Fakultät für Mathematik und Informatik
Lehrgebiet Programmiersysteme
Prof. Dr. F. Steimann

Spezifikation der statischen Semantik von C# in Refacola

Masterarbeit

vorgelegt an der

Fakultät für Mathematik und Informatik

der FernUniversität in Hagen

Betreuer: Prof. Dr. Friedrich Steimann
Bearbeiter: Frank Rogel (Matrikelnummer: 8628793)

Beginn der Arbeit: 01.01.2014
Abgabe der Arbeit: 01.07.2014

Kurzzusammenfassung

Durch die Refacola erstellte constraintbasierte Refaktorierungswerkzeuge beruhen gegenwärtig zumeist auf den Sprachdefinitionen und Regeln, die für die Programmiersprache Java erstellt wurden. Ziel dieser Arbeit ist es, die statische Semantik von C# in Form einer Refacola-Sprachdefinition und den darauf aufbauenden Refacola-Regeln und Refaktorierungen zu spezifizieren und diese mit NUnit-Tests abzusichern. Grundlage dafür ist die für Java existierende Implementierung mit den vorhandenen JUnit-Tests. Es wird dargestellt, wie die Schnittstelle zwischen C# und Java realisiert wird und beschrieben, wie Microsofts Roslyn für die Umsetzung genutzt wird. Einige für C# erstellte Regeln, die mit Referenzen in die C# 5.0 Spezifikation annotiert sind, werden den Java-Regeln gegenübergestellt. Weiter zeigt die Arbeit beispielhaft die Nutzung einer für C# erstellten Refaktorierung durch eine Erweiterung für Visual Studio.

Abstract

Constraint-based Refactoring-Tools generated by the Refacola are currently mostly based on the language definitions and rules, which were provided for the programming language Java. The goal of this master thesis is to specify the static semantics of C# in form of a Refacola language definition and Refacola rules based on it as well as refactorings and to secure them with NUnit tests. This work is based on the existing Java implementation with the subsisting JUnit tests. It is shown how the interface between C# and Java is realized and described how Microsofts Roslyn is used for the implementation. Some rules are presented and annotated with references to the C# 5.0 Specification and compared to the corresponding Java rules. Further this work exemplifies the use of a refactoring created for C# within an Extension for Visual Studio.

Inhaltsverzeichnis

1. Einleitung

Mit der Refacola[1] wurde am Lehrgebiet Programmiersysteme der FernUniversität in Hagen eine domänenspezifische Sprache sowie ein Framework entwickelt, mit deren Einsatz constraintbasierte Refaktorierungswerkzeuge erstellt und ausgeführt werden können. Grundlage dafür sind eine abstrakte Sprachdefinition, in der die Programmelemente der jeweils betrachteten Programmiersprache dargestellt sind, und Regeln, die semantische Überprüfungen des Compilers bei der Analyse dieser Sprache abbilden. Durch Definition und Anwendung der eigentlichen Refaktorierungen wird aus den Regeln, durch Verknüpfung mit den gegebenen Fakten eines Programmstücks, ein Constraint-System erstellt, welches gelöst wird oder aufgrund einer aufgetretenen Regelverletzung fehlschlägt.

Zum Zeitpunkt des Verfassens dieser Arbeit wird die Refacola als Sprache und Framework erfolgreich für Java, hauptsächlich in Zusammenspiel mit der Programmierumgebung Eclipse, eingesetzt. Für Java ist somit eine komplette Refacola Sprachdefinition mit Regeln und Refaktorierungen vorhanden. Darüber hinaus existiert ein Satz von JUnit-Tests für die einzelnen Regelpakete, der zum Testen und Überprüfen der Korrektheit[2] der Regeln und der Refaktorierung eingesetzt wird.

Aufbauend auf der für Java existierenden Sprachdefinition und den Regeln, wird in dieser Arbeit ein Pendant für C# geschaffen, welches die überprüfbare statische Semantik mit Hilfe der Regeln abbildet und diese Regeln über geeignete Refaktorierungen auf existierende Programme anwendet. Die C# Spezifikation 5.0 dient indes dazu existierende Spezialfälle zu finden und die für C# geltenden Regeln in Refacola abzubilden. Dabei werden die Unterschiede zwischen Java und C# aufgezeigt und die daraus resultierenden Regeln gegenübergestellt.

Die entstandenen Regeln werden, in Anlehnung an die existierenden Testfälle für Java, mit Tests für C#, unter zu Hilfenahme des Test-Frameworks NUnit, abgesichert.

Darüber hinaus wird im Rahmen dieser Arbeit gezeigt, wie erstellte Refacola-Refaktorierungen in die Entwicklungsumgebung Visual Studio (ab Version 11.0) durch eine

1 Abkürzung für Refactoring Constraint Language; nähere Beschreibung dazu in Abschnitt 2.2.

2 „Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.“ (Dijkstra, Edsger W., The Humble Programmer, ACM Turing Lecture, 1972, URL: http://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html (letzter Zugriff: 24.05.2014))

Erweiterung (Visual Studio Extension) genutzt werden können. Dadurch wird es im Rahmen dieser Arbeit möglich, die Zugreifbarkeit von einzelnen Programmelementen zu analysieren und gewünschte Änderungen durch das von der Refacola erstellte Constraint-System prüfen zu lassen. Bei Erfolg wird die mögliche Änderung angezeigt und kann dann in den Programmcode eingebracht werden. Wurde durch das Constraint-System zwar eine Lösung gefunden, diese jedoch nur aufgrund einer (noch) fehlenden Regel, wird der vom Compiler bei der Überprüfung der Kompilierbarkeit des refaktorierten Programmcodes gemeldete Fehler zur Anzeige gebracht. Die Umsetzung dieser Erweiterung ermöglicht es dem Entwickler, die spezifizierten Refacola-Regeln in einer realen Umgebung zu testen und zu nutzen.

Bei allen Programmänderungen durch eine Refaktorierung steht natürlich das Beibehalten der Funktionalität und Kompilierbarkeit des Programms im Vordergrund. Eine Refaktorierung darf niemals ein existierendes Programm in Logik und Bedeutung verändern – es geht lediglich um eine Verbesserung der Softwarequalität, um der Software-Fäulnis und somit Problemen, die durch ständige Weiterentwicklung einer Softwarelösung auftreten können, entgegenzuwirken.

In Kapitel 2 werden die nötigen theoretischen Grundlagen beschrieben und es wird auf einzelne Komponenten eingegangen, die in dieser Arbeit zur Umsetzung genutzt werden.

Kapitel 3 beschäftigt sich zum einen mit den Unterschieden zwischen Java und C#, die sich in den in dieser Arbeit erstellten Regeln für C# widerspiegeln, zum anderen geht es um die Korrektheit der Regeln und deren Überprüfbarkeit.

Mit der nötigen Implementierung für C# setzt sich Kapitel 4 auseinander. Zunächst werden die Architektur und Funktionalität beschrieben, wie aus C# heraus eine Refacola-Refaktorierung zur Ausführung gebracht und die Faktenbasis durch Verwendung der Microsoft Roslyn CTP erstellt werden kann. Weitergehend werden die erstellten Tests betrachtet und gezeigt, wie die Überprüfung der Korrektheit in dieser Arbeit umgesetzt wird. Abschließend wird die entwickelte Visual Studio Erweiterung zum Verringern der Sichtbarkeit einer Klasse oder eines Members auf die minimal nötige Sichtbarkeit erläutert.

Die Betrachtung einiger der für C# erstellten Regeln erfolgt in Kapitel 5. Es wird dabei ein Vergleich mit den für Java existierenden Regeln, mit Bezug auf die Besonderheiten beider Sprachen z.B. bei den Strukturierungskonzepten, erstellt und diese mit den Regeln der C#-

Spezifikation unterlegt.

Kapitel 6 stellt das Fazit dieser Arbeit dar und zeigt im Ausblick auf, welche Verbesserungen noch nötig sind, um die Refacola Sprachdefinition vollständig auf .NET-Seite für C# nutzen und die damit definierten Refaktorierungen effizient einsetzen zu können.

2. Theoretische Grundlagen

Anfang der 90er Jahre wurde erstmals von Refaktorierungen (engl. Refactorings) als einem Teil der Softwareentwicklung gesprochen. Dabei stand auch die Automatisierung der Refaktorierungen im Vordergrund, welche geeignete Werkzeuge, die durch den Programmierer genutzt werden können, voraussetzt – die Refacola bietet ein Rahmenwerk zur Erstellung geeigneter, sprachunabhängiger[3] Refaktorierungswerkzeuge. Eine Analyse des abstrakten Syntaxbaumes (engl.: Abstract Syntax Tree, kurz: AST) stellt die Grundlage zur Bereitstellung von Fakten über ein gegebenes Programm dar, des Weiteren müssen Änderungen wieder in den AST und damit den Programmcode eingebracht werden – mit dem Projekt Microsoft Roslyn wurde eine API geschaffen, mit der Informationen aus dem Compiler und dessen zur Verfügung stehenden Syntaxbaum abgerufen und Veränderungen wieder zurückgeschrieben[4] werden können.

Im Folgenden wird auf diese Grundlagen zum Verständnis der Arbeit näher eingegangen.

2.1 Refaktorierung von Programmcode

Eine Refaktorierung eines existierenden Software-Systems beschreibt den Prozess der Veränderung der internen Struktur, welche die Verbesserung der Qualität des Programmcodes zum Ziel hat.[5] Diese Verbesserung der Qualität besteht im Wesentlichen aus einer Optimierung der Lesbarkeit und Verständlichkeit von Methoden, Klassen und Hierarchien einer Software, der besseren Erweiterbarkeit und der Vermeidung von Redundanz. Oberstes Ziel bei allen Refaktorierungen ist es, das beobachtbare Verhalten des Software-Systems keinesfalls zu verändern.[5] Es gilt also zum einen die Kompilierbarkeit nach einer oder mehrerer Änderungen des Codes durch eine Refaktorierung zu gewährleisten, zum anderen das beobachtete Verhalten und die Funktionalität beizubehalten, sodass das Programm vor und nach der Refaktorierung dieselben Ergebnisse erzielt – dies hoffentlich jedoch effizienter und verständlicher.

3 Die Sprachunabhängigkeit der abzubildenden Sprache ist durch die Definition der Sprache, dessen semantischer Regeln und Refaktorierungen in einer domänenspezifischen Sprache gegeben.

4 Ein SyntaxTree und dessen SyntaxNodes sind bei Roslyn stets „immutable“ - somit wird nicht der eigentliche SyntaxNode verändert, sondern immer eine Kopie des Nodes mit den gewünschten Änderungen erzeugt. Dazu genaueres in 4.2.2.

5 vgl. [FOW02], Seite 9, 51

Fowler beschreibt in seinem Buch, als eine Art Katalog, ausführlich die Einzelschritte von Refaktorierungen, die nötig sind, um z.B. eine Methode in einer Klasse eines Projekts umzubenennen. Aufbauend auf diesen noch einfachen und kleinen Änderungen werden komplexe Refaktorierungen möglich, die durch das schrittweise Ausführen einer Reihe von kleineren Anpassungen zu strukturierten, disziplinierten und dadurch nachvollziehbaren Veränderungen führen. Durch diese klare Strukturierung wird das Risiko minimiert, eine funktionsfähige Software anzupassen und zu verbessern, jedoch gleichzeitig Fehler einzubauen, die im Nachhinein schwer zu beheben und nachzuvollziehen sind.[6]

Ein Vorteil eines solchen Katalogs von Refaktorierungen und der schrittweisen Anleitung zur Überführung von Code hin zu einem auch in Zukunft verständlichen, wartbaren und erweiterbaren Programmcode ist es, die für einfache Refaktorierungen nötigen Einzelschritte zu implementieren und automatisch durch Werkzeuge innerhalb einer Entwicklungsumgebung ausführen zu lassen. Somit können auch komplexe Refaktorierungen durch das nacheinander Ausführen von kleinen nachvollziehbaren Änderungen ermöglicht werden. Dadurch wird der Entwickler in seiner täglichen Arbeit unterstützt und entlastet, indem eine Integration des Refaktorierens in den eigentlichen Entwicklungsprozess möglich wird, welches gerade innerhalb der agilen Softwareentwicklung z.B. mit Scrum als eine Art Daueraufgabe angesehen wird, bei der während jeder Erweiterung oder Anpassung gleichzeitig auf mögliche Refaktorierungen geachtet wird, die dann entweder direkt mit eingebracht oder über eine Backlog gesammelt werden.[7] Durch geeignete automatisierte Refaktorierungsvorgänge innerhalb einer Entwicklungsumgebung fällt für den Programmierer eine große Barriere bei der Durchführung von Refaktorierungen.

Ein Vorreiter auf diesem Gebiet der Automatisierung stellt der Refactoring Browser für Smalltalk dar, der von Brant und Roberts erstellt wurde und unter anderem neben einem RenameClass/Method auch z.B. ein ExtractMethod anbietet, um selektierten Code in eine eigene Methode auszulagern, oder eine äquivalente Methode zu nutzen.[8] Bis heute gibt es lediglich im Bereich der Programmiersprache Java die Entwicklungsumgebungen Eclipse und IntelliJ IDEA, die einen relativ großen Umfang an integrierten Refaktorierungswerkzeugen

6 vgl. [FOW02]

7 vgl. [COH10], „Ron Jeffries sagt: 'Bei der agilen Vorgehensweise muss das Design zu Anfang einfach sein und sich dann weiterentwickeln. Die Maßnahme dazu ist das Refactoring.'“ (aus [COH10], Seite 189)

8 vgl. [BraRo]

als Grundausstattung enthalten. Im .NET-Bereich existieren innerhalb von z.B. Visual Studio lediglich ein paar Refaktorierungen, die dann durch Plugins (z.B. ReSharper) erweitert werden können.

Diese Arbeit zeigt im Verlauf, wie mit der Refacola erzeugte Refaktorierungswerkzeuge auf .NET-Seite eingebunden und genutzt werden können und stellt diese dem Entwickler als Erweiterungen der IDE zur Verfügung. Zunächst jedoch wird die von der Refacola genutzte constraintbasierte Refaktorierung betrachtet.

2.1.1 Constraintbasierte Refaktorierung

Mit Hilfe von Constraints, also Bedingungen, die für eine Programmiersprache eines Programms erfüllt sein müssen, wird bei der constraintbasierten Refaktorierung sichergestellt, dass die syntaktische Korrektheit und die Semantik des Programmcodes nach einer Refaktorierung erhalten bleiben.

Innerhalb der Constraints kommen sogenannte Constraint-Variablen zum Einsatz, die Eigenschaften eines Programms repräsentieren, die durch eine Refaktorierung verändert werden können. Jede Constraint-Variable besitzt einen Wertebereich, der deren mögliche Wertbelegung angibt; so hat z.B. die Variable „Accessibility“ den definierten Wertebereich {private, protected, protected internal, internal, public}. Zusammen bilden nun Constraint-Variablen, deren Wertebereiche und die eigentlichen Constraints ein Constraint-Satisfaction-Problem (CSP), dessen Lösungsraum eine Menge von Refaktorierungen ist, die auf dem Code zu einer gültigen Änderung führt.[9]

Um nun für ein betrachtetes Programm das CSP aufzustellen und zu lösen, ist es erst einmal nötig, Informationen über den Code zu erhalten. Dazu dient der AST des Programms, aus dem Fakten (program queries[9]) zu den einzelnen Programmelementen gesammelt werden können, z.B.: es existieren zwei Klassen C_1 und C_2, C_2 ist Sub-Klasse von C_1. Als Constraint ist z.B. für C# gegeben, dass eine abgeleitete Klasse (hier C_2) die Sichtbarkeit der vererbenden Klasse (hier C_1) nicht erhöhen darf.[10] Dies setzt erst einmal voraus – es wird davon ausgegangen, dass C_1 und C_2 top-level Klassen sind –, dass die Sichtbarkeit beider Klassen entweder public,

9 vgl. [STvP12]

10 vgl. [CSharp5], „§ 3.5.4 Accessibility constraints: - The direct base class of a class type must be at least as accessible as the class type itself“

internal oder „none“ (also kein Zugriffsmodifizierer gesetzt) ist. Hier ist zu beachten, dass die Sichtbarkeit die eigentliche Constraint-Variable ist, die über das CSP gelöst werden soll. Sind die Fakten und Constraints, die zur Anwendung kommen müssen, gefunden, können so genannte Constraint-Regeln[11] aufgestellt werden.

Allgemein sind diese von folgender Form:

$$\frac{\text{program queries (Fakten)}}{\text{constraints (Regeln)}}$$ [11]

Abbildung 1: Allgemeine Form einer Constraint-Regel

Für das Beispiel der Sichtbarkeit bei Vererbung ergibt sich folgende Form:

$$\frac{\text{sub}(C_2, C_1)}{\text{accessibility}(C_1) \geq \text{accessibility}(C_2)}$$ [12]

Abbildung 2: Form einer angewendeten Constraint-Regel

Soll jetzt durch eine Refaktorierung die Sichtbarkeit (Constraint-Variable) einer der Klassen C_1 oder C_2 verändert werden, muss die obige Constraint-Regel (und alle die für die Refaktorierung zu betrachtenden Regeln) nach Veränderung der Constraint-Variablen innerhalb des CSP lösbar sein. Möglicherweise kann das CSP nur dadurch gelöst werden, dass eine weitere Constraint-Variable verändert wird. Letztendlich müssen alle Constraint-Regeln des CSPs erfüllt sein, damit ein Refaktorierung erfolgreich ist und durchgeführt werden kann, sonst ist eine Refaktorierung der gewünschten Eigenschaft eines Programmelements auf einen bestimmten Wert nicht möglich. Es muss beachtet werden, dass die initiale Wertbelegung der Constraint-Variablen bereits eine Lösung des CSP darstellt.[11]

Sind die Regeln einer Programmiersprache definiert und Fakten über die Programmelemente eines Programms vorhanden, wird das zu lösende Constraint-Satisfaction-Problem gebildet, werden also die benötigten Constraint-Regeln erzeugt und die Constraint-Variablen entsprechend belegt. Das Refacola-Framework übernimmt diese Aufgabe und bedient sich eines Constraint-Solvers zum Lösen des CSPs.

11 vgl. [STvP12]

12 Zur Vereinfachung der Darstellung wurde in der Constraint-Rule eine Prüfung der Sichtbarkeit mit $\geq$ angewendet. In C# kann für die Zugriffsmodifizierer lediglich eine Halb-Ordnung angegeben werden, weshalb in den eigentlichen Refacola-Regeln eine explizite Prüfung vorgenommen werden muss. (siehe 3.1)

2.2 Refacola – die Refactoring Constraint Language

Die Refacola ist zum einen eine domänenspezifische Sprache, zum anderen ein Framework zur Generierung von Constraint-Regeln, die in einer constraintbasierten Refaktorierung Anwendung finden. Die domänenspezifische Sprache der Refacola besteht aus drei Sprachmodulen, die zur Abbildung einer betrachteten Programmiersprache und zur Definition von Refaktorierungen genutzt werden. Die drei Sprachmodule dienen dazu:

- die Elemente einer Programmiersprache zu definieren
- die für die Sprache geltenden Regeln abzubilden
- die Refaktorierung zu spezifizieren

Die Sprachelemente der zu betrachtenden Ziel-Programmiersprache werden mit ihren benötigten Eigenschaften in einer Subtypen-Hierarchie definiert. Jede Eigenschaft eines definierten Refacola-Typs besitzt einen Wertebereich innerhalb einer Domäne, die entweder, wie z.B. bei der Sichtbarkeit, durch eine Aufzählung von möglichen Werten gegeben ist, oder auf bereits definierte Programmelemente eingeschränkt ist. Zusätzlich zu den Programmelementen, den Eigenschaften und deren Wertebereichen werden Anfragen definiert, die die eigentlichen Fakten über ein existierendes Programm darstellen.[13] Der grobe Aufbau einer Sprachdefinition wird nun exemplarisch gezeigt.

Listing 1: Exemplarische Darstellung einer Refacola-Sprachdefinition

```
1   language NameDerSprache
2   kinds
3         abstract ProgrammElement <: ENTITY
4         Element <: ProgrammElement { eigenschaft1 }
5         SubElement <: Element { eigenschaft2, eigenschaft3 }
6
7   properties
8         eigenschaft1 : Identifier
9         eigenschaft2 : Elem
10        eigenschaft3 : WertDomäne
11
12  domains
13        Elem = [ ProgrammElement ]
14        WertDomäne = { wert1, wert2, wert3 }
15
16  queries
17        istSubElementVon(sub: ProgrammElement, super: ProgrammElement)
```

13 vgl. [SKP11]

Eine Refacola-Sprachdefinition, wie in Listing 1 exemplarisch dargestellt, besitzt zuallererst einen Namen, der die Sprachelemente innerhalb der Regeln und Refaktorierungen eindeutig einer Sprache zuordnet. Die 'kinds' spezifizieren die vorhandenen Sprachelemente mit deren Eigenschaften. Diese Eigenschaften werden unter 'properties' mit den für sie geltenden Wertebereichen erstellt. Das Schlüsselwort 'domains' leitetet die Definition dieser Domänen für die Eigenschaften ein. Eine Domäne kann entweder auf einzelne Sprachelemente beschränkt sein, wie z.B. die Domäne 'Elem'. Wie im Fall der 'WertDomäne' können aber auch explizit Werte angegeben werden; so ist z.B. die Sichtbarkeit innerhalb der Sprachdefinition auf solch eine Wertdomäne festgelegt. Mit 'queries' werden letztendlich Fakten innerhalb eines Programms repräsentiert, die dann im Bedingungsteil der erstellten Regeln zur Anwendung kommen.

Ist nun die abzubildende Programmiersprache mit allen benötigten Programmelementen spezifiziert, können unter Verwendung dieser Elemente Regeln in Refacola definiert werden, die syntaktische und semantische Bedingungen der Programmiersprache darstellen. Die Prüfung dieser Bedingungen wird normalerweise in der Analysephase des Compilers durchgeführt, und sorgt bei Anwendung einer Refaktorierung aus Refacola für die Einhaltung der Kompilierbarkeit. Diese Regeln geben also die Constraints an, die dann bei Anwendung einer Refaktorierung als Grundlage zur Erstellung des CSP dienen. Die Regeln können je nach Prüfungsbereich also in Regeln für die Namensgebung oder in Regeln für die Sichtbarkeit usw. eingeteilt werden. Im Folgenden Listing 2 wird ein exemplarischer Regelsatz für die in Listing 1 (siehe Seite 10) dargestellte Refacola-Sprachdefinition verdeutlicht.

Listing 2: Exemplarische Darstellung eines Refacola Regelsatzes

```
18  import "Example.language.refacola"
19
20  ruleset BeispielRegeln
21  language NameDerSprache
22  rules
23
24  EineRegel
25  for all
26      sub : NameDerSprache.Element
27      super : NameDerSprache.Element
28  do
29      if
30          NameDerSprache.istSubElementVon(sub, super)
31      then
32          sub.eigenschaft1 != super.eigenschaft1
33  end
```

Nach dem Import der Sprachdefinition und erfolgter Definition des Regelsatzes ('ruleset') mit der dazu verwendeten Sprache ('language'), werden dessen Regeln ('rules') definiert. Eine Regel gibt zuerst an, welche Sprachelemente betrachtet werden sollen ('for all') und definiert für diese eine Variable zur weiteren Verwendung. Im Bedingungsteil der Regel wird geprüft, ob die Regel auf die in der Faktenbasis gefundenen Elemente angewendet werden darf. Ist dies der Fall, wird für die Sprachelemente eine Constraint-Regel generiert und diese dem CSP hinzugefügt.

Zur Erstellung der eigentlichen Refaktorierung werden nun die erstellte Sprachdefinition und die Regeln importiert und bei Anwendung zur Lösung des CSPs genutzt. Innerhalb der Refaktorierung wird spezifiziert, welche Eigenschaften der Programmelemente der Sprache angepasst werden sollen ('forced changes') und welche Änderungen es zusätzlich geben darf ('allowed changes'). Bei der zweiten Kategorie, den erlaubten Änderungen, müssen Werte angegeben werden, die zur Änderung benutzt werden dürfen. Eine exemplarische Refaktorierung zur Darstellung des Aufbaus wird nun in Listing 3 gezeigt.

Listing 3: Exemplarische Darstellung einer Refacola Refaktorierung

```
34   import "Example.language.refacola"
35   import "Example.ruleset.refacola"
36
37   refactoring RenameField
38   languages NameDerSprache
39   uses BeispielRegeln
40
41   forced changes
42        eigenschaft1 of NameDerSprache.Element as NeueEigenschaft1
43   allowed changes
44        eigenschaft2 of NameDerSprache.SubElement
45                                   {initial, NeueEigenschaft1 @forced, new}
```

Für eine detaillierte Beschreibung und Bedeutung aller Schlüsselwörter der Refacola sei auf [SKP11] (Kapitel 5) verwiesen.

Aus der deklarativ erstellten Sprach-Definition, den Regelsätzen und den Refaktorierungen, wird durch die Refacola unter Verwendung der Frameworks Xtext[14] und Xpand/Xtend[15] Java Code generiert, der die definierten Refaktorierungen als Klassen zur Nutzung bereitstellt. Die

14 Xtext ist ein Framework zur Entwicklung von (domänenspezifischen) Sprachen und ist ein Teil des Eclipse Modeling Frameworks (EMF) (http://www.eclipse.org/Xtext/)

15 Xtend ist ebenfalls Teil des EMF (http://www.eclipse.org/xtend/)

Umsetzung der in Refacola deklarierten Definitionen in Java-Quellcode stellt die Grundlage dar, um eine Refaktorierung zu nutzen und letztendlich das Refaktorierungswerkzeug zu erstellen.

Die Voraussetzung der Nutzung des eigentlichen Refaktorierungswerkzeugs ist zum einen, aus einem existierenden Programmcode eine Faktenbasis zu erstellen, die durch Analyse des ASTs und Erzeugung der benötigten Objekte (Java-Objekte, generiert aus den Elementen der Refacola-Sprachdefinition) realisiert wird. Zum anderen müssen die generierten Änderungen, die eine gültige Lösung des erstellten CSPs für eine Refaktorierung darstellen, in den Programmcode wieder zurückschreiben – also Programmelemente verschieben, löschen oder eine Änderung z.B. des Zugriffsmodifizierers vornehmen. Dies wird durch eine Abfrage- und Rückschreib-Komponente realisiert, die entweder direkt die benötigten Java-Objekte erstellt oder die Informationen über den Austausch von Dateien (Faktenbasis / Changeset) weitergibt.

Die folgende Grafik stellt schematisch die Refacola und die beteiligten Komponenten dar. Dabei sind die Informationen über das zu refaktorierende Programm getrennt dargestellt, um zu zeigen, dass die für die Constraint-Generierung von der Refacola benötigten Fakten und die Ergebnisse der Refaktorierung über den Austausch von Dateien realisiert werden kann. Diese Methode des Datenaustauschs wird auch im Rahmen dieser Arbeit genutzt.

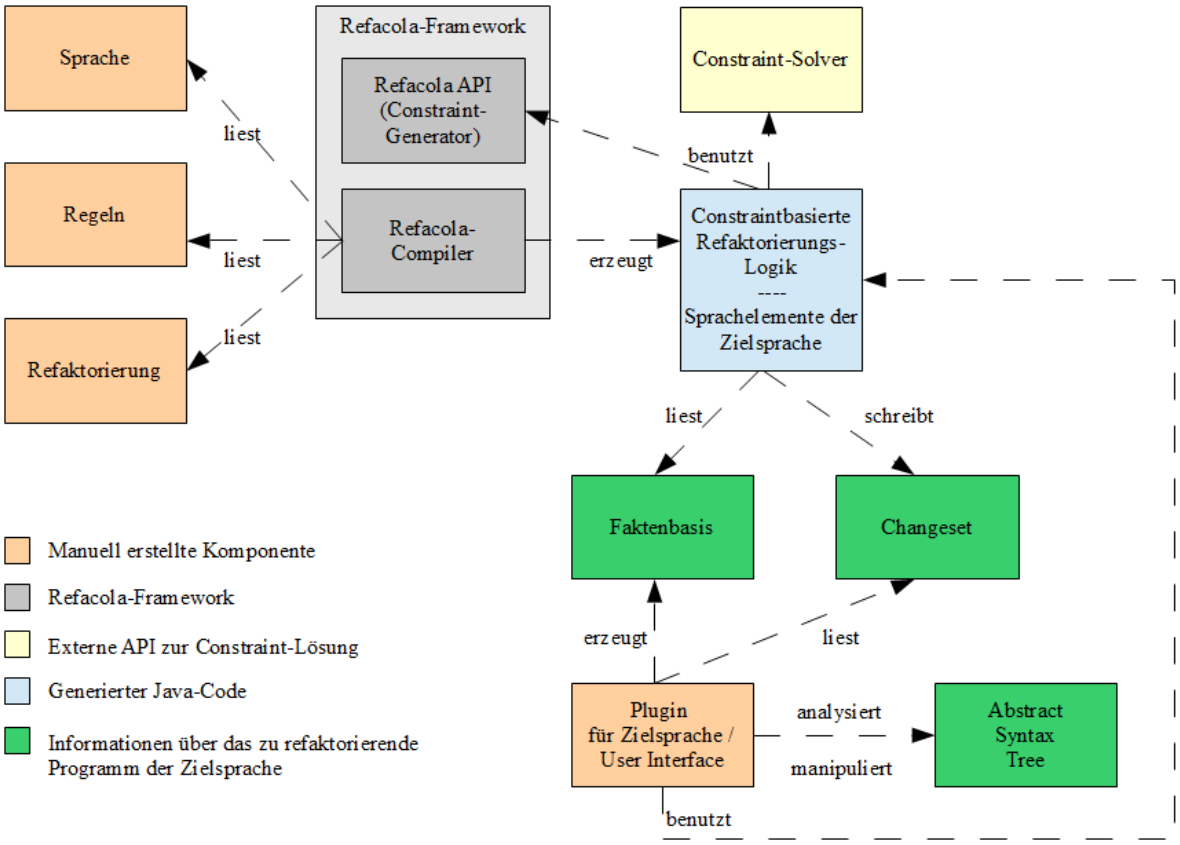


Abbildung 3: Darstellung der Refacola und beteiligter Komponenten (eigene Darstellung)

Das Schaubild zeigt das Zusammenspiel der Refacola-Komponenten: Aus der definierten Sprache, den Regeln und der Refaktorierung erzeugt der Refacola-Compiler die constraintbasierte Refaktorierungslogik (das Werkzeug) und die Klassen der Sprachelemente. Das Werkzeug bekommt die eingelesene Faktenbasis und erzeugt durch die Refacola API das CSP, welches über den Constraint-Solver gelöst wird. Die mögliche Refaktorierung wird in das Changeset geschrieben, welches durch das Plugin der Zielsprache in den AST und somit den Programmcode eingebracht wird.

Wie bereits erwähnt, wird der Datenaustausch im Rahmen dieser Arbeit über Dateien realisiert. Dies erfordert natürlich Dateisystemzugriffe, die möglicherweise Performance-Verschlechterungen mit sich bringen können, sowie Konfigurationen für den Ablageort der Dateien und Zugriffsberechtigungen auf diesen. Denkbar sind weiter einige Ansätze, welche die Nutzung der Refacola aus .NET heraus vereinfachen würden. Auf diese Überlegungen wird im Ausblick dieser Arbeit (in Kapitel 6) eingegangen.

Nach dem Überblick über die Refacola beschreibt das folgende Kapitel nun die Microsoft Roslyn API, die in dieser Arbeit genutzt wird, um zum einen ein existierendes C# Programm zu analysieren und die Faktenbasis bereitzustellen, zum anderen, um die von der Refacola erzeugten Änderungen über das Changeset einzulesen und den Programmcode zu refaktorieren.

2.3 Microsoft Roslyn CTP – der Compiler als API

Die Community Technology Preview (CTP) der Microsoft Roslyn API wurde im September 2012 von Microsoft herausgegeben, um interessierten Entwicklern einen Überblick über deren Funktionalität und die neuen Möglichkeiten im Bereich der Metaprogrammierung zu geben. Ein Release-Termin war mit Beginn dieser Arbeit nicht veröffentlicht, weshalb sich die Arbeit auf die verfügbare Microsoft Roslyn September 2012 CTP (Version v.3) bezieht.[16] Im Folgenden werden der bisherige C#-Compiler und die Möglichkeiten von Roslyn beschrieben und dabei auf die in dieser Arbeit verwendeten Funktionen eingegangen.

2.3.1 Der C# .NET-Compiler bisher

Der C# .NET-Compiler als Teil des .NET-Frameworks übersetzt den in C# geschriebenen Programmcode in einen standardisierten Zwischencode. Dieser Zwischencode, die Common Intermediate Language (CIL), wird von der Laufzeitumgebung Common Language Runtime (CLR) ausgeführt. Ziel dieser Kompilierung in die Intermediate Language ist es, den Entwicklern aus verschiedenen Hochsprachen des .NET-Framework (z.B. VB.NET, F#) zu ermöglichen, Bibliotheken, die in einer Sprache geschrieben wurden, aus einer anderen Sprache heraus zu nutzen, die dann erst durch die Laufzeitumgebung in systemeigenen Programmcode für verschiedene Plattformen übersetzt werden. Es ist somit eine Interoperabilität gegeben, für die das allgemeine Typsystem, das Common Type System (CTS), innerhalb der CLR genutzt wird.

16 Während des Verfassens dieser Arbeit, wurde im April 2014 Roslyn von Microsoft als End-User-Preview und Open-Source-Projekt veröffentlicht (http://roslyn.codeplex.com/ letzter Zugriff: 24.05.2014)). Die gesamte Implementierung für diese Arbeit beruht jedoch auf der älteren Roslyn CTP, weshalb im Rahmen der Arbeit eine Umstellung auf die neuere Version nicht möglich ist.

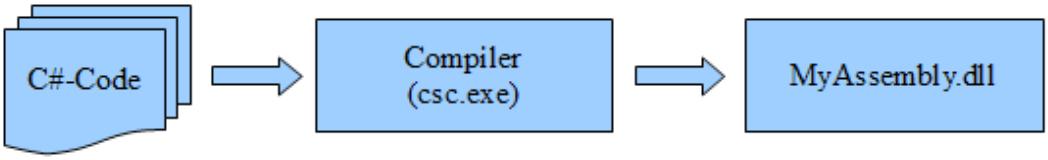


Abbildung 4: Darstellung des Kompilierungsprozesses (eigene Darstellung in Anlehnung an [HaBo13] Seite 288)

Das bisherige Kompilieren einer Anwendung, welches in Abbildung 4 dargestellt ist, erfolgt prinzipiell durch einen einfachen Aufruf des Compilers, der den geschriebenen Programmcode als Eingabe bekommt, intern syntaktische und semantische Prüfungen und Optimierungen vornimmt. Letztendlich wird der Programmcode in die CIL übersetzt und ein Assembly erzeugt, das über die CLR ausgeführt werden kann oder durch andere Programme genutzt wird. Dieser Ablauf ist für einen Entwickler nicht transparent – der Compiler ist im Prinzip eine Blackbox, in die Programmcode hineingegeben wird und dann eine Assembly als Ausgabe erzeugt wird. Die komplexe Architektur des Compilers ist somit erst einmal verborgen. Für einen Entwickler ist diese interne Architektur für die Programmierung zunächst nicht relevant, jedoch besitzt der Compiler detaillierte Informationen über das eingegebene Programm, welche z.B. im Rahmen der Meta-Programmierung sinnvoll genutzt werden können. Diese Informationen gehen jedoch nach der Kompilierung verloren und können nicht weiter genutzt werden.

Mit Hilfe der Microsoft Roslyn API wird sich der Zugriff auf die Informationen des Compilers grundlegend verändern, da diese nun anhand von Objekt-Modellen abgefragt und genutzt werden können. Im Folgenden wird die Roslyn API beschrieben.

2.3.2 Die Zukunft des .NET-Compilers[17]

Mit Microsofts Roslyn-Projekt wird im .NET-Bereich der Compiler aus dem Status einer Blackbox herausgehoben, um die ganze Fülle an Informationen, die der Compiler über den zu kompilierenden Programmcode besitzt, für den Entwickler verfügbar zu machen. Somit ist es möglich, dass Tools, wie z.B. Refaktorierungen und Code-Analysen sowie der Entwickler selbst, diese Informationen nutzen und einsetzen können. Um dies zu realisieren, wird mit

17 vgl. [ROS12]

Roslyn eine Programmierschnittstelle geschaffen, die Code-Informationen als eine Art Service für den Programmierer bereitstellt und diese innerhalb von Anwendungen nutzbar macht. Roslyns Architektur ist unterteilt in drei Schichten, die jeweils eine oder mehrere APIs beinhalten. In folgender Abbildung sind diese Schichten dargestellt.

Abbildung 5: Die drei API-Schichten von Roslyn (aus [ROS12])

Die Editor Services API stellt alle Funktionen bereit, die innerhalb der Visual Studio IDE angeboten werden, z.B. IntelliSense, Refaktorierungen und Code-Formatierung. Diese Schicht stellt die Schnittstelle für den Entwickler dar, um Visual Studio durch eigene Werkzeuge zu erweitern. Die Editor Services Schicht nutzt die darunter liegenden Services APIs, welche die Schnittstelle bilden, um Informationen über eine Solution und die darin existierenden Projekte zu erhalten. Alle Informationen einer kompletten Solution werden durch ein einziges Objekt-Modell bereitgestellt, welches direkten Zugriff auf die von den Compiler APIs erzeugten Objekt-Modelle bietet. Zusätzlich bietet die Services API Programmierschnittstellen zur Code-Generierung an, um z.B. Klassen oder Methoden zur erzeugen. In der Schicht der Compiler APIs, die von den Services APIs genutzt wird, existieren die Programmierschnittstellen, die sowohl syntaktische als auch semantische Informationen bereitstellen, welche innerhalb der einzelnen Phasen der Kompilierung gesammelt werden.

Um Informationen abzufragen, stehen verschiedene in den einzelnen Kompilierungsphasen erzeugte Objekt-Modelle zur Verfügung. In der Analysephase wird der Syntax-Baum erzeugt und über die Syntax Tree API bereitgestellt, um generelle Informationen über den Aufbau eines Programms zu erhalten. Die weiter analysierten und zur Verfügung stehenden semantischen Informationen werden durch die Symbol API zugreifbar und erlauben gezielte Abfragen von Informationen zu einzelnen Programmelementen. Folgende Grafik gibt eine Übersicht über die Compiler-Pipeline von Roslyn und den APIs der Compiler API.

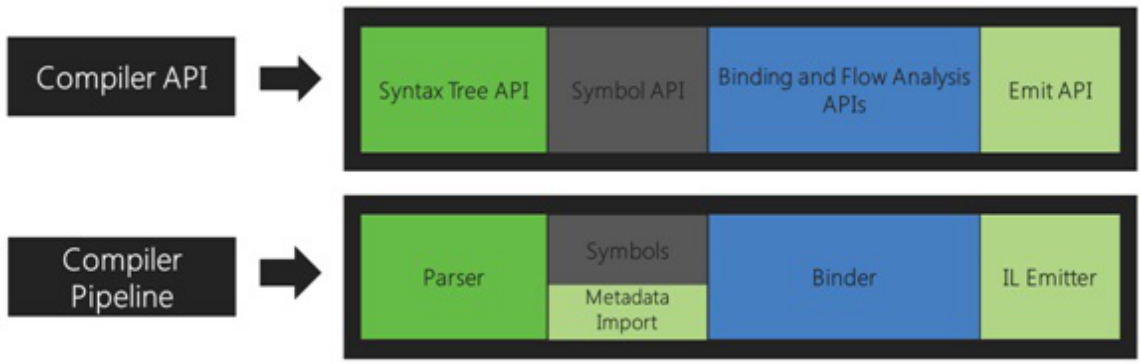


Abbildung 6: Übersicht zu Roslyns Compiler API (aus [ROS12])

Zu den Objekt-Modellen der Compiler API gehören z.B. der aus dem Code erzeugte (abstrakte) Syntax-Baum (`Roslyn.Compilers.CSharp.SyntaxTree`), der die Grundlage ist um den Programmcode strukturell zu analysieren und Änderungen einzubringen. Weiter existiert das semantische Modell (`Roslyn.Compilers.CSharp.SemanticModel`), um semantische Informationen über einzelne Knoten des Syntax-Baumes abzurufen, die während der Binding-Phase des Compilers gesammelt werden.

Die vorliegende Arbeit nutzt Microsoft Roslyn:

- bei der Analyse von zu refaktorierenden Programmen, um die für Refacola nötige Faktenbasis zu erstellen, (siehe Kapitel 4.2.1)
- um die von den Refacola-Refaktorierungen erzeugten Changesets in den Quellcode zu übernehmen, (siehe Kapitel 4.2.2)
- um die Kompilierbarkeit des veränderten Programmcodes zu prüfen (siehe Kapitel 4.3), und letztendlich
- um eine Refaktorierung in die Visual Studio IDE als Extension zu integrieren (siehe Kapitel 4.4).

Dieses Kapitel hat einen Überblick über die Grundlagen und die in dieser Arbeit genutzten Frameworks und Programmierschnittstellen gegeben, die zum Verständnis der weiteren Kapitel beitragen sollen.

In Kapitel 3 wird nun zunächst auf die Unterschiede von Java und C# eingegangen, welche

sich in den in Kapitel 5 betrachteten Regeln wiederfinden. Weiterhin wird erläutert, wie die Überprüfung der Korrektheit der erstellten Regeln in dieser Arbeit gesichert wird.

3. Vorbetrachtung

Das vorliegende Kapitel befasst sich mit den beiden Sprachen Java und C# und geht auf Unterschiede ein, die sich in den erstellten Regeln für die Refacola zur Spezifikation der statischen Semantik von C# (siehe Kapitel 5) wiederfinden. Dabei wird es nicht um einen kompletten Vergleich der beiden Sprachspezifikationen gehen, sondern es werden speziell die Eigenschaften von C# beleuchtet, die im Rahmen dieser Arbeit innerhalb der Sprach-Definition und Regeln für Refacola zur Anwendung gekommen sind.

Im zweiten Teil des Kapitels (siehe 3.2) wird dann erläutert wie die Überprüfung der Korrektheit der erstellten Regeln gesichert wird.

3.1 C# vs. Java

C# und Java sind zwei objektorientierte Programmiersprachen, die sich auf den ersten Blick ziemlich ähnlich sind. Eigenschaften und Funktionen werden in Form von Klassen gekapselt, die untereinander durch Nachrichtenaustausch miteinander kommunizieren. Die Klassen werden in Dateien deklariert, besitzen einen eindeutigen Namen und sind die Vorlage für die Erstellung von Objekten eines Typs. Beide Sprachen werden bei Kompilierung zunächst in eine Zwischensprache übersetzt, welche dann durch eine Laufzeitumgebung auf der eigentlichen Maschine zur Ausführung gebracht wird.

Wird die Spezifikation der Sprachen detailliert betrachtet, werden die Unterschiede der statischen Semantik von Java und C# deutlich. Diese Semantik spiegelt sich in den Regeln der Refacola-Definitionen wider, die für Java existieren und für C# im Rahmen dieser Arbeit erstellt werden. Auf die Unterschiede in der Semantik wird nun eingegangen.

3.1.1 Packages, Namespaces und Assemblies

Deklarierte Typen, wie z.B. Klassen oder die in C# existierenden Structs, besitzen einen eindeutigen Bezeichner, der diese identifiziert. Der qualifizierte Name der Typen setzt sich jedoch nicht nur aus dem Namen des deklarierten Typs zusammen, sondern beginnt immer mit dem Namen der Domäne, die diesen Typen enthält. Somit ist es möglich, dass ein Typ gleich benannt ist, jedoch zu einer anderen Domäne gehört und dadurch Namenskonflikte vermieden

werden. In Java existiert dazu das Konzept der *packages*. *Packages* und Klassen werden auf Verzeichnisse und Dateien abgebildet. Eine Datei enthält genau eine (top-level) Klasse und gehört zu einem *package*, der Dateiname muss mit dem Namen der Klasse übereinstimmen. Verdeutlicht wird dies in Listing 4.

Listing 4: Klasse in einem package

```
46    Datei: D.java
47
48    package a.b.c;
49    class D { }
```

Die Abbildung auf das Dateisystem ist in Java direkt gegeben.

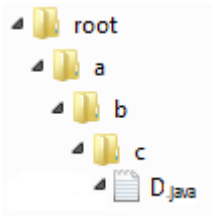


Abbildung 7: Abbildung eines packages auf das Dateisystem (eigene Darstellung)

In C# hingegen existieren dazu Namensräume, eingeleitet durch das Schlüsselwort *namespace*. Mehrere *Namespaces* können in C# innerhalb einer Datei definiert werden, der Dateiname kann frei gewählt werden und muss nicht wie in Java mit dem enthaltenen Klassennamen korrelieren. Eine Abbildung auf das Dateisystem ist in C# nicht gegeben. Listing 5 stellt dies dar.

Listing 5: Klasse in einem namespace

```
50    Datei: EineKlasse.cs
51
52    namespace A.B.C
53    {
54        class D { }
55    }
```

Um nun Typen eines *packages* oder *namespaces* nutzen zu können, müssen diese importiert werden. In Java werden dabei immer Klassen oder Sammlungen von Klassen (*-import) importiert. Für C# importiert man *namespaces*, es sei denn, dass über einen Alias eine Klasse direkt importiert wird. Nur durch einen Alias ist dies möglich.

Während in C# *namespaces* dazu dienen, Namenskonflikte zu verhindern und spezielle Typ-Domänen zu bilden, geht Java mit den *packages* weiter – es gibt einen eigenen Zugriffsmodifizierer *package*, der die Sichtbarkeit von Klassen und Membern nur auf das diese beinhaltende *package* einschränkt. In C# gibt es im Bereich der Namensräume keinen Modifizierer. Eine Einschränkung der Sichtbarkeit ist dort erst auf der Ebene der Assemblies möglich. Ein Assembly stellt in C# die kleinst mögliche Einheit zur Bereitstellung von Programmcode dar. Typen können darin den Zugriff mit des Modifizierer *internal* nur auf ihr eigenes Assembly beschränken. Findet z.B. ein Zugriff aus einem anderen Assembly statt, reicht es nicht, dass der referenzierte Typ die Sichtbarkeit *internal* deklariert, da so die Prüfung der Regel *TypeAccess* (siehe Kapitel 5.2.2, Tabelle 6) fehlschlägt.

Nachfolgend werden die Unterschiede bei den Zugriffsmodifikatoren beider Programmiersprachen erläutert.

3.1.2 Zugriffsmodifikatoren

Die Zugriffsmodifizierer in Java und C# regeln den Zugriff auf die Member eines Typs und den Typ selbst. Die für C# und Java existierenden Modifizierer berücksichtigen die im vorhergehenden Abschnitt 3.1.1 geschilderten Unterschiede bei der Strukturierung der Typen. Wie schon erwähnt, existiert in Java z.B. der Modifizierer *package*, der dort die Default-Sichtbarkeit[18] eines Members angibt. C# hingegen kennt das Konzept des *Packages* nicht. Vergleichbar existiert in C# jedoch das Schlüsselwort *internal*, welches die Zugreifbarkeit eines Members eines Typs (oder den Typ selbst) auf das den Typ beinhaltende Assembly beschränkt.

18 Die Default-Sichtbarkeit wird sowohl in Java als auch in C# durch das Weglassen eines Zugriffsmodifizierers deklariert.

In folgenden Tabellen werden die Zugriffsmodifizierer beider Programmiersprachen aufgelistet und die Zugreifbarkeit auf Member oder Typen durch andere Typen dargestellt.

Tabelle 1: Zugriffsmodifizierer von Java und Zugreifbarkeit auf Member.

Modifizierer	Klasse	Selbes Package	Sub-Klasse (anderes Package)	„Welt“
public	Ja	Ja	Ja	Ja
protected	Ja	Ja	Ja	Nein
package (default)	Ja	Ja	Nein	Nein
private	Ja	Nein	Nein	Nein

Der Modifizierer `package` schränkt, wie in Tabelle 1 zu sehen ist, den Zugriff z.B. auf eine Methode innerhalb einer Klasse auf die Klasse selbst und andere in demselben Package vorhandenen Klassen ein. Der `protected`-Modifizierer erweitert diese Beschränkung um Sub-Klassen, die sich in anderen Assemblies befinden.

Für C# kann die Zugreifbarkeit wie folgt dargestellt werden:

Tabelle 2: Zugriffsmodifizierer von C# und Zugreifbarkeit auf Member.

Modifizierer	Klasse	Sub-Klasse (selbes Assembly)	Assembly	Fremde Assemblies	Sub-Klasse in fremden Assemblies
public	Ja	Ja	Ja	Ja	Ja
protected internal	Ja	Ja	Ja	Nein	Ja
internal	Ja	Ja	Ja	Nein	Nein
protected	Ja	Ja	Nein	Nein	Ja
private	Ja	Nein	Nein	Nein	Nein

Die in Tabelle 2 gezeigten Modifizierer für C# stellen (genau wie in Java) also eine Einschränkung der Zugreifbarkeit von Typen oder Membern dar. Folgende Beschreibung geht

darauf noch einmal ein und zeigt die Default-Sichtbarkeiten:[19]

public

Der Zugriff ist nicht beschränkt. Dies ist der implizit gegebene Modifizierer für Member in `enums` oder `interfaces`.

internal

Der Zugriff ist auf das beinhaltende Assembly oder befreundete Assemblies beschränkt. `internal` ist der Default-Modifizierer für top-level Typen, also Typen, die nicht in anderen verschachtelt sind.

protected

Zugreifbar nur aus dem beinhaltenden Typ oder Sub-Typen dieses Typs.

protected internal

Vereint die Sichtbarkeit von `protected` und `internal`, erlaubt also mehr als diese alleine.

private

Der Zugriff ist nur aus dem beinhaltenden Typ möglich. Dies ist die default-Sichtbarkeit für Member in Klassen oder `structs`.

Die vier existierenden Zugriffsmodifizierer in Java können durch ihre spezifizierte Sichtbarkeit in einer Reihenfolge angegeben werden, die einer Totalordnung unterliegt wie in Abbildung 8 gezeigt:

$$\text{private} < \text{package} < \text{protected} < \text{public}$$

Abbildung 8: Totalordnung der Zugriffsmodifizierer in Java

Dies gilt jedoch nicht für die in C# vorhandenen Zugriffsmodifizierer, da durch die beiden Modifizierer `internal` und `protected`, die untereinander in keiner Reihenfolge stehen, lediglich eine Halbordnung angegeben werden kann. Dieser Zusammenhang wird in Abbildung 9 dargestellt:

19 vgl. [ALB12], Seite 84

$$\text{private} < \{\text{protected}, \text{internal}\} < \text{protected internal} < \text{public}$$

Abbildung 9: Halbordnung der Zugriffsmodifizierer in C#

Aus diesen beschriebenen Unterschieden resultiert, dass für C# innerhalb der Refacola-Regeln, die sich auf die Prüfung der Sichtbarkeit beziehen (z.B TypeAccess (siehe Kapitel 5.2.2)), explizit auf die einzelnen Modifizierer geprüft werden muss, da Vergleiche wie in Java mit $\leq$ oder $\geq$ nicht möglich sind.

3.1.3 Properties

In Java können für ein (als *private*) deklariertes Feld einer Klasse Zugriffsmethoden angegeben werden, die dieses abfragen oder verändern. Diese Zugriffsmethoden werden auch als Getter und Setter bezeichnet. Die Getter und Setter werden in Java wie Methoden verwendet. Folgendes Listing 6 verdeutlicht dies an einem Beispiel, in dem über den Setter der Name einer Regel gesetzt wird.

Listing 6: Zugriffsmethoden in Java (Getter, Setter)

```
56   public class Rule {
57
58        private String ruleName;
59
60        public String getRuleName() {
61              return ruleName;
62        }
63
64        public void setRuleName(String ruleName) {
65              this.ruleName = ruleName;
66        }
67   }

68   Rule rule = new Rule();
69   rule.setRuleName("MyRule");
```

In C# existieren die sogenannten Properties, die ebenfalls als Zugriffsmethoden genutzt werden, um den Zugriff auf ein Feld zu kapseln. Dazu definieren die Properties einen get- und/oder set-Block, welche auch als Accessor bezeichnet werden. Der set-Accessor besitzt einen impliziten Parameter *value*, der den Typ des Properties besitzt. Das besondere an den Properties ist, dass sie von einem Aufrufer genau wie ein Feld benutzt werden können (siehe

Code-Zeile 81 in Listing 7), wenn das Property den entsprechenden Accessor deklariert. Man kann also Read-Only- oder Write-Only-Properties definieren, oder den lesenden/schreibenden Zugriff durch entsprechende Modifizierer einschränken.

Listing 7: Properties in C#

```
70   public class Rule
71   {
72        private string ruleName;
73
74        public string RuleName
75        {
76              get { return this.ruleName; }
77              set { this.ruleName = value; }
78        }
79   }

80   Rule rule = new Rule();
81   rule.RuleName = "MyRule";
```

Bei Zuweisung an das Property 'RuleName' aus dem Beispiel wird der Wert an den impliziten set-Parameter *value* gebunden und letztendlich in dem Feld 'ruleName' gespeichert. Innerhalb der get-/set-Blöcke können komplexe Prüfungen vor der Zuweisung an das Feld und beim Auslesen des Feldes vorgenommen werden. Soll über das Property lediglich das Schreiben und Lesen an ein mit gleichem Typ (des Properties) implementierten Feld realisiert werden, bietet C# die Nutzung von automatisch implementierten Properties an. Dabei generiert der Compiler automatisch ein privates Feld mit dem Typ des Properties. Dieses Feld wird nur über die Properties verwendet. Durch diese Möglichkeit reduziert sich die Implementierung des obigen Properties. Listing 8 stellt nachfolgend die Syntax der automatisch implementierten Properties dar:

Listing 8: Darstellung der Syntax der automatisch implementierten Properties

```
82   public class Rule
83   {
84        public string RuleName { get; set; }
85   }
```

Die Properties betreffenden Regeln sind in Kapitel 5.2.6 dargestellt.

3.1.4 Partielle Klassen und Methoden

Partielle Klassen ermöglichen es in C#, die Deklaration eines Typs über mehrere verteilte Deklarationen, die sich meist in unterschiedlichen Dateien befinden, zu realisieren. Zur Laufzeit werden die partiellen Definitionen zu einem einzigen Typ zusammengefasst und als solcher behandelt.[20] Dies erlaubt z.B., dass ein Typ durch zwei partielle Deklarationen implementiert ist, bei denen der eine Teil durch ein Tool oder Framework[21] deklariert wird, der andere die manuell hinzugefügten Methoden und Properties enthält. Partielle Klassen, auch partielle Structs und Interfaces sind möglich, werden durch den Modifizierer *partial* gekennzeichnet, welcher bei Typdeklaration direkt vor den Schlüsselwörtern *class*, *struct* oder *interface* stehen muss.[22]

Partielle Klassen können partielle Methoden deklarieren, die z.B. Erweiterungspunkte darstellen. Eine partielle Methode deklariert dies ebenfalls mit dem Modifizierer *partial*, der direkt vor dem für partielle Methoden obligatorischen Rückgabetyp *void* auftauchen muss. Eine partielle Methode als Erweiterungspunkt besteht aus einer Definition und der Implementierung in einer partiellen Methode einer anderen partiellen Klasse. Partielle Methoden sind implizit *private*. In Listing 9 wird ein partielle Methode (als Erweiterungspunkt) dargestellt:

Listing 9: Partielle Klassen und Methoden

```
86   Rule_Generated.cs:
87   public partial class Rule {
88
89       partial void Execute();
90   }
91
92   Rule.cs:
93   public partial class Rule {
94
95       partial void Execute()
96       {  //... }
97   }
```

20 vgl. [CSharp5], Seite 282

21 Ein Beispiel dafür ist das Entity Framework (EF), aktuell in Version 6 verfügbar, welches ein objekt-relationales Mapping (ORM) zwischen einer Datenquelle und den eigentlichen Objekten zur Verfügung stellt. Dabei werden aus den Datenmodell-Entitäten partielle Klassen erzeugt, die vom Framework genutzt werden und durch den Entwickler manuell erweitert werden können.

22 Man beachte, dass partial in C# kein Schlüsselwort ist und an anderen Stellen, außer vor den genannten Schlüsselwörtern, als normaler Bezeichner für z.B. Variablen genutzt werden kann.

Das Konzept der partiellen Klassen und Methoden existiert in Java nicht. In Kapitel 5.2.5 wird weiter auf die existierenden Regeln und deren Umsetzung eingegangen.

3.1.5 Structs und Enums

In C# werden Typen eingeteilt in Referenz-Typen (reference types) und Wert-Typen (value types). Bei den Referenz-Typen sind alle Klassen, Arrays, Delegates und Interface Typen inbegriffen. Die Wert-Typen umfassen z.B. alle vorhandenen numerischen Typen oder den *char*-Typ und die manuell erstellten *struct*- und *enum*-Typen.[23]

Structs[24]

Ein Struct ist in C# ähnlich einer Klasse. Das Struct kann fast alle Member deklarieren, die auch für eine Klasse möglich sind, mit Ausnahmen z.B. der parameterlosen Konstruktoren und virtuellen Methoden. Zu beachten ist, dass Struct-Typen immer Wert-Semantik besitzen. Das Fehlen von virtuellen Methoden (siehe 3.1.7) deutet schon auf die fehlende Vererbung bei Structs hin. Structs erben implizit von *System.ValueType* (welche von *object* erbt), können aber keine Klassen explizit erweitern, lediglich eine Implementierung von Interfaces ist möglich.

Structs benötigen durch ihre Wert-Semantik keine Speicherallokation für ein Objekt auf dem Heap, welche bei vielen Instanzen des Typen einen Vorteil bringen kann.

Enums (Enumeration)

Wie Structs sind auch Enums Wert-Typen. Enums spezifizieren jedoch als Member keine Felder und Methoden, sondern benannte numerische Konstanten, die der Reihenfolge nach den dem Enum unterliegenden Wert-Typen, als Standard-Typ ist dies *int*, zugewiesen bekommen. Eine explizite Angabe von Werten für einzelne Konstanten ist ebenfalls möglich.

3.1.6 Delegates

Ein Delegat ist ein Referenz-Typ, der von *System.Delegate* ableitet und eine Methodensignatur definiert. Bei Instanziierung können dem Delegaten Methoden mit gleicher Signatur

23 vgl. [ALB12], Seite 16

24 vgl. [ALB12], Seite 83

zugeordnet werden.[25] Die Methoden können benannt oder auch anonym, genauso wie durch Lambda-Ausdrücke gegeben sein. Der Aufruf dieser Methoden erfolgt dann über die Instanz des Delegaten, welcher daraufhin die ihm zugeordneten Methoden ausführt. Delegaten lassen sich durch den Operator '+=' mehrfach belegen und stellen dadurch die Grundlage für Events dar. Folgendes Listing 10 macht dies an einem Beispiel deutlich.

Listing 10: (Multicast-)Delegates und mögliche Methodenzuordnung in C#[26]

```
98     class FirstRule {
99         public void Execute(int value) {
100            Console.WriteLine("FirstRule: " + value);
101        }}
102
103    class SecondRule {
104        public static void Execute(int value) {
105            Console.WriteLine("SecondRule: " + value);
106        }}
107
108    // Definition des Delegaten: Alle Methoden mit Rückgabetyp 'void'
109    // und einem Parameter des Typs 'int' können zugeordnet werden.
110    delegate void RuleExecuter(int parameter);
111
112    class DelegateTest
113    {
114        void Test() {
115          // Zuordnung einer Methode mit gleicher Signatur zu einer Instanz
116          // des Delegaten
117          RuleExecuter executer = new RuleExecuter(new FirstRule().Execute);
118
119          // Zuordnung einer statischen Methode
120          executer += SecondRule.Execute;
121
122          // Zuordnung einer anonymen Methode
123          executer += delegate(int i)
124                          { Console.WriteLine("Anonym: " + i); };
125
126          // Zuordnung eines Lambda-Ausdrucks
127          executer += (value => Console.WriteLine("Lambda: " + value));
128
129          // Ausführung: alle dem Delegaten zugeordnete Methoden werden
130          // hier mit Parameter '5' aufgerufen
131          executer(5);
132      }}
133      // Ausgabe in der Reihenfolge der Zuweisung an den Delegaten:
134      // FirstRule: 5
135      // SecondRule: 5
136      // Anonym: 5
137      // Lambda: 5
```

25 vgl. [ALB12], Seite 101

26 vgl. MSDN C#-Referenz http://msdn.microsoft.com/de-de/library/900fyy8e.aspx (letzter Zugriff: 21.05.2014)

Der Delegat des Beispiels wird in Zeile 110 definiert. Eine Instanziierung des Delegaten erfolgt in Zeile 117 mit der ersten zugeordneten Methode. Wie schon erwähnt, können dem Delegaten auch statische Methoden (Zeile 120), anonyme Methoden (Zeile 123) und auch Lambda-Ausdrücke (Zeile 127) zugewiesen werden. Durch Ausführung des Delegaten (Zeile 131) werden alle dem Delegaten zugeordneten Methoden zur Ausführung gebracht. Das Ergebnis der Ausgabe wird ab Zeile 134 verdeutlicht.

3.1.7 Vererbung in C#

In beiden objektorientierten Programmiersprachen Java und C# existiert das Konzept der Vererbung. Dabei werden bereits existierende Klassen durch neue Klassen spezialisiert oder generalisiert, die dadurch eine Klassenhierarchie bilden. Um zum Beispiel bei der Spezialisierung das Verhalten einer Klasse anzupassen, ist es nötig Methoden innerhalb der Sub-Klasse zu verändern. An dieser Stelle wird der Unterschied bei der Vererbung zwischen Java und C# deutlich. In Java kann von einer Sub-Klasse jede Methode der Super-Klasse überschrieben und somit angepasst werden. Die überschreibende Methode kann mit der Annotation `@override` den Compiler anweisen zu überprüfen, dass die überschriebene Methode in einer Super-Klasse tatsächlich existiert. Weitergehende Kennzeichnungen (des Überschreibens) gibt es in Java nicht. Bei späteren Änderungen an der Basisklasse kann dies zu Problemen führen, die auch als Fragile-Base-Class-Problem bekannt sind.

In C# ist bei der Vererbung und dem Überschreiben von Methoden der Super-Klasse eine explizite Angabe nötig, die innerhalb der Super-Klasse anzeigt, dass eine gegebene Methode überschrieben werden darf, und in der Sub-Klasse angibt, dass eine Methode der Super-Klasse überschrieben wird. Dazu existieren in C# die Schlüsselwörter `virtual` und `override`. Eine als `virtual` deklarierte Methode erlaubt somit abgeleiteten Klassen das Überschreiben, wenn diese es mit dem Schlüsselwort `override` angeben. Bei einer fehlenden `override`-Deklaration wird eine Warnung durch den Compiler erzeugt, da die Methode der Sub-Klasse die Methode der Super-Klasse ausblendet. Durch Hinzufügen von `override` wird die Methode überschrieben, mit dem Schlüsselwort `new` kann das gewünschte Ausblenden einer Methode explizit gemacht werden. Dies spiegelt sich in einigen erstellten Regeln wider, die in Kapitel 5.2 zu finden sind.

Es werden nun in einem Beispiel die Unterschiede zwischen den einzelnen Deklarationen

virtual, *override* und *new* gezeigt.[27] Zur Erläuterung sei auf die Kommentare des Beispiels in Listing 11 verwiesen.

Listing 11: Verwendung der Schlüsselwörter `override` *und* `new`.

```
138   class BaseClass {
139       public virtual void Method1() { Console.WriteLine("Base – Method1");}
140
141       public virtual void Method2() { Console.WriteLine("Base – Method2");}
142   }
143
144   class DerivedClass : BaseClass {
145       public override void Method1()
146                   { Console.WriteLine("Derived - Method1");}
147
148       public new void Method2() { Console.WriteLine("Derived – Method2");}
149   }
150
151   class Program {
152       static void Main(string[] args) {
153           BaseClass bc = new BaseClass();
154           DerivedClass dc = new DerivedClass();
155           BaseClass bcdc = new DerivedClass();
156
157           // Die beiden Aufrufe führen wie erwartet
158           // die Methoden der BaseClass aus.
159           bc.Method1();
160           bc.Method2();
161           // Ausgabe:
162           // Base - Method1
163           // Base - Method2
164
165           // Die beiden Aufrufe führen wie erwartet
166           // die Methoden der DerivedClass aus.
167           dc.Method1();
168           dc.Method2();
169           // Ausgabe:
170           // Derived - Method1
171           // Derived - Method2
172
173           // Die beiden Aufrufe führen zu unterschiedlichen Ergebnissen,
174           // je nachdem welche Methode, entweder override (Method1) oder
175           // new (Method2), verwendet wird.
176           bcdc.Method1();
177           bcdc.Method2();
178           // Ausgabe:
179           // Derived - Method1
180           // Base - Method2
181       }
182   }
```

27 vgl. MSDN: C# Programming Guide, http://msdn.microsoft.com/en-us/library/ms173153.aspx (letzter Zugriff: 21.05.2014)

3.1.8 Generics

Generics existieren sowohl in Java als auch in C#. Sie stellen durch das Konzept der Typ-Parameter bei Definition einer Klasse eine Menge von Klassen dar, die erst durch Instanziierung des Typ-Parameters der Klasse zu einem konkreten Typ wird. Häufig werden Generics bei Definition von Collection-Klassen genutzt, um z.B. die durch Generics gegebene Typ-Sicherheit zur Übersetzungszeit zu nutzen.[28] Die verwendeten Typ-Parameter können in beiden Programmiersprachen durch Typ-Constraints weiter eingeschränkt werden. In C# existieren bei diesen Typ-Constraints mehr Einschränkungsmöglichkeiten als dies in Java der Fall ist. In Java kann nach dem Schlüsselwort `extends` spezifiziert werden, dass der Typ-Parameter eine bestimmte Basisklasse haben muss oder ein Interface implementiert. C# geht an dieser Stelle weiter und ermöglicht es nach dem Schlüsselwort `where`, neben der Angabe der Vererbungshierarchie des Typ-Parameters weitere Anforderungen zu stellen, also z.B., dass der Typ des Typ-Parameters ein Wert-Typ (und nicht Nullable) sein muss, oder, dass der Typ des Typ-Parameters einen parameterlosen Konstruktor besitzen muss. Folgende Tabelle zeigt die möglichen Constraints vollständig auf.

Tabelle 3: Constraint-Typen für Generics in C#[29]

Constraint	**Beschreibung**
where T : struct	T muss ein Wert-Typ, außer ein Nullable-Typ, sein.
where T : class	T muss ein Referenz-Typ sein (Klasse, Interface, Delegat oder Array-Typs).
where T : new()	T muss einen parameterlosen Konstruktor besitzen.
where T : <Basis-Klasse>	T muss diese Basis-Klasse spezifizieren.
where T : <Interface>	T muss diese Interfaces (durch Komma getrennt) implementieren. Das Interface kann wiederum generisch sein.
where T : U	T muss vom zweiten angegebenen Typ-Argument U ableiten.

Das Konzept der Typ-Constraints ist wie gesehen in beiden Sprachen vorhanden, hat jedoch im Rahmen dieser Arbeit keinen Einzug gefunden. C# unterstützt jedoch bei den Generics

28 vgl. MSDN: Benefits of Generics: http://msdn.microsoft.com/en-us/library/b5bx6xee.aspx (letzter Zugriff: 21.05.2014)

29 vgl. MSDN: Constraints on Type Parameters: http://msdn.microsoft.com/en-us/library/d5x73970.aspx (letzter Zugriff: 21.05.2014)

keine Wildcards, wie dies in Java möglich ist. Auf die Java Wildcards wird in dieser Arbeit nicht weiter eingegangen, da diese bei der Erstellung der Regeln für C# keine Relevanz haben.

3.2 Überprüfung der Korrektheit

Nachdem im vorangegangen Kapitel auf einige Unterschiede von C# und Java eingegangen wurde, die in dieser Arbeit in den Regeln berücksichtigt werden, geht der nun folgende Abschnitt darauf ein, wie die Korrektheit der erstellten Regeln in dieser Arbeit gesichert wird. Dabei soll es nicht um formale Beweise der Korrektheit gehen, sondern dargestellt werden, wie die Überprüfung der partiellen Korrektheit gesichert wird. Korrektheit bedeutet erst einmal, dass ein Programm einer Spezifikation genügt. In dieser Arbeit werden die existierenden Regeln für C#, die semantische Überprüfungen des Compilers darstellen, in Refacola-Regeln (siehe Kapitel 2.2) abgebildet. Diese erstellten Regeln werden durch Anwendung einer Refacola-Refaktorierung genutzt, um die Semantik der Sprache über Constraints während einer Refaktorierung zu prüfen.

Im Rahmen dieser Arbeit wurden zur Überprüfung der Korrektheit NUnit-Tests erstellt, die teilweise auf den für Java existierenden Testfällen aufbauen. Dabei geht es darum, ein kompilierbares Programm durch eine Refaktorierung so zu verändern, dass die Semantik erhalten bleibt und das Programm sich im Anschluss wieder kompilieren lässt. Durch automatisierte Tests ist diese Prüfung jederzeit möglich und vereinfacht so die Überprüfung der Funktionalität nach erfolgten Änderungen.

Weiter wird nun beschrieben, wie diese Tests zur Prüfung konzipiert werden. Danach wird auf das Testen in der Entwicklungsumgebung Visual Studio eingegangen.

3.2.1 Tests mit NUnit

Ein Beispiel eines für diese Arbeit erstellten Tests ist in Kapitel 4.3 exemplarisch dargestellt. In vorliegendem Abschnitt wird es darum gehen, wie die Prüfung der Korrektheit durch die erstellten Tests partiell gesichert werden kann.

In den NUnit-Tests wird ein zu refaktorierendes Programmstück zur Prüfung gegeben, welches ein Beispielszenario darstellt, um eine Regel oder den Fehlschlag der Regel während ei-

ner Refaktorierung zu prüfen. Dieser Programmcode muss zuallererst kompilierbar sein. Dies wird geprüft, bevor für diesen Programmcode begonnen wird die Faktenbasis zu erzeugen. Daraufhin wird die Refacola-Refaktorierung gestartet. An dieser Stelle findet die nächste Überprüfung statt, bei der es darum geht, ob durch Anwendung der Refaktorierung überhaupt Constraints durch die Refacola generiert wurden. Ist dies nicht der Fall, wird über ein Error-Objekt die fehlende Constraint-Generierung kenntlich gemacht. Die fehlende Constraint-Generierung kann auf zwei mögliche Fehler hindeuten. Zum einen kann es sein, dass für das zu prüfende Element keine Refacola-Regel gefunden wurde, die auf die übergebenen und zu bearbeitenden Refacola-Programmelemente angewendet werden kann. Zum anderen kann es ein Hinweis sein, dass zwar eine Regel existiert, jedoch auf C# Seite nicht für alle in dem Programmcode enthaltene Elemente ein Fakt erzeugt wurde, sodass die Variablen der Constraint-Regeln möglicherweise nicht korrekt belegt werden können.

Eine erfolgreiche Refaktorierung durch die Refacola stellt ein Changeset zur Verfügung, welches die nötigen Änderungen beschreibt, die in den Programmcode eingebracht werden müssen. Dieses Changeset kann eine oder mehrere Änderungen beinhalten, je nachdem, ob für die zu testende Refaktorierung weitere Änderungen erlaubt sind (allowed changes) und ob durch den Algorithmus der Refacola zur Constraint-Generierung weitere Regeln gefunden werden, die durch die Änderung einer Constraint-Variablen zusätzlich überprüft werden müssen.

Nach Einarbeitung der gefundenen Änderungen wird als nächstes geprüft, ob der resultierende Programmcode kompilierbar ist. Für den Fall, dass das refaktorierte Programm nicht kompiliert, werden die durch den Compiler gefundenen Fehler zur Anzeige gebracht. Im letzten Schritt der Prüfung wird der aufgrund der Refacola-Refaktorierung geänderte Programmcode mit dem erwarteten Ergebnis, das als textuelle Repräsentation des Programms vorab gegeben ist, überprüft.

In Testfällen, bei denen ein Fehlschlag zu erwarten ist, funktioniert die Prüfung in den ersten zwei Schritten analog dem positiven Fall - Prüfung der Kompilierbarkeit und der erzeugten Constraints. Diese Informationen der Refacola zum Fehlschlag der Refaktorierung werden ebenfalls in das Changeset geschrieben. Es handelt sich dabei jedoch nicht um eine Änderung, sondern nur um den Austausch der Informationen zwischen der Refacola und C#. Über diese Information wird dann geprüft, ob die Refaktorierung aufgrund der erwarteten und im Test spezifizierten Regel nicht ausgeführt werden konnte. Schlägt der Test aufgrund der erwarteten

Regel fehl, ist davon auszugehen, dass die erzeugten Constraint-Regeln erfolgreich erzeugt und geprüft wurden und Änderungen an Constraint-Variablen vorgenommen wurden, die dann letztendlich zum Fehlschlag eines Constraints führten. Schlägt der Test zwar fehl, jedoch wird eine andere Regel als fehlgeschlagen zurückgemeldet, sind möglicherweise nicht alle semantischen Überprüfungen innerhalb der Regeln spezifiziert, sodass eine Constraint-Variable durch eine Regel einen ungültigen Wert erhält, der dann zum Fehlschlag einer nicht erwarteten Regel führt. Folgendes Ablaufdiagramm stellt die Durchführung eines Tests grafisch dar.

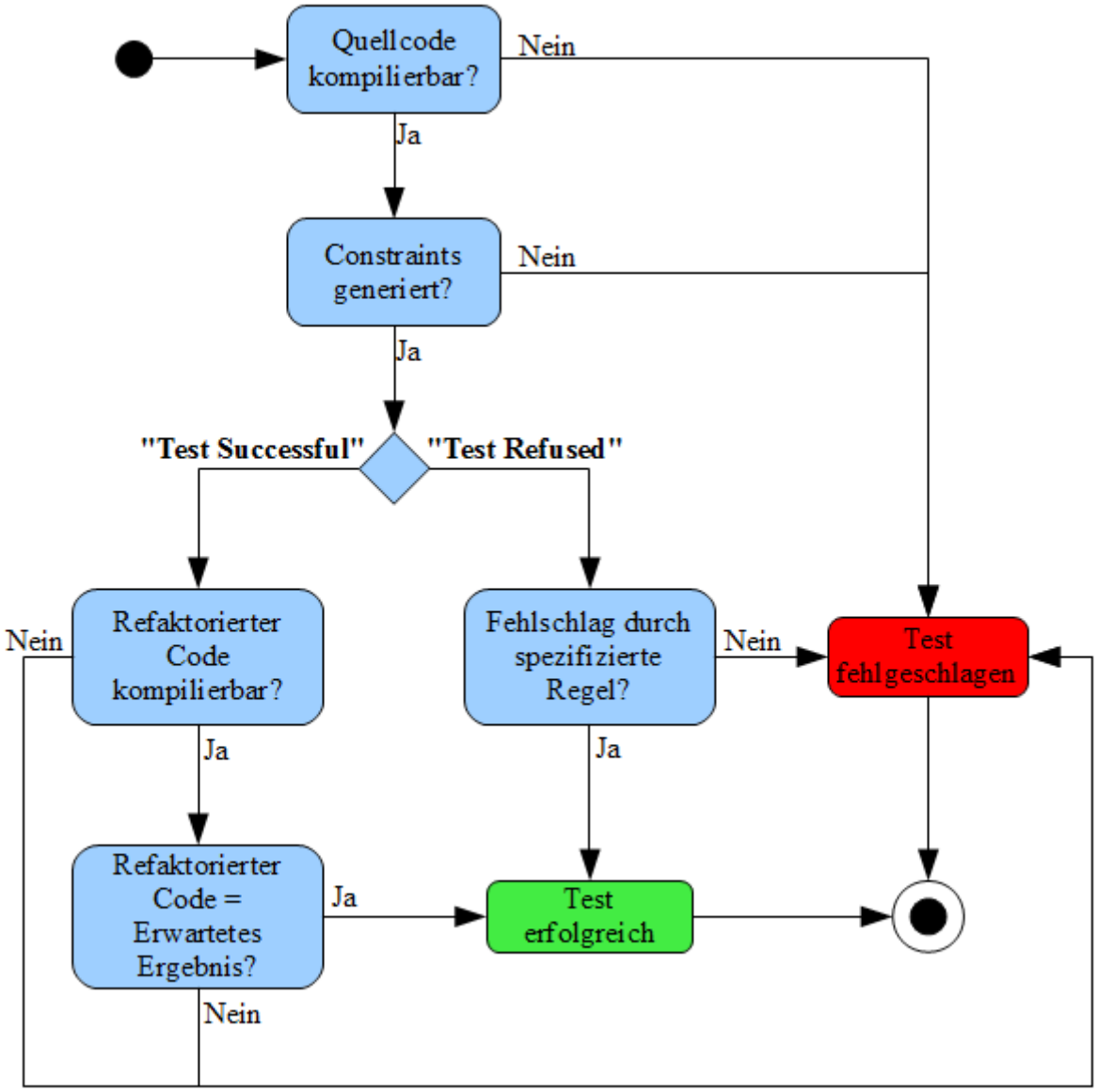


Abbildung 10: Grafische Darstellung des Testablaufs als Ablaufdiagramm (eigene Darstellung)

Der kommende Abschnitt geht auf die erstellte Visual Studio Extension und die dadurch gewonnene Möglichkeit des Testens in einer IDE ein.

3.2.2 Tests durch IDE-Erweiterung

Die in Kapitel 3.2.1 beschriebenen Tests überprüfen anhand kleiner Code-Fragmente eine gewünschte Refaktorierung. Die spezifizierten Code-Fragmente decken dabei einen großen Teil an Code-Beispielen ab, die mit C# erstellt werden können. Dabei wird häufig auftretender Programmcode betrachtet, der jedoch innerhalb der NUnit-Tests statisch gegeben ist.

Die erstellte Visual Studio Extension, die in Kapitel 4.4 beschrieben wird, dient dazu Tests von Refaktorierungen während der Entwicklung eines Programms direkt in die Entwicklungsumgebung Visual Studio zu integrieren. Diese Tests werden so mehr auf die direkte Benutzerinteraktion ausgerichtet und können so eine gewisse Dynamik in den Testprozess bringen. Somit lassen sich schnell neue Beispiele erstellen, die durch die Refacola refaktoriert werden sollen. Dadurch wird letztendlich die Funktionsabdeckung auf C#-Seite geprüft und werden noch nötige Änderungen aufgezeigt, da z.B. C#-Konstrukte noch nicht in die für die Refacola benötigten Fakten abgebildet werden.

Im Rahmen dieser Arbeit wird eine einfache Refaktorierung als Extension bereitgestellt, die sich auf die Prüfung der Sichtbarkeit von Klassen und Membern bezieht und auf die minimal benötigte Sichtbarkeit ermittelt. Es ist dadurch möglich die Regeln für die Sichtbarkeit von Elementen zu testen und direkt Änderungen der Sichtbarkeit in den Code einzubringen.

Im Kapitel 3.2 wurde auf die in dieser Arbeit eingesetzten Tests zur Überprüfung der partiellen Korrektheit eingegangen und mögliche Szenarios und deren Ergebnisse wurden betrachtet.

Kommender Abschnitt beschriebt nun, wie die Umsetzung zur Nutzung der Refacola-Refaktorierungen in dieser Arbeit erfolgt.

4. Umsetzung

In Kapitel 3 wurden die für diese Arbeit wichtigen Unterschiede zwischen Java und C# beschrieben. Auf die Umsetzung der Unterschiede innerhalb der Regeln wird in Kapitel 5 genauer eingegangen.

Das vorliegende Kapitel verschafft zuerst einen detaillierten Überblick über die in dieser Arbeit geschaffene Architektur zur Verwendung der Refacola aus C# heraus. Dazu werden die beteiligten Komponenten auf Java- und .NET-Seite beschrieben, die für einen Refacola-Refaktorierungsaufruf benötigt werden. Weiter werden in diesem Kapitel die erstellten NUnit-Tests und die dort abgebildeten Testfälle erläutert. Den Abschluss bildet die Beschreibung der erstellten Visual Studio Erweiterung zur Refaktorierung der Sichtbarkeit von Klassen und Membern. Eine Installationsanleitung zur Verwendung der Tests und der Erweiterung wird im Anschluss dieser Arbeit in Anhang B bereitgestellt.

4.1 Architektur

Die Refacola als sprachunabhängiges Refaktorierungsframework gewährleistet durch die deklarative Beschreibung der Sprachelemente und Regeln einer Zielsprache den Einsatz zur Spezifikation von Refaktorierungen für verschiedene Programmiersprachen. Unabhängig von der gewählten Programmiersprache werden auf Grundlage der Regeln und Fakten durch den Constraint-Generator Constraints erstellt, die durch einen Constraint-Solver gelöst werden. Die entscheidende Frage ist, wie die eigentlichen Fakten (Programmelemente und Anfragen) eines Programms der Refacola zur Constraint-Erstellung übergeben werden. Im Fall von Java existieren sowohl eine Abfrage- als auch eine Rückschreibkomponente. Aus einem Programmcode werden über Reflection direkt die benötigten Java-Objekte, die aus der Refacola-Sprachdefinition durch den Refacola-Compiler erzeugt wurden, erstellt und können somit verarbeitet werden. Im vorliegenden Fall, bei dem es um eine Refaktorierung von C#-Programmen geht, müssen die Fakten des Programms der Refacola auf andere Weise übergeben werden. Um dies zu realisieren, existiert auf Java-Seite die Klasse *de.feu.ps.refacola.FactbaseReader*, mit der es möglich ist, Dateien einzulesen und aus den darin enthaltenen Fakten die benötigten Java-Objekte der Zielsprache zu erzeugen. Dazu wird dem *FactbaseReader* lediglich die bei der Kompilierung erzeugte

LanguageFactory der Zielsprache und der Dateipfad zur Faktenbasis übergeben. Der erzeugte *FactbaseReader* wird dann bei Anwendung einer Refaktorierung als Datenquelle genutzt. Die Nutzung des *FactbaseReaders* ist momentan ebenfalls für die Programmiersprache Eiffel die Vorgehensweise zur Übergabe der Programminformationen und wird auch in dieser Arbeit verwendet, um die Refacola aus C# heraus zu nutzen. Der Ablauf zur Verwendung einer Refacola-Refaktorierung kann wie folgt grob dargestellt werden (Es wird von einer gefundenen Lösung des CSPs ausgegangen):

1. Analyse des Programms über den AST (**C#**)
2. Erzeugen der Faktenbasis als Datei (*.refacola.factbase) (**C#**)
3. Aufruf der gewünschten Refacola-Refaktorierung mit dem zu ändernden Programmelement und dem neuen Wert (**C# ↔ Java**)
    - Einlesen der Faktenbasis (**Java**)
    - Initialisieren der Refaktorierung (**Java**)
    - Ausführen der Refaktorierung (**Java**)
    - Erzeugen des Changesets und Speicherung als Datei (*.refacola.changeset) (**Java**)
4. Einlesen des Changesets (**C#**)
5. Anpassung des Programmcodes und Überprüfung der Kompilierbarkeit (**C#**)

Die generelle Architektur mit Bezug auf den geschilderten Ablauf verdeutlicht das folgende Schaubild (siehe Abbildung 11): Der Original-Code wird mithilfe der Roslyn API über den *SyntaxTreeParser* analysiert. Der *SyntaxTreeParser* erstellt die Faktenbasis und wird von den NUnit-Tests und der Visual Studio Extension erzeugt und verwendet. Die Tests und die Extension rufen über den *RefacolaInvoker* die eigentliche Refacola-Refaktorierung auf. Die Refaktorierung liest über den *FactbaseReader* die Programminformationen ein und verwendetet die Refacola zur Erstellung des CSP (welches über eine Constraint-Solver gelöst wird). Gefundene Änderungen (Lösung des CSP) werden im Changeset bereitgestellt, welches auf C#-Seite wieder über den *SyntaxTreeParser* eingelesen und verarbeitet wird und zum Refactored-Code führt.

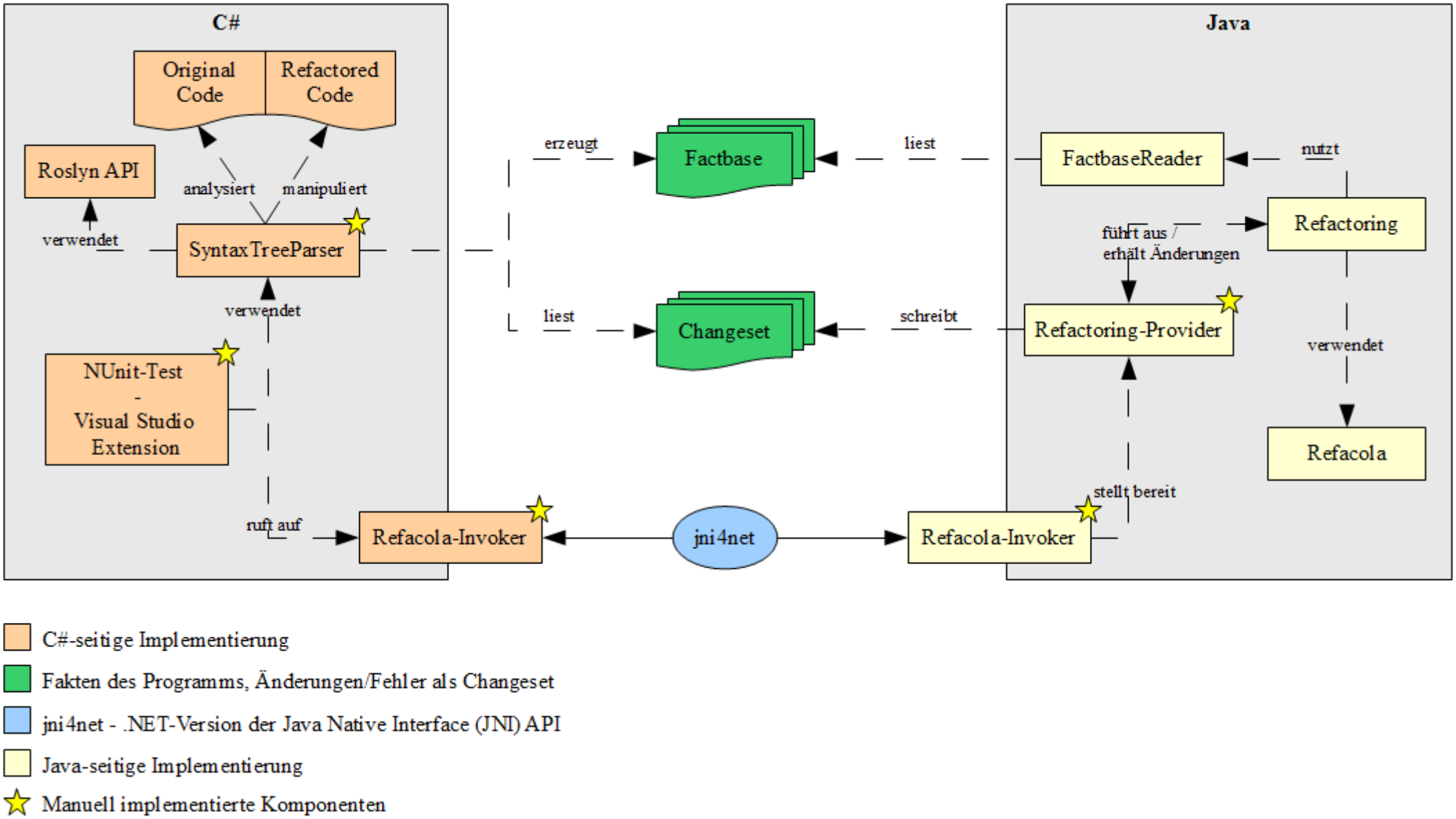


Abbildung 11: Darstellung der Architektur zur Verwendung der Refacola aus .NET (eigene Darstellung)

Im weiteren Verlauf dieses Kapitels wird detailliert auf die einzelnen Komponenten, deren Funktionsweise und die Implementierung eingegangen. Zunächst wird der *SyntaxTreeParser* beschrieben, der unter Verwendung der Roslyn API (siehe Kapitel 2.3) den Syntax-Baum eines Programms analysiert und über den *FactbaseWriter* die Fakten in die *.refacola.factbase Dateien ausgibt. Ebenfalls verarbeitet der *SyntaxTreeParser* über den *ChangesetReader* die von der Refacola-Refaktorierung gelieferten Änderungen oder Ausgaben über die *.refacola.changeset Dateien. Im Anschluss daran wird die erstellte Schnittstelle zwischen C# und Java erläutert und auf die jni4net API eingegangen.

4.2 Funktionsweise der Komponenten

Nachdem im vorhergehenden Abschnitt die generelle Architektur beschrieben wurde, wird nun auf einzelne Komponenten näher eingegangen um die Funktionsweise zu verdeutlichen. Zunächst werden die Komponenten zur Analyse und Manipulation des ASTs beschrieben, bevor dann auf die Implementierung der Schnittstelle zwischen C# und Java eingegangen wird.

4.2.1 Analyse des AST

Die Analyse des AST stellt die Grundlage dar, um Informationen aus dem zu refaktorierenden Programm zu erhalten und diese in geeigneter Form an ein Refacola-Refaktorierungswerkzeug zu übergeben.

Während der Analysephase des Compilers, genauer während der syntaktischen Analyse, wird der Syntax-Baum durch den Parser erstellt, wenn das analysierte Quellprogramm der Grammatik der verwendeten Sprache entspricht. Dieser Syntax-Baum wird dann im weiteren Verlauf der Kompilierung mit semantischen Information angereichert. Mit der Roslyn API (siehe Kapitel 2.3) können in jeder Phase des Kompilierungsprozesses Informationen über das kompilierte Programm abgefragt werden. Diese ersten Informationen stellt die Syntax API in einem geeigneten Objekt-Modell zur Verfügung. Der komplette Syntax-Baum einer Kompilationseinheit wird über eine Instanz der Klasse *Roslyn.Compilers.CSharp.SyntaxTree* repräsentiert. Der *SyntaxTree* besteht aus mehreren *SyntaxNodes,* die sich weiter in *SyntaxToken* und *SyntaxTrivia* unterteilen lassen. Dies wird kurz in Abbildung 12 dargestellt.

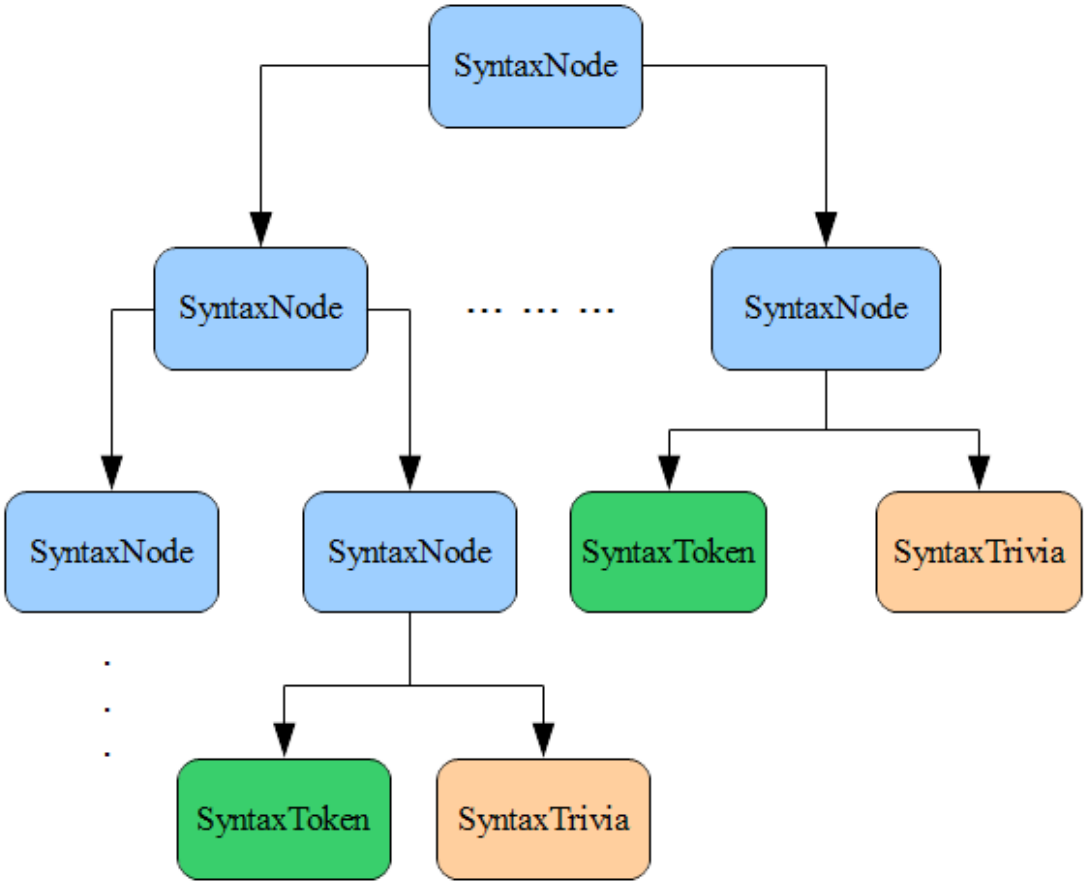


Abbildung 12: Darstellung der Komponenten eines SyntaxTrees (eigene Darstellung)

Der komplette Programmcode wird über die hierarchisch angeordneten `SyntaxNodes` dargestellt. Die `SyntaxNodes` sind wiederum in `SyntaxToken`, die z.B. Schlüsselwörter oder Bezeichner darstellen, und `SyntaxTrivia`, die z.B. Kommentare und Leerzeichen repräsentieren, unterteilt.[30] An dem Beispiel-Programmcode aus Listing 12 wird dies weiter verdeutlicht.

Listing 12: Beispielcode zur Darstellung der SyntaxNodes der Syntax API

```
ClassC.cs:

namespace a
{
    public class C
    {
        public void M() { }
    }
}
```

30 vgl. [SynA12]

Dieser Beispielcode wird nun exemplarisch im folgenden Schaubild mit dem Objekt-Modell der Roslyn API in Verbindung gebracht. Dabei wurden die zugehörigen Code-Elemente farbig hinterlegt und mit den entsprechenden Elementen der Syntax API dargestellt.

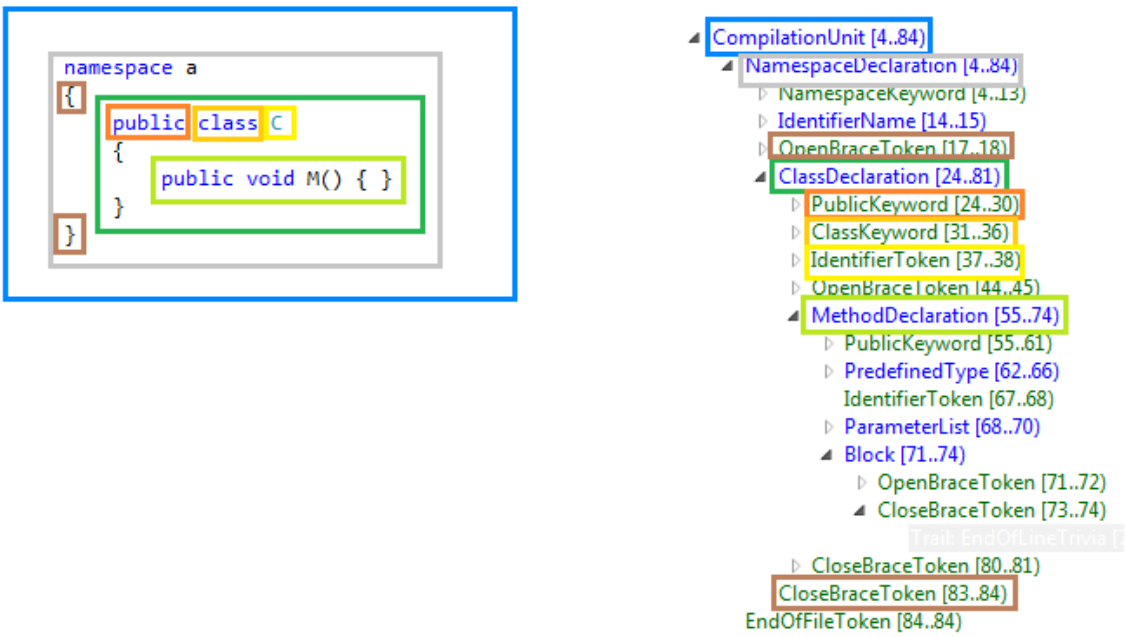


Abbildung 13: Programmcode und dessen Struktur als Syntax-Baum der Syntax API (eigene Darstellung)

Der oberste *SyntaxNode* ist die *CompilationUnitSyntax*, welcher der Quellcode-Datei entspricht. Die *NamespaceDeclarationSyntax* ist in diesem Fall der erste *SyntaxNode* der alle anderen *SyntaxNodes*, *SyntaxTokens* und *SyntaxTrivia* enthält. Betrachtet man die Klassendeklaration, kann man weiter die einzelnen Komponenten des *ClassDeclaration*-SyntaxNodes erkennen: das *public*- und das *class*-Keyword, der eigentliche Identifier „C“ der Klasse usw.. Die Syntax API von Roslyn stellt nun Klassen bereit, um die Elemente des Syntax-Baums zu besuchen und Informationen darüber abzufragen. Der Einstiegspunkt zur Analyse stellt das Interface *IDocument* dar, welches eine Programmdatei innerhalb eines Projekts oder einer Solution repräsentiert. Über das *IDocument* wird dann der zu analysierende *SyntaxTree* erstellt. Über das *SemanticModel*, welches ebenfalls über das *IDocument* erzeugt wird, können semantische Fragen über einzelne *SyntaxNodes* gestellt und so Informationen z.B. über eine Klasse gesammelt werden, wie z.B., ob die betrachtete Klasse *abstract* oder *static* ist, welches ihre Basisklasse ist oder ob sie eine ge-

nerische Klasse ist.

In dieser Arbeit übernimmt das Analysieren des Programmcodes und das Schreiben der Faktenbasis die Klasse *SyntaxTreeParser*. Der *SyntaxTreeParser* initialisiert dazu die Klasse *FactbaseWriter*, die letztendlich die eigentliche Faktenbasis unter einem geeigneten Namen (bei den NUnit-Tests z.B. der Testname) auf dem Dateisystem ablegt. Weiter wird der *ElementVisitor*[31] vorbereitet, der die Analyse des Syntax-Baums übernimmt. Der *ElementVisitor* erzeugt aus den benötigten *SyntaxNodes* die für die Faktenbasis benötigten Elemente. Das dazugehörige Objekt-Modell entspricht den in Refacola erzeugten Programmelementen der Refacola-Sprachdefinition für C#.

Für das obige Beispiel aus Listing 12 (siehe Seite 41) ergibt sich nach Analyse des Syntax-Baums die in Listing 13 (siehe Seite 44) gezeigte Faktenbasis. Die textuelle Repräsentation der vom *SyntaxTreeParser* erzeugten Objekte, entspricht dabei der Form, die der *FactbaseReader* (siehe Kapitel 4.1) benötigt, um die mit der Refacola-Sprachdefinition erzeugten Java-Objekte zu erstellen. Beginnend in Zeile 223 (siehe Listing 13 auf Seite 44) wird z.B. eine Methode angegeben, die dem definierten Refacola-Sprachelement *RegularTypedInstanceMethod* entspricht. Die definierten Eigenschaften des Sprachelements müssen alle mit entsprechenden Werten belegt sein; die Eigenschaft *Identifier* (Listing 13, Zeile 224) der Methode muss als Refacola-Typ *Identifier* gegeben sein, der einer Zeichenfolge entspricht und in Anführungszeichen dargestellt wird. Im Fall der Eigenschaft *Accessibility* (siehe Listing 13, Zeile 226) wird deren Wert aus der definierten Domäne als Literal mit einer vorangestellten Raute ('#') dargestellt. Der Top-Level-Owner (*TlOwner*, siehe Listing 13, Zeile 229) ist in der Sprachdefinition auf bereits definierte Sprachelemente eingeschränkt. In dem betrachten Fall ist der Wert des *TlOwner* der Name des Elements, hier *TopLevelClass_C*, welches die Methodendeklaration beinhaltet und in Listing 13, Zeile 243 definiert ist. Abschließend endet die Repräsentation eines Sprachelements mit einem Semikolon.

31 Der `ElementVisitor` implementiert die abstrakte Klasse `Roslyn.Compiler.CSharp.SyntaxWalker` welche wiederum von `SyntaxVisitor` (Implementierung des Visitor-Pattern) abgeleitet ist. Durch den `ElementVisitor` wird in dieser Arbeit der `SyntaxTree` durchlaufen und Fakten über das Programm gesammelt.

Listing 13: Erzeugte Faktenbasis für das Beispiel aus Listing 12 (siehe Seite 41)

```
192  ClassC_cs {
193       Assembly     Assembly_f7ca5881c35b4349b379149b63c0a161
194                    Identifier "f7ca5881c35b4349b379149b63c0a161";
195
196       Assembly     Assembly_mscorlib
197                    Identifier "mscorlib";
198
199       DefaultConstructor  DefaultConstructor_C__ctor
200                           Identifier "C__ctor"
201                           Assembly
                                    Assembly_f7ca5881c35b4349b379149b63c0a161
202                           Accessibility #public
203                           HostNamespace Namespace_a
204                           Parameters RegularTypedFormalParameter_Empty
205                           Tlowner TopLevelClass_C
206                           Owner TopLevelClass_C;
207
208       instantiates(DefaultConstructor_C__ctor,TopLevelClass_C);
209
210        Namespace    Namespace_a
211                     Identifier "a";
212
213       Namespace    Namespace_System
214                    Identifier "System";
215
216       Null Null_null;
217
218       RegularTypedFormalParameter     RegularTypedFormalParameter_Empty
219                                       Identifier "Empty"
220                                       DeclaredParameterType Null_null
221                                       DeclaredType Null_null;
222
223       RegularTypedInstanceMethod
                  RegularTypedInstanceMethod_C_ClassCTest_cs_6_8__6_27_M
224                Identifier "M"
225                Assembly Assembly_f7ca5881c35b4349b379149b63c0a161
226                Accessibility #public
227                HostNamespace Namespace_a
228                DeclaredType Struct_Void
229                Tlowner TopLevelClass_C
230                Modifier #notAvailable
231                Parameters RegularTypedFormalParameter_Empty
232                Owner TopLevelClass_C;
233
234       returnStatement(ValueTypedExpression_System_Void,
                 RegularTypedInstanceMethod_C_ClassCTest_cs_6_8__6_27_M);
235
236       Struct Struct_Void
237              Identifier "Void"
238              Assembly Assembly_mscorlib
239              Accessibility #public
240              Tlowner Struct_Void
241              Modifier #notAvailable
242              HostNamespace Namespace_System;
```

```
243       TopLevelClass TopLevelClass_C
244                     Identifier "C"
245                     Assembly Assembly_f7ca5881c35b4349b379149b63c0a161
246                     Accessibility #public
247                     Modifier #notAvailable
248                     HostNamespace Namespace_a
249                     Tlowner TopLevelClass_C
250                     EnclosingTypes TopLevelClass_C;
251
252       ValueTypedExpression ValueTypedExpression_System_Void
253                           ExpressionType Struct_Void
254                           InferredValueType Struct_Void;
255
256   }
```

Diese Faktenbasis dient einer Refacola-Refaktorierung als Basis zur Erzeugung der zu prüfenden Constraints. Soll z.B. mit der Refaktorierung *ChangeAccessibilityRefactoring* die Zugreifbarkeit der Klasse C auf den Default-Modifizierer (der durch das Weglassen „spezifiziert“ wird) geändert werden, wird durch den Refacola-Constraint-Generator das zu lösende CSP erstellt und die gewünschte Änderung durch die Constraint-Regeln geprüft. In diesem Beispiel ist eine Änderung der Sichtbarkeit möglich, sodass ein Changeset mit den Änderungen erzeugt wird.

Das nächste Kapitel beschreibt, wie diese Änderungen in den Programmcode eingebracht werden.

4.2.2 Modifikation des AST

Konnte das für ein zu refaktorierendes Programm erstellte CSP gelöst werden, werden die zu ändernden Eigenschaften der Programmelemente und deren neuer Wert in einem Changeset bereitgestellt. Diese Änderungen müssen in den existierenden Programmcode auf C#-Seite eingebracht werden. Dazu wird in dieser Arbeit ebenfalls die Roslyn API genutzt, bei der es bezüglich der Transformation zu beachten gilt, dass ein erzeugter *SyntaxTree* stets unveränderbar ist. Das bedeutet, dass eine spezifische Änderung, die in einen *SyntaxTree* eingebracht wird, einen neuen, wiederum unveränderlichen *SyntaxTree* erzeugt[32]. Sind alle Änderungen vollzogen, wird letztendlich das Quellcodedokument (die

32 Die Unveränderbarkeit in Roslyn erlaubt es z.B. Informationen des `SyntaxTrees` durch mehrere Analyse-Tools gleichzeitig zu analysieren ohne das der betrachtete `SyntaxTree` inzwischen durch ein anderes Tool verändert worden wäre. (vgl. [SynT12])

CompilationUnitSyntax – siehe Kapitel 4.2.1) verändert, in dem die Textrepräsentation des neuen *SyntaxTrees* in das Dokument geschrieben wird. Diese Funktionalität wird in dieser Arbeit durch den *ElementRewriter* realisiert, der ebenfalls (wie der *ElementVisitor*) über den *SyntaxTreeParser* initialisiert wird. Das Einlesen der von der Refacola (über den *RefacolaInvoker* – mehr dazu in 4.2.3) durchgeführten Änderungen werden zuerst über den *ChangesetReader* eingelesen und in den Programmcode eingebracht. Um im Anschluss an die erfolgten Änderungen zu prüfen, ob die Kompilierbarkeit nach der Refaktorierung weiter gegeben ist, werden alle Dokumente auf Fehler überprüft. Dazu können mit Roslyn über die aktuelle Kompilierung diagnostische Informationen abgerufen werden, die alle aktuellen Fehler und Warnungen enthalten.

Für das obige Beispiel aus Listing 12 (siehe Seite 41), bei dem es sich um die Änderung der Sichtbarkeit von „C“ auf „default“ handelt, wurde von der Refacola folgende Änderung als Changeset erstellt:

Listing 14: Von der Refacola erzeugte Änderung für das Bsp. aus Listing 12 (siehe S. 41)

```
257    TopLevelClass_C.accessibility -> none;
```

Diese Änderung stellt eine gültige Refaktorierung der Sichtbarkeit dar – dies wurde durch das Lösen der Constraint-Regeln (die sich aus den Regeln und den Fakten ergeben) von der Refacola geprüft. Nach den Änderungen ist die Kompilierbarkeit ebenfalls gegeben, sodass die refaktorierte Klasse wie in folgendem Listing 15 definiert ist:

Listing 15: Beispielcode der refaktorierten Klasse

```
258    namespace a
259    {
260        class C
261        {
262            public void M() { }
263        }
264    }
```

Dieses Beispiel stellt natürlich eine sehr einfache Refaktorierung dar, bei der durch die Refacola nur sehr wenige Regeln geprüft werden und auch nur ein Change erzeugt wird. Wird die Klasse aus dem obigen Beispiel jedoch von einer anderen Klasse aus referenziert, und liegt die referenzierende Klasse in einem anderen Assembly, ergeben sich weit mehr

Constraint-Regeln, die erzeugt und geprüft werden müssen. Somit sind möglicherweise auch mehrere gefundene Changes zu verarbeiten.

In den vorangegangen zwei Abschnitten wurde grob dargestellt, wie die Analyse und Modifikation des AST in dieser Arbeit umgesetzt wird. Zur detaillierten Betrachtung sei auf die in den Kapiteln genannten und der Arbeit beiliegenden Klassen verwiesen.

Als nächstes wird darauf eingegangen, wie die Schnittstelle zwischen C# und der in Java implementierten Refacola realisiert wird.

4.2.3 Schnittstelle C# ↔ Refacola

Damit die durch die Refacola erstellten Refaktorierungswerkzeuge für C# automatisiert genutzt werden können, ist es nötig, aus dem .NET-Framework heraus die Möglichkeit zu nutzen Java-Code aufzurufen. Dafür wurden in dieser Arbeit die beiden Implementierungen IKVM.NET[33] und jni4net[34] betrachtet. IKVM.NET stellt eine in .NET implementierte Java Virtual Machine (JVM) zur Verfügung, die Java-Bytecode in die dem .NET-Framework zu Grunde liegende CIL (siehe Kapitel 2.3.1) konvertiert und ausführt. Aufgrund von Schwierigkeiten bei der Kompilierung der für Refacola nötigen Archive (und deren Abhängigkeiten) und der Verwendung von IKVM.NET wurde diese Implementation nicht weiter betrachtet.

Stattdessen wird in dieser Arbeit die Implementierung jni4net verwendet, auf welche nun eingegangen wird. Jni4net nutzt das von der JVM bereitgestellte Java Native Interface (JNI). Das JNI ermöglicht der JVM zum einen, plattformspezifische Funktionen zu verwenden, zum anderen ermöglicht es anderen Programmen, die JVM zu kontrollieren, also z.B. Klassen zu laden oder Methoden auf erzeugten Instanzen aufzurufen. Durch jni4net wird für die benötigten Java-Klassen ein .NET-Proxy erstellt, der die Schnittstelle zur Implementierung der Java-Klasse bildet, die innerhalb der JVM über JNI aufgerufen wird.[35] Die genaue Beschreibung zur Erstellung dieses Proxies erfolgt in der Installationsanleitung in Anhang B.3.

Um nun aus C# heraus die entsprechenden Refacola-Refaktorierungen für eine gegebene

33 http://www.ikvm.net/ und http://de.wikipedia.org/wiki/IKVM.NET

34 http://jni4net.sourceforge.net/

35 vgl. [SAV09]

Faktenbasis nutzen zu können, wurde für Java die Klasse *de.feu.ps.refacola.csharp.bridge.RefacolaInvoker* implementiert. Über deren Konstruktor werden der Pfad und der Name der erzeugten Faktenbasis übergeben um den *FactbaseReader* zu initialisieren.

Für jede mögliche Refaktorierung stellt der *RefacolaInvoker* eine Methode bereit, welche die gewünschte Refaktorierung durch die Refacola ausführen lässt. Nach erfolgreicher Ausführung wird eine Changeset-Datei geschrieben. Diese enthält entweder die gefundenen Änderungen, die zu einer gültigen Refaktorierung führen, oder den Namen der Regel, die eine Änderung der Eigenschaft eines Programmelements verhindert, also ein Constraint erzeugt, welches nicht gelöst werden kann. Die für diese Arbeit umgesetzten und in der Java-Spezifikation vorhandenen Refaktorierungen *ChangeAccessibility*, *ChangeType* und *PullUpMember* können über die drei Methoden des *RefacolaInvokers*, die in folgendem Listing 16 dargestellt sind, genutzt werden.

Listing 16: Bereitgestellte Methoden des RefacolaInvokers

```
265       public Boolean InvokeRefacolaChangeAccessibilityRefactoring
266                   (String forcedChangeMember, String forcedAccessibility)
267
268       public Boolean InvokeRefacolaChangeDeclaredTypeRefactoring
269                   (String memberToChange, String newType,
270                    Boolean changeType, Boolean changeAccess)
271
272       public Boolean InvokeRefacolaPullupMemberRefactoring
273                   (String memberToChange, String destination,
274                    Boolean changeAccess, Boolean moveOthers)
```

Mit Hilfe von jni4net wird der .NET-Proxy für die Klasse *RefacolaInvoker* erstellt, der die Nutzung der genannten Methoden aus C# heraus ermöglicht. Um eine Refaktorierung über den *RefacolaInvoker* anzustoßen, wird nun erst einmal über die Klasse *net.sf.jni4net.Bridge* die JVM gestartet. Dieser werden alle Java Archive, die zur Ausführung der Refacola-Refaktorierungswerkzeuge benötigt werden, zur Verfügung gestellt. Über die *Bridge* wird die Proxy-Klasse registriert und eine Instanz dieser erstellt. Über JNI wird nun der Methodenaufruf auf der Proxy-Instanz an die Methode des in der JVM geladenen Java-Objekts der Klasse *RefacolaInvoker* weitergeleitet und von dieser ausgeführt. Die Ausführung der Refaktorierung läuft nun komplett innerhalb der JVM ab – das Resultat der Refaktorierung wird als boolescher Wert zurückgegeben, um die weitere Verarbeitung ent-

sprechend vorzunehmen. Da die erzeugte JVM und die CLR innerhalb desselben Prozesses ablaufen, ist die Ausführung der Refacola-Refaktorierung aus .NET heraus sehr schnell und daher auch innerhalb der Visual Studio Erweiterung, die eine direkte Benutzerinteraktion ermöglicht, nutzbar.

Die beschriebene Schnittstelle zwischen C# und Java über jni4net hat sich in dieser Arbeit zur Ausführung der Refacola-Refaktorierungen bewährt. Die Komponenten zur Analyse (Kapitel 4.2.1) und Modifikation (Kapitel 4.2.2) des AST und die eben vorgestellte Schnittstelle befinden sich in der zur Arbeit gehörenden Assembly `RefacolaCSharpLanguageApi.dll`. Diese DLL wird von den in dieser Arbeit erstellten NUnit-Tests und der Visual Studio Extension genutzt. Auf diese beiden Komponenten wird in den nächsten zwei Kapiteln eingegangen.

4.3 Tests zur Validierung

Die in dieser Arbeit erstellten Refacola-Regeln und die sie nutzenden Refaktorierungswerkzeuge können durch die in Kapitel 4.2.3 beschriebene Schnittstelle aus C# heraus genutzt werden. Dies ermöglicht es, die erstellten Regeln durch automatisierte Tests abzusichern. Diese Tests prüfen verschiedene Refaktorierungen anhand von Beispielszenarios und vergleichen das von der Refacola gelieferte Ergebnis (die mögliche Refaktorierung), welches in den Programmcode eingebracht wird, mit dem erwarteten Ergebnis. In Testfällen bei denen ein Fehlschlag zu erwarten ist, wird geprüft, ob eine Refaktorierung aufgrund der erwarteten Regel nicht ausgeführt werden konnte. Durch diese automatisierte Testausführung können Änderungen an bestehenden Regeln oder neu hinzugefügte Regeln, die noch nicht betrachtete Spezifikationen prüfen, schnell und einfach auf deren Korrektheit geprüft und deren Funktion verifiziert werden.

In dieser Arbeit werden für die einzelnen Refaktorierungen die für Java existierenden Tests übernommen und für C# angepasst. Für die neu entwickelten Regeln werden ebenfalls Tests erstellt, die diese Regeln mit Hilfe von Testszenarios überprüfen können.

In Listing 17 (siehe Seite 50) wird eine erstellte Testdefinition gezeigt und erläutert.

Listing 17: Beispieldarstellung eines erstellten NUnit-Tests

```
275     [Test]
276     public void TopLevelTypeAccessibility3()
277     {
278        ProjectId projectId1;
279        DocumentId documentId1;
280
281        ISolution solution = Solution.Create(SolutionId.CreateNewId())
282               .AddProject("Project1", "Project1.dll",
283                           LanguageNames.CSharp, out projectId1)
284               .AddDocument(projectId1, "Test1",
285                         new StringText("namespace a { public class A {}}"),
286                         out documentId1)
287               .AddMetadataReference(projectId1,
                     MetadataReference.CreateAssemblyReference("mscorlib"));
288
289        List<IDocument> documents = new List<IDocument>();
290        documents.Add(solution.GetDocument(documentId1));
291
292        string testName = TestContext.CurrentContext.Test.Name;
293
294        IDocument document = solution.GetDocument(documentId1);
295
296        SyntaxNode nodeToChange = document.GetSyntaxRoot()
297                                          .DescendantNodes()
298                                          .OfType<ClassDeclarationSyntax>()
299                                          .FirstOrDefault();
300
301        ChangeAccessibilityTestHelper changeAccessibility =
302                                    new ChangeAccessibilityTestHelper();
303        changeAccessibility.InitializeFactbase(testName, documents);
304
305        changeAccessibility.RunAssertSuccess(nodeToChange,
306                                         ExtendedAccessibility.Internal,
307                                         "namespace a { internal class A {}}");
308     }
```

Dieser einfache Test aus Listing 17 dient als Beispiel dafür, wie die Überprüfung der Regeln implementiert wird. Getestet wird hierbei die *`ChangeAccessibility`*-Refaktorierung einer top-level Klasse A.

In Kapitel 4.2.1 wurde dargestellt, wie mit Roslyn die Analyse des ASTs auf komfortable Weise möglich ist. Mit Roslyn ist es nun ebenfalls möglich, aus einer textuellen Repräsentation ein Programm zu erstellen und dieses z.B. über die Scripting-Engine[36] ausführen zu lassen. Innerhalb der Tests wird jedoch kein Programmstück direkt zur Ausführung gebracht, sondern

36 Die Scripting-Engine wurde für diese Arbeit nicht weiter betrachtet. Für nähere Informationen dazu sei auf [ScrI12] verwiesen.

dieses lediglich analysiert. Die Erzeugung des zu analysierenden Programms erfolgt durch die Instanziierung einer Solution in Listing 17 (siehe Seite 50) in Zeile 281. Die Solution besteht aus einem Projekt, welches ein Code-Dokument enthält und eine Referenz auf das Assembly mscorlib.dll[37] hat. Das zu analysierende Programm wird dem zu erzeugenden Dokument als Textrepräsentation übergeben (Listing 17 (siehe Seite 50) in Zeile 285). Über das Dokument kann nun das Programmelement gefunden werden, dessen Eigenschaften durch eine Refaktorierung verändert werden sollen (Listing 17 (siehe Seite 50) in Zeile 296). Über die Hilfsklasse *ChangeAccessibilityTestHelper* werden Methoden zur Verfügung gestellt, die über den *SyntaxTreeParser* die Erzeugung der Faktenbasis übernehmen, die Refaktorierung anstoßen und die Änderungen einarbeitet und das zurückgeliefert Ergebnis auswerten. Der konkrete Aufruf der Refaktorierung erfolgt in Listing 17 (siehe Seite 50) in Zeile 305: Es wird erwartet, dass dieser Test erfolgreich ausgeführt wird (Methode *RunAssertSuccess*(...)). Als Parameter werden der Methode zum einen das Programmelement, dessen Eigenschaft – hier die Zugreifbarkeit – geändert werden soll, zum anderen die gewünschte Zugreifbarkeit des Elements die überprüft werden soll (hier *internal*), und die textuelle Repräsentation des refaktorierten Programmstücks, mit dem die durch die Refacola gefundene und über den *SyntaxTreeParser* in den Programmcode eingebrachte Änderung verglichen wird, übergeben. Dieser Beispieltest stellt einen Test für ein positiv erwartetes Ergebnis dar.

Ein Refaktorierungsaufruf, dessen Fehlschlag erwartet wird, stellt Listing 18 dar:

Listing 18: Beispielaufruf einer Refaktorierung deren Fehlschlag erwartet wird.

```
309     changeAccessibility.RunAssertRefused(
                    nodeToChange, ExtendedAccessibility.Internal,
                    "failed: Accessibility.NestedTypeAccess");
```

Analog des erfolgreich erwarteten Tests aus Listing 17 (siehe Seite 50), werden der Methode in Listing 18 Zeile 309 ebenfalls drei Parameter übergeben. Der dritte Parameter gibt bei diesem Fall an, dass erwartet wird, dass das Ändern der Sichtbarkeit auf *internal* eines Elements des betrachteten Programms fehlschlägt, weil die Regel *NestedTypeAccess* eine Constraint-Regel erzeugt, die nicht gelöst werden kann.

Alle Tests befinden sich in der erstellten Solution in dem Projekt *RefacolaTestSuite* (die-

37 mscorlib steht für Multilanguage Standard Common Object Runtime Library. Das Assembly enthält grundlegende Typen und Methoden, die von der CLR genutzt werden.

ser Arbeit beiliegend). Aktuell existieren 286 entwickelte Tests, die anhand der drei Refaktorierungen `ChangeAccessibility`, `ChangeType` und `PullUpMember` unterteilt sind.

Die erstellten Tests ermöglichen ein kontinuierliches Testen der Regeln nach Anpassungen oder Hinzufügen neuer Features. Die betrachteten Testszenarios wurden anhand gängiger Beispiele erstellt, die ein Großteil der C#-Sprachspezifikation und dessen Möglichkeit abdecken. Um nun nicht nur die Tests (mit den statischen Programmcodes) zur Prüfung der Funktionalität der Refaktorierungen und Korrektheit der Regeln zur Verfügung zu haben, wird zusätzlich eine Visual Studio Extension implementiert, um den praktischen Einsatz der entwickelten Refacola-Refaktorierungswerkzeuge für C# zu zeigen. Diese Erweiterung beschreibt das nächste Kapitel.

4.4 Visual Studio Extension – Accessibility Refactoring

Die entwickelte Visual Studio Extension ermöglicht es dem Entwickler, direkt in der Entwicklungsumgebung Refaktorierungen durchzuführen. Die Erweiterung zeigt, wie die mit der Refacola erstellten Refaktorierungswerkzeuge im produktiven Einsatz genutzt werden können.

Im Rahmen dieser Arbeit wurde beispielhaft eine Visual Studio Erweiterung zur Reduktion der Sichtbarkeit von Klassen und deren Membern implementiert. Durch Selektion des Bezeichners eines Programmelements wird die Refaktorierung der Sichtbarkeit für dieses Element geprüft. Es werden dazu die in C# möglichen Zugriffsmodifizierer an die Refaktorierung übergeben und diese ausgeführt. Eine erfolgreiche Refaktorierung präsentiert dabei dem Entwickler die nötigen Änderungen am existierenden Programmcode.

Zur Umsetzung bietet die Microsoft Roslyn CTP (siehe Kapitel 2.3) über die Editor Services API die Möglichkeit, dass im Editor aktuell geöffnete Dokument zu analysieren. Dazu wird das Interface `Roslyn.Services.ICodeRefactoringProvider` implementiert. Als nächstes wird erst einmal geprüft, ob eine Refaktorierung für das selektierte Element vorgesehen ist. Im positiven Fall wird dann die gewünschte Änderung (hier die Sichtbarkeit) in Form eines Refaktorierungsaufrufs überprüft. Ist eine Änderung an einem Codeelement möglich, wird in der entsprechenden Codezeile angezeigt, welche Anpassungen durch die Refaktorierung vorzunehmen sind. Dazu werden die veränderten Codezeilen zur Anzeige gebracht. Eine Transformation wird dann vom Entwickler durch Selektion der Aktion „Change

accessibility...“ in den Programmcode eingebracht.
Mit dieser Erweiterung für Visual Studio wird gezeigt, dass es auf einfache Weise möglich ist – über die in dieser Arbeit entwickelte Schnittstelle zwischen C# und Java – erzeugte Refacola-Refaktorierungswerkzeuge, die aufgrund der erstellten Refacola-Regeln für C# Überprüfungen vornehmen, zu nutzen.
Wie die Erweiterung installiert und benutzt wird, welche Voraussetzungen gegeben sein müssen und welche Ordner im Dateisystem zur Erzeugung der Faktenbasis und des Changesets bereitgestellt werden müssen, ist in der Installationsanleitung in Anhang B zu finden.

Rückblickend wurde in Kapitel 4 die Architektur des entwickelten Systems erläutert und die implementierten Funktionen der einzelnen Komponenten wurden beschrieben. Es wurde gezeigt, wie aus C# heraus die von der Refacola erstellten Refaktorierungswerkzeuge erfolgreich genutzt werden können.

Die wichtigsten Teile, die diese Refaktorierungswerkzeuge für C# ermöglichen, sind die Refacola-Sprachdefinition, die spezifizierten Regeln und die Refaktorierungsdefinitionen. Im folgenden Kapitel werden nun die für die C# Spezifikation erstellten Regeln detailliert betrachtet und einzelne Regeln mit den für Java existierenden verglichen, um die in Kapitel 3 gezeigten Unterschiede anhand der Regeln der Sprachen zu verdeutlichen.

5. Die Refacola-Definitionen

Die bisherige Arbeit hat gezeigt, wie constraintbasierte Refaktorierungswerkzeuge der Refacola aus C# heraus genutzt werden können. Dabei wurden die verwendeten Frameworks und Technologien erläutert und geschildert, wie diese zur Erstellung der Testfälle und der Visual Studio Erweiterung eingesetzt werden.

Die in der Vorbetrachtung beschriebenen Unterschiede der beiden Programmiersprachen Java und C# werden nun in folgendem Kapitel bei der Umsetzung der Sprachdefinition für C# in Refacola weiter erläutert und die Implementierung anhand der Regeln verdeutlicht. Als Grundlage für die Refacola-Definitionen für C# dienen die bereits existierenden Definitionen für Java, die im Rahmen dieser Arbeit an die Semantik von C# angepasst und erweitert werden. Wie bereits erwähnt, liefert die C# Spezifikation 5.0 Spezialfälle und Bedingungen, die dann in Refacola abgebildet werden.

5.1 Refacola-Sprachdefinition

Die Sprachdefinition von C# in Refacola stellt die Grundlage dar, auf welcher die zu überprüfenden Regeln in Refacola spezifiziert werden können. Für die in Kapitel 3.1 geschilderten Unterschiede werden einige neue Sprachelemente in die Refacola-Sprachdefinition aufgenommen, die als nächstes betrachtet werden.

5.1.1 Erweiterte Modifikatoren

Um die Sichtbarkeit der definierten Programmelemente zu prüfen, wurden die in Kapitel 3.1.2 genannten Zugriffsmodifizierer in Refacola abgebildet. Zusätzlich zu den genannten Schlüsselwörtern für die Zugreifbarkeit wurde explizit in die Zugriffsmodifizierer-Domäne das Literal `none` aufgenommen. Dadurch ist es innerhalb der Regeln möglich, das Setzen des Default-Modifizierers, also das Weglassen des Modifizierers, zu fordern. Das Weglassen des Modifizierers ist zum Beispiel im Hinblick auf die Properties und deren Accessoren nötig: Deklariert ein Property den Modifizierer `private`, dürfen die Accessoren keinen Zugriffsmodifizierer haben.[38]

38 vgl. [CSharp5], 10.7.2 Accessors, Seite 325

Eine zweite Domäne für Modifikatoren stellt weitere Schlüsselwörter und Modifizierer zur Nutzung innerhalb der Regeln bereit. Dazu gehören *partial* zur Spezifizierung partieller Klassen oder Methoden, *static* zur Formulierung einiger Regeln für statische Klassen (*StaticClassOnlyStaticMembers*, *StaticClassIsNoType*), *abstractKeyword* zur Eingrenzung abstrakter Klassen oder Methoden, *notAvailable* zur Bestimmung, dass eine Klasse oder Methode keinen weiteren Modifizierer besitzt, *newKeyword* zur Kennzeichnung des Ausblendens und Neudefinition z.B. einer vererbten Methode, *virtual* zur Abgrenzung der Möglichkeit des Überschreibens von Membern und *override* zur Markierung eines überschriebenen existierenden Members.

Die Zugriffsmodifikatoren werden innerhalb der Regeldefinition für die Sichtbarkeit (Accessibility.ruleset.refacola) genutzt. Ebenso sind darin Prüfungen auf die erweiterten Modifizierer gegeben z.B. zur Überprüfung der Modifizierer bei Vererbung in der Regel *OverridingModifier*.

Die Regeldefinitionen der Sichtbarkeit umfassen 50 Regeln, von denen in Abschnitt 5.2 einige den für Java existierenden Regeln gegenübergestellt werden.

5.1.2 Properties

Zur Abbildung der Eigenschaften von Objekten eines Typs, den Properties, wurden der Refacola-Sprachdefinition einige Sprachelemente hinzugefügt. Wie schon in 3.1.3 erwähnt, können Properties einen Get- und einen Set-Block besitzen. Die Accessoren werden über *AccessorGet* und *AccessorSet* definiert. In Listing 19 wird die Sprachdefinitionen für Properties gezeigt.

Listing 19: Refacola-Definitionen für C# Properties

```
310    abstract Accessor <: AccessibleEntity, NamedEntity
311    AccessorGet <: Accessor
312    AccessorSet <: Accessor
313    Property <: Member, RegularTypedEntity { getter, setter }
```

Sowohl das Accessor-Element als auch das Property-Element können einen Zugriffsmodifizierer definieren. Getter und setter als Attribute des Property-Elements sind vom Typ *Accessor*.

Um den Zugriff auf ein *Property* abzubilden, wurde ein Subtyp der *MethodOrFieldReference*, eine *AccessorReference*, hinzugefügt. Diese besitzt als explizites Attribut den *Accessor* (get oder set), der referenziert wird. Listing 20 stellt die Definition dar:

Listing 20: Referenz für Zugriff auf Properties

```
314    AccessorReference <: MethodOrFieldReference { accessor }
```

5.1.3 Wert-Typen für Structs und Enums

Zur Repräsentation von Structs und Enums, die in C# als Wert-Typen behandelt werden (siehe 3.1.5), wurde die Kategorie *ValueType* in die Sprachdefinition aufgenommen, um die in C# existierende Unterscheidung zwischen Referenz- und Wert-Typen abzubilden. Über das Sprachelement *ValueTypedExpression* kann z.B. der Rückgabewert *void* (System.Void) einer Methode in Refacola repräsentiert werden. Nachstehendes Listing 21 stellt die Wert-Typ-Elemente dar.

Listing 21: Wert-Typ zur Definition von Structs und Enums

```
315    abstract ValueType <: NamedType
316
317    abstract EnumUnderlyingType <: Struct, RegularTypedEntity
318
319    Enumeration <: ValueType, TopLevelEntity, OwnerType { underlyingType }
320
321    Struct <: ValueType, TopLevelEntity, OwnerType
322
323    ValueTypedExpression <: Expression { inferredValueType}
```

Wie in 3.1.5 beschrieben, besteht eine Enumeration (Enum) aus benannten Konstanten, deren Werte alle von einem bestimmten Struktur-Typ sind. Dieser Struktur-Typ kann System.Byte (byte), System.SByte (sbyte), System.Int16 (short), System.UInt16 (ushort), System.Int32 (int), System.UInt32 (uint), System.Int64 (long) oder System.UInt64 (ulong) sein und muss als sogenannter *SimpleType* gegeben sein.[39] Zur Definition ist das Attribut *underlyingType* vom Typ *EnumUnderlyingType* vorgesehen.

39 vgl. [CSharp5], Seite 79 (4.1.4): SimpleTypes sind reservierte Wörter, die als Aliase für die entsprechenden Struktur-Typen definiert sind, z.B: int für System.Int32 (die SimpleTypes sind jeweils in Klammern hinter den Typen angegeben).

Die Structs erben von *OwnerType* das Attribut *modifier*. Dies ermöglicht, explizit die Modifizierer, die für Struktur-Typen nicht gesetzt werden dürfen, auszuschließen, so z.B. in der Regel *StructAndMemberModifier*, in der unter anderem ausgeschlossen ist, dass ein *Struct* als abstrakt definiert werden kann.

Um nun in der Sprachdefinition die Wert-Typen zu nutzen und genau wie Referenz-Typen diese als top-level zu markieren und als Besitzer von Membern auszuzeichnen, wurden die Kategorien ('kinds') *TopLevelEntity* und *OwnerType* aufgenommen.

Listing 22: TopLevelEntity und OwnerType der Sprachdefinition

```
324       abstract TopLevelEntity <: NamedType, AccessibleEntity, TlOwnedEntity
325       abstract OwnerType <: Type, TlOwnedEntity { modifier }
```

Das Aufnehmen der *TopLevelEntity* ist durch Hinzufügen der Wert-Typen *Struct* und *Enum* nötig geworden, um diese als top-level Typ zu deklarieren und auch als Besitzer einer Referenz zu setzen. Der *OwnerType* gilt ebenfalls als Markierung dafür, dass auch Wert-Typen Member wie Properties und Methoden besitzen können.

5.1.4 Delegates

Für die in 3.1.6 beschriebenen Delegaten wurde zu den Kinds der Sprachdefinition folgende Spezifikation aufgenommen:

Listing 23: Definition der Delegates

```
326   Delegate <: NamedClassOrInterfaceType, RegularTypedEntity,
327               TopLevelEntity { parameters }
```

Über das von *RegularTypedEntity* vererbte Attribut *declaredType* kann innerhalb der Regeln geprüft werden, welchen Rückgabe-Typ eine Methode besitzt und ob diese zum Delegaten passt. Ebenso sind die Parameter des Delegaten (*parameters)*, die als eine Liste von formalen Parametern (*FormalParameters)* gegeben sind, entscheidend, um die Prüfung bei Zuweisung an einen Delegaten zu validieren. Die entsprechende Regel *DelegateTypeCompatibility*, dargestellt in Abschnitt 5.2, wird z.B. über den erstellten Test *DelegateMethodType* (enthalten in *de.feu.ps.master.fro.refacola.tests.RefacolaTestSuite.ChangeTypeTests.ChangeTypeTestsCT*) geprüft.

Dieser Abschnitt hat einen Ausschnitt aus den existierenden Definitionen behandelt. Zum Überblick über alle vorhandenen Typen sei auf die der Arbeit beiliegende Definition *CSharp.Language.refacola* verwiesen (auf CD enthalten unter Code/Java-Packages/de.feu.ps.refacola.lang.csharp).

Folgend wird auf einige der Refacola-Regeldefinitionen eingegangen. Diese werden erläutert und mit Referenzen zur C# Spezifikation 5 versehen.

5.2 Refacola-Regeldefinition

Um die spezifizierte Semantik von C# in Refacola-Regeln abzubilden, wird neben den existierenden Regeln für Java, die C# Spezifikation 5.0 als Grundlage verwendet. Auf Basis dessen wird ein Großteil der Semantik abgebildet, um so eine Grundlage für weitere Arbeiten zu schaffen.

Im weiteren Verlauf dieses Kapitels werden einzelne für C# definierte Regeln den jeweils äquivalenten Regeln von Java, falls vorhanden, gegenübergestellt und Details sowie Unterschiede erläutert. Hierfür werden die Regeln tabellarisch gegenübergestellt, bevor näher auf die Beschreibung derselbigen eingegangen wird.

5.2.1 Top-Level-Typen

Die Top-Level-Typen sind nicht-verschachtelte Typen, die in einem Namensraum (namespace) liegen. Die betreffenden Regeln sind sehr einfach und dienen dadurch als einleitendes Beispiel.

Tabelle 4: Gegenüberstellung der Regeln zur Zugreifbarkeit von Top-Level-Typen

Regel für C#	Regel für Java

`TopLevelTypeAccessibility`
`  for all`
`  tlt: CSharp.TopLevelType`
`  do if`
`  all(tlt)`
`  then`
`  tlt.accessibility = #internal or`
`  tlt.accessibility = #public or`
`  tlt.accessibility = #none`
`  end`
	`TopLevelTypeAccessibility`
	`  for all`
	`  tlt: Java.TopLevelType`
	`  do if`
	`  all(tlt)`
	`  then`
	`  tlt.accessibility = #package or`
	`  tlt.accessibility = #public`
	`  end`

Die Regel *`TopLevelTypeAccessibility`* prüft für alle Top-Level-Typen, ob der spezifizierte Zugriffsmodifizierer entweder *`public`* oder *`internal`* ist. Der Default-Modifizierer für Top-Level-Typen ist *`internal`* und kann durch das Weglassen des Zugriffsmodifizierers gesetzt werden. Dazu wurde der Modifizierer *`none`* zur Domäne hinzugefügt (siehe Kapitel

5.1.1) auf welchen deshalb explizit geprüft wird.[40] In Java hingegen kann lediglich der Zugriffsmodifizierer *public* deklariert werden. Der Modifizierer *package* entspricht dort dem Default-Modifizierer, der ebenfalls durch Weglassen deklariert wird.

Tabelle 5: Gegenüberstellung der Regeln für die Bezeichner von Top-Level-Typen

Regel für C#	Regel für Java

UniqueTopLevelTypeIdentifier
for all
  tl1: CSharp.TopLevelType
  tl2: CSharp.TopLevelType
do
  if
  tl1 != tl2
  then
  tl1.hostNamespace = tl2.hostNamespace
  -> tl1.identifier != tl2.identifier
  or (tl1.modifier = #partial and
  tl2.modifier = #partial)
end
	UniqueTopLevelTypeIdentifier
	for all
	  tlt1: Java.TopLevelType
	  tlt2: Java.TopLevelType
	do
	  if
	  tlt1 != tlt2
	  then
	  tlt1.hostPackage = tlt2.hostPackage
	  -> tlt1.identifier != tlt2.identifier
	end

Ein Top-Level-Typ muss einen eindeutigen Namen in einem Namensraum haben. Eine Ausnahme in C# stellt dabei die Definition von partiellen Typen dar. Die Regel prüft also für unterschiedliche Typen, die sich im selben Namensraum befinden, ob der Bezeichner unterschiedlich gewählt wurde, oder es sich um eine partielle Typdefinition handelt. Dazu wurde, wie in 5.1.1 beschrieben, eine zusätzliche Domäne erstellt, die weitere Schlüsselwörter von C# enthält.
Die Regeln für partielle Typen werden im weiteren Verlauf betrachtet.

5.2.2 Zugriff auf Typen

Der Zugriff auf einen Top-Level-Typ aus einem anderen Typ heraus erfordert die Zugreifbarkeit des Ersteren. Dabei ist entscheidend, ob sich der referenzierte Typ in demselben Assembly befindet. Wie in Abschnitt 3.1.2 beschrieben, müssen durch die fehlende Ordnung der Zugriffsmodifizierer in C# diese explizit geprüft werden. In Tabelle 6 wird eine erste Re-

40 vgl. [CSharp5], § 3.5.1 Declared Accessibility, Seite 60

gel für den Zugriff auf Top-Level-Typen angegeben.

Tabelle 6: Gegenüberstellung der Regeln für den Zugriff auf Top-Level-Typen

Regel für C#	Regel für Java

TypeAccess
for all
  tr: CSharp.TypeReferenceInType
  type: CSharp.TopLevelEntity
do
  if
  all(type), all(tr)
  then
  tr.typeBinding = type and
  tr.tlowner != type.tlowner
  -> type.accessibility = #internal
  or type.accessibility = #none
  or type.accessibility = #public,
  tr.typeBinding = type and
  tr.assembly != type.assembly
  -> (type.accessibility = #public)
end
	TypeAccess
	for all
	  tr: Java.TypeReferenceInType
	  type: Java.TlOwnedType
	do
	  if
	  all(type), all(tr)
	  then
	  tr.typeBinding = type and
	  tr.tlowner != type.tlowner
	  -> type.accessibility >= #package,
	  tr.typeBinding = type and
	  tr.hostPackage != type.hostPackage
	  -> type.accessibility >= #protected
	end

Ein Top-Level-Typ wird, wie in Abschnitt 5.1.3 beschrieben, über die Definition *TopLevelEntity* markiert. Ist die Referenz an einen Typ gebunden und ist der Typ nicht im selben Assembly wie die Referenz, muss die Zugreifbarkeit *public* sein. Befindet sich die Referenz auf den Typ in demselben Assembly sind alle drei Zugriffsmodifizierer für Top-Level-Typen möglich.

Weiter ist zu betrachten, dass ein verschachtelter Typ aus einem Top-Level-Typ heraus referenziert wird und es zwischen diesen beiden Typen keine Subtypenbeziehung gibt. Diese Regel ist im Vergleich zu der in Java existierenden Regel komplexer, da wiederum das Strukturierungskonzept der Assemblies mit in die Regel integriert wird. Daraus ergibt sich die in Tabelle 7 (siehe Seite 62) dargestellte erweiterte Regel für C#:

Tabelle 7: Gegenüberstellung der Regeln für den Zugriff auf verschachtelte Typen

Regel für C#	Regel für Java

NestedTypeAccessFromTopLevel
for all
tr: CSharp.TypeReferenceInType
referencingType: CSharp.TopLevelEntity
nestedType: CSharp.NestedType
do if
all(referencingType), all(tr),
all(nestedType)
then
tr.typeBinding = nestedType and
tr.tlowner != nestedType.tlowner
and referencingType = tr.owner
and Csharp.sub*!(
referencingType.enclosingTypes,
nestedType.owner)
-> (nestedType.accessibility !=
#private and
nestedType.accessibility !=
#none and
nestedType.accessibility !=
#protected),
tr.typeBinding = nestedType and
tr.assembly != nestedType.assembly
and referencingType = tr.owner
and Csharp.sub*!(
referencingType.enclosingTypes,
nestedType.owner)
-> (nestedType.tlowner.accessibility
= #public and
nestedType.accessibility =
#public)
end
	MemberTypeAccess
	for all
	tr: Java.TypeReferenceInType
	referencingType: Java.TlOwnedType
	type: Java.MemberType
	do
	if
	all(type), all(tr),
	all(referencingType)
	then
	(tr.typeBinding = type)
	and (referencingType = tr.owner)
	and (tr.hostPackage !=
	type.hostPackage)
	and Java.sub*!(
	referencingType.enclosingTypes,
	type.owner)
	-> type.accessibility = #public
	end

Die komplexere Regel für C# wird durch die Unterscheidung nötig, ob sich der Typ, aus dem referenziert wird, in demselben Assembly befindet oder nicht. Befindet sich sowohl der referenzierte verschachtelte Typ, als auch der referenzierende Typ innerhalb eines Assemblies, ist bei einem Zugriff die Voraussetzung, dass der verschachtelte Typ weder *private* (*none*) noch *protected* ist (Vererbung ist explizit ausgeschlossen in dieser Regel). Für den Typ, welcher den *NestedType* deklariert, ist keine weitere Einschränkung nötig. Befindet sich dagegen der *NestedType* innerhalb eines anderen Assemblies, muss dieser und der deklarierende Top-Level-Typ als *public* deklariert werden (ebenfalls aufgrund fehlender Vererbung). Analog dazu existiert die Regel zur Prüfung des Falls, dass die Referenz innerhalb eine *NestedType* stattfindet (*NestedTypeAccessFromNestedType*).

Im Weiteren wird die Gegenüberstellung der Regeln für Typen, die innerhalb einer Vererbungshierarchie definiert sind, und deren Zugriff behandelt.

5.2.3 Vererbung

Bei der Vererbung in C# gibt es, wie schon in Kapitel 3.1.7 erwähnt, einige Unterschiede bei der Deklaration von spezialisiertem Verhalten, genauer gesagt, der Definition von überschreibenden und überschriebenen Membern.

Auch die vorhandenen Zugriffsmodifizierer *protected* und *protected internal* (siehe Abschnitt 3.1.2) sind an dieser Stelle wichtige Unterschiede im Vergleich zu Java bei der Spezifikation der Sichtbarkeit im Hinblick auf verschachtelte Typen und den existierenden Zugriffsbeschränkungen.

Zunächst werden die Regeln für die Kennzeichnung des Überschreibens von Methoden und die damit zusammenhängenden Regeln betrachtet.

Wie in 3.1.7 beschrieben, existieren in C# die Schlüsselwörter *virtual* und *override*, welche in Java nicht vorhanden sind. Für eine überschriebene Methode müssen einige Regeln gelten, die sich in der in Tabelle 8 (Seite 64) dargestellten Regel widerspiegeln.

Tabelle 8: Darstellung der Regel zur Prüfung der Modifizierer beim Überschreiben

Regel für C#	Regel für Java

OverridingModifier
for all
overridingMethod: CSharp.InstanceMethod
overriddenMethod: CSharp.InstanceMethod
do
if
Csharp.overridesDirectly(overridingMethod,
overriddenMethod)
then
(overriddenMethod.modifier = #virtual or
overriddenMethod.modifier = #abstractKeyword or
overriddenMethod.modifier = #override) and
overriddenMethod.modifier != #sealed and
overridingMethod.modifier = #override
end
	Nicht vorhanden

Existiert eine überschriebene Methode, die durch das Refacola-Faktum *overridesDirectly* abgebildet wird, muss diese entweder als *virtual* die Möglichkeit des Überschreibens anzeigen, als eine abstrakte Methode einer Klasse deklariert sein oder bereits eine Methode einer Super-Klasse überschreiben. Das bedeutet, dass die Methode weder nicht-virtuell noch statisch sein darf. Auch darf die überschriebene Methode nicht als *sealed* markiert sein, da dadurch die Möglichkeit des Überschreibens entfällt.[41]

Für Methoden, die eine existierende Methode überschreiben, gilt für C# ebenfalls wie für Java, dass zum einen die Methoden denselben Parameter-Typ besitzen müssen, zum anderen auch der Return-Typ der beiden Methoden identisch sein muss. Zur Prüfung werden die beiden Regeln *OverridingReturnType* und *OverridingSameParameterType* (aus *Types.ruleset.refacola*) genutzt.

Ein weiterer Unterschied wird bei der Definition der Sichtbarkeit deutlich, der sich wiederum auf die Assemblies und deren einschränkenden Modifizierer bezieht und in Tabelle 9 (Seite 65) angegeben ist.

41 vgl. [CSharp5], Seite 317, § 10.6.4 Override Methods

Tabelle 9: Gegenüberstellung der Regeln zur Prüfung der Sichtbarkeit beim Überschreibe von Methoden

Regel für C#	Regel für Java
OverridingAccessibility2 for all overridingMethod: CSharp.InstanceMethod overriddenMethod: CSharp.InstanceMethod do if Csharp.overridesDirectly (overridingMethod, overriddenMethod) then (overriddenMethod.assembly != overridingMethod.assembly) and (overriddenMethod.accessibility = #protectedinternal) -> overridingMethod.accessibility = #protected, (overriddenMethod.assembly = overridingMethod.assembly) and (overriddenMethod.accessibility != #protectedinternal) -> overriddenMethod.accessibility = overridingMethod.accessibility end	OverridingAccessibility2 for all overridingMethod: Java.InstanceMethod overriddenMethod: Java.InstanceMethod do if Java.overridesDirectly (overridingMethod, overriddenMethod) then overridingMethod.accessibility >= overriddenMethod.accessibility end

Für Java fällt auf, dass eine überschreibende Methode die Sichtbarkeit einer überschriebenen Methode erweitern darf, sie muss also mindestens dieselbe Sichtbarkeit wie die überschriebene Methode besitzen. In C# gilt dies nicht, dort muss eine überschreibende Methode dieselbe Sichtbarkeit besitzen. Für C# wird daher auf die Gleichheit der Sichtbarkeit geprüft (*OverridingAccessibility 1-3*). Einen Spezialfall stellt das Überschreiben einer als *protected internal* deklarierten Methode dar, die in einem anderen Assembly überschrieben wird. Um nun die Sichtbarkeit nicht weiter auf das andere Assembly zu erweitern, muss die überschreibende Methode die Sichtbarkeit *protected* spezifizieren[42], sodass nur weitere abgeleitete Klassen Zugriff auf diese haben.

42 vgl. [CSharp5], Seite 317, § 10.6.4 Override Methods

Die Sichtbarkeit von Typen bei der Vererbung unterliegt in C# weiter der Einschränkung, dass die Basisklasse eines Typs mindestens genauso sichtbar sein muss wie der Typ selbst. Ein abgeleiteter Typ darf also niemals die Sichtbarkeit der Basisklasse erweitern. Daraus resultieren für C# vier weitere Regeln, welche die Vererbung und die Sichtbarkeit zwischen Top-Level-Typen und verschachtelten Typen prüfen. Dazu gibt Tabelle 10 beispielhaft eine der Regeln an.

Tabelle 10: Darstellung der Regel zur Prüfung der Sichtbarkeit von verschachtelten Typen bei Vererbung

Regel für C#	Regel für Java

InheritedNestedTypeAccessibility
for all
super: CSharp.NestedType
sub: CSharp.NestedType
do
if
CSharp.sub(sub,super)
then
(sub.accessibility = #public and
super.accessibility = #public) or
(sub.accessibility = #internal and
(super.accessibility = #public or
(super.accessibility = #internal and
sub.assembly = super.assembly))) or
(sub.accessibility = #protected and
(super.accessibility = #public or
super.accessibility = #protected or
super.accessibility = #protectedinternal)) or
(sub.accessibility = #protectedinternal and
super.accessibility = #public)
end
	Nicht vorhanden

Besteht eine Subtypenbeziehung und ist der Sub-Typ öffentlich sichtbar, muss dies auch für den Super-Typ gelten. Deklariert ein Sub-Typ die Sichtbarkeit *`protected`*, sind für den Super-Typ die drei Zugriffsmodifikatoren *`public`*, *`protected`* und *`protected internal`* möglich.

5.2.4 Delegaten

Für die Prüfung der Kompatibilität zwischen Delegaten D und deren zugeordneten Methoden M wurde die Regel *DelegateTypeCompatibility* erstellt. Die Kompatibilität setzt z.B. voraus, dass D und M die gleiche Anzahl von Parametern haben und diese sowie die Rückgabetypen ineinander konvertiert werden können oder diese in einer Vererbungshierarchie stehen. Die Regel zur Prüfung der Typ-Kompatibilität von Delegaten ist in Tabelle 11 angegeben.

Tabelle 11: Darstellung der Regel zur Prüfung der Typ-Kompatibilität bei Delegaten

Regel für C#	Regel für Java
DelegateTypeCompatibility for all delegate: CSharp.Delegate method: CSharp.RegularTypedMethod do if all(delegate), all(method) then CSharp.sub*(method.parameters, delegate.parameters) and CSharp.sub*(delegate.declaredType, method.declaredType) end	Delegaten nicht vorhanden

5.2.5 Partielle Typen

Partielle Typen, die eine Definition auf mehrere Dateien verteilen können, erfordern, dass die Typen im gleichen Namespace deklariert werden. Im Fall von verschachtelten Typen ist auch der top-level Typ zu beachten. Somit kann eine erste Regel für partielle Typen, wie in Tabelle 12 gezeigt, definiert werden.

Tabelle 12: Darstellung der Regel zur Prüfung von partiellen Typen

Regel für C#	Regel für Java

PartialTypes
for all
type1: CSharp.OwnerType
type2: CSharp.OwnerType
do
if
all(type1), all(type2),
type1 != type2
then
type1.modifier = #partial and
type2.modifier = #partial and
type1.identifier = type2.identifier
-> type1.tlowner = type2.tlowner or
type1.hostNamespace = type2.hostNamespace
end
	Keine partiellen Typen in Java

Spezifiziert ein partieller Typ eine Sichtbarkeit, müssen alle dazugehörigen partiellen Typen ebenfalls dieselbe Sichtbarkeit definieren. Wird kein Modifizierer angegeben, wird wiederum der jeweilige Default-Modifizierer angenommen.[43] Die dies prüfende Regel ist *PartialTypesAccessibility*.

Wie in 3.1.4 gesehen dürfen nicht nur Typen eine partielle Definition deklarieren, sondern ebenfalls Methoden, die dadurch Erweiterungspunkte ermöglichen. Partielle Methoden dürfen keine Zugriffsmodifizierer besitzen und sind implizit *private*.[44] Dies wird durch die in Tabelle 13 (Seite 69) dargestellte Regel geprüft.

43 vgl. [CSharp5], Seite 283, § 10.2.2 Modifiers

44 vgl. [CSharp5], Seite 285, § 10.2.7 Partial methods

Tabelle 13: Darstellung der Regel zur Prüfung der Sichtbarkeit von partiellen Methoden

Regel für C#	Regel für Java

PartialMethods
for all
method1: CSharp.InstanceMethod
method2: CSharp.InstanceMethod
do
if
all(method1), all(method2),
method1 != method2
then
(method1.tlowner = method2.tlowner) and
(method1.identifier = method2.identifier) and
(method1.modifier = #partial and
method2.modifier = #partial)
-> (method1.accessibility = #none and
method2.accessibility = #none)
end
	Keine partiellen
	Methoden in Java

Der Paragraph 10.2.7 aus der C# Spezifikation fordert weiter, dass der Rückgabe-Typ der partiellen Methoden *`void`* ist und die Parameter keine *`out`*-Parameter definieren dürfen. Dies wurde im Rahmen dieser Arbeit nicht weiter umgesetzt. Ersteres würde eine Prüfung auf einen konkreten Typ erfordern, welche in Refacola nicht möglich ist. Die Parameter *`ref`* und *`out`* wurden nicht implementiert.

5.2.6 Properties

Der Zugriff auf Eigenschaften einer Klasse erfolgt oft über Properties, die mit ihren Accessoren den Zugriff auf die Werte eines Feldes ermöglichen. Accessoren (get / set) können selbst Zugriffsbeschränkungen besitzen. Es ist jedoch lediglich einem der Accessoren erlaubt, einen Zugriffsmodifizierer explizit zu definieren.[45] Tabelle 14 zeigt die dazu definierte Regel.

Tabelle 14: Darstellung der Regel für Accessoren und deren Zugriffsmodifizierer

Regel für C#	Regel für Java

PropertyAccessorOnlyOneModifier
for all
prop: CSharp.Property
accRef: CSharp.AccessorReference
accRef2: CSharp.AccessorReference
do
if
CSharp.binds(accRef, prop),
CSharp.binds(accRef2, prop),
accRef != accRef2
then
(accRef.accessor.accessibility = #none or
accRef2.accessor.accessibility = #none)
end
	Getter und Setter werden an dieser Stelle wie normale Methoden behandelt

Darüber hinaus gibt es Einschränkungen, welche Sichtbarkeit ein Accessor haben darf, wenn das Property selbst einen bestimmten Zugriffsmodifizierer deklariert. Ist das Property z.B. als *internal* deklariert, muss der Accessor (wie beschrieben darf nur ein Accessor einen Modifizierer haben) als *private* definiert werden. Für einen kompletten Überblick über die weiteren Einschränkungen sei auf die in Tabelle 15 angegebene Regel und auf die C# Spezifikation verwiesen.[45]

45 vgl. [CSharp5], Seite 324, § 10.7.2 Accessors

Tabelle 15: Darstellung der Regel zur Sichtbarkeitsprüfung von Properties und Accessoren

Regel für C#	Regel für Java

PropertyAccessorAccessibility
for all
prop: CSharp.Property
accRef: CSharp.AccessorReference
accRef2: CSharp.AccessorReference
do
if
CSharp.binds(accRef, prop),
CSharp.binds(accRef2, prop),
accRef != accRef2
then
(prop.accessibility = #public)
->(accRef.accessor.accessibility != #public),
(prop.accessibility = #protectedinternal)
-> (accRef.accessor.accessibility != #public and
accRef.accessor.accessibility !=
#protectedinternal),
(prop.accessibility = #internal or
prop.accessibility = #protected)
-> (accRef.accessor.accessibility = #private or
accRef.accessor.accessibility = #none),
((prop.accessibility = #private) or
(prop.accessibility = #none))
-> ((accRef.accessor.accessibility = #none) and
(accRef2.accessor.accessibility = #none))
end
	Keine spezielle Regel, da es sich wie erwähnt, um normale Zugriffsmethoden handelt.

Für einen Überblick über alle in dieser Arbeit definierten Regeln sei auf die Refacola-Definitionen verwiesen, die dieser Arbeit auf CD beiliegen. (siehe Anhang A.)

In folgendem Kapitel 5.3 der Refacola-Definitionen wird die Refaktorierung der Sichtbarkeit, wie sie für diese Arbeit definiert ist, dargestellt.

5.3 Refacola-Refaktorierung

Es wird nun eine der Refaktorierungen aus dieser Arbeit dargestellt. Sie findet unter anderem Anwendung in der im Rahmen dieser Arbeit implementierten Visual Studio Erweiterung.

Tabelle 16: Darstellung der Refacola-Refaktorierung ChangeAccessibility

ChangeAccessibility-Refaktorierung für C#

```
import "CSharp.language.refacola"
import "Accessibility.ruleset.refacola"
import "Types.ruleset.refacola"
import "Locations.ruleset.refacola"
import "Names.ruleset.refacola"

refactoring ChangeAccessibility
languages CSharp
uses Accessibility, Types, Locations, Names
forced changes
    accessibility of CSharp.AccessibleEntity as Accessibility
allowed changes
    accessibility of CSharp.Accessor
    accessibility of CSharp.Property
    accessibility of CSharp.Method
    accessibility of CSharp.DeclaredConstructor
    accessibility of CSharp.TlOwnedArrayType
    accessibility of CSharp.ClassOrInterfaceType
```

Die dargestellte Refaktorierung ändert die Sichtbarkeit von Typen (auch Structs und Enums), Delegaten, Methoden, definierten Konstruktoren und Feldern. Als erlaubte Änderungen werden weiter andere Typen, Methoden, Properties und Accessoren spezifiziert. Die Aufnahme der Typen, als erlaubte Änderung zum Beispiel, ist durch die partiellen Typen nötig, da durch Änderung der Zugreifbarkeit eines partiell definierten Typs alle dazugehörigen partiellen Typen mit verändert werden müssen.

Abschließend wird nun das Fazit der Arbeit gezogen und ein Ausblick auf mögliche zukünftige Arbeiten gegeben.

6. Fazit und Ausblick

Refaktorierungen, die auf Basis von Constraints die gewünschten und nötigen Änderungen prüfen und dadurch die syntaktische Korrektheit und die semantische Äquivalenz des analysierten und angepassten Programmcodes gewährleisten, stellen eine konsequente und überprüfbare Methode dar, um geeignete Werkzeuge zu erstellen, die durch den Entwickler innerhalb einer Entwicklungsumgebung genutzt werden können. Diese Refaktorierungen und die Erzeugung der Constraints werden auf Grundlage einer Sprachdefinition der jeweiligen Programmiersprache und den für sie existierenden Regeln gebildet. Dies setzt voraus, dass alle in der betreffenden Sprachspezifikation existierenden Regeln vollständig in Form von Refacola-Regeln abgebildet sind.

Der Einsatz der Refacola als Sprache zur Abbildung der Elemente der betrachteten Programmiersprache und deren Regeln, sowie als Framework zur Erzeugung der benötigten Constraints, schafft eine sehr gute und sinnvolle Möglichkeit, Refaktorierungswerkzeuge sprachunabhängig zu erstellen und einzusetzen.

Im Rahmen dieser Arbeit wurde für C# eine Refacola-Sprachdefinition erstellt, auf deren Grundlage die für die C# Spezifikation 5.0 existierenden Regeln in Refacola abgebildet wurden. Dabei wurde die für Java verfügbare Sprachdefinition als Vorlage genommen und für C# angepasst. Abschnitt 3.1 zeigt dazu existierende Unterschiede auf, die sich in den erstellten Refacola-Definitionen (siehe Kapitel 5) wiederfinden. Aufgrund des großen Umfangs wurde auf eine vollständige Abbildung aller in der C# Spezifikation enthaltenen Regeln verzichtet und es wurde sich auf den wesentlichen Teil beschränkt. Das Ziel ist es, einen relativ großen Umfang an möglichen Programmelementen und Regeln von C# abzudecken, der sich in den angepassten und erstellten Tests widerspiegelt. Es kann dadurch zum Beispiel bei Anwendung der Visual Studio Erweiterung vorkommen, dass bestimmte Code-Konstrukte eine Fehlermeldung zur Anzeige bringen, da in C# die Betrachtung des betreffenden Konstruktes noch nicht implementiert ist.

Die entwickelte Visual Studio Erweiterung stellt eine Beispielimplementierung dar, um Refacola-Refaktorierungen interaktiv in einer IDE einzusetzen. Der erste Aufruf zur Überprüfung der Möglichkeit zur Refaktorierung der Sichtbarkeit eines Elementes dauert durch die Initialisierung der Schnittstelle auch bei kleineren Testklassen relativ lange. Die folgenden

Prüfungen laufen daraufhin recht schnell ab (ohne dass explizit Laufzeitmessungen vorgenommen werden), sodass der Einsatz der Erweiterung durch einen Entwickler in produktiven Umgebungen möglich ist.

Gezeigt wird weiter, wie die Lücke zwischen dem zu analysierenden C#-Code und der Javaseitig implementierten Refacola über die in Kapitel 4.2.3 entwickelte Schnittstelle geschlossen wurde, sodass überhaupt erst eine automatische Ausführung von Refaktorierungen aus C# heraus möglich ist. Diese Schnittstelle bedient sich zum Austausch der Fakten und der erzeugten Änderungen der Refacola des Dateisystems, weshalb Schreib- und Lesevorgänge auf beiden Seiten nötig sind, was zu einer zeitlichen Verzögerung des Vorgangs führt.

Eine Möglichkeit, die Nutzung des Dateisystems zum Datenaustausch zu umgehen, ist eine direkte Erzeugung der aus der Refacola-Sprachspezifikation erstellten (Java-)Objekte aus .NET heraus und eine direkte Verwendung der über die Refacola erzeugten Refaktorierungswerkzeuge. Die in Abschnitt 4.2.3 erwähnte Implementierung IKVM.NET, bei der es am Anfang der Arbeit zu Problemen bei der Konvertierung der benötigten Archive (aufgrund von Abhängigkeiten der Archive) gekommen ist, könnte dazu genutzt werden, sämtliche Archive und die Java-Objekte (der C#-Sprachelemente) in die .NET Intermediate Language zu übersetzen, sodass aus .NET heraus die Faktenbasis in Form der erzeugten Objekte direkt übergeben wird und zur Lösungsfindung genutzt werden kann. Ausstehend bleibt an dieser Stelle jedoch die Definition der Sprache und Regeln in Refacola, für die ein Editor für Eclipse existiert und somit natürlich auch Java essentiell benötigt wird.

In einer Arbeit von Herrn Hertel[46] wurde die Portierung der Refacola API auf die .NET-Plattform betrachtet, welche eine weitere Möglichkeit darstellt, um die Refacola direkt aus .NET heraus zu nutzen. Diese Portierung beruht jedoch auf einer älteren Refacola-Version, sodass diese im Rahmen dieser Arbeit nicht ohne Anpassungen zum Einsatz kommen konnte.

Zur Analyse[47] und Manipulation[48] des AST wurde Roslyn[49] eingesetzt. An dieser Stelle ist zu überlegen, ob es nicht sinnvoll ist, als Refacola-Sprachdefinition direkt das Roslyn unterliegende Objekt-Modell mit dessen semantischen Informationen zu den einzelnen Elementen zu

46 vgl. [HER11]

47 siehe Kapitel 4.2.1

48 siehe Kapitel 4.2.2

49 siehe Kapitel 2.3

nutzen und in Refacola abzubilden und aufbauend auf diesem die Regeln zu formulieren. Somit ist der Zwischenschritt über ein eigenes Objekt-Modell nicht nötig, sodass eine mögliche Fehlerquelle vermieden wird und direkt auf den Objekte-Modellen von Roslyn Regeln definiert werden. Dafür ist wiederum eine Portierung der Refacola auf die .NET-Plattform sinnvoll, um direkt auf dem Roslyn Objekt-Modell Regeln zu spezifizieren.

Zusammenfassend ist zu sagen, dass mit dieser Arbeit und den darin erstellten Regeln für die Zugreifbarkeit, Typbindung und Namensgebung und die Absicherung und Prüfung der Regeln anhand von Beispielszenarios innerhalb der Tests, eine Grundlage für die weitere Betrachtung der Refacola-Sprachdefinition und -Refaktorierungen für C# geschaffen wurde. Änderungen und zusätzliche Regeln können leicht erstellt und durch die Testausführung überprüft werden.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literaturverzeichnis

[ALB12] Albahari, Joseph, Albahari Ben: C# 5.0 Pocket Reference, O'Reilly Media, 2012

[BraRo] Brant, John, Roberts, Don: Refactorings (zum Refactoring Browser für Smalltalk), 1998, http://st-www.cs.illinois.edu/users/brant/refactory/Refactorings.html, zuletzt abgerufen: 14.05.2014

[COH10] Cohn, Mike: Agile Softwareentwicklung: Mit Scrum zum Erfolg!, Addison-Wesley, 2010

[CSharp5] Microsoft Corporation: C# Language Specification, Version 5.0, 2012

[FOW02] Fowler et al.: Refactoring: Improving the Design of Existing Code, Addison-Wesley, 2002

[HaBo13] Hazzard, Kevin, Bock, Jason: Metaprogramming in .NET, Manning, 2013

[HER11] Hertel, Marcel: Portierung der RefaCola – API auf die .NET-Plattform, Masterarbeit, FernUniversität in Hagen, 2011, verfügbar unter http://www.fernuni-hagen.de/ps/arbeiten/

[MAI12] Mainka, André Michael: Typ-Constraints für generische Typen in Refacola, Diplomarbeit, FernUniversität in Hagen, 2012, verfügbar unter http://www.fernuni-hagen.de/ps/arbeiten/

[QF12] Microsoft Corporation: How to Write a Quick Fix (CSharp), 2012, Download verfügbar unter http://www.microsoft.com/en-us/download/details.aspx?id=27745 (letzter Zugriff: 18.05.2012)

[ROS12] Microsoft Corporation: The Roslyn Project, September 2012, http://msdn.microsoft.com/en-us/vstudio/hh500769, zuletzt abgerufen: 27.02.2014

[SAV09] Savara, Pavel: How Calling from .NET to Java works in jni4net, zamboch.blogspot.de/2009/10/how-calling-from-net-to-java-works.html, zuletzt abgerufen 02.04.2014

[ScrI12] Microsoft Corporation: Interactive - Scripting Introduction, 2012, Download verfügbar unter http://www.microsoft.com/en-us/download/details.aspx?id=27745 (letzter Zugriff: 18.05.2012)

[SemA12] Microsoft Corporation: Getting Started with Semantic Analysis (CSharp), 2012, Download verfügbar unter http://www.microsoft.com/en-us/download/details.aspx?id=27745 (letzter Zugriff: 18.05.2012)

[SKP11] Steimann, Friedrich, Kollee, Christian, von Pilgrim, Jens: A Refactoring Constraint Language and its Application to Eiffel, ECOOP 2011 (Seite 255-280)

[ST09] Steimann, Freidrich, Thies, Andreas: From Public to Private to Absent: Refactoring JAVA Programs under Constrained Accessibility, ECOOP 2009, Seite 419-443

[STU11] Stumpf, Kevin: Automatisierte Analyse von C#-Programmen für das Pull-Up-Field-Refactoring in MonoDevelop, Bachelorarbeit, FernUniversität in Hagen, 2011, verfügbar unter http://www.fernuni-hagen.de/ps/arbeiten/

[STvP12] Steimann, Friedrich, von Pilgrim, Jens: Constraint-Based Refactoring with Foresight, erschienen in ECOOP 2012

[SynA12] Microsoft Corporation: Getting Started with Syntax Analysis (CSharp), 2012, Download verfügbar unter http://www.microsoft.com/en-us/download/details.aspx?id=27745 (letzter Zugriff: 18.05.2012)

[SynT12] Microsoft Corporation: Getting Started with Syntax Transformations (CSharp), 2012, Download verfügbar unter http://www.microsoft.com/en-us/download/details.aspx?id=27745 (letzter Zugriff: 18.05.2012)

[TFK+11] Tip, Frank, Fuhrer, Robert M., Kiezun, Adam, Ernst, Michael D., Balaban, Ittai, de Sutter, Bjorn: Refactoring Using Type Constraints, ACM Transactions on Programming Languages and Systems, Vol. 33, No. 3, Article 9, 2011

A. Inhalt der CD

Der Inhalt der CD, welche dieser Arbeit beiliegt, wird nun tabellarisch dargestellt:

Verzeichnis / Datei	Inhalt
Spezifikation der statischen Semantik von C# in Refacola.pdf	Diese Arbeit im .pdf-Format.
Code/C#-Solution	Enthält die erstellte Solution dieser Arbeit mit den Projekten: RefacolaCsharpLanguageApi, RefacolaRefactoringVSAddIn, RefacolaTestSuite
Code/Java-Packages	Enthält die Packages, die die Java-Schnittstelle und die Refacola-Definitionen enthält: de.feu.ps.refacola.csharp.bridge (Schnittstelle), de.feu.ps.refacola.csharp.test (Testklasse), de.feu.ps.refacola.lang.csharp (Refacola-Definitionen)
Tools/jni4net	Paket von jni4net zur Erstellung der Proxy-Klassen (siehe B.3).
Tools/Roslyn CTP	Die in dieser Arbeit genutzte Version der Roslyn CTP zur Installation (siehe B.4).
Tools/Visual Studio SDK	Das von Roslyn benötigte SDK (siehe B.4).
Visual Studio Extension	Die Dateien zur Installation der Erweiterung (siehe B.5).

B. Installationsanleitung

Diese Installationsanleitung beschreibt die nötigen Schritte, um die in dieser Arbeit erstellten Komponenten einzusetzen und zu testen. Die benötigten Abhängigkeiten innerhalb der Solution liegen bereits in kompilierter Form vor und sind in den Projekten referenziert. Werden Änderungen an der Implementierung für die Refacola vorgenommen, müssen die benötigten Java-Packages in das `RefacolaAPI.jar` erneut exportiert werden. Ändert sich die Schnittstelle, muss diese in das `RefacolaApiBridge.jar` exportiert und die Proxy-Klasse neu erstellt werden. Zuerst werden kurz die System-Anforderungen spezifiziert. Folgend wird auf die benötigten Java-Packages für den Export eingegangen. Danach wird die Erstellung der für die Schnittstelle zwischen C# und Java benötigten Proxy-Klasse über jni4net betrachtet. Anschließend wird erläutert, welche Installationen gebraucht werden, um Roslyn mit Visual Studio zu nutzen, bevor die erstellte Erweiterung und deren Anforderungen beschrieben werden.

B.1 System-Anforderungen

Um die erstellten NUnit-Tests auszuführen, erfordert deren Testeinstellung X64 als standardmäßige Prozessorarchitektur. Eine Ausführung unter der Architektur X86 führte im Zusammenspiel mit Roslyn (bei der Erstellung der Test-Solution innerhalb der Testfälle) zu einer Ausnahme, die bis zum Verfassen dieser Arbeit nicht gelöst werden konnte.

Somit wird zum Ausführen der Tests und Erweiterungen ein 64-bit System mit Windows 7 benötigt. Weitere Anforderungen an z.B. die Entwicklungsumgebung werden in B.4 bei der Installation von Roslyn geschildert.

B.2 Erstellung der Java Archive

Der Export einiger Java-Packages ist nötig, um diese innerhalb des C# Projekts `RefacolaCsharpLanguageApi`, welches die Schnittstelle C# ↔ Java zur Verfügung stellt, einzubinden. Wurde lediglich an den Regel-Definitionen für C# oder der Refacola selbst etwas geändert, reicht es das Archiv `RefacolaAPI.jar` zu exportieren und über Visual Studio in den Ordner `RefacolaInterface/JniBridge/` im Projekt

RefacolaCsharpLanguageApi zu kopieren.

Hat es Änderungen z.B. aufgrund neuer Refaktorierungen in der Klasse *de.feu.ps.refacola.csharp.bridge.RefacolaInvoker* gegeben, muss zusätzlich zum *RefacolaAPI.jar*, das *RefacolaApiBridge.jar* erstellt werden. Diese Archive werden dann in B.3 zur Generierung der Proxy-Klasse benötigt.

Aufgelistet werden nun, die für die beiden Java Archive benötigten Java-Packages.

RefacolaAPI.jar:

- de.feu.ps.refacola.api
- de.feu.ps.refacola.csharp.bridge
- de.feu.ps.refacola.dsl
- de.feu.ps.refacola.factbase
- de.feu.ps.refacola.lang.csharp
- de.feu.ps.refacola.lang.java.jdt
- de.feu.ps.refacola.solvers

RefacolaApiBridge.jar:

- de.feu.ps.refacola.csharp.bridge

B.3 jni4net – Generierung der Proxy-Klasse

Durchgeführte Änderungen an der in Java implementierten und für die Proxy-Generierung notwendigen Klasse *de.feu.ps.refacola.csharp.bridge.RefacolaInvoker* erfordern die erneute Erstellung der im C# Projekt *RefacolaCsharpLanguageApi* benötigten *RefacolaApiBridge.j4n.dll*. Dazu werden die beiden in B.1 erstellten Java Archive benötigt.

Das Tool zur Proxy-Generierung „proxygen.exe“ ist in dem Paket von jni4net (auch auf CD vorliegend) vorhanden.

Über die Kommandozeile muss folgender Befehl abgesetzt werden:

```
proxygen    "<<Pfad zum Archiv>>\RefacolaApiBridge.jar"
            -wd generated
            -cp "<<Pfad zum Archiv>>\RefacolaApi.jar"
```

Im erzeugten Ordner „generated“ befinden sich nun die vorbereiteten Proxy-Klassen für die CLR und JVM. Zusätzlich wird ein Skript erstellt (build.cmd) in dem manuell die Pfade zum JDK1.6 (32bit Version) für die Aufrufe von „javac.exe“ und „jar.exe“ und auch der Pfad zur „csc.exe“ des .NET-Frameworks angepasst werden müssen. Im Ordner jni4net\bin auf der beiliegenden CD existiert bereits eine angepasste Version („buildWithPathsAdapted.cmd“). Alternativ können auch die Windows-Umgebungsvariablen auf die entsprechenden Pfade gesetzt werden. Dabei ist zu beachten, dass die richtige Version genutzt wird, also Java 1.6 (32bit) und .NET 4.5 (32bit).

Durch Ausführung des Skripts werden die beiden Dateien *RefacolaApiBridge.j4n.dll* und *RefacolaApiBridge.j4n.jar* erstellt. Diese müssen wie auch die beiden erstellten Archive aus B.1, in den Ordner *RefacolaInterface/JniBridge/* im Projekt *RefacolaCsharpLanguageApi* kopiert werden.

B.4 Installation von Roslyn

Die Installation von Roslyn ist Voraussetzung, um die in dieser Arbeit erstellen NUnit-Tests und die Visual Studio Extension nutzen zu können.

!WICHTIG!:

Die Entwicklungen dieser Arbeit benutzen die Microsoft Roslyn September 2012 CTP (auf CD verfügbar). Im April 2014, während des Verfassens dieser Arbeit, wurde Microsofts Roslyn als Open-Source-Projekt veröffentlicht, welche den letzten Entwicklungsstand von Roslyn widerspiegelt. Diese Version wird nicht mehr berücksichtigt, da diese auf Visual Studio 2013 aufbaut und einige Code-Änderungen nötig macht, die im Rahmen dieser Arbeit aus zeitlichen Gründen nicht mehr eingebracht werden können.

Für die Installation von Roslyn (beiliegend auf CD unter Tools/Roslyn CTP) werden folgende Betriebssysteme unterstützt:[50]

- Windows 7
- Windows 8
- Windows Server 2008 R2
- Windows Server 2012

Weiter wird Visual Studio 2012 ab der Professional Edition erwartet, welches zusätzlich durch das Visual Studio SDK erweitert werden muss (beiliegend auf CD unter Tools/Visual Studio SDK). Mit der Installation von Visual Studio 2012 wird auch das benötigte .NET-Framework 4.5 mitinstalliert.

Hinweis: Eine Nutzung der Visual Studio Express Editionen ist dazu nicht möglich, da diese keine Unterstützung für AddIns anbieten. Somit kann weder Roslyn noch die erstellte Erweiterung verwendet werden.

50 Roslyn: Download und Anforderungen: http://www.microsoft.com/en-us/download/details.aspx?id=34685 (letzter Zugriff: 24.05.2014)

B.5 Visual Studio Extension – RefacolaAccessibilityRefactoring

Die erstellte Visual Studio Extension nutzt die in dieser Arbeit entwickelte Schnittstelle zur Refacola und die in Refacola entwickelten Refaktorierungen, die auf den Regeln für C# aufbauen. Um die Erweiterung nutzen zu können, ist die Installation von Roslyn und dessen Abhängigkeiten Voraussetzung (siehe B.4).

Zur Installation der eigentlichen Erweiterung sind nur wenige Schritte nötig. Zuerst muss das VSIX-Paket[51] durch Ausführung der RefacolaRefactoringVSAddIn.vsix installiert werden. Dazu ist die installierte Version von Visual Studio zu wählen für die diese Erweiterung installiert werden soll. Für diese Arbeit ist dies Visual Studio 2012.

Die Installation erfolgt in das Visual Studio Installations-Verzeichnis

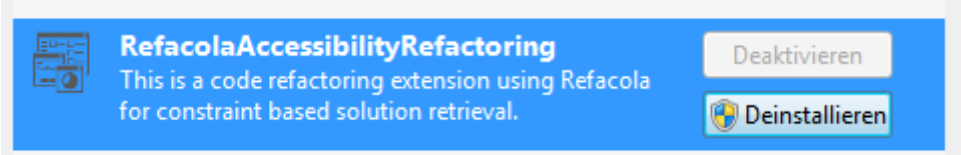
<<VisualStudioInstallationFolder>>\Common7\IDE\Extensions\<< Generated 8.3 name>>

Nach der Installation ist es nötig, den im genannten Verzeichnis vorhandenen Order „RefacolaInterface“ in den Ordner <<VisualStudioInstallationFolder>>\Common7\IDE zu kopieren.

Die Schnittstelle C# ↔ Java erfordert mit der ausgelieferten Konfiguration, das zum Schreiben und Lesen der Faktenbasis und des Changeset, die beiden Ordner

C:\GeneratedFactbase und C:\GeneratedChangeset vorhanden sind. Generell benötigt auf die konfigurierten Ordner sowohl die CLR-Runtime als auch die JVM Zugriff.

Zur Deinstallation muss über Visual Studio – Tools – Erweiterungen und Updates die Erweiterung RefacolaAccessibilityRefactoring deinstalliert werden.



Um nun die Erweiterung innerhalb der IDE zu nutzen, ist es nötig Roslyn in Visual Studio zu

51 Das VSIX-Paket ist im Prinzip ein gepacktes Archiv, welches dem Open Packaging Convention Standard (OPC) folgt. Dieses Paket enthält alle zur Ausführung nötigen Dateien (vgl. http://msdn.microsoft.com/en-us/library/ff363239%28v=vs.110%29.aspx (letzter Zugriff: 24.05.2014))

aktivieren. Dies geschieht über das Ausführen von Visual Studio mit dem Startparameter */rootsuffix Roslyn.* Dazu wird am besten eine neue Verknüpfung angelegt, sodass diese die IDE mit Roslyn startet.

B.6 Konfiguration im Projekt der Schnittstelle

Das Projekt *RefacolaCsharpLanguageApi*, in dem die Schnittstelle implementiert ist, besitzt eine .settings-Datei, in der die benötigten Pfade zu definieren sind. Auch die in B.5 genannten Pfade können dort angepasst werden. Eine Anpassung erfordert jedoch das Neuerstellen der Projekte und möglicherweise der De- und Neuinstallation der Erweiterung. Die folgende Tabelle zeigt nun die Einstellungen mit den aktuell gesetzten Werten:

Name	Wert	Beschreibung
JavaHomePathTo1_6_32Bit	C:\Program Files (x86)\Java\jre6	Pfad zur 32bit Version von Java 1.6; genutzt für die Visual Studio Erweiterung
ChangesetPath_NUnitTests	../../GeneratedChangeset/	Absoluter Pfad zur Ablage der Faktenbasis; genutzt für die NUnit-Tests in Visual Studio
FactbasePath_NUnitTests	../../GeneratedFactbase/	Relativer Pfad zur Ablage der Changeset-Datei im Projekt; genutzt für die NUnit-Tests in Visual Studio
ChangesetPath_VSAddIn	C:\GeneratedChangeset\	Absoluter Pfad zur Ablage der Changeset-Datei; genutzt für die Visual Studio Erweiterung
FactbasePath_VSAddIn	C:\GeneratedFactbase\	Absoluter Pfad zur Ablage der Faktenbasis; genutzt für die Visual Studio Erweiterung
JavaHomePathTo1_6_64Bit	C:\Program Files\Java\jdk1.6.0_39	Pfad zur 64bit Version von Java 1.6; genutzt für die NUnit-Tests in Visual Studio

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Frank Rogel, Frankfurt/Main, den 10. Juni 2014