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

Simon Birk*, **Andreas Rößler**, **Reinhard Schmidt**

Fakultät Informationstechnik der Hochschule Esslingen - University of Applied Sciences

Wintersemester 2015/2016

„Clean Code“ von Robert C. Martin ist derzeit eines der meist diskutierten und einflussreichsten Bücher über Code-Qualität. Darin werden anhand von Fallstudien Best Practices, Heuristiken und Code Smells identifiziert. Der Autor möchte ein Bewusstsein für die Problematiken schaffen, die durch schlecht geschriebenen Code entstehen: unsaubere Programme können mit der Weiterentwicklung zunehmend verwahrlosen und werden so immer anfälliger für Fehler. Die Implementierung neuer Features nimmt dadurch immer mehr Zeit in Anspruch. Ein Code Smell ist ein oberflächliches Symptom einer meist tiefer liegenden Problematik des Systems. In der Regel sind Code Smells leicht zu erkennen, wie zum Beispiel eine Methode, die aus vielen Zeilen Quelltext besteht. Allerdings müssen Code Smells nicht immer ein Indikator für Probleme sein. Manchmal ist es durchaus in Ordnung eine lange Methode zu schreiben - es kommt auf den Kontext an [1].

Das Ziel dieser Arbeit ist, ein Programmpaket von Code-Analysewerkzeugen zu entwickeln, mit deren Hilfe einige von Martin definierte Code Smells identifiziert werden können. Dies geschieht auf Basis der .NET-Compiler-Plattform Roslyn: Diese bietet umfassende Möglichkeiten, um Code zu analysieren und zu generieren.

Mit der Veröffentlichung von Roslyn schlägt Microsoft neue Wege ein: Die meisten Compiler sind als Black Box-Systeme implementiert - Entwickler haben keinen Zugriff auf den Funktionsumfang der einzelnen Compiler-Module. Roslyn bietet Schnittstellen, mit denen Quelltext auf syntaktische und semantische Aspekte untersucht werden kann. Folgende Grafik zeigt den schematischen Aufbau der in vier Komponenten aufgeteilten Compiler-Pipeline, sowie den dazugehörenden APIs:

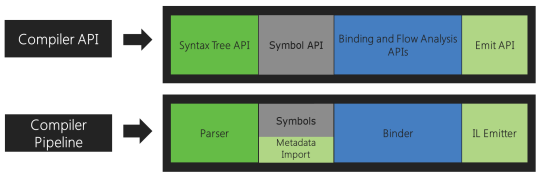

Abbildung 1: Systemarchitektur von Roslyn

Zunächst zerlegt der **Parser** den Quelltext in die Bestandteile der Programmiersprache. Diese werden in einem sogenannten Syntax-Baum abgelegt, der eine vollständige Repräsentation des Codes darstellt. Anschließend werden alle benannten Symbole[1] in einer Symboltabelle abgelegt. Der **Binder** verknüpft nun die im Syntax-Baum gespeicherten Bezeichner mit den Einträgen der Symboltabelle. Das Ergebnis ist ein semantisches Modell des Codes. Schließlich werden aus allen gewonnen Informationen vom **Emitter** die fertigen Assemblies generiert [2].

Mit Hilfe der Roslyn-API wird das Analysewerkzeug dieser Arbeit entwickelt. Das Tool ist in der Lage, einige der gängigsten Smells aus Martins Buch zu identifizieren, darunter auch die folgenden:

Ringabhängigkeiten in Vererbungshierarchien: Beim Entwurf objektorientierter Systeme werden oft Vererbungshierarchien entworfen. Dabei ist es wichtig, dass die Basisklasse unabhängig von den abgeleiteten Klassen ist, damit das Konzept der Basisklasse von den Konzepten der abgeleiteten Klassen getrennt ist. Sind in einer Basisklasse Abhängigkeiten zu einer von ihr abgeleiteten Klasse vorhanden, so ist möglicherweise mit dem Code etwas nicht in Ordnung [3].

Flag-Argumente: Die Übergabe eines booleschen Arguments ist ein klarer Hinweis, dass in der betreffenden Funktion mehr als nur eine Aufgabe erfüllt wird. Mit dem Zustand des Arguments wird über eine Kontrollstruktur bestimmt, welcher Code-Pfad innerhalb der Funktion ausgeführt wird. In den meisten Fällen sollte auf Flag-Argumente verzichtet und stattdessen zwei Methoden geschrieben werden [3].

Auskommentierter Code: Oft wird Code in dem Glauben auskommentiert, er würde nochmals gebraucht werden. Meistens ist dies nicht der Fall. Der auskommentierte Code bleibt bestehen, bläht den restlichen Quelltext unnötig auf und erschwert die Lesbarkeit. Code sollte nicht auskommentiert, sondern gelöscht werden! Sollte der Code nochmals gebraucht werden, kann eine ältere Version aus dem Versionskontrollsystem bezogen werden [3].

*Diese Arbeit wurde durchgeführt bei der Firma IT-Designers GmbH, Esslingen

[1] Namen von Klassen, Methoden, Variablen usw.

Verletzungen des Law Of Demeters: Das LoD besagt, dass ein Modul nichts über das Innere der Objekte wissen sollte, die es manipuliert. Das LoD ist erfüllt, wenn eine Methode `m` einer Klasse `K` nur Methoden der folgenden Komponenten aufruft:

- `K`,
- ein Objekt, das von `m` erstellt wird,
- ein Objekt, das als Argument an `m` übergeben wird,
- ein Objekt, das eine Instanzvariable von `K` ist,

Bei Verstößen gegen das LoD weiß die aufrufende Funktion zu viel über die innere Struktur der Objekte, mit denen sie arbeitet. Dadurch werden vermehrt Abhängigkeiten geschaffen. Verletzungen des LoD werden auch als `Train Wreck` (Zugkatastrophe) bezeichnet, weil die Struktur der aneinander gereihten Aufrufe wie ein zusammengekoppelter Eisenbahnwagen aussieht: `string street = person.Address.Street.ToString();` [3]

Abgesehen von regelbasierten Bewertungen von Code kann dieser mit Hilfe von Metriken auch quantitativ bewertet werden. Es handelt sich dabei um Kennzahlen, die eine Aussage über verschiedene Aspekte des Codes treffen. Mit Metriken kann Quellcode verschiedener Projekte oder Arbeitsstände formal verglichen werden.

Die Kennzahl **Lines Of Code (LOC)** gibt beispielsweise an, aus wie vielen Zeilen Code eine Softwareeinheit besteht. Dabei ist allerdings nicht definiert, wie Kommentare oder Leerzeilen behandelt werden. Neben dem einfach LOC-Maß gibt es noch erweiterte Definitionen, die diese Fragen klären.

Number Of Methods (NOM) gibt an, aus wie vielen Methoden eine Klasse besteht und **Number Of Classes (NOC)** gibt an, aus wie vielen Klassen ein Projekt besteht. Die **zyklomatische Komplexität (CC)** gibt an, aus wie vielen linear unabhängigen Pfaden eine Softwareeinheit besteht. Die Metrik lässt sich wie folgt bestimmen: Die Anzahl möglicher Pfade durch eine Softwareeinheit (entspricht der Anzahl von Kontrollstrukturen und Schleifen) wird mit eins addiert.

Durch Kombination von Metriken lassen sich neue Kennzahlen bestimmen, die qualitative Aussagen über Code treffen: Beispielsweise bestimmt Quotient

$$\frac{NOM}{NOC}$$

die durchschnittliche Anzahl an Methoden je Klasse – ein Anzeichen für die Kohäsion von Klassen. Ein hoher Wert lässt vermuten, dass die betreffende Klasse zu viele Verantwortlichkeiten übernimmt.

Die durchschnittliche Anzahl von Codezeilen pro Methode

$$\frac{LOC}{NOM}$$

beschreibt den Umfang von Funktionen. Hohe Werte lassen auf zu große Funktionen schließen, denn „Funktionen sollten kaum jemals länger als 20 Zeilen sein“ [2].

Die durchschnittliche zyklomatische Komplexität pro Codezeile

$$\frac{CC}{LOC}$$

bestimmt die Dichte von Verzweigungspunkten im Code.

Mit Hilfe dieser drei Quotienten lassen sich Aussagen über den Umfang von Klassen und Methoden sowie über die Komplexität eines Projekts treffen [4].

Um statistische Vergleichswerte für C#-Projekte zu haben, wird im Rahmen dieser Arbeit ein **Metrik-Rechner** erstellt. Mit diesem Tool werden 32 Open Source-Projekte mit insgesamt über 2,5 Millionen Zeilen Code untersucht. Für jedes der Projekte werden die oben genannten Quotienten errechnet, kumuliert und anschließend die Schwellenwerte ermittelt. Die Ergebnisse sind in der folgenden Tabelle zusammengetragen:

Metrik	Gering	Durchschn.	Hoch	Sehr Hoch
CC/LOC	0,16	0,30	0,44	0,66
LOC/Methode	3	8	13	20
NOM/Klasse	3	7	11	17

Abbildung 2: Schwellenwerte von C#-Projekten

Mit den ermittelten Daten lässt sich schnell ein Überblick über eine Softwareeinheit gewinnen und macht sie mit anderen vergleichbar. Werden diese Kennzahlen für das eigene Projekt berechnet, ist anhand der Schwellenwerte schnell erkennbar, ob und bei welchen Klassen und Methoden Refaktorisierungsbedarf besteht.

[1] http://martinfowler.com/bliki/CodeSmell.html
[2] https://github.com/dotnet/roslyn/wiki/Roslyn
[3] Robert C. Martin, Clean Code
[4] Radu Marinescu und Michele Lanza, Object-Oriented Metrics in Practice

Abbildung 1: https://github.com/dotnet/roslyn/wiki/Roslyn