantwortlichkeiten

Das Ablaufdiagramm bezieht sich auf **ein** Element eines Typs. Da sich diese Regel lediglich auf Felder bezieht, wird zu Beginn der Analyse eine Typüberprüfung durchgeführt. Handelt es sich bei dem zu analysierenden Element um ein Feld, wird die Analyse fortgesetzt. Im Zuge der Analyse sollen lediglich Kontrollklassen beachtet werden. Aus diesem Grund werden alle als Datentyp eingestuften Typen, sowie vom Compiler generierte Typen von der Analyse ausgeschlossen.

Handelt es sich bei dem Typen des aktuell betrachteten Feldes um eine Kontrollklasse, wird die Analyse fortgeführt. Dazu werden zunächst alle Wertzuweisungen an das Feld ermittelt. Handelt es sich bei mindestens einem dieser Wertzuweisungen um den Aufruf eines Kont-

struktors, wird eine Regelverletzung erzeugt. Die Überprüfung für das aktuell betrachtete Feld wird beendet, sobald ein Regelverstoß erkannt wurde oder alle Wertzuweisungen überprüft wurden.

4.9. Überprüfung von Codegestaltungsrichlinien

Die Gestaltung des Sourcecodes nimmt keinen Einfluss auf dessen Funktionalität. Trotzdem sollte auf eine einheitliche und durchgängige Formatierung des Sourcecodes. Wie bereits beschrieben wird hierdurch die Lesbarkeit und Verstehbarkeit deutlich verbessert. So wird beispielsweise durch die einheitliche Anordnung von Klasseninstanzvariablen, Eigenschaften und Methoden unnötiges Suchen innerhalb einer Sourcecodedatei minimiert.

Bei den Codegestaltungsrichtlinien handelt es sich in der Regel um umgebungsspezifischen Formatierungsvorschriften. Die Gestaltungsrichtlinien zweier Softwareentwicklungsunternehmen unterscheiden sich mit großer Wahrscheinlichkeit. Die Erstellung allgemeingültiger und unumstoßlicher Formatierungsregeln ist also nicht sinnvoll. Allerdings ist es sinnvoll, bei der Erstellung spezifischer Komponenten die empfohlenen Gestaltungsrichtlinien der verwendeten Programmiersprache einzuhalten. Damit sind z.B. Namen von Methoden gemeint. Innerhalb der Java-Plattform besteht die Konvention, Methodennamen mit einem Kleinbuchstaben zu beginnen, wohingegen Methodennamen innerhalb der .NET Plattform mit einem großen Buchstaben beginnen.

Tools wie FxCop bringen von Haus aus eine Sammlung von Analyseregeln zur Einhaltung der Gestaltungsrichtlinien und -empfehlungen für die .NET Plattform mit. So beinhaltet FxCop allein in der Kategorie „Design Rules“ mehr als 60 vordefinierte Analyseregeln. Zudem werden viele weitere Analyseregeln mitgeliefert, die unter anderen Kategorien geführt werden. So gibt es beispielsweise die Kategorie „Naming“, unter die die o.g. Regelung der Methodennamen fällt.

4.10. Einhaltung des Gesetzes von Demeter

Das Gesetz von Demeter definiert im Gegensatz zu vielen anderen Paradigmen sehr genaue Regel zur Verringerung der Kopplung. Das vollständige Regelwerk ist bereits in 2.5.1 aufgeführt. Da es bei diesem Prinzip vorrangig um das Aufrufen von Methoden innerhalb des Kontextes anderer Methoden geht, liegt der Realisierungsaufwand für diese Regel klar in der Analyse der Methodenrümpfe.

Zur Durchführung der Analyse wird für jede zu analysierende Klasse eine Datenbasis benötigt. Die Datenbasis beinhaltet die Typen, deren Methoden laut den Vorgaben des Paradigmas genutzt werden dürfen. Werden andere Methodenaufrufe erkannt, stellt dies einen Regelverstoß dar.

Die Informationen, die zur Erstellung der Datenbasis benötigt werden, ergeben sich aus den Regeln, die das Gesetz von Demeter definiert. Es werden Informationen über die Methoden der eigenen, sowie aller assoziierter Klassen benötigt. Anhand dieser Informationen wird der Rumpf einer jeden Methode der zu analysierenden Klasse untersucht. Werden Methodenaufrufe von Klassen erkannt, die nicht Bestandteil der zuvor erstellten Datenbasis sind, liegt ein Regelverstoß vor.

4.11. Verwendung von Schnittstellen anstelle konkreter Klassen

Abhängigkeiten zwischen Klassen sollten, wie bereits in 2.5.2 beschrieben, über Schnittstellen hergestellt werden. Die Einhaltung dieses Paradigmas lässt sich im ersten Ansatz recht einfach realisieren. An allen Stellen innerhalb einer Klasse, an denen auf externe Typen verwiesen wird, müssen diese Verweise über Schnittstellen oder abstrakte Klassen abgebildet werden. Verweise können z.B. Methodenparameter, Klasseninstanzvariablen oder lokale Variablen sein.

Wie bei nahezu jeder Regel gibt es aber auch hier Ausnahmen. So ist es oftmals nicht sinnvoll, den Zugriff auf Datentypen über Schnittstellen zu realisieren. Dies kann speziell bei einer Anwendung, die sich aus verschiedenen, miteinander in Relation stehenden Komponenten, zu Missverständissen führen. Werden Daten nur über Schnittstellen zwischen den Komponenten hin- und hergereicht, würde jede Komponente eine eingenständige Implementierung der Datentypschnittstelle benötigen. Die Tatsache, dass ein und der selbe Datentyp mehrfach implementiert werden muss, verstößt massiv gegen das **Don't Repeat Yourself** Paradigma (Siehe 2.4.1). Zusätzlich besteht die Gefahr, dass Operationen, wie z.B. `Equals()`, auf unterschiedliche Art und Weise implementiert werden und es dadurch zu Fehlern im System kommt.

Aus diesem Grund wird die Analyseregel so konzipiert, dass Datentypklassen ignoriert werden. Alle weiteren Abhängigkeiten hingegen müssen über Schnittstellen abgebildet werden. Im ersten Ansatz soll lediglich die Referenzierung fremder Klassen über Methodenparameter oder durch Anlegen von Klasseninstanzvariablen berücksichtigt werden.

Durch die Kategorisierung in Daten- und Kontrollklassen gestaltet sich die Formulierung der Analyseregel relativ einfach. Handelt es sich bei einem verwendeten Typen **nicht** um eine Datenklasse, muss es sich entweder um eine Schnittstelle oder eine abstrakte Klasse handeln.

Ist dies nicht der Fall, liegt ein Regelverstoß vor. Mit Hilfe der von FxCop bereitgestellten Klasse `ClassNode` kann die komplette Analyse durchgeführt werden. Die Klasse beinhaltet alle Informationen, die für die Analyse von Nöten sind.

4.12. Trennung von Schnittstellen anhand ihrer Zuständigkeiten

Die Vermischung von Zuständigkeiten innerhalb einer Schnittstelle lässt sich aufgrund der fehlenden semantischen Informationen nicht direkt erkennen. Die Analyse wird aus diesem Grund auf eine andere Art und Weise durchgeführt. Dabei wird nicht die Schnittstelle als isoliertes Analyseobjekt betrachtet, sondern vielmehr die Klassen, die eine bestimmte Schnittstelle benutzen. Anhand des in 2.5.3 genannten Beispiels wird ersichtlich, dass dieser Ansatz durchaus sinnvoll und erfolgversprechend ist.

Die Betrachtung der Schnittstellenmethoden, die innerhalb einer Klasse Verwendung finden, ermöglicht einen Rückschluss auf die verwendete Funktionalität. Nutzt eine Klasse nur einen Bruchteil der von der Schnittstelle angebotenen Funktionalität, kann dies ein Hinweis auf eine vermischte Funktionalität sein. Allerdings garantiert dieses Verfahren keine fehlerfreie Erkennung, sondern liefert lediglich Hinweise auf Schnittstellen, die verschiedene Funktionalitäten vermischen. Es obliegt daher dem Entwickler abzuwägen, ob gemeldete Regelverstöße eine Refaktorisierung rechtfertigen oder nicht.

Ermittelung der genutzten Schnittstellen

Da die Analyse auf Basis von Klassen durchgeführt werden soll, müssen zunächst die von der Klasse genutzten Schnittstellen ermittelt werden. Hierzu ist es erforderlich, die Typen aller Felder und Eigenschaften zu ermitteln. Zusätzlich müssen auch Typen der Rückgabewerte und Eingabeparameter aller Methoden ermittelt werden. Alle Typen, bei denen es sich um Schnittstellen handelt, werden in einer Liste gesammelt und im weiteren Verlauf der Analyse verwendet.

Ermittelung nicht genutzter Methoden

Die Ermittelung der genutzten Schnittstellenmethoden erfolgt mit Hilfe der bereits bekannten Methode `CallGraph.CallersFor`. Mit Hilfe dieser Methode wird überprüft, ob eine bestimmte Klasse eine Methode einer verwendeten Schnittstelle nutzt oder nicht. Befindet sich die aktuell analysierte Klasse **nicht** unter den Aufrufern der betrachteten Schnittstellenmethode, wird ein

Zähler erhöht. Dieser Zähler wird am Ende der Analyse dazu verwendet, den prozentualen Anteil der nicht genutzten Schnittstellenmethoden zu ermitteln.

Auslöseschwelle der Analyseregel

Ausschlaggebend für die Erkennung einer potentiellen Regelverletzung ist eine Auslöseschwelle. Über diese Auslöseschwelle wird definiert, wie hoch der prozentuale Anteil der genutzten Schnittstellenmethoden **mindestens** sein muss. Ergibt sich durch das im vorangangenen Abschnitt beschriebene Verfahren ein geringerer Nutzungsanteil, wird dies als Verstoß gegen die Analyseregel angesehen. Als Voreinstellung wird ein Wert von **50%** gewählt. Dieser Wert kann im Zuge der Auswertung, die im nächsten Kapitel beschrieben wird, angepasst werden, sollte sich der voreingestellte Wert als nicht praktikabel erweisen.

4.13. Reflexion der Realisierungsphase

Während der Realisierungsphase hat sich gezeigt, das die Komplexität der Code Smells und Paradigmen maßgeblichen Einfluss auf den Implementierungsaufwand nimmt. Einfache Metriken, wie z.B. die Ausmaße einer Klasse, ließen sich ohne große Umstände implementieren. Komplexere Regeln erzeugten durch die Vielzahl an Einschränkungen und Sonderfällen deutlich mehr Konzeptions- und Implementierungsaufwand. Allein durch den Zugriff auf die erforderlichen Informationen mussten viele Hilfsklassen implementiert werden. Dies war für die Implementierung der weniger komplexen Regeln nicht erforderlich.

Zusätzlich zum erhöhten Implementierungsaufwand komplexer Analyseregeln erhöhte sich auch der Testaufwand deutlich. Die vielen Sonderfälle erforderten wiederum viele Testfälle, was den Implementierungsaufwand deutlich erhöhte.

Durch die Umsetzung der unterschiedlich komplexen Regeln ist allerdings auch deutlich geworden, über welch umfassende Fähigkeiten FxCop verfügt. Wurde die Umsetzung von Analyseregeln zu Beginn dieses Kapitels noch aufgrund der durch das Framework vorgegebenen Kompliziertheit kritisch betrachet, so kann im Nachhinein festgestellt werden, dass sich dieser Eindruck als falsch erwiesen hat. Lediglich die fehlende Dokumentation des Frameworks hat sich als nachteilig erwiesen und verzögerte die Realisierung der Analyseregeln, je nach Komplexität, stark.

5. Auswertung

5.1. Beschreibung des Prüflings

Bei dem gewählten Prüfling handelt es sich um eine Software zur Auswertung und Aufarbeitung von Messdaten, die von externen Hardwarekomponenten geliefert werden. Zusätzlich übernimmt die Software die Ansteuerung und Auswertung digitaler Ein- und Ausgabekontakte über eine gesonderte Hardware. Die Software verwendet das Microsoft WCF[1] Framework, um aufbereitete Messdaten für eine gesonderte Visualisierungsanwendung bereitzustellen. Das folgende Schaubild verdeutlicht den grundsätzlichen Aufbau des Gesamtsystems.

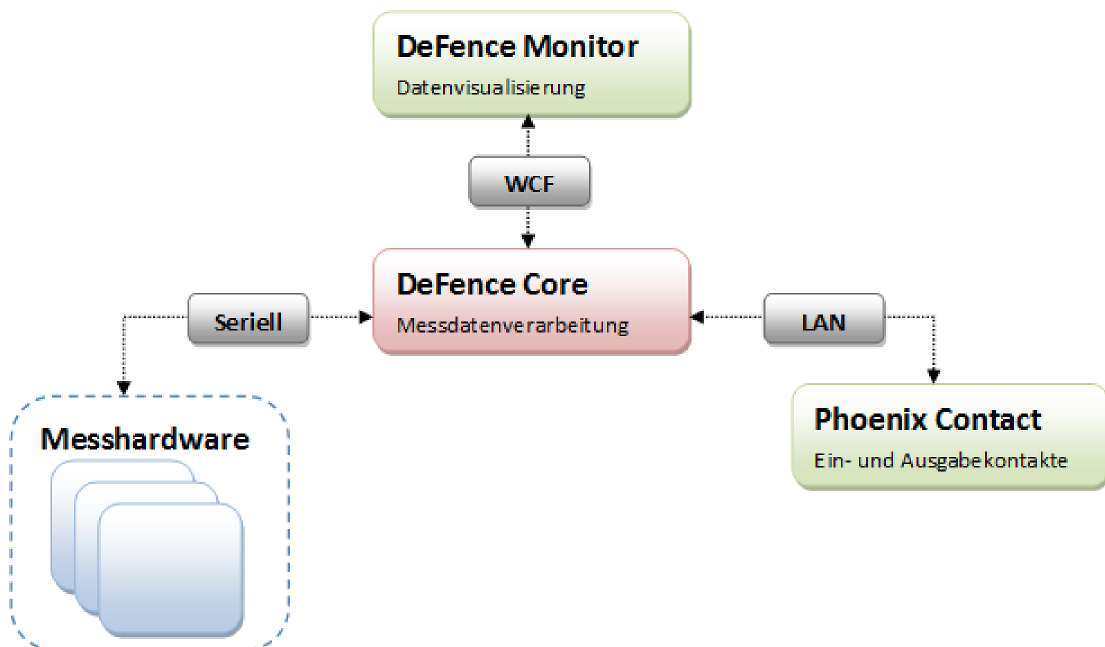

Abbildung 5.1.: Systemübersicht

[1] WCF - Windows Communication Foundation

Sämliche eigenentwickelte Softwarekomponenten wurden unter Verwendung des Microsoft .NET 3.5 Frameworks entwickelt. Im Rahmen dieser Auswertung wird allerdings nur die Anwendung **„DeFence Core"** betrachtet. Hierbei handelt es sich um das zentrale Bindeglied zwischen der Messhardware und dem Datenvisualisierungstool, das vom Endanwender zur Messadatenanalyse genutzt wird.

Die Verbindung zur Messhardware wird über eine oder mehrere serielle Schnittstellen hergestellt. Die grundlegende Verwaltung der seriellen Kommunikation wird mit Hilfe einer gesonderten Programmbibliothek realisiert. Diese Bibliothek stellt eine einheitliche Basis für verschiedene Kommunikationstechnologien und -Schnittstellen zur Verfügung. Die Auswertung vernachlässigt diese Softwarekomponente zur Gewährleistung der Übersichtlichkeit.

Die Anwendung **„DeFence Core"** befindet sich zum aktuellen Zeitpunkt noch in der Entwicklungsphase. Bei der betrachteten Revision der Anwendung handelt es sich um eine Vorabversion, die bereits einige der geplanten Features realisiert. Zu aktuellen Zeitpunkt existieren für die Anwendung **keine** automatisierten Testfälle.

Statistische Werte

Vor Beginn der eigentlichen Auswertung sind mit Hilfe des Tools **Campwood Software SourceMonitor 3.1.5**[2] einige grundlegende statistische Werte ermittelt worden. Diese ermittelten Werte erlauben eine Abschätzung des Umfangs und der Komplexität des verwendeten Prüflings. Aus der Erfassung der statistischen Werte ergeben sich die folgenden Messergebnisse:

- **52** Dateien
- **53** Klassen
- **9761** Codezeilen (inkl. Kommentarzeile)
- Methoden pro Klasse: **5,83**
- Maximale zykl. Komplexität: **19**
- Durchschnittliche zykl. Komplexität: **2,62**
- Maximale Blocktiefe: **8**
- Durchschnittliche Blocktiefe: **2,23**

Die gesammelten Daten beziehen sich ausschließlich auf die Anwendung „DeFence Core" und beinhalten daher keine Daten der referenzierten Bibliotheken.

[2] `http://www.campwoodsw.com/sourcemonitor.html`

5.2. Testläufe

Während der Realisierung der Analyseregeln und der damit zusammenhängenden Erstellung der Testfälle hat sich gezeigt, dass eine Analyseregeln die Beachtung bestimmter Ausnahmen erfordern. Es ist daher damit zu rechnen, dass im Zuge der Auswertung der Analyseregeln gegen den Prüfling weitere Ausnahmen zu Tage treten, die ggf. in das bestehende Regelwerk eingearbeitet werden müssen. Aus diesem Grund wird die Auswertung in mehreren Iterationen durchgeführt.

5.2.1. Erste Iteration

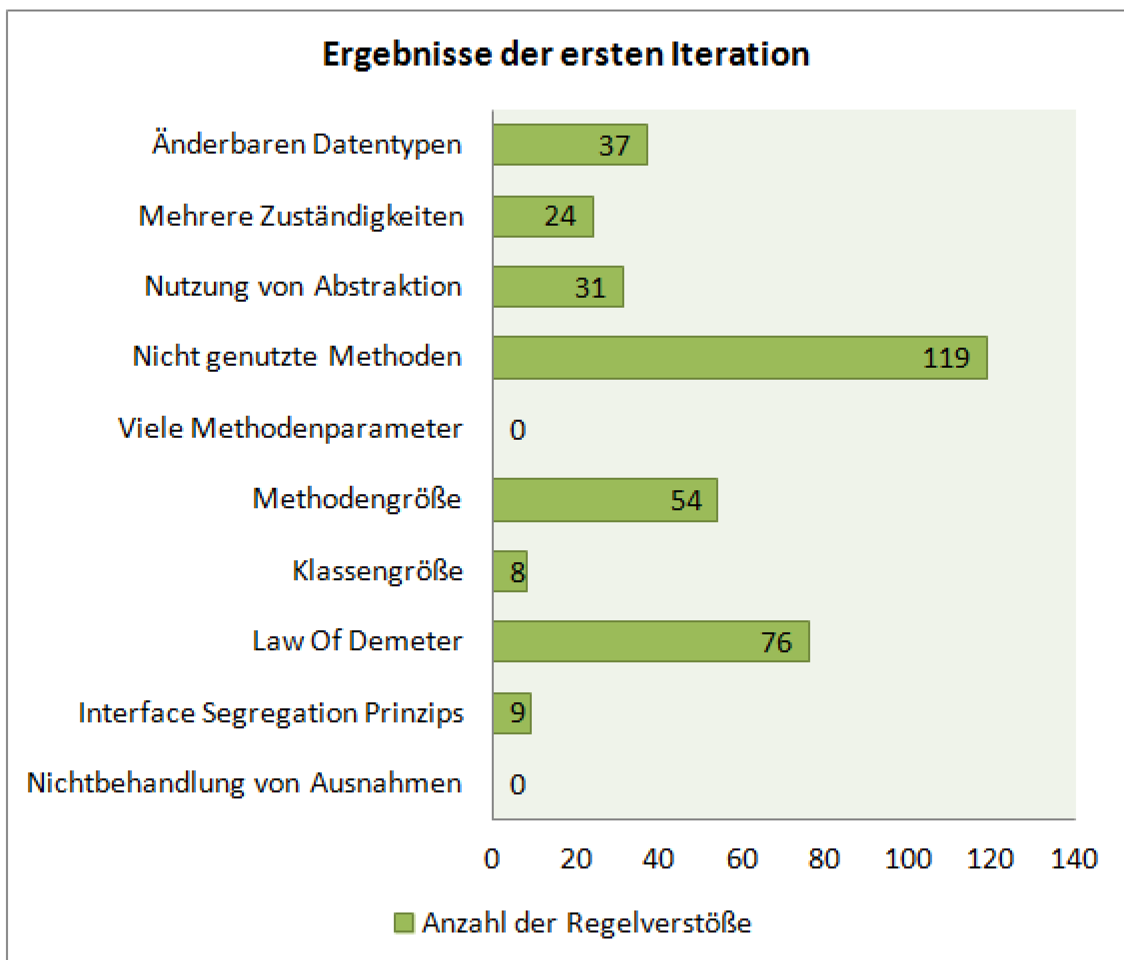

Abbildung 5.2.: Ergebnisse der ersten Iteration

Der erste Testlauf wurde durchgeführt nachdem die sämtliche Tests der Analyseregeln gegen die selbstdefinierten Testfälle zu den erwarteten Ergebnissen geführt haben. Die Durchführung des ersten Testlaufes unter Verwendung des o.g. Prüflings lieferte insgesamt **358** Regelverstöße.

Auffallend hoch ist die Anzahl der nicht genutzten Elemente. Bei genauerer Betrachtung der erkannten Verstöße gegen diese Analyseregel wird deutlich, dass ein Teil der Regelverstöße aufgrund einer bisher nicht aufgefallenen Schwäche innerhalb des FxCop Frameworks entstehen. Wie in 4.4 beschrieben, wird das Zählen von Methodenaufrufen zur Durchführung der Analyse verwendet. Hierfür wird die Klasse `CallGraph` verwendet. Offenbar liefert die Methode `GetCallersFor` allerdings nur explizite Aufrufe einer Methode. Dies führt zu einer fehlerhaften Erkennung von Methodenaufrufen, sofern abstrakte Basisklassen verwendet werden. Im Falle von „DeFence Core“ tritt dieser Fehler innerhalb einer Vererbungshierarchie zutage. Die Klasse `AbstractAnalysis` stellt eine abstrakte Basis für alle konkreten Messdatenanalysmechanismen zur Verfügung. Konkrete Implementierungen der abstrakten Basisklasse sind die Klassen `DispersionAnalysis` und `DriftAnalysis`. Der Zugriff auf Instanzen dieser beiden Klassen erfolgt anwendungsintern immer über die abstrakte Basisklasse `AbstractAnalysis`, was zu einer fehlerhaften Erkennung von Regelverstößen führt.

Wie bereits in der Beschreibung des Prüfungs (siehe 5.1) erwähnt, stellt „DeFence Core“ einer anderen Anwendung aufbereitete Messdaten zur Anzeige bereit. Die Bereitstellung der Messdaten erfolgt mit Hilfe von **WCF**. Zur Realisierung dieses Dienstes ist es erforderlich, einen entsprechendes Service-Objekt innerhalb der Anwendung „DeFence Core“ bereitzustellen, das die Messdaten über öffentlich zugängliche Methoden zur Verfügung stellt. Diese Methoden werden allerdings erst zur Laufzeit aufgerufen und führen daher im Zuge dieser Analyse zu Fehlerkennungen. Zur Umgehung dieser Fehlerkennung ist es empfehlenswert, die entsprechende Klasse mit Hilfe des folgenden Attributs zu versehen. Mit Hilfe dieses Attributs kann die Überprüfung dieser Regel für diesen speziellen Klassentypus übersprungen werden.

Listing 5.1: **Attribut zum Überspringen der Überprüfung**

```
[SuppressMessage("CleanCodeRules.CleanCodeRules", "CC0008")]
```

Auch die Analyseregel **„Einhaltung des Law Of Demeter“** generiert eine überdurchschnittlich hohe Anzahl an Regelverstößen. Hauptgrund hierfür sind die vom Autor strikt eingehaltenen Regelvorgaben, die durch dieses Paradigma vorgegeben werden. Verschiedene Internet-Quellen empfehlen, die vorgegebenen Regeln nur auf Kontroll- nicht aber auf Datenklassen anzuwenden[3]. Der Autor folgte dieser Empfehlung bewusst **nicht**, da auch über Datenklassen

[3] http://clean-code-developer.de/Grüner-Grad.ashx#Law_of_Demeter_2 - Zugriff am 25.10.2011

ungewünschte Abhängigkeiten entstehen können. Viele der erkannten Regelverstöße beziehen sich daher auf verschachtelte Zugriffe auf Datenklassen. Der folgende Ausschnitt zeigt beispielhaft eine Methode der Anwendung, die gegen diese Analyseregel verstößt.

Listing 5.2: **Verletzung des Law Of Demeter**

```
public override void updateSettings()
{
    mDispersionThreshold = DeFence.Properties.↩
       DeFenceCoreSettings.Default.DispersionThreshold;
    mDispersionAnalysisInterval = DeFence.Properties.↩
       DeFenceCoreSettings.Default.DispersionIntervalSize;
    mPreAlarmPercent = DeFence.Properties.DeFenceCoreSettings.↩
       Default.DispersionPreAlarmBarrier;
    mIsEnabled = DeFence.Properties.DeFenceCoreSettings.Default↩
       .DispersionAnalysisEnabled;
}
```

Einige der erkannten Regelverstöße machen allerdings deutlich, dass auch diese Analyseregel eine Überarbeitung erfordert. So erzeugen beispielsweise Zugriffe auf Sprachelemente des .NET Frameworks ebenfalls Regelverstöße. Hierbei handelt es sich allerdings um Elemente, die ohnehin für die gesamte Anwendung zugänglich sind. Durch die bisherige Nichtbeachtung dieses Umstandes wird u.A. das folgende Beispiel als Regelverstoß erkannt.

Listing 5.3: **Zugriff auf Elemente des .NET Frameworks**

```
Console.WriteLine(string.Format("Caught {0} while executing ↩
   callback.", exception.GetType()), exception);
```

Für den Typen `Type`, der durch Aufruf von `exception.GetType()` zurückgegeben wird, existiert keine entsprechende lokale oder klassenglobale Variable. Dies führt zur Verletzung der Vorgaben der Analyseregel. Zur Vermeidung derartiger Fehlerkennungen werden Zugriffe auf Elemente des .NET Frameworks nach dem oben aufgeführten Prinzip als Ausnahme zur Analyseregel hinzugefügt.

Die Ergebnisse der weiteren Analyseregeln werden im Rahmen der folgenden Iteration zusammen mit den Ergebnissen der fehlerkorrigierten Analyseregeln betrachtet.

5.2.2. Zweite Iteration

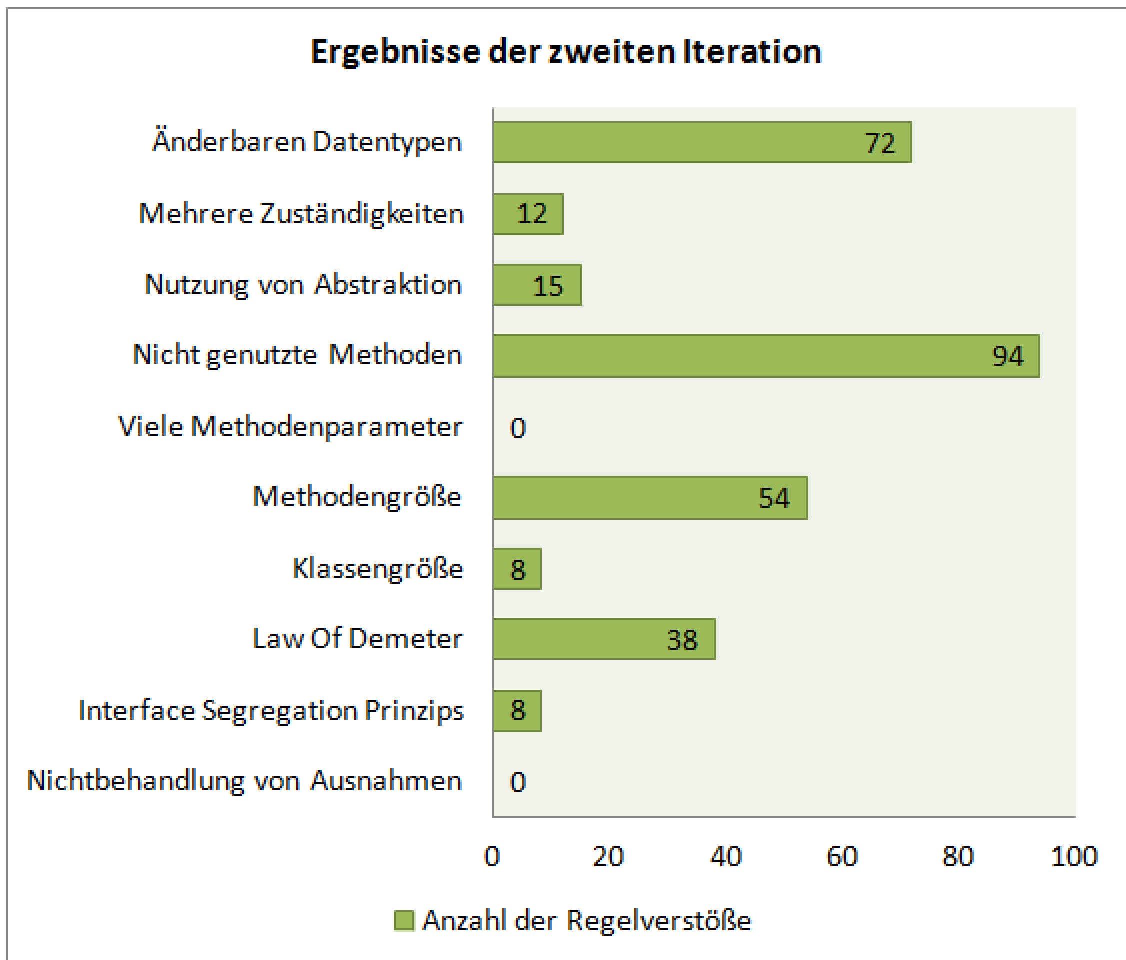

Abbildung 5.3.: Ergebnisse der zweiten Iteration

Vor Durchführung des nächsten Testslaufes wurden die im vorangegangenen Abschnitt identifizierten Fehler innerhalb der Analyseregeln beseitigt. Der zweite Testlauf lieferte insgesamt **301** Regelverstöße, also **57** Regelverstöße weniger, als zuvor. Anhand dieser Zahlen wird deutlich, wie verherend sich selbst kleinste Fehler innerhalb der Analyseregeln auf das Messergebnis auswirken. Die Fehler der Analyseregeln, die während der ersten Iteration zu Tage getreten sind, ließen sich mit einfachsten Mitteln (mitunter durch nur eine zusätzliche Überprüfung) beseitigen. Anhand dieser Erfahrungen wird ersichtlich, wie sich selbst kleinste Fehler innerhalb der Analyseregeln auf das Gesamtresultat und die daraus entstehenden Analyseaufwände für den Softwareentwickler auswirken. Anzumerken ist, dass es sich bei

dem verwendeten Prüfling um eine überschaubare Anwendung handelt. Inwieweit sich die verfälschten Analyseergebnisse bei einer weitaus umfangreicheren Anwendung auswirken, kann an dieser Stelle nicht beurteilt werden.

Wie der Grafik zu entnehmen ist, hat sich die Anzahl der Regelverstöße, die das **Law Of Demeter** betreffen, um 50% reduziert. Die Menge der zu Unrecht als nicht genutzt gemeldeten Elemente hat sich hingegen lediglich um etwa 21% verringert. Bei erneuter Betrachtung der **94** Verstöße, die diese Regel betreffen, stellt sich heraus, dass eine Vielzahl der erkannten Regelverstöße aufrund einer nicht mit in die Analyse einbezogenen Bibliothek entstanden sind. Wie in 5.1 beschrieben, übernimmt diese zusätzliche Bibliothek die Verwaltung der Kommunikation zur Messhardware. Dazu gibt die Bibliothek u. A. den allgemeinen Aufbau von Kommunikationstelegrammen vor. Konkrete Telegramme zur Kommunikation mit der Messhardware werden im Kontext der Anwendung „**DeFence Core**" implementiert und sind somit Bestandteil der Analyse. Die Verarbeitung dieser Telegramme erfolgt allerdings in der Bibliothek, die nicht mit in die Analyse miteinbezogen wurde. Aufgrund der Tatsache, dass es sich hierbei um keinen Fehler in den Analyseregeln handelt, wird diese Häufung an Regelverstößen nicht weiter betrachtet. Alle weiteren Analyseregeln haben plausible Ergebnisse geliefert, die in den nun folgenden Abschnitten genauer betrachtet werden.

Vermeidung von änderbaren Datentypen

Insgesamt wurden für diese Analyseregel **72** Regelverstöße gemeldet. Die deutliche Steigerung der gemeldeten Regelverstöße erklärt sich durch die Umstellung der Heuristik zur Erkennung von Datentypen. Diese sieht nun auch Listen als Datentypen an.

Die Regel bezieht sich auf einzelne Felder einer Datenklasse oder Struktur. Besitzt ein analysiertes Element also mehrere veränderbare Felder, so wird für jedes dieser Felder ein Regelverstoß generiert. Die gemeldeten Regelverstöße beziehen sich auf insgesamt **23** verschiedene Datenklassen, die allesamt von der in 4.7 genannten Heuristik als Solche erkannt wurden.

Vermeidung der Vermischung von Konfiguration und Nutzung

Durch die Änderung der Heuristik zur Erkennung von Datentypen ergibt sich auch bei dieser Analyseregel ein verändertes Analyseergebnis. Statt der im ersten Iterationsschritt gemeldeten **24** Regelverstöße werden nun lediglich **12** Verstöße gemeldet. Zur Veranschaulichung wird nun ein konkreter Regelverstöße im Detail analysiert.

Der Regelverstoß, der betrachtet werden soll, meldet die direkte Verwendung einer Kontrollklasse, die vom .NET Framework bereitgestellt wird. Der Regelverstoß kommt zustande,

weil eine Klasse eine interne Variable vom Typ `System.Timers.Timer` besitzt. Bei diesem Typ handelt es sich laut Heuristik um eine Kontrollklasse, welche nach Vorgabe der Analyseregel nicht klassenintern initialisiert werden darf.

Listing 5.4: **Regelwidrige Konfiguration im Konstruktor**

```
public AnalysisController(MultiplexerModel modelRef)
{
    mModelRef = modelRef;
    mAnalysisList.Add(new DispersionAnalysis());
    mAnalysisList.Add(new DriftAnalysis());
    mDriftTimer = new Timer(mDirftRecalcTime);
    mDriftTimer.Elapsed += new ElapsedEventHandler(↩
        driftTimerElapsedHandler);
}
```

Durch die Initialisierung von `mDriftTimer` im Konstruktor der Klasse `AnalysisController` wird ein Regelverstoß erzeugt. Anhand dieses Beispiels kann anschaulich verdeutlicht werden, warum eine interne Konfiguration der Variable aus Sicht der Softwarequalität als problematisch anzusehen ist. Soll für die o.g. Klasse ein automatisierter Test verfasst werden, muss der Verfasser des Tests immer das Verhalten des Timers beachten. Das Verhalten der Klasse `AnalysisController` kann daher nicht isoliert getestet werden, da für die Variable `mDriftTimer` kein Mock-Objekt erzeugt werden kann.

Nutzung von Abstraktion

An insgesamt **15** Stellen werden konkrete Kontrolltypen anstelle von Schnittstellen verwendet. Die Überprüfung bezieht sich sowohl auf Instanzvariablen, als auch auf Parameter- und Rückgabetypen von Methoden. Die deutliche Verringerung der Regelverstöße gegenüber der ersten Iteration wurde, wie auch schon bei den vorangangenen Regeln, durch die Modifikation der Heuristik zur Erkennung von Datentypen hervorgerufen.

Viele Regelverstöße, die diese Analyseregel meldet, überschneiden sich mit den gemeldeten Verstößen der vorangegangenen Regel. Aus diesem Grund wird an dieser Stelle Bezug auf das Listing 5.4 aus dem vorherigen Abschnitt genommen. Die Klasse `AnalysisController` besitzt eine Instanzvariable `mDriftTimer` vom Typ `System.Timers.Timer`. Dieser Typ wird als Kontrollklasse angesehen und sollte daher idealerweise über eine Schnittstelle angesprochen werden.

Insgesamt **3** Regelverstöße machen darauf aufmerksam, dass der Typ `Object` durch einen abstrakten Basistypen angesprochen werden sollte. Dies ist selbstverständlich nicht möglich, da dieser Typ die Wurzel der .NET Vererbungshierarchie darstellt und von keinem anderen Typen erben kann. Es wird daher eine Ausnahme in die Analyseregel integriert, um den Typen `Object` während der Analyse zu ignorieren.

Umfang von Klassen und Methoden

Die Analyse hat festegestellt, dass **15%** der vorhandenen Klassen, sowie rund **18%** der vorhandenen Methoden die vorgegeben Richtwerte in Hinblick auf ihren Umfang überschreiten. Für Klassen wurden die folgenden Richtwerte für die Analyse verwendet:

- Maximal **9** Instanzvariablen
- Maximal **9** Eigenschaften
- Maximal **9** Methoden

Die Analyse beinhaltet allerdings auch Konstanten, die innerhalb einer Klasse deklariert wurden. Diese Tatsache führt dazu, dass Klassen deren Umfang für den Betrachter als überschaubar eingestuft wird, von der Analyseregel trotzdem als potenziell zu mächtig angesehen wird. Als Gegensatz dazu meldete die Analyseregel aber auch die Klasse `MainController`. Diese Klasse besitzt insgesamt **6** Instanzvariablen und **20** Methoden, wobei allein **3** eine sehr hohe zyklomatische Komplexität aufweisen. Insgesamt ist die Klasse über **800** Zeilen (inkl. Kommentaren) lang. Bei der anschließenden Begutachtung der Klasse stellte sich heraus, dass die Klasse mehrere Verantwortlichkeiten besitzt und der zu erwartende Pflegeaufwand sehr hoch ist.

Methoden werden ab einer Größe von **25** Zeilen als potentiell zu mächtig angesehen. Dabei werden allerdings nicht die Kommentarzeilen, sondern lediglich die Anweisungen einer Methode berücksichtigt. In Anbetracht der gelieferten Ergebnisse erscheint eine nachträgliche Erhöhung dieses Wertes als sinnvoll. Viele Regelverstöße weisen auf Methoden hin, die eine sehr niedrige zyklomatische Komplexität aufweisen und nur wenig Verantwortung besitzen. Eine Optimierung dieser Methoden ist nicht sinnvoll. Aufgrund der vielen unnötig gemeldeten Regelverstöße gehen die tatsächlich übermächtigen Methoden in der Menge der gemeldeten Verstöße unter. Die herausragenste Methode ist `processMultiplexerTelegrams` der Klasse `MainController`. Diese Methode besitzt nicht nur die mit Abstand höchste Komplexität des Programms, sondern überschreitet mit **138** Zeilen (inkl. Kommentarzeilen) das vordefinierte Limit deutlich.

Einhaltung des Interface Segregation Prinzips

Mit insgesamt **8** gemeldeten Regelverstößen verhält sich die Anwendung in Hinblick auf diese Analyseregel eher unauffällig. Die überwiegende Anzahl der gemeldeten Regelverstöße bezieht sich auf die Schnittstelle `IList`. Diese Schnittstelle bietet viele verschiedene Operationen in Hinblick auf Listen an, wobei viele dieser Operationen nicht benötigt werden. Oftmals wird nur die Methode `Add` der Schnittstelle verwendet, was zu einer Verletzung dieser Analyseregel führt.

Um diese Art der Regelverletzung zu ignorieren, werden Schnittstellen, die aus dem .NET Framework stammen, ab sofort von der Analyse ausgeschlossen.

5.3. Bewertung der Ergebnisse

Die durchgeführten Testläufe haben gezeigt, dass die entwickelten Analyseregeln für reell existierende .NET Anwendungen einsetzbar sind. Durch die Analyseregeln wurden viele Design- und Architekturschwächen innerhalb des betrachteten Prüflings aufgedeckt, die nach den Maßstäben des Clean Code Development als problematisch anzusehen sind. Viele erkannte Regelverstöße wiesen auf einen hohen Kopplungsgrad zwischen verschiedenen Klassen der Anwendung hin. Erwähnenswert ist, dass die Vielzahl der erkannten Regelverstöße auf eine schlechte Testbarkeit hinweisen. Viele der erkannten Regelverstöße machen es nahezu unmöglich, automatisierte Testfälle für die einzelnen Klassen der Anwendung zu erstellen.

Die verfälschten Analyseergebnisse des ersten Iterationsschritts haben gezeigt, dass selbst kleinste Konzeptions- und Implementierungsfehler innerhalb der verfassten Analyseregeln massiven Einfluss auf die gemeldeten Resultate haben können. Bei etwa **16%** der gemeldeten Regelverstöße der ersten Iteration handelte es sich um Falschmeldungen, die in Folge einer fehlerhaften Implementierung der Analyseregeln entstanden sind. Derartig verfälschte Messergebnisse erzeugen in einem produktiven Umfeld unnötigen Analyseaufwand für den Softwareentwickler und sind in diesem Fall eher kontraproduktiv als hilfreich.

6. Schluss

6.1. Fazit

Im Rahmen dieser Arbeit wurden mit Hilfe des Software-Toolkits **Micrsoft FxCop** Analyseregeln zur Unterstützung des Clean Code Development konzipiert, realisiert und im Rahmen einer Feldstudie unter Verwendung eines Prüflings getestet. Es wurde gezeigt, dass die formulierten Analyseregeln Probleme und Schwachstellen innerhalb einer .NET Anwendung erkennen können und es dem Entwickler so ermöglicht wird, diese Schwachstellen zeitnah zu beseitigen und somit die innere Qualität der Software zu steigern.

Anhand der realisierten Analyseregeln erhält der Leser einen Eindruck davon, mit welchem Implementierungs- und Testaufwand zu rechnen ist, wenn eigene Regeln erstellt werden sollen. Zudem wurde auf die Problematik bei der Formulierung eigener Analyeregeln hingewiesen. Im Rahmen dieser Arbeit hat sich gezeigt, dass Konzept- und Implementierungsfehler negativ auf das Resultat einer Analyse auswirken und dadurch zusätzlicher und unnötiger Aufwand für den Softwareentwickler entsteht.

6.2. Ausblick

Bei den entwickelten Analyseregeln handelt es sich um den Versuch, möglichst allgemeine Regeln zu definieren, die zur Analyse verschiedenster .NET Anwendungen verwendet werden können. Die Validierung der Analyseregeln erfolgte lediglich gegen selbst definierte Testfälle, sowie eine spezielle Revision eines reellen Prüflings. Um den tatsächlichen Nutzen der Regeln identifizieren zu können, müssen diese im Rahmen einer kontinuierlichen Analyse über einen längeren Zeitraum zum Einsatz kommen. Nur so ließe sich zweifelsfrei nachweisen, dass sich Design- und Architekturschwächen mit Hilfe der Analyseregeln frühzeitig erkennen lassen und so den Entwicklungsprozess einer Anwendung aktiv unterstützen.

A. Inhalt der CD

Die mitgelieferte CD beinhaltet dieses Dokument als **PDF**-Datei. Desweiteren sind folgende Bestandteile auf der CD enthalten.

Quellcode

Der Quellcode der formulierten Analyseregeln sowie der verwendeten Testprojekte. Zum Öffnen der kompletten Solution (*.**sln**) oder eines der einzelnen Projekte (*.**csproj**) wird das **Microsoft Visual Studio 2010** benötigt.

Dokumentation

Die technische Dokumentation des Quellcodes. Die Dokumentation wird im **HTML**-Format geliefert und wurde mit Hilfe der Tools Doxygen 1.7.5.1 und GraphViz 2.28.0 erstellt. Es handelt sich dabei um eine textuelle und grafische Dokumentation der Analyseregeln, die anhand der vorhandenen Quellcodekommentierung sowie der internen Strukturierung der Analysebibliothek generiert wurde. Zum Betrachten der Dokumentation muss die Datei **index.html** in einem Webbrowser geöffnet werden.

Installation

Die vorkompilierten Analyseregeln als Setup-Datei zur Installation in das FxCop-Verzeichnis. Vor der Installation der Analyseregeln ist es erforderlich, **Microsoft FxCop 10.0** zu installieren. Zur Durchführung der Installation werden ggf. lokale Administratorrechte benötigt.

B. Glossar

Assembly

Zusammenfassung kompilierter Sourcecodedateien und weiterer Ressourcen zu einer funktionellen Softwareeinheit. Üblicherweise handelt es sich hierbei um eine ausführbare Datei (*.exe) oder eine Programmbibliothek (*.dll).

Halstead-Metrik

Maß für die Komplexität einer Anwendung, dessen Wert anhand der vorhandenen Operatoren und Operanden ermittelt wird.

Language Integrated Query

Abk. LINQ - Eine in das .NET Framework integrierte Sprache zur einheitlichen und technologieunabhängigen Abfrage von Daten.

Mock-Objekt

Objekt zur Nachstellung des Verhaltens einer anderen Klasse. Mock-Objekte werden beim isolierten Testen einzelner Bestandteile einer Software dazu genutzt, das Verhalten referenzierter Klassen zu simulieren.

Windows Communication Foundation

Abk. WCF - Eine in das .NET Framework integrierte Bibliothek zur Realisierung verteilter Anwendungen.

Zyklomatische Komplexität

Auch McCabe-Metrik - Maß für die Komplexität eines Softwaremoduls, dessen Wert anhand der Verzweigungen innerhalb des Sourcecodes ermittelt wird.

Literaturverzeichnis

[Andrew Hunt 1999] Andrew Hunt, David T.: *The Pragmatic Programmer.* Addison-Wesley, 1999. – ISBN 0-201-61622-X

[Dictus 2010] Dictus, Roy: *Code Contracts By Example.* Website. 2010. – http://www.agilitylux.com/practices/code-contracts-by-example/ - Letzter Zugriff 21.10.2011

[Fowler 1999] Fowler, Martin: *Refactoring - Improving The Design Of Existing Code.* Addison-Wesley, 1999. – ISBN 0-201-48567-2

[Hoffmann 2008] Hoffmann, Dirk W.: *Software-Qualität.* Springer, 2008. – ISBN 978-3-540-76322-2

[Jürgen Dunkel 2003] Jürgen Dunkel, Andreas H.: *Softwarearchitektur für die Praxis.* Springer, 2003. – ISBN 3-540-00221-9

[Kresowaty 2008] Kresowaty, Jason: *FxCop and Code Analysis: Writing Your Own Custom Rules.* Website. 2008. – http://www.binarycoder.net/fxcop/html/index.html - Letzter Zugriff 20.10.2011

[Krysztof Cwalina 2009] Krysztof Cwalina, Brad A.: *Framework Design Guidelines - Conventions, Idioms, and Patterns for Reuseable .NET Libraries.* Addison Wesley, 2009. – ISBN 0-321-54561-3

[Martin 1996a] Martin, Robert C.: *The Dependency Inversion Principle.* Website. 1996. – http://www.objectmentor.com/resources/articles/dip.pdf - Letzter Zugriff 31.09.2011

[Martin 1996b] Martin, Robert C.: *The Interface Segregation Principle.* Website. 1996. – http://www.objectmentor.com/resources/articles/isp.pdf - Letzter Zugriff 31.09.2011

[Martin 1996c] Martin, Robert C.: *The Liskov Substitution Principle.* Website. 1996. – http://www.objectmentor.com/resources/articles/lsp.pdf - Letzter Zugriff 02.10.2011

[Martin 2009] MARTIN, Robert C.: *Clean Code - A Handbook of Agile Software Craftmanship*. Prentice Hall, 2009. – ISBN 0-13-235088-2

[Masak 2010] MASAK, Dieter: *Der Architekturreview*. Springer, 2010. – ISBN 978-3-642-01658-5

[Microsoft 2011] MICROSOFT: *Code Contracts User Manual*. Website. 2011. – http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf - Letzter Zugriff 20.10.2011

[Schneider 2007] SCHNEIDER, Kurt: *Abenteuer Softwarequalität - Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement*. dpunkt.verlag, 2007. – ISBN 978-3-89864-472-3

[Siedersleben 2004] SIEDERSLEBEN, Johannes: *Moderne Softwarearchitektur - Umsichtig planen, robust bauen mit Quasar*. dpunkt.verlag, 2004. – ISBN 3-89864-292-5

[Smith 2009] SMITH, Steve: *Don't Repeat Yourself*. Website. 2009. – http://programmer.97things.oreilly.com/wiki/index.php/Don%27t_Repeat_Yourself - Letzter Zugriff 29.09.2011

[Venners 2003] VENNERS, Bill: *Orthogonality and the DRY Principle*. Website. 2003. – http://www.artima.com/intv/dry.html - Letzter Zugriff 29.09.2011

[Vlissides 2000] VLISSIDES, Erich Gamma Richard Helm Ralph Johnson J.: *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, 2000. – ISBN 0-201-63361-2

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 2. Dezember 2011 Sebastian Ptasik